- Axon ----------

Axon.AdaptiveCommsComponent

"Adaptive Comms Components" - can add and remove inboxes and outboxes

An AdaptiveCommsComponent is just like an ordinary component but with the ability to create and destroy extra inboxes and outboxes whilst it is running.

- An AdaptiveCommsComponent is based on an Axon.Component.component

There are other variants on the basic component:

- Axon.ThreadedComponent.threadedcomponent
- Axon.ThreadedComponent.threadedadaptivecommscomponent

If your component needs to block - eg. wait on a system call; then make it a 'threaded' component. If it needs to change what inboxes or outboxes it has at runtime, then make it an 'adaptive' component. Otherwise, simply make it an ordinary component!

Adding and removing inboxes and outboxes

To add a new inbox or outbox call self.addInbox() or self.addOutbox() specifying a base name for the inbox/outbox. The created inbox or outbox is immediately ready to be used.:

```
actualInboxName = self.addInbox("inputData")
actualOutboxName = self.addOutbox("outputData")
```

You specify a name you would ideally like the inbox or outbox to be given. If that name is already taken then a variant of it will be generated. Calls to addInbox() and addOutbox() therefore return the actual name the inbox or outbox was given. You should always use this returned name. It is unwise to assume your ideal choice of name has been allocated!

To remove a box, call self.deleteInbox() or self.deleteOutbox() specifying the name of the box to be deleted:

```
self.deleteInbox(actualInboxName)
self.deleteOutbox(actualOutboxName)
```

When deleting an inbox or outbox, try to make sure that any linkages involving that inbox/outbox have been destroyed. This includes not only linkages created by your component, but any created by other components too.

Tracking resources

adaptivecommscomponent also includes an ability to track associations between resources and inboxes, outboxes and other information.

For example, you might want to associate another component (that your component is interacting with) with the set of inboxes, outboxes and any other info that are being used to communicate with it.

You can also associate particular inboxes or outboxes with those resources. This therefore allows you to map both ways: "which resource relates to this inbox?" and "which inboxes relate to this resource?"

For example, suppose a request leads to your component creating an inbox and outbox to deal with another component. You might store these as a tracked resource, along with other information, such as the 'other' component and any state or linkages that were created; and associate this resource with the inbox from which data might arrive:

```
def wireUpToOtherComponent(self, theComponent):
    newIn  = self.addInbox("commsIn")
    newOut = self.addOutbox("commsOut")

    newState = "WAITING"
    inLinkage  = self.link((theComponent,itsOutbox),(self,newIn))
    outLinkage = self.link((theComponent,itsInbox), (self,newOut))

    resource = theComponent

    inboxes = [newIn]
    outboxes = [newOut]
    info = (newState, inLinkage, outLinkage)
    self.trackResourceInformation(resource, inboxes, outboxes, info)

    self.trackResource(resource, newIn)
```

If a message then arrives at that inbox, we can easily look up all the information we might need know where it came from and how to handle it:

```
def handleMessageArrived(self, inboxName):
    msg = self.recv(inboxName)

    resource = self.retrieveResource(inboxName)
    inboxes, outboxes, info = self.retrieveResourceInformation(resource)
    theComponent=resource

    ...
```

When you are finished with a resource and its associated information you

can clean it up with the ceaseTrackingResource() method which removes the association between the resource and information. For example when you get rid of a set of linkages and inboxes or outboxes associated with another component you might want to clean up the resource you were using to track this too:

```
def doneWithComponent(self, theComponent):
    resource=theComponent
    inboxes, outboxes, info = self.retrieveResourceInformation(resource)

    for name in inboxes:
        self.deleteInbox(name)
    for name in outboxes:
        self.deleteOutbox(name)

    state,linkages = info[0], info[1:]
    for linkage in linkages:
        self.unlink(thelinkage=linkage)

    self.ceaseTrackingResource(resource)
```

Implementation

AdaptiveCommsComponent's functionality above and beyond the ordinary Axon.Component.component is implemented in a separate mixin class _AdaptiveCommsable. This enables it to be reused for other variants on the basic component that need to inherit this functionality - such as the threadedadaptivecommscomponent.

When adding new inboxes or outboxes, name clashes are resolved by permuting the box name with a suffixed unique ID number until there is no longer any clash.

Other

AdaptiveCommsComponent

> Base class for a component that works just like an ordinary component but can also 'adapt' its comms by adding or removing inboxes and outboxes whilst it is running.
>
> Subclass to make your own.
>
> See Axon.AdaptiveCommsComponent._AdaptiveCommsable for the extra methods that this subclass of component has.

Axon.AxonExceptions

Axon Exceptions

AxonException is the base class for all axon exceptions defined here.

Other

AccessToUndeclaredTrackedVariable

Attempt to access a value being tracked by the coordinating assistant tracker that isn't actually being tracked yet!

Arguments:

- the name of the value that couldn't be accessed
- the value that it was to be updated with (optional)

Possible causes:

- Attempt to update or retrieve a value with a misspelt name?
- Attempt to update or retrieve a value before it starts being tracked?

ArgumentsClash

Supplied arguments clash with each other.

Possible causes:

- meaning of arguments misunderstood? not allowed this given combination of arguments or values of arguments?

AxonException

Base class for axon exceptions.

Any arguments listed are placed in self.args

BadComponent

The object provided does not appear to be a proper component.

Arguments:

- the 'component' in question

Possible causes:

- Trying to register a service (component,boxname) with the coordinating assistant tracker supplying something that isn't a component?

BadInbox

The inbox named does not exist or is not a proper inbox.

Arguments:

- the 'component' in question
- the inbox name in question

Possible causes:

- Trying to register a service (component,boxname) with the coordinating assistant tracker supplying something that isn't a component?

BadParentTracker

Parent tracker is bad (not actually a tracker?)

Possible causes:

- creating a coordinatingassistanttracker specifying a parent that is not also a coordinatingassistanttracker?

BoxAlreadyLinkedToDestination

The inbox/outbox already has a linkage going *from* it to a destination.

Arguments:

- the box that is already linked
- the box that it is linked to
- the box you were trying to link it to

Possible causes:

- Are you trying to make a linkage going from an inbox/outbox to more than one destination?
- perhaps another component has already made a linkage from that inbox/outbox?

MultipleServiceDeletion

Trying to delete a service that does not exist.

Possible causes:

- Trying to delete a service (component,boxname) from the coordinating assistant tracker twice or more times?

NamespaceClash

Clash of names.

Possible causes:

- two or more requests made to coordinating assistant tracker to track values under a given name (2nd request will clash with first)?
- should have used updateValue() method to update a value being tracked by the coordinating assistant tracker?

ServiceAlreadyExists

A service already exists with the name you specifed.

Possible causes:

- Two or more components are trying to register services with the coordinating assistant tracker using the same name?

invalidComponentInterface

Component does not have the required inboxes/outboxes.

Arguments:

- *"inboxes"* or *"outboxes"* - indicating which is at fault
- the component in question
- (inboxes,outboxes) listing the expected interface

Possible causes:

- Axon.util.testInterface() called with wrong interface/component specified?

noSpaceInBox

Destination inbox is full.

Possible causes:

- The destination inbox is size limited?
- It is a threaded component with too small a 'default queue size'?

normalShutdown

NO DOCS

Axon.Base

Axon base classes

What is defined here is a metaclass that is used as a base class for some key classes in Axon.

It was originally created to allow super class calling in a slightly nicer manner in terms of syntactic sugar easier to get right that still has the good effects of "super" in a multiple inheritance scenario. **Use of this particular feature has been deprecated** because of more subtle issues in inheritance situations.

However this metaclass has been retained (and is still used) for possible future uses.

- AxonObject is the base class for Axon.Microprocess.microprocess and Axon.Linkage.linkage

Other

AxonObject

Base class for axon objects.

AxonType

NO DOCS

Axon.Box

Axon postboxes - inboxes and outboxes

The objects used to implement inboxes and outboxes. They store and handle linkages and delivery of messages from outbox to inbox.

- Components create postboxes and use them as their inboxes and outboxes.

This is an Axon internal. If you are writing components you do not need to understand this as you will normally not use it directly.

Developers wishing to use Axon in other ways or understand its implementation shoudl read on with interest!

Example Usage

Creation Creating an outbox:

```
def outboxNotify():
    print("A message was collected from an inbox that this outbox is linked to.")
```

```
myOutbox = makeOutbox(outboxNotify)
```

Creating an inbox:

```
def inboxNotify():
    print("A new message has arrived at this inbox.")

myInbox = makeInbox(inboxNotify)
```

Creating an inbox that is limited to holding 10 items:

```
mySizeLimitedInbox = makeInbox(inboxNotify, size=10)
```

Alternative syntax to do the same:

```
mySizeLimitedInbox = makeInbox(inboxNotify)
mySizeLimitedInbox.setSize(10)
```

Adding/Removing Linkages Create outboxes A and B, and inboxes C and D, then linking them in a chain A to B to C to D:

```
boxA = makeOutbox( <notify callback> )
boxB = makeOutbox( <notify callback> )

boxC = makeInbox( <notify callback> )
boxD = makeInbox( <notify callback> )

boxB.addsource(boxA)
boxC.addsource(boxB)
boxD.addsource(boxC)
```

We can also remove one of those linkages:

```
boxC.removeSource(boxB)
```

More detail

Call makeInbox() or makeOutbox() to make an inbox or outbox respectively.

Both inboxes and outboxes are instances of the postbox class. postboxes provide a subset of the python list interface to let you add and remove items from it:

- **postbox.append(data)** - ie. send a message
- **postbox.pop(data)** - ie. collect a message
- **postbox.___len___()** - ie. len(myPostbox)

Inboxes An inbox is a postbox with storage. Calling append() will put a message into that inbox. Calling len() will report the number of items in the inbox, and pop() will enable you to take items out.

Inboxes can be size limited. If it becomes full then trying to append() will raise an Axon.AxonExceptions.noSpaceInBox exception.

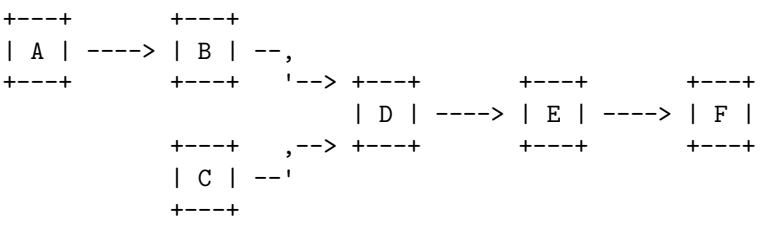
Outboxes An outbox is a postbox with no storage. Calling append() will silently discard the message. len() will report the box as containing zero items; and calling pop() will, as expected, raise an IndexError exception.

Linking them together Boxes can be wired together, so that posting a message to one actually results in the message appearing in another. Axon does this when you make a link between postboxes on different components. Links have direction. Messages flow only one way along a link - from source to target/destination/sink.

Boxes can be wired up in a many-to-one tree structure - where many sources feed their messages, along one or more hops through inbetween postboxes, towards a single destination:

- postbox.addsource(source_postbox)
- postbox.removeSource(source_postbox)

Suppose you wire up boxes to form a tree:



Sending a message using the append() method from A,B,C,D or E will result in the message being sent to F. Make sure F is an outbox, otherwise the message will be lost!

When a box is wired to another, it diverts calls to append() to the final destination instead of its own local storage; so A,B,C,D and E can be inboxes or outboxes - it doesn't matter.

You are not allowed to create links going from one source to two or more destinations (one-to-many arrangements). If you try, an Axon.AxonExceptions.BoxAlreadyLinkedToDestination exception will be raised.

How is it implemented?

Calling makeInbox() or makeOutbox() creates an instance of the postbox class. A postbox behaves like a simple piece of storage, accessed using the append(), pop() and len() methods. However, if the postbox is linked to others, then the storage that is actually accessed belongs to the target postbox (the final destination in the chain).

This storage is therefore actually a separate object, held inside a postbox. When postboxes are wired together, they all reconfigure themselves so that calls to

append(), len() and pop() actually access the same storage in the target postbox. In the postbox class, where the messages actually get sent to is referred to as the sink.

For inboxes, this is an instance of the realsink class (that actually stores stuff). But for outboxes, it is an instance of the nullsink class (that just discards stuff given to it, and always appears empty). This is so that messages that end up at outboxes don't pile up, uncollected.

For example, suppose we link three postboxes in a chain:

```
+-------------------+      +-------------------+      +----------------------+
|     postbox A     | ---> |     postbox B     | ---> |      postbox C       |
|                   |      |                   |      |                      |
| A.target = C      |      | B.target = C      |      | C.target = None      |
| A.sink   = C.sink |      | B.sink   = C.sink |      | C.sink   = C.storage |
+-------------------+      +-------------------+      +----------------------+
```

The target of postboxes A and B is postbox C. The sinks used by all three is the storage beloinging to postbox C. Calls to append(), pop() and len() made to any of the three postboxes are all direected to the storage in postbox C.

The links between postboxes are represented internally as a list of sources for each postbox. For example:

```
+---+       +---+
| A | ----> | B | --,
+---+       +---+   '--> +---+       +---+       +---+
                        | D | ----> | E | ----> | F |
            +---+   ,--> +---+       +---+       +---+
            | C | --'
            +---+
```

```
A.sources = []
B.sources = [A]
C.sources = []
D.sources = [B,C]
E.sources = [D]
F.sources = [E]
```

Links are created an destroyed by calling addsource() or removeSource(). So for example, to wire up postbox D in the above example, the following calls were made:

```
D.addsource(B)
D.addsource(C)
```

Internally, addsource() and removeSource() calls _retarget() which recurses back up the chain of linkages, updating any other boxes that feed into the source, to make sure they all now point at the new target too.

addsource() also delivers any messages waiting in the source's storage to the new destination's storage. This ensures that messages do not get lost halfway along a chain of linkages when the chain is extended.

Because all postboxes in a chain end up redirecting calls to the target postbox's storage; a separate self.local_len() method is provided to allow a component to find out whether there is any items waiting in its own postbox. A component's inbox might not be the final destination in a chain, so it is important that if the component attempts to examine its own inbox for new items it should not inadvertently query the final destination instead.

Notification that a message has been delivered When creating an inbox, you provide a notification callback that will be called whenever a new message arrives at that box. Axon uses this to wake the component that owns that inbox.

The realsink object keeps note of this callback, and calls it when a new message is delivered to it (ie. its append() method is called).

Notification that a message has been collected When a message is collected; some parties in the chain of linked boxes may wish to be notified. Axon uses this to wake owners of outboxes linked to the destination inbox from which the message has been collected. You therefore provide a notification callback when creating an outbox.

The realsink object keeps a 'wakeOnPop' list of callbacks to call when its pop() method is called.

When linkages are added or removed, the storage of all inboxes downstream of where the change has occurred must update their list of 'wakeOnPop' callbacks. Therefore addsource() or removeSource() also call _addNotifys() or _removeNotifys() respectively, which recurse down the chain of linkages towards the target, updating the list of callbacks as they go.

Notifications - performance All this climbing up and down of the chain of linkages to update lists of callbacks takes time - O(n) where n is the number of postboxes in the chain.

Paying this cost upfront means that the overheads of actually delivering or collecting messages is substantially less because all the data is already there and up to date. In general, it is felt that messages are likely to be sent far more often than linkages are created and destroyed - which should justify this tradeoff.

Other

ShowAllTransits

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

makeInbox

Returns a new postbox object suitable for use as an Axon inbox.

Keyword arguments:

- notify -- notify() will be called whenever a message arrives at this inbox.
- size -- None, or a limit on the maxmimum number if items this inbox can hold (default=None)

makeOutbox

Returns a new postbox object suitable for use a an Axon outbox.

Keyword arguments:

- notify -- notify() will be called whenever a message is collected from an inbox that this outbox delivers to.

nullsink

nullsink() -> new nullsink object

A dummy piece of storage for postboxes, that behaves a bit like a list.

Discards data given to it by calling append() and always reports that it contains no items.

postbox

postbox(storage[,notify]) -> new postbox object.

Creates a postbox, using the specified storage as default storage. Storage should have the interface of list objects.

Also takes optional notify callback, that will be called whenever an item is taken out of a postbox further down the chain.

realsink

realsink(notify[,size]) -> new realsink object.

A working piece of storage for postboxes, that behaves a bit like a list.

Stores data given to it by calling append(), up to a limit after which Axon.AxonExceptions.noSpaceInBox exceptions are raised.

Calls the 'notify' callback when append() is called. Calls any callbacks in the self.wakeOnPop list when pop() is called.

Keyword arguments:

- notify -- notify() is called whenever append() is called
- size -- None, or the maximum number of items this storage can hold

NEED TO TIGHTEN UP IMPORTS from Axon.Ipc import *

Axon.Component

Components - the basic building block

A component is a microprocess with a microthread of control and input/output queues (inboxes and outboxes) which can be connected by linkages to other components. Ie. it has code that runs whilst the component is active, and mechanisms for sending and receiving data to and from other components.

- A component is based on a microprocess - giving it its thread of execution.

There are other variants on the basic component:

- Axon.AdaptiveCommsComponent.AdaptiveCommsComponent
- Axon.ThreadedComponent.threadedcomponent
- Axon.ThreadedComponent.threadedadaptivecommscomponent

If your component needs to block - eg. wait on a system call; then make it a 'threaded' component. If it needs to change what inboxes or outboxes it has at runtime, then make it an 'adaptive' component.

The basics of writing a component

Here's a simple example:

```
class MyComponent(Axon.Component.component):

    Inboxes = { "inbox"   : "Send the FOO objects to here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Emits BAA objects from here",
                 "signal" : "NOT USED",
               }

    def main(self):
        while 1:
```

```
        if self.dataReady("inbox"):
            msg = self.recv("inbox")
            result = ... do something to msg ...
            self.send(result, "outbox")

        yield 1
```

Or, more specifically:

1. **Subclass the component class**. Don't forget to call the superclass initializer if you write your own ___init___ method:

   ```
   class MyComponent(Axon.Component.component):

       def __init__(self, myArgument, ...):
           super(MyComponent, self).__init__()
           ...
   ```

2. **Declare any inboxes or outboxes you expect it to have** - do this as static members of the class called "Inbox" and "Outbox". They should be dictionaries where the key is the box name and the value serves as documentation describing what the box is for:

   ```
   Inboxes = { "inbox"   : "Send the FOO objects to here",
               "control" : "NOT USED",
             }
   Outboxes = { "outbox" : "Emits BAA objects from here",
                "signal" : "NOT USED",
              }
   ```

   You can also do this as lists, but doing it as a dictionary with documentation values is much more useful to people wanting to use your component.

   If you don't specify any then you get the default "inbox" and "control" inboxes and "outbox" and "signal" outboxes.

3. **Write the main() method** - it must be a generator - namely just like an ordinary method or function, but with 'yield' statements regularly interspersed through it. This is the thread of execution in the component -just use yield statements regularly so other components get time to execute.

   If you need to know more, these might help you:

   - A tutorial on "How to write components"
   - the Mini Axon tutorial
   - the Axon.Microprocess.microprocess class

Running a component

Once you have written your component class; simply create instances and activate them; then start the scheduler to begin execution:

```
x = MyComponent()
y = MyComponent()
z = AnotherComponent()
x.activate()
y.activate()
z.activate()

scheduler.run.runThreads()
```

If the scheduler is already running, then simply activating a component will start it executing.

Communicating with other components - creating linkages

Components have inboxes and outboxes. The main() thread of execution in a component communicates with others by picking up messages that arrive in its inboxes and sending out messages to its outboxes. For example here is a simple block of code for the main() method of a component that echoes anything it receives to the console and also sends it on:

```
if self.dataReady("inbox"):
    msg = self.recv("inbox")
    print str(msg)
    self.send(msg,"outbox")

yield 1
```

Use the dataReady() method to find out if there is (one or more items of) data waiting at the inbox you name. recv() lets you collect a message from an inbox. Use the send() method to send a message to the outbox you name. There is also an anyReady() method that will tell you if *any* inbox has data waiting in it.

A message gets from one component's outbox to another one's inbox if there is a linkage between them (going from the outbox to the inbox). A component can create and destroy linkages by using the link() and unlink() methods.

For example, we could create a linkage from a component's outbox called "outbox" to a different component's inbox called "inbox":

```
theLink = self.link( (self, "outbox"), (otherComponent, "inbox") )
```

Using the handle that we were given, we can destroy that linkage later:

```
self.unlink(theLinkage = theLink)
```

Linkages normally go from an outbox to an inbox - after all the whole idea is to get messages that one component sends to its own outbox to arrive at another component's inbox. However you can also create 'passthrough' linkages from an inbox to another inbox; or from an outbox to another outbox.

This is particularly useful if you want to encapsulate a child component - hide it from view so other components only need to be wired up to you.

For example, your component may want any data being sent to one of its inboxes to be forwarded automatically onto an inbox on the child. This is a type '1' passthrough linkage:

```
thelink = self.link( (self,"inbox"), (myChild,"inbox"), passthrough=1 )
```

Or if you want anything a child sends to its outbox to be sent out of one of your own outboxes, which is a type '2' passthrough:

```
thelink = self.link( (myChild,"outbox"), (self,"outbox"), passthrough=2 )
```

The alternative, of course, is to add extra inboxes and outboxes to communicate with the child, and to write a main() method that simply passes the data on. Passthrough linkages are more efficient and quicker to code!

There is no performance penalty for delivering a message along a such a chain of linkages (eg. outbox ---> outbox ---> inbox ---> inbox ---> inbox). Axon resolves the chain and delivers straight to the final destination inbox!

Child components

Components can create and activate other components. They can adopt them as children:

```
newComponent = FooComponent()
self.addChildren(newComponent)
newComponent.activate()
```

Making a component your child means that:

- you will be woken (if asleep) when your child terminates
- the removeChild() method provides a convenient way to make sure any linkages you have made involving that child are destroyed.
- calling childComponents() lists all children you currently have

Whether another component is your child or not, you can tell if it has terminated yet by calling its _isStopped() method.

For example, a component might want to create a child component, make a linkage to it then wait until that child terminates before cleaning it up. Achieve this by writing code like this in the main() body of the component:

```
src = RateControlledFileReader("myTextFile",readmode="lines")
dst = ConsoleEchoer()

lnk = self.link( (src,"outbox"), (dst,"inbox") )

self.addChildren(src,dst)    # we will be woken if they terminate
src.activate()
```

```
dst.activate()

while not dst._isStopped() and not src._isStopped():
    self.pause()
    yield 1

self.unlink(thelinkage = lnk)
self.removeChild(src)
self.removeChild(dst)
```

Going to sleep (pausing)

When a component has nothing to do it can pause. This means it will not be executed again until it is woken up. Pausing is a good thing since it relinquishes cpu time for other parts of the system instead of just wasting it.

When would you want to pause? - usually when there is nothing left waiting in any of your inboxes and there is nothing else for your component to do:

```
class Echoer(Axon.Component.component):

    def main(self):
        while 1:
            if self.dataReady("inbox"):
                msg = self.recv("inbox")
                print str(msg)
                self.send(msg,"outbox")

            if not self.anyReady():
                self.pause()

            yield 1
```

Calling the pause() method means that *at the next yield statement* your component will be put to sleep. It doesn't happen as soon as you call pause() -only when execution reaches the next `yield`.

What will wake up a paused component? - any of the following:

- a message arriving at any inbox (even one with messages already waiting in it)
- a message being collected from an inbox that is linked to one of our outboxes
- a child component terminating

Your component *cannot* wake itself up - only the actions of other components can cause it to be woken. Why? You try writing some code that stops executing (pauses) yet can issue a method call to ask to be woken! :-)

Old style main thread

Components also currently support an 'old' style for writing their main thread of execution.

This way of writing components is likely to be deprecated in the near future. We suggest you write a main() method as a generator instead.

To do old-style; instead of writing a generator you write 3 functions - one for initialisation, main execution and termination. The thread of execution for the component is therefore effectively this:

```
self.initialiseComponent()

while 1:
    result = self.mainBody()
    if not result:
        break

self.closeDownComponent()
```

initialiseComponent() is called once when the component first starts to execute.

mainBody() is called every execution cycle whilst it returns a non zero value. If it returns a zero value then the component starts to shut down.

closeDownComponent() is called when the component is about to shut down.

A simple echo component might look like this:

```
class EchoComponent(Axon.Component.component):

    def initialiseComponent(self):
        # We can perform pre-loop initialisation here!
        # In this case there is nothing really to do
        return 1

    def mainBody(self):
        # We're now in the main loop
        if self.dataReady("inbox"):
            msg = self.recv("inbox")
            print msg
            self.send(msg,"outbox")

        if not self.anyReady():
            self.pause()

        # If we wanted to exit the loop, we return a false value
        # at the end of this function
        # If we want to continue round, we return a true, non-None,
```

```
        # value. This component is set to run & run...
        return 1

def closeDownComponent(self):
        # We can't get here since mainLoop above always returns true
        # If it returned false however, we would get sent here for
        # shutdown code.
        # After executing this, the component stops execution
        return
```

Internal Implementation

Component is a subclass of microprocess (from which it inherits the ability to have a thread of execution). It includes its own postoffice object for creating and destroying linkages. It has a collection of inboxes and outboxes.

Attributes:

- **Inboxes** and **Outboxes** - static members defining the list of names for inboxes and outboxes a component should be given when it is created.
- **inboxes** and **outboxes** - the actual collections of inboxes and outboxes that are set up when the component is initialised. Implemented as dictionaries mapping names to the postbox objects.
- **postoffice** - a postoffice object, used to create, destroy and manage linkages.
- **children** - list of components registered as children of this component.
- **__callOnCloseDown** - list of callback functions to be called when this component terminates. Used to notify parents of children that their child has terminated.

Other

TraceAllRecvs

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

TraceAllSends

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

WaitComplete

WaitComplete(generator) -> new WaitComplete object.

Message to ask the scheduler to temporarily suspect this microprocess and run a new one instead based on the generator provided; resuming the original when the new one completes.

Use within a microprocess by yielding one back to the scheduler.

Arguments:

- the generator to be run as the separate microprocess

component

Base class for an Axon component. Subclass to make your own.

A simple example:

```
class IncrementByN(Axon.Component.component):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
                value = value + self.n
                self.send(value,"outbox")

            if not self.anyReady():
                self.pause()

            yield 1
```

errorInformation

errorInformation(caller[,exception][,message]) -> new errorInformation ipc message.

A message to indicate that a non fatal error has occured in the component. It may skip processing errored data but should respond correctly to future messages.

Keyword arguments:

- caller -- the source of the error information. Assigned to self.caller
- exception -- Optional. None, or the exception that caused the error. Assigned to self.exception
- message -- Optional. None, or a message describing the problem. Assigned to self.message

ipc

Message base class

newComponent

newComponent(*components) -> new newComponent ipc message.

Message used to inform the scheduler of a new component that needs a thread of control and activating.

Use within a microprocess by yielding one back to the scheduler.

Arguments:

- the components to be activated

notify

notify(caller,payload) -> new notify ipc message.

Message used to notify the system of an event. Subclass to implement your own specific notification messages.

Keyword arguments:

- caller -- a reference to whoever/whatever issued this notification. Assigned to self.caller
- payload -- any relevant payload relating to the notification. Assigned to self.object

producerFinished

producerFinished([caller][,message]) -> new producerFinished ipc message.

Message to indicate that the producer has completed its work and will produce no more output. The receiver may wish to shutdown.

Keyword arguments:

- caller -- Optional. None, or the producer who has finished. Assigned to self.caller
- message -- Optional. None, or a message giving any relevant info. Assigned to self.message

reactivate

reactivate(original) -> new reactivate ipc message.

Returned by Axon.Microprocess.microprocess._closeDownMicroprocess() to the scheduler to get another microprocess reactivated.

Keyword arguments:

- original -- The original microprocess to be resumed. Assigned to self.original

shutdown

Message used to indicate that the component recieving it should shutdown.

Due to legacy mistakes, use shutdownMicroprocess instead.

shutdownMicroprocess

shutdownMicroprocess(*microprocesses) -> new shutdownMicroprocess ipc message.

Message used to indicate that the component recieving it should shutdown. Or to indicate to the scheduler a shutdown knockon from a terminating microprocess.

Arguments:

- the microprocesses to be shut down (when used as a knockon)

status

status(status) -> new status ipc message.

General Status message.

Keyword arguments:

- status -- the status.

wouldblock

wouldblock(caller) -> new wouldblock ipc message.

Message used to indicate to the scheduler that the system is likely to block now.

Keyword arguments:

- caller -- who it is who is likely to block (presumably a micro-process). Assigned to self.caller

Axon.CoordinatingAssistantTracker

Co-ordinating Assistant Tracker

The co-ordinating assistant tracker is designed to allow components to register services and statistics they wish to make public to the rest of the system. Components can also query the co-ordinating assistant tracker to create linkages to specific services, and for specific global statistics.

- A co-ordinating assistant tracker is shared between several/all components
- Components can register an inbox as a service with a name
- Components can retrieve a service by its name

Accessing the Co-ordinating Assistant Tracker

Co-ordinating assistant trackers are designed to work in a singleton manner; accessible via a local or class interface (though this is not enforced).

The simplest way to obtain the global co-ordinating assistant tracker is via the getcat() class (static) method:

```
from Axon.CoordinatingAssistantTracker import coordinatingassistanttracker

theCAT = coordinatingassistanttracker.getcat()
```

The first time this method is called, the co-ordinating assistant tracker is created. Subsequent calls, wherever they are made from, return that same instance.

Services

Components can register a named inbox on a component as a named service. This provides a way for a component to provide a service for other components - an inbox that another component can look up and create a linkage to.

Registering a service is simple:

```
theComponent = MyComponentProvidingServiceOnItsInbox()
theComponent.activate()
```

```
theCAT = coordinatingassistanttracker.getcat()
theCAT.registerService("MY_SERVICE", theComponent, "inbox")
```

Another component can then retrieve the service:

```
theCAT = coordinatingassistanttracker.getcat()
(comp, inboxname) = theCAT.retrieveService("MY_SERVICE")
```

Because services are run by components - these by definition die and so also need to be de-registered:

```
theCAT = coordinatingassistanttracker.getcat()
theCAT.deRegisterService("MY_SERVICE")
```

Tracking global statistics

Microprocesses can also use the co-ordinating assistant tracker to log/retrieve statistics/information.

Use the trackValue() method to initially start tracking a value under a given name:

```
value = ...

theCAT = coordinatingassistanttracker.getcat()
theCAT.trackValue("MY_VALUE", value)
```

This can then be easily retrieved:

```
theCAT = coordinatingassistanttracker.getcat()
value= theCAT.retrieveValue("MY_VALUE")
```

Call the updateValue() method (not the trackValue() method) to update the value being tracked:

```
newvalue = ...

theCAT = coordinatingassistanttracker.getcat()
theCAT.updateValue("MY_VALUE", newvalue)
```

Hierarchy of co-ordinating assistant trackers

Although at initialisation a parent co-ordinating assistant tracker can be specified; this is not currently used.

Other

coordinatingassistanttracker

coordinatingassistanttracker([parent]) -> new coordinatingassistant-tracker object.

Co-ordinating assistant tracker object tracks values and (component,inboxname) services under names.

Keyword arguments:

- parent -- Optional. None, or a parent coordinatingassistant-tracker object.

Polite Notice

The code you are using includes using Axon.Handle. This code is currently experimental - we'd welcome any issues you may find/experience with this code.

Axon.Handle

Communicating with components from non Axon code

The Handle component wraps another component and allows data to be sent to and received from its standard inboxes ("inbox" and "control") and standard outboxes ("outbox" and "signal"). It provides this via thread safe, non-blocking get() and put() methods.

This is particularly useful in combination with Axon.background - allowing communication with components running in the background of a non Axon based piece of code.

Still Experimental

This code is currently experimental - we'd welcome reports of any issues you may encounter when using this code.

Example Usage

Here, Axon/Kamaelia is used to connect to a server then receive text, chunking it into individual lines. This is done by using Axon in the background (since other code in this hypothetical system is not Axon based).

NOTE: To see how this could be done without using Axon.Handle, see the examples in the documentation for Axon.background.

1. We create a background object and call its start() method:

   ```
   from Axon.background import background

   background().start()
   ```

2. We create and activate our Kamaelia pipeline of components, wrapped in an instance of the Handle class:

```
    from Axon.Handle import Handle
    from Kamaelia.Chassis.Pipeline import Pipeline
    from Kamaelia.Internet.TCPClient import TCPClient
    from Kamaelia.Visualisation.PhysicsGraph import chunks_to_lines

    queue = queue()

    connection = Handle(
        Pipeline(
            TCPClient("my.server.com", 1234),
            chunks_to_lines()
        )
    ).activate()
```

We can now fetch items of data when they arrive, using the Handle, from the "outbox" outbox of the pipeline:

```
from queue import Empty

while 1:
   try:
       received_line = connection.get("outbox")
       print( "Received line:", received_line)
   except Empty:
       # no data yet
       time.sleep(0.1)
```

We can also send data, back to the server, by sending it to the "inbox" inbox of the pipeline:

```
connection.put("Bytes to send to server\n", "inbox")
```

Behaviour

Handle is a threaded component. It does not have the standard inboxes ("inbox" and "control") or standard outboxes ("outbox" and "signal"). The only way to communicate with Handle is via its get() and put() methods.

Instantiate Handle, passing it a component to wrap. Upon activation, Handle automatically wires inboxes and outboxes of its own to the "inbox" and "control" inboxes and "outbox" and "signal" outboxes of the component it is wrapping. Handle then activates the wrapped component.

To send data to the wrapped component's "inbox" or "control" inboxes, call the put() method, specifying, as arguments, the item of data and the name of the inbox it is destined for. The data is queued and sent at the next opportunity.

To retrieve data sent out by the wrapped component's "outbox" or "signal" outboxes, call the get() method, specifying, as an argument, the name of the

outbox in question. This method is *non blocking* - if there is data waiting, then the oldest item of data is returned, otherwise a queue.Empty exception is immediately thrown.

When the wrapped component terminates, Handle will immediately terminate. Handle does not respond to shutdown messages received from the wrapped component. Handle cannot be sent shutdown messages since it has no "control" inbox on which to receive them.

Limitations

Limited to standard inboxes and outboxes Handle currently only provides access to the standard "inbox" and "control" inboxes and standard "outbox" and "signal" outboxes of the component it wraps.

If access is required to a different inbox or outbox, try wrapping the component within a Kamaelia.Chassis.Graphline component and specifying linkages to connect the inbox or outbox in question to one of the standard inboxes or outboxes of the Graphline.

CPU Usage The current implememntation of Handle involves a degree of polling. However it does use a slight (approximately 1 centisecond) delay between pollings.

Therefore when idle, CPU usage of this component will be slightly greater than zero.

Design rationale and history

This component is the successor to the earlier "likefile" component. Likefile suffered from some design issues that resulted in occassional race conditions.

We dropped the name "LikeFile" since whilst it derives from the concept of a file handle, it doesn't use the same API as file() for some good reasons we'll come back to.

A file handle is an opaque thing that you can .write() data to, and .read() data from. This is a very simple concept and belies a huge amount of parallel activity happening concurrently to your application. The file system is taking your data and typically buffering it into blocks. Those blocks then may need padding, and depending on the file system, may actually be written immediately to the end of a cyclone buffer in a journal with some write operations. Then periodically those buffers get flushed to the actual disk.

Based on the fact that file handles are a very natural thing for people to work with, based on their ubiquity, and the fact that it masks the fact you're accessing a concurrent system from a linear one, that's why we've taken this approach for integrating Kamaelia components (which are naturally parallel) with non-Kamaelia code, which is typically not parallel.

For simplicity of implementation, initially the implementation of Handle supports only the equivalent of non-blocking file handles. This has two implications:

- Reading data from a Handle may fail, since there may not be any ready yet. This is chosen in preference to a blocking operation
- Writing data to a Handle may also fail, since the component may not actually be ready to receive data from us.

Other

Handle

NO DOCS

Axon.Introspector

Detecting the topology of a running Axon system

The Introspector is a component that introspects the current local topology of an Axon system - that is what components there are and how they are wired up.

It continually outputs any changes that occur to the topology.

Example Usage

Introspect and display whats going on inside the system:

```
MyComplexSystem().activate()

pipeline( Introspector(),
          AxonVisualiserServer(noServer=True),
        )
```

More detail

Once activated, this component introspects the current local topology of an Axon system.

Local? This component examines its scheduler to find components and postmen. It then examines them to determine their inboxes and outboxes and the linkages between them. In effect, it determines the current topology of the system.

If this component is not active, then it will see no scheduler and will report nothing.

What is output is how the topology changes. Immediately after activation, the topology is assumed to be empty, so the first set of changes describes adding nodes and linkages to the topology to build up the current state of it.

Subsequent output just describes the changes - adding or deleting linkages and nodes as appropriate.

Nodes in the topology represent components and postboxes. A linkage between a component node and a postbox node expresses the fact that that postbox belongs to that component. A linkage between two postboxes represents a linkage in the Axon system, from one component to another.

This topology change data is output as string containing one or more lines. It is output through the "outbox" outbox. Each line may be one of the following:

- "DEL ALL"
  - the first thing sent immediately after activation - to ensure that the receiver of this data understand that we are starting from nothing
- "ADD NODE <id> <name> randompos component"
- "ADD NODE <id> <name> randompos inbox"
- "ADD NODE <id> <name> randompos outbox"
  - an instruction to add a node to the topology, representing a component, inbox or outbox. <id> is a unique identifier. <name> is a 'friendly' textual label for the node.
- "DEL NODE <id>"
  - an instruction to delete a node, specified by its unique id
- "ADD LINK <id1> <id2>"
  - an instruction to add a link between the two identified nodes. The link is deemed to be directional, from <id1> to <id2>
- "DEL LINK <id1> <id2>"
  - an instruction to delete any link between the two identified nodes. Again, the directionality is from <id1> to <id2>.

the <id> and <name> fields may be encapsulated in double quote marks ("). This will definitely be so if they contain space characters.

If there are no topology changes then nothing is output.

This component ignores anything arriving at its "inbox" inbox.

If a shutdownMicroprocess message is received on the "control" inbox, it is sent on to the "signal" outbox and the component will terminate.

How does it work?

Every execution timeslice, Introspector queries its scheduler to obtain a list of all components. It then queries the postoffice in each component to build a picture of all linkages between components. It also builds a list of all inboxes and outboxes on each component.

This is mapped to a list of nodes and linkages. Nodes being components and postboxes; and linkages being what postboxes belong to what components, and what postboxes are linked to what postboxes.

This is compared against the nodes and linkages from the previous cycle of processing to determine what has changed. The changes are then output as a sequence of "ADD NODE", "DEL NODE", "ADD LINK" and "DEL LINK" commands.

Components

Introspector

Introspector() -> new Introspector component.

Outputs topology (change) data describing what components there are, and how they are wired inside the running Axon system.

Axon.Ipc

IPC message classes

Some standard IPC messages used by Axon. The base class for all IPC classes is ipc.

Some purposes for which these can be used are described below. This is not exhaustive and does *not* cover all of the Ipc classes defined here!

Shutting down components

- Axon.Ipc.shutdownMicroprocess
- Axon.Ipc.producerFinished

If you want to order a component to stop, most will do so if you send either of these ipc objects to their "control" inbox. When a component does stop, most send on the same message they received out of their "signal" outbox (or a message of their own creation if they decided to shutdown without external prompting)

Producer components, such as file readers or network connections will send a producerFinished() ipc object rather than shutdownMicroprocess() to indicate that the shutdown is due to them finishing producing data.

You can therefore link up components in a chain - linking "signal" outboxes to "control" inboxes to allow shutdown messages to cascade - making it easier to shutdown and clean up a system.

How should components behave when they receive either of these Ipc messages? In most cases, components simply shut down as soon as possible and send the same message on out of their "signal" outbox. However many components behave slightly more subtley to ensure the last few items of data passing throuhg a chain of components are not accidentally lost:

- If the message is a producerFinished() message, then a component may wish to finish processing any data still left in its inboxes or internal buffers before terminating and passing on the producerFinished() message.
- If the message is a shutdownMicroprocess() message, then a component should ideally try to terminate rather than finish what it is doing.

Many components therefore containg logic similar to this:

```
class MyComponent(Axon.Component.component):

    def main(self):

        while still got things to do and not received "shutdownMicroprocess":
            ..do things..
            ..check "control" inbox..
            yield 1

        if not received any shutdown message:
            self.send(Axon.Ipc.shutdownMicroprocess(), "signal")
        else:
            self.send(message received, "signal")
```

producerFinished() can be likened to notification of a clean shutdown - rather like a unix process closing its stdout file handle when it finishes. shutdownMicroproces() is more like a hard termination due to a system being interrupted.

Knock-on shutdowns between microprocesses

- Axon.Ipc.shutdownMicroprocess

When a microprocess terminates, the scheduler calls its Axon.Microprocess.microprocess._closeDownMicroprocess method. This method can return an Axon.Ipc.shutdownMicroprocess ipc object, for example:

```
def _closeDownMicroprocess(self):
    return Axon.Ipc.shutdownMicroprocess(anotherMicroprocess)
```

The scheduler will ensure that other microprocess is also shut down.

shutdown vs shutdownMicroprocess

- Axon.Ipc.shutdown
- Axon.Ipc.shutdownMicroprocess

You may notice that shutdownMicroprocess appears to be used for two purposes -knock-on shutdowns and signalling component shutdown.

Axon.Ipc.shutdown was originally intended to be used rather than Axon.Ipc.shutdownMicroprocess; however because most components sup-

port the latter (which was an accidental mistake) the latter should continue to be used.

Axon may at some stage make these two Ipc classes synonyms for each other to resolve this issue, but this decision has not been taken yet.

Setting off a new microprocess and waiting for it to complete

- Axon.Ipc.WaitComplete
- Axon.Ipc.reactivate

Used by:

- components / microprocesses
- Axon.Scheduler.scheduler

A microprocess can yield a WaitComplete() Ipc message to the scheduler to ask for another microprocess to be started. When that second microprocess completes, the original one resumes - it waits until the second one completes.

This is a nice little way to sidestep the restriction in python that you can't nest yield statements for a given generator inside methods/functions it calls.

For example, here's a clean way to wait for data arriving at the "inbox" inbox of a component:

```
class MyComponent(Axon.Component.component):

    def main(self):
        ...
        yield WaitComplete(self.waitForInbox())
        msg = self.recv("inbox")

    def waitForInbox(self):
        while not self.dataReady("inbox"):
            yield 1
```

Internally, the scheduler uses Axon.Ipc.reactivate to ensure the original microprocess is resumed after the one that was launched terminates.

Other

WaitComplete

WaitComplete(generator) -> new WaitComplete object.

Message to ask the scheduler to temporarily suspect this microprocess and run a new one instead based on the generator provided; resuming the original when the new one completes.

Use within a microprocess by yielding one back to the scheduler.

Arguments:

- the generator to be run as the separate microprocess

errorInformation

errorInformation(caller[,exception][,message]) -> new errorInformation ipc message.

A message to indicate that a non fatal error has occured in the component. It may skip processing errored data but should respond correctly to future messages.

Keyword arguments:

- caller -- the source of the error information. Assigned to self.caller
- exception -- Optional. None, or the exception that caused the error. Assigned to self.exception
- message -- Optional. None, or a message describing the problem. Assigned to self.message

ipc

Message base class

newComponent

newComponent(*components) -> new newComponent ipc message.

Message used to inform the scheduler of a new component that needs a thread of control and activating.

Use within a microprocess by yielding one back to the scheduler.

Arguments:

- the components to be activated

notify

notify(caller,payload) -> new notify ipc message.

Message used to notify the system of an event. Subclass to implement your own specific notification messages.

Keyword arguments:

- caller -- a reference to whoever/whatever issued this notification. Assigned to self.caller
- payload -- any relevant payload relating to the notification. Assigned to self.object

producerFinished

producerFinished([caller][,message]) -> new producerFinished ipc message.

Message to indicate that the producer has completed its work and will produce no more output. The receiver may wish to shutdown.

Keyword arguments:

- caller -- Optional. None, or the producer who has finished. Assigned to self.caller
- message -- Optional. None, or a message giving any relevant info. Assigned to self.message

reactivate

reactivate(original) -> new reactivate ipc message.

Returned by Axon.Microprocess.microprocess._closeDownMicroprocess() to the scheduler to get another microprocess reactivated.

Keyword arguments:

- original -- The original microprocess to be resumed. Assigned to self.original

shutdown

Message used to indicate that the component recieving it should shutdown.

Due to legacy mistakes, use shutdownMicroprocess instead.

shutdownMicroprocess

shutdownMicroprocess(*microprocesses) -> new shutdownMicroprocess ipc message.

Message used to indicate that the component recieving it should shutdown. Or to indicate to the scheduler a shutdown knockon from a terminating microprocess.

Arguments:

- the microprocesses to be shut down (when used as a knockon)

status

status(status) -> new status ipc message.

General Status message.

Keyword arguments:

- status -- the status.

wouldblock

wouldblock(caller) -> new wouldblock ipc message.

Message used to indicate to the scheduler that the system is likely to block now.

Keyword arguments:

- caller -- who it is who is likely to block (presumably a microprocess). Assigned to self.caller

Axon.Linkage

Linkages

Components only have input & output boxes. For data to get from a producer (eg a file reader) to a consumer (eg an encryption component) then the output of one component, the source component, must be linked to the input of another component, the sink component.

- linkage objects are handles describing a linkage from one postbox to another
- Axon.Postoffice.postoffice creates and destroys linkages

All components have a postoffice object, this performs the creation and destruction of linkages. Ask it for a linkage between inboxes and outboxes and a linkage object is returned as a handle describing the linkage. When a message is sent to an outbox, it is immediately delivered along linkage(s) to the destination inbox.

This is *not* the usual technique for software messaging. Normally you create messages, addressed to something specific, and then the message handler delivers them.

However the method of communication used here is the norm for *hardware* systems, and generally results in very pluggable components - the aim of this system, hence this design approach rather than the normal. This method of communication is also the norm for one form of software system - unix shell scripting - something that has shown itself time and again to be used in ways the inventors of programs/components never envisioned.

Other

linkage

linkage(source, sink[, passthrough]) -> new linkage object

An object describing a link from a source component's inbox/outbox to a sink component's inbox/outbox.

Keyword arguments: - source -- source component - sink -- sink component - sourcebox -- source component's source box name (default="outbox") - sinkbox -- sink component's sink box name (default="inbox") - passthrough -- 0=link is from inbox to outbox; 1=from inbox to inbox; 2=from outbox to outbox (default=0)

Axon.Microprocess

Microprocess - A class supporting concurrent execution

A microprocess is a class supporting parallel execution, provided by forming a wrapper around a generator. It also provides a place for context to be stored about the generator.

- A component is based on a microprocess - giving it its thread of execution.
- The Scheduler runs microprocesses that have been 'activated'

This is an Axon internal. If you are writing components you do not need to understand this as you will normally not use it directly.

Developers wishing to use Axon in other ways or understand its implementation shoudl read on with interest!

Basic Usage

Making and using a microprocess is easy:

1. Subclass microprocess writing your own main() generator method
2. Create and 'activate' it
3. Run the scheduler so it is executed

Specifically, classes that subclass microprocess, and implement a main() generator function can be activated, and scheduled by the scheduler/microthread systems. Essentially a microprocess provides a minimal runtime context for the scheduling & thread handling system.

In more detail:

1. Subclass a microprocess, overriding the main() generator method to make your own that yields non-zero/False values:

```
class Loopy(microprocess):
    def __init__(self, num):
        self.num = num
        super(Loopy, self).__init__()
    def main(self):
        yield 1
```

```
        while 1:
            print "we loop forever", self.num
            yield 1
```

2. Instantiate and activate a few (note these are two separate steps!):

```
mp1=Loopy(1)
mp1.activate()

mp2=Loopy(2)
mp2.activate()

mp3=Loopy(3).activate()     # a more convenient shorthand
```

3. If you haven't already, start the scheduler to cause them to be run. The call will return when all microprocesses have finished executing (which is *never* in this example case):

```
>>> scheduler.run.runThreads()
we loop forever 1
we loop forever 2
we loop forever 3
we loop forever 1
we loop forever 2
we loop forever 3
we loop forever 1
we loop forever 2
... etc ...
```

Pause a microprocess whilst it is running by calling the pause() method. Wake it up again by calling unpause(). Pausing a microprocess means that it will cease to be executed until something else unpauses it. When unpaused it picks up from where it left off.

More detail

Essentially a microprocess provides a context for scheduling generators, and treating them similar to processes/threads. It provides basic facilities to support the activation (starting), pausing, unpausing and termination of a generator.

To start a microprocess running, you must create it and then activate it. Activation is a separate step to allow you to control exactly when you want a microprocess to actually start running. Once activated, running the scheduler will cause your generator to be executed along with all other active microprocesses.

Every yield statement in your generator hands back control, allowing Axon to schedule other microprocesses that may be running.

You can yield any value you like except zero or False (which are reserved for future use).

When a microprocess finishes, the scheduler calls its _closeDownMicroprocess() method. You can either override this in your subclass, or specify a closeDown-Value when initialising microprocess. The scheduler will act on the return value if it recognises it - see the Scheduler module for more details.

Alternative ways of defining the generator/thread Subclass microprocess and write your generator as a differently named method, for example foo(), and to then specify the *name* of the "mainmethod" when you ask the microproces to activate:

```
class MyMicroprocess(microprocess):
    def foo(self):
        yield 1
        while 1:
            print "we loop forever!"
            yield 1

mp = MyMicroprocess()
mp.activate(mainmethod="foo")
scheduler.run.runThreads()
```

Alternatively, you can instantiate a microprocess providing your own generator:

```
def bar():
    yield 1
    while 1:
        print "we loop forever!"
        yield 1

mp = MyMicroprocess(thread=bar())
mp.activate()
scheduler.run.runThreads()
```

Note that this last approach removes the ability of the microprocess to be prematurely stopped by calling its stop() method.

Microprocess lifecycle in detail

In terms of runtime a microprocess can be viewed to have 2 different life cycles - that which an external user sees, and that which the microprocess sees.

In terms of runtime life cycle viewed externally, a microprocess is created, activated, and then has its next method repeatedly called until a StopIteration exception is raised, at which point the microprocess is deleted. In terms of a more traditional approach the next call approximates to a timeslice being allocated to a process/thread.

The value returned by next() should be non-zero (reserved for future use). The scheduler calling next() may also recognise some specific values - see the

Axon.Scheduler.scheduler class for more information.

The runtime life cycle from the view of the microprocess stems from the fact that a generator wraps a thread of control, by effectively treating the program counter like a static variable. The following describes this runtime from the microprocess's point of view.

First the '___init___' function is called during initialisation at object creation time. This results in a non-active, non-running microprocess. Activation has been deliberately separated from creation and initialisation. At some point in the future, the microprocess's activate method is called, activating the object. When the object is activated, an internal call to a '_microprocessGenerator' occurs. This function in fact results in the return object being a generator, which can then have its next method called repeatedly. This generator is then stored as an attribute of the microprocess class.

The following describe the flow of control the generator takes when the generator is provided with a flow of control/time slice via it's next method. Initially, it creates a local generator object - 'pc' - by calling the object's main method. (This allows the client of the microprocess class to provide their own generator if they wish.) This is necessary due to the fact that any function containing a 'yield' keyword is a generator -the 'yield' keyword cannot be abstracted away. Next, inside a loop, the microprocess calls the next() method of its local generator object 'pc' -effectively providing a time slice to the user of the microprocess class. Any result provided by the timeslice is then yielded (returned) to the client of the generator. However if the microprocess has its stopped flag set, the microprocess generator simply yields a null value, followed by stopping.

This all boils down to checking to see if the microprocess is not stopped prior to running the body of a generator formed from the main method of the class. The intent here is that users will inherit from the microprocess class, and then reimplement the main method, which periodically yields control. If the user/inheriting class does not implement a main method, then the system provides a stub that simply returns.

Pausing and unpausing of microprocesses has been delegated to the scheduler to allow Axon systems to not consume CPU cycles when idle. When a microprocess is paused the scheduler simply never calls its next() method until it is unpaused. As such, calls to pause() and unpause() are actually relayed to the scheduler.

The microprocess class uses a dummy scheduler _NullScheduler until it is actually activated. This is done so pause() and unpause() calls can be silently absorbed whilst a microprocess is not yet active.

Essentially the microprocess provides a context for scheduling generators, and treating them similar to processes/threads.

Clients are not expected to use the microprocess class itself directly -they are expected to subclass the microprocess class. Subclasses do need however to call the microprocess constructor. A minimal client class could look like this:

```
from Axon.Microprocess import microprocess
class automaton(microprocess):
    def __init__(self):
        self.Microprocess() # Call superclass constructor
    def main:
        while 1:
        yield 1
        print "Hello Again"
```

This microprocess would then be run by a wrapper as follows:

```
import Axon.Microprocess as microprocess
import Axon.Scheduler as scheduler
s = scheduler.scheduler()
a = automaton()
a.activate()
s.runThreads()
```

The component class does this, and adds further facilities for inter-microprocess communication. Likewise, the scheduler class subclasses microprocess so that it can be scheduled in parallel with other tasks.

As noted previously, every microprocess object has access to a debugger, which is accessed via the local attribute self.debugger, which we shall return to later. Likewise every microprocess object contains a reference to a scheduler.

Internal flags/state

- **id** and **name** - unique identifiers. No other Axon entity will have the same name or id.
- **init** - a flag indicating if the microprocess has been correctly initialised.
- **stopped** - Indicates that the microprocess has run and since stopped.
- **___thread** - the generator object that gets executed whenever next() is called. Is actually an internally created generator that wraps the one created by the main() method.
- **scheduler** - The scheduler that controls execution of this microprocess. When not yet activated a dummy scheduler (NullScheduler) is used instead.
- **tracker** - The coordinating assistant tracker to be used by this microprocess.
- **debugger** - A local debugging object. (See the debug class docs for more detail)

Note that the paused/awake state of a microprocess is something maintained and managed by the scheduler; not the microprocess itself.

Other

Axon - the core concurrency system for Kamaelia

Axon is a component concurrency framework. With it you can create software "components" that can run concurrently with each other. Components have "inboxes" and "outboxes" through with they communicate with other components.

A component may send a message to one of its outboxes. If a linkage has been created from that outbox to another component's inbox; then that message will arrive in the inbox of the other component. In this way, components can send and receive data - allowing you to create systems by linking many components together.

Each component is a microprocess - rather like a thread of execution. A scheduler takes care of making sure all microprocesses (and therefore all components) get regularly executed. It also looks after putting microprocesses to sleep (when they ask to be) and waking them up (for example, when something arrives in one of their inboxes).

Base classes for building your own components

- **Axon.Component**
  - defines the basic component. Subclass it to write your own components.
- **Axon.AdaptiveCommsComponent**
  - like a basic component but with facilties to let you add and remove inboxes and outboxes during runtime.
- **Axon.ThreadedComponent**
  - like ordinary components, but which truly run in a separate thread - meaning they can perform blocking tasks (since they don't have to yield control to the scheduler for other components to continue executing)

Underlying concurrency system

- **Axon.Microprocess**
  - Turns a python generator into a schedulable microprocess - something that can be started, paused, reawoken and stopped. Subclass it to make your own.
- **Axon.Scheduler**
  - Runs the microprocesses. Manages the starting, stopping, pausing and waking of them. Is also a microprocess itself!

Services, statistics, Instrospection

- **Axon.CoordinatingAssistantTracker**

  - provides mechanisms for components to advertising and discover services they can provide for each other.
  - acts as a repository for collecting statistics from components in the system
- **Axon.Introspector**
  - outputs live topology data describing what components there are in a running axon system and how they are linked together.

Exceptions, Messages and Misc

- **Axon.Base**
  - base metaclass for key Axon classes
- **Axon.AxonExceptions**
  - classes defining various exceptions in Axon.
- **Axon.Ipc**
  - classes defining various IPC messages in Axon used for signalling shutdown, errors, notifications, etc...
- **Axon.idGen**
  - unique id value generation
- **Axon.util**
  - various miscellaneous support utility methods

Integration with other systems

- **Axon.background**
  - use Axon components within other python programs by wrapping them in a scheduler running in a separate thread
- **Axon.Handle**
  - a Handle for getting data into and out of the standard inboxes and outboxes of a component from a non Axon based piece of code. Useful in combination with Axon.background

Internals for implementing inboxes, outboxes and linkages

- **Axon.Box**
  - The base implementation of inboxes and outboxes.
- **Axon.Postoffice**
  - All components have one of these for creating, destroying and tracking linkages.
- **Axon.Linkage**
  - handles used to describe linkages from one postbox to another

What, no Postman? Optimisations made to Axon have dropped the Postman. Inboxes and outboxes handle the delivery of messages themselves now.

Debugging support

- **Axon.debug**
  - defines a debugging output object.
- **Axon.debugConfigFile**
  - defines a method for loading a debugging configuration file that determines what debugging output gets displayed and what gets filtered out.
- **Axon.debugConfigDefaults**
  - defines a method that supplies a default debugging configuration.

microprocess

> microprocess([thread][,closeDownValue]) -> new microprocess object
>
> Creates a new microprocess object (not yet activated). You can optionally specify an alternative generator to be used instead of the one the microprocess would ordinarily create for itself.
>
> Keyword arguments:
>
> - thread -- None, or an alternative generator to be the thread of execution in this microprocess.
> - closeDownValue -- Value to be returned when the microprocess has finished and _closeDownMicroprocess() is called (default=0)

Axon.Postoffice

Axon postoffices

A postoffice object looks after linkages. It can create and destroy them and keeps records of what ones currently exist. It hands out linkage objects that can be used as handles to later unlink (remove) the linkage.

THIS IS AN AXON INTERNAL! If you are writing components you do not need to understand this.

Developers wishing to understand how Axon is implemented should read on with interest!

How is this used in Axon?

Every component has its own postoffice. The component's link() and unlink() methods instruct the post office to create and remove linkages.

When a component terminates, it asks its post office to remove any outstanding linkages.

Example usage

Creating a link from an inbox to an outbox; a passthrough link from an inbox to another inbox; and a passthrough link from an outbox to another outbox:

```
po = postoffice()
c0 = component()
c1 = component()
c2 = component()
c3 = component()

link1 = po.link((c1,"outbox"), (c2,"inbox"))
link2 = po.link((c2,"inbox"), (c3,"inbox"), passthrough=1)
link3 = po.link((c0,"outbox"), (c1,"outbox"), passthrough=2)
```

Removing one of the linkages; then all linkages involving component c3; then all the rest:

```
po.unlink(thelinkage=link3)

po.unlink(thecomponent=c3)

po.unlinkAll()
```

More detail

A postoffice object keeps creates and destroys objects and keeps a record of which ones currently exist.

The linkage object returned when a linkage is created serves only as a handle. It does not form any operation part of the linkage.

Multiple postoffices can (in fact, will) exist in an Axon system. Each looks after its own collection of linkages. A linkage created at one postoffice will *not* be known to other postoffice objects.

Other

postoffice

> The post office looks after linkages between postboxes, thereby ensuring deliveries along linkages occur as intended.
>
> There is one post office per component.
>
> A Postoffice can have a debug name - this is to help differentiate between postoffices if problems arise.

Axon.STM

STM

Support for basic in-process software transactional memory.

What IS it?

Software Transactional Memory (STM) is a technique for allowing multiple threads to share data in such a way that they know when something has gone wrong. It's been used in databases (just called transactions there really) for some time and is also very similar to version control. Indeed, you can think of STM as being like variable level version control.

Why is it useful?

Why do you need it? Well, in normal code, Global variables are generally shunned because it can make your code a pain to work with and a pain to be certain if it works properly. Even with linear code, you can have 2 bits of code manipulating a structure in surprising ways - but the results are repeatable. Not-properly-managed-shared-data is to threaded systems as not-properly-managed-globals are to normal code. (This code is one way of helping manage shared data)

Well, with code where you have multiple threads active, having shared data is like an even nastier version of globals. Why? Well, when you have 2 (or more) running in parallel, the results of breakage can become hard to repeat as two pieces of code "race" to update values.

With STM you make it explicit what the values are you want to update, and only once you're happy with the updates do you publish them back to the shared storage. The neat thing is, if someone else changed things since you last looked, you get told (your commit fails), and you have to redo the work. This may sound like extra work (you have to be prepared to redo the work), but it's nicer than your code breaking :-)

The way you get that message is the .commit raises a ConcurrentUpdate exception.

Also, it's designed to work happily in code that requires non-blocking usage -which means you may also get a BusyRetry exception under load. If you do, you should as the exception suggests retry the action that you just tried. (With or without restarting the transaction)

Apologies if that sounds too noddy :)

Using It

Accessing/Updating a single shared value in the store You can have many single vars in a store of course... If they're related though or updated as a group, see the next section:

```
from Axon.STM import Store

S = Store()
greeting = S.usevar("hello")
print repr(greeting.value)
greeting.set("Hello World")
greeting.commit()
```

Accessing/Updating a collection of shared values in the store Likewise you can use as many collections of values from the store as you like:

```
from Axon.STM import Store

S = Store()
D = S.using("account_one", "account_two", "myaccount")
D["account_one"].set(50)
D["account_two"].set(100)
D.commit()
S.dump()

D = S.using("account_one", "account_two", "myaccount")
D["myaccount"].set(D["account_one"].value+D["account_two"].value)
D["account_one"].set(0)
D["account_two"].set(0)
D.commit()
S.dump()
```

What can (possibly) go wrong?

You can have 2 people trying to update the same values at once. An example of this would be - suppose you have the following commands being executed by 2 threads with this mix of commands:

```
S = Store()
D = S.using("account_one", "account_two", "myaccount")
D["myaccount"].set(0)
D["account_one"].set(50)
D["account_two"].set(100)
D.commit() # 1
S.dump()

D = S.using("account_one", "account_two", "myaccount")
D["myaccount"].set(D["account_one"].value+D["account_two"].value)
E = S.using("account_one", "myaccount")
E["myaccount"].set(E["myaccount"].value-100)
E["account_one"].set(100)
E.commit() # 2
```

```
D["account_one"].set(0)
D["account_two"].set(0)
D.commit() # 3 - should fail
S.dump()
```

You do actually want this to fail because you have concurrent updates. This will fail on the third commit, and fail by throwing a ConcurrentUpdate exception. If you get this, you should redo the transaction.

The other is where there's lots of updates happening at once. Rather than the code waiting until it acquires a lock, it is possible for either the .using, .usevar or .commit methods to fail with a BusyRetry exception. This means exactly what it says on the tin - the system was busy & you need to retry. In this case you do not have to redo the transaction. This is hard to replicate except under load. The reason we do this however is because most Kamaelia components are implemented as generators, which makes blocking operation (as a .acquire() rather than .acquire(0) would be) an expensive operation.

Other

BusyRetry

NO DOCS

Collection

Collection() -> new Collection dict

A dictionary which belongs to a thread-safe store

Again, you do not instantiate these yourself

ConcurrentUpdate

NO DOCS

Store

Store() -> new Store object

A thread-safe versioning store for key-value pairs

You instantiate this as per the documentation for this module

Value

Value(version, value, store, key) -> new Value object

A simple versioned key-value pair which belongs to a thread-safe store

Arguments:

- version -- the initial version of the value
- value -- the object's initial value
- store -- a Store object to hold the value and it's history
- key -- a key to refer to the value

Note: You do not instantiate these - the Store does that

NEED TO TIGHTEN UP IMPORTS from Axon.Ipc import *

Axon.Scheduler

Scheduler - runs things concurrently

The Scheduler runs active microprocesses - giving a regular timeslice to each. It also provides the ability to pause and wake them; allowing an Axon based system to play nicely and relinquish the cpu when idle.

- The Scheduler runs microprocesses that have been 'activated'
- The Scheduler is itself a microprocess

Using the scheduler

The simplest way is to just use the default scheduler `scheduler.run`. Simply activate components or microprocesses then call the runThreads() method of the scheduler:

```
from Axon.Scheduler import scheduler
from MyComponents import MyComponent, AnotherComponent

c1 = MyComponent().activate()
c2 = MyComponent().activate()
c3 = AnotherComponent().activate()

scheduler.run.runThreads()
```

Alternatively you can create a specific scheduler instance, and activate them using that specific scheduler:

```
mySched = scheduler()

c1 = MyComponent().activate(Scheduler=mySched)
c2 = MyComponent().activate(Scheduler=mySched)
c3 = AnotherComponent().activate(Scheduler=mySched)

mySched.runThreads()
```

The runThreads() method is the way of bootstrapping the scheduler. Being a microprocess, it needs something to schedule it! The runThreads() method does exactly that.

The activate() method is fully thread-safe. It can handle multiple simultaneous callers from different threads to the one the scheduler is running in.

Pausing and Waking microprocesses

The Scheduler supports the ability to, in a thread safe manner, pause and wake individual microprocesses under its control. Because it is thread safe, any thread of execution can issue pause and wake requests for any scheduled microprocess.

The pauseThread() and wakeThread() methods submit requests to pause or wake microprocesses. The scheduler will process these when it is next able to - the requests are queued rather than processed immediately. This is done to ensure thread safety. It can handle multiple simultaneous callers from different threads to the one the scheduler is running in.

Pausing a microprocess means the scheduler removes it from its 'run queue'. This means that it no longer executes that microprocess. Waking it puts it back into the 'run queue'.

If no microprocesses are awake then the scheduler relinquishes cpu usage by blocking.

If however this scheduler is itself being scheduled by another microprocess then it does not block. Ideally it should ask its scheduler to pause it, but instead it busy-waits - self pausing functionality is not yet implemented.

'yielding' new components for activation and replacement generators

In general, the main() generator in a microprocess (its thread of execution) can return any values it likes when it uses the `yield` statement. It is recommended to not yield zeros or other kinds of 'false' value as these are reserved for possible future special meaning.

However, this scheduler does understand certain values that can be yielded:

- **Axon.Ipc.newComponent** - a microprocess can yield this to ask the scheduler to activate a new component or microprocess:

  ```
  def main(self):
      ...
      x=MyComponent()
      yield Axon.Ipc.newComponent(x)
      ...
  ```

  This is simply an alternative to calling x.activate().

- **Axon.Ipc.WaitComplete** - this is a way for a microprocess to substitute itself (temporarily) with another one that uses a new generator. For example:

```
def main(self):
    ...
    yield Axon.Ipc.WaitComplete(self.waitOneSecond())
    ...

def waitOneSecond(self):
    t=time.time()
    while time.time() < t+1.0:
        yield 1
```

  This is a convenient way to modularise parts of your main() code. But there is an important limitation with the current implementation:

  - self.pause() will not cause the replacement generator to pause. (Where 'self' is the original microprocess - as in the example code above)

What happens when a microprocess finishes?

The scheduler will stop running it! It will call the microprocess's stop() method. It will also call the _closeDownMicroprocess() method and will act on the return value if it is one of the following:

- **Axon.Ipc.shutdownMicroprocess** - the specified microprocess will also be stopped. Use with caution as the implementation is currently untested and likely to fail, possibly even crash the scheduler!
- **Axon.Ipc.reactivate** - the specified microprocess will be (re)activated. The scheduler uses this internally to pick up where it left off when a Axon.Ipc.WaitComplete instigated detour finishes (see above).

Querying the scheduler (Introspection)

The listAllThreads() method returns a list of all activated microprocesses -both paused and awake.

The isThreadPaused() method lets you determine if an individual microprocess is paused. Note that the result returned by this method is conservative (the default assumption is that a thread is probably awake). the result will vary depending on the exact moment it is called!

Both these methods are thread safe.

Slowing down execution (for debugging)

It also has a slow motion mode designed to help with debugging & testing. Call runThreads() with the slowmo argument set to the number of seconds the

scheduler should pause after each cycle of executing all microprocesses. For example, to wait half a second after each cycle of execution:

```
scheduler.run.runThreads(slowmo=0.5)
```

How does it work internally?

The scheduler keeps the following internal state:

- **time** - updated to time.time() every execution cycle - can be inspected by microprocesses instead of having to call time.time() themselves.
- **threads** - a dictionary containing the state of activated microprocesses (whether they are awake or not)
- **wakeRequests** and **pauseRequests** - the thread safe queues of requests to wake and pause individual microprocesses
- Internal to the main() generator:
  - **runqueue** - the list of active and awake microprocesses being run
  - **nextrunqueue** - the list of microprocesses to be run next time round

The scheduler uses a simple round robin approach - it walks through its run queue and calls the next() method of each microprocess in turn. As it goes, it builds a new run queue, ready for the next cycle. If a microprocess terminates (raises a StopIteration exception) then it is not included in the next cycle's run queue.

After it has gone through all microprocesses, the scheduler then processes messages in its wakeRequests and sleepRequests queues. Sleep requests are processed first; then wake requests second. Suppose there is a sleep and wake request queued for the same microprocess; should it be left awake or put to sleep? By processing wake requests last, the scheduler can err on the side of caution and prefer to leave it awake.

Microprocesses are all in one of three possible states (recorded in the `threads` dictionary):

- **ACTIVE** - the microprocess is awake. It should be in the run queue being prepared for the next execution cycle.
- **SLEEPING** - the microprocess is asleep/paused. It should *not* be in the run queue for the next cycle.
- **GOINGTOSLEEP** - the microprocess has been requested to be put to sleep.

A request to put a microprocess to sleep is handled as follows:

- If the microprocess is already *sleeping*, then nothing needs to happen.
- If the microprocess is *active*, then it is changed to "going to sleep". It is not removed from the run queue immediately. Instead, what happens is:
  - on the next cycle of execution, as the scheduler goes through items in the run queue, it doesn't execute any that are "going

to sleep" and doesn't include them in the next run queue it is building. It also sets them to the "sleeping" state,

Wake requests are used to both wake up sleeping microprocesses and also to activate new ones. A request to wake a microprocess is handled like this:

- If the microprocess is already *active*, then nothing needs to happen.
- If the microprocess is *sleeping* then it is added to the next run queue and changed to be *active*.
- If the microprocess is *going to sleep* then it is only changed to be *active* (it will already be in the run queue, so doesn't need to be added)

If the request contains a flag indicating that this is actually an activation request, then this also happens:

- If the microprocess is not in the `threads` dictionary then it is added to both the run queue and `threads`. It is set to be *active*.

This three state system is a performance optimisation: it means that the scheduler does not need to waste time searching through the next run queue to remove items - they simply get removed on the next cycle of execution.

Wake requests and sleep requests are handled through thread-safe queues. This enables other threads of execution (eg. threaded components) to safely make requests to wake or pause components.

Other

WaitComplete

WaitComplete(generator) -> new WaitComplete object.

Message to ask the scheduler to temporarily suspect this microprocess and run a new one instead based on the generator provided; resuming the original when the new one completes.

Use within a microprocess by yielding one back to the scheduler.

Arguments:

- the generator to be run as the separate microprocess

errorInformation

errorInformation(caller[,exception][,message]) -> new errorInformation ipc message.

A message to indicate that a non fatal error has occured in the component. It may skip processing errored data but should respond correctly to future messages.

Keyword arguments:

- caller -- the source of the error information. Assigned to self.caller
- exception -- Optional. None, or the exception that caused the error. Assigned to self.exception
- message -- Optional. None, or a message describing the problem. Assigned to self.message

ipc

Message base class

newComponent

newComponent(*components) -> new newComponent ipc message.

Message used to inform the scheduler of a new component that needs a thread of control and activating.

Use within a microprocess by yielding one back to the scheduler.

Arguments:

- the components to be activated

notify

notify(caller,payload) -> new notify ipc message.

Message used to notify the system of an event. Subclass to implement your own specific notification messages.

Keyword arguments:

- caller -- a reference to whoever/whatever issued this notification. Assigned to self.caller
- payload -- any relevant payload relating to the notification. Assigned to self.object

producerFinished

producerFinished([caller][,message]) -> new producerFinished ipc message.

Message to indicate that the producer has completed its work and will produce no more output. The receiver may wish to shutdown.

Keyword arguments:

- caller -- Optional. None, or the producer who has finished. Assigned to self.caller
- message -- Optional. None, or a message giving any relevant info. Assigned to self.message

reactivate

reactivate(original) -> new reactivate ipc message.

Returned by Axon.Microprocess.microprocess._closeDownMicroprocess() to the scheduler to get another microprocess reactivated.

Keyword arguments:

- original -- The original microprocess to be resumed. Assigned to self.original

scheduler

Scheduler - runs microthreads of control.

shutdown

Message used to indicate that the component recieving it should shutdown.

Due to legacy mistakes, use shutdownMicroprocess instead.

shutdownMicroprocess

shutdownMicroprocess(*microprocesses) -> new shutdownMicroprocess ipc message.

Message used to indicate that the component recieving it should shutdown. Or to indicate to the scheduler a shutdown knockon from a terminating microprocess.

Arguments:

- the microprocesses to be shut down (when used as a knockon)

status

status(status) -> new status ipc message.

General Status message.

Keyword arguments:

- status -- the status.

vrange

range(stop) -> range object range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1. start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3. These are exactly the valid

indices for a list of 4 elements. When step is given, it specifies the increment (or decrement).

wouldblock

wouldblock(caller) -> new wouldblock ipc message.

Message used to indicate to the scheduler that the system is likely to block now.

Keyword arguments:

- caller -- who it is who is likely to block (presumably a micro-process). Assigned to self.caller

Axon.ThreadedComponent

Thread based components

A threaded component is like an ordinary component but where the main() method is an ordinary method that is run inside its own thread. (Normally main() is a generator that is given slices of execution time by the scheduler).

This is really useful if your code needs to block - eg. wait on a system call, or if it is better off being able to run on another CPU (though beware python's limited ability to scale across multiple CPUs).

If you don't need these capabilities, consider making your component an ordinary Axon.Component.component instead.

- threadedcomponent - like an ordinary Axon.Component.component, but runs in its own thread
- threadedadaptivecommscomponent - a threaded version of Axon.AdaptiveCommsComponent.AdaptiveCom

Just like writing an ordinary component

This is nearly identical to writing an ordinary Axon.Component.component. For example this ordinary component:

```
class MyComponent(Axon.Component.component):

    Inboxes = { "inbox"   : "Send the FOO objects to here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Emits BAA objects from here",
                 "signal" : "NOT USED",
               }

    def main(self):
```

```
        while 1:
            if self.dataReady("inbox"):
                msg = self.recv("inbox")
                result = ... do something to msg ...
                self.send(result, "outbox")

            yield 1
```

Can be trivially written as a threaded component simply by removing the `yield` statements, turning main() into a normal method:

```
class MyComponent(Axon.ThreadedComponent.threadedcomponent):

    Inboxes = { "inbox"   : "Send the FOO objects to here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Emits BAA objects from here",
                 "signal" : "NOT USED",
               }

    def main(self):
        while 1:
            if self.dataReady("inbox"):
                msg = self.recv("inbox")
                result = ... do something to msg ...
                self.send(result, "outbox")
```

What can a threaded component do?

Exactly the same things any other component can. The following method calls are all implemented in a thread safe manner and function exactly as you should expect:

- self.link()
- self.unlink()
- self.dataReady()
- self.anyReady()
- self.recv()
- self.send()

self.pause() behaves slightly differently:

- calling self.pause() pauses immediately - not at the next yield statement (since there are no yield statements!)
- self.pause() has an extra optional 'timeout' argument to allow you to write timer code that can be interrupted, for example, by incoming messages.

In addition, threadedadaptivecommscomponent also supports the extra methods in Axon.AdaptiveCommsComponent.AdaptiveCommsComponent:

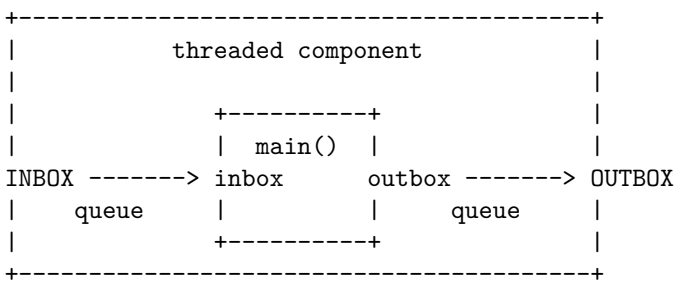
- self.addInbox()
- self.deleteInbox()
- self.addOutbox()
- self.deleteOutbox()
- *etc..*

Inbox and Outbox queues

There is one difference: because the main() method runs in a different thread it does not actually interact directly with its own inboxes and outboxes. Internal queues are used to get data between your thread and the component's actual inboxes and outboxes. This is hidden for the most part - the method calls you make to receive and send messages are exactly the same.

When initialising a threadedcomponent you may wish to specify the size limit (queue length) for these queues. There is a size limit so that if your threaded component is delivering to a size limited inbox, the effects of the inbox becoming full propagate back to your thread.

In some ways this is a bit like nesting one component within another - where all the parent component's inboxes and outboxes are forwarded to the child:



What does this mean in practice?

- *More messages get buffered.* - Suppose your threaded component has an internal queues of size 5 and is delivering messages to an inbox on another component with a size limit of 10. From the perspective of your threaded component you will actually be able to send 15 messages before you might start to get Axon.AxonExceptions.noSpaceInBox exceptions.
- *Threaded components that output lots of messages might see unexpected 'box full' exceptions* - Suppose your threaded component has a small internal queue size but produces lots of messages very quickly. The rest of the system may not be able to pick up those messages quickly enough to put them into the destination inbox. So even though the destination might not have a size limit you may still get these exceptions.

The secret is to choose a sensible queue size to balance between it being able to buffer enough messages without generating errors whilst not being so large as to render a size limited inbox pointless!

Regulating speed

In addition to being able to pause (with an optional timeout), a threaded component can also regulate its speed by briefly synchronising with the rest of the system. Calling the sync() method simply briefly blocks until the rest of the system can acknowledge.

Stopping a threaded component

Note that it is *not* safe to forcibly stop a threaded component (by calling the stop() method before the microprocess in it has terminated). This is because there is no true support in python for killing a thread.

Calling stop() prematurely will therefore kill the internal microprocess that handles inbox/outbox traffic on behalf of the thread, resulting in undefined behaviour.

When the thread terminates...

threadedcomponent will terminate - as you would expect. However, there are some subtleties that may need to be considered. These are due to the existence of the intermediate queues used to communicate between the thread and the actual inboxes and outboxes (as described above).

- When main() terminates, even if it has just recently checked its inqueues (inboxes) there might still be items of data at the inboxes. This is because there is a gap between data that arriving at an inbox, and it being forwarded into an inqueue going to the thread.
- When main() terminates, threadedcomponent will keep executing until it has finished successfully sending any data in outqueues, out of the respective "outboxes". This means that anything main() thinks it has sent is guaranteed to be sent. But if the destination is a size limited inbox that has become full (and that never gets emptied), then threadedcomponent will indefinitely stall because it cannot finish sending.

How is threaded component implemented?

threadedcomponet subclasses Axon.Components.component. It overrides the activate() method, to force activation to use a method called _localmain() instead of the usual main(). The code that someone writes for their main() method is instead run in a separate thread.

The code running in main() does not directly access inboxes our outboxes and doesn't actually create or destroy linkages itself. _localmain() can be thought of as a kind of proxy for the thread - acting on its behalf within the main scheduler run thread.

main() is wrapped by _threadmain() which tries to trap any unhandled exceptions generated in the thread and pass them back to _localmain() to be

rethrown.

Internal state:

- **__threadrunning** - flag, cleared by the thread when it terminates
- **queuelengths** - size to be used for internal queues between thread and inboxes and outboxes
- **__threadmainmethod** - the main method to be run as a thread
- **__thethread** - the thread object itself

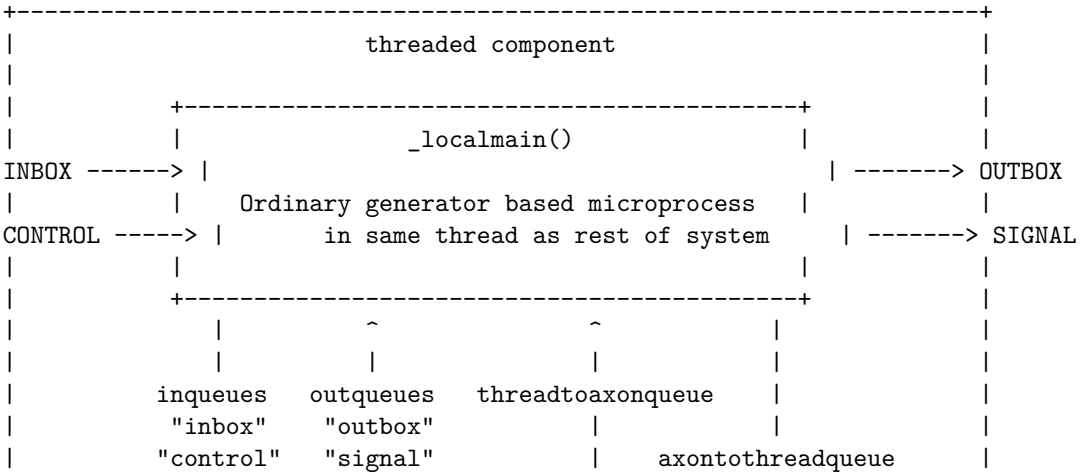
Internal to _localmain():

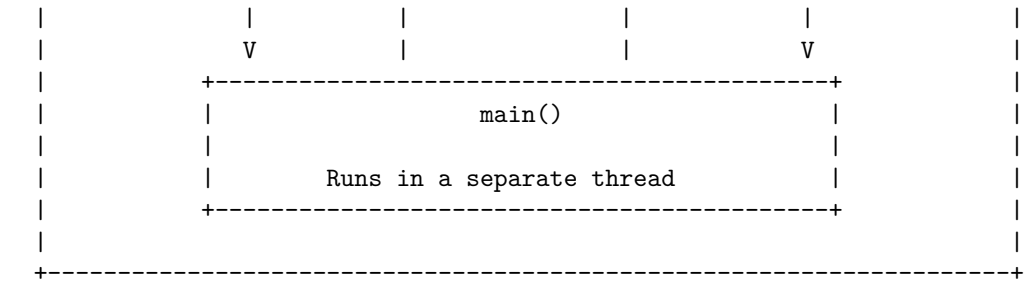
- **running** - flag tracking if the thread is still runnning
- **stuffWaiting** - flag tracking if there is are things that need to be done (if there is stuff waiting then _localmain() should not pause or terminate until it finishes)

Communication between the thread and _localmain():

- **inqueues** - dictionary of thread safe queues for getting data from inboxes to the thread
- **outqueues** - dictionary of thread safe queues for getting data from the thread to outboxes
- **threadtoaxonqueue** - thread safe queue for making requests to _localmain()
- **axontothreadqueue** - thread safe queue for replies from _localmain()
- **threadWakeUp** - thread safe event flag for waking up the thread if sleeping
- **__threadId** - unique id that is given to the thread as its 'name'
- **__localThreadId** - the thread id (name) of the thread _localmain() and the scheduler run in

The relationship between _localmain() and the main() method (running in a separate thread) looks like this:



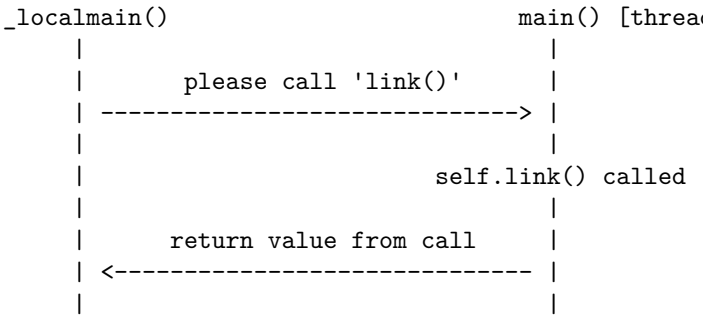


When a message arrives at an inbox, _localmain() collects it and places it into the thread safe queue self.inqueues[boxname] from which the thread can collect it. self.dataReady() and self.recv() are both overridden so they access the queues instead of the normal inboxes.

Similarly, when the thread wants to send to an outbox; self.send() is overridden so that it is actually sent to a thread safe queue self.outqueues[boxname] from which _localmain() collects it and sends it on.

Because all queues have a size limit (specified in at initialisation of the threaded component) this enables the effects of size limited inboxes to propagate up to the separate thread, via the queue. The implementation of self.send() is designed to mimic the behaviour

For other methods such as link(), unlink() and (in the case of threadedadaptivecommscomponent) addInbox(), deleteInbox(), addOutbox() and deleteOutbox(), the _localmain() microprocess also acts on the thread's behalf. The request to do something is sent to _localmain() through the thread safe queue self.threadtoaxonqueue. When the operation is complete, an acknowledgement is sent back via another queue self.axontothreadqueue:



The thread does not continue until it receives the acknowledgement - this is to prevent inconsistencies in state. For example, a call to create an inbox might be followed by a query to determine whether there is data in it - the inbox therefore must be fully set up before that query can be handled.

This is implemented by the _do_threadsafe() method. This method detects whether it is being called from the same thread as the scheduler (by com-

paring thread IDs). If it is not the same thread, then it puts a request into self.threadtoaxonqueue and blocks on self.axontothreadqueue waiting for a reply. The request is simply a tuple of a method or object to call and the associated arguments. When _localmain() collects the request it issues the call and responds with the return value.

self.pause() is overridden and equivalent functionality reimplemented by blocking the thread on a threading.Event() object which can be signalled by _localmain() whenever the thread ought to wake up.

The 'Adaptive' version does *not* ensure the resource tracking and retrieval methods thread safe. This is because it is assumed these will only be accessed internally by the component itself from within its separate thread. _localmain() does not touch these resources.

XXX *TODO*: Thread shutdown - no true support for killing threads in python (if ever). stop() method therefore doesn't stop the thread. Only stops the internal _localmain() microprocess, which therefore cuts the thread off from communicating with the rest of the system.

Other

DefaultQueueSize

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.___int___(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

UnHandledException

Used for passing exceptions unhandled by the thread back to the main thread for capture and either reporting or masking

threadedadaptivecommscomponent

threadedadaptivecommscomponent([queuelengths]) -> new threadedadaptivecommscomponent

Base class for a version of an Axon adaptivecommscomponent that runs main() in a separate thread (meaning it can, for example, block).

Subclass to make your own.

Internal queues buffer data between the thread and the Axon inboxes and outboxes of the component. Set the default queue length at initialisation (default=1000).

Like an adaptivecommscomponent, inboxes and outboxes can be added and deleted at runtime.

A simple example:

```
class IncrementByN(Axon.ThreadedComponent.threadedcomponent):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
                value = value + self.n
                self.send(value,"outbox")

            if not self.anyReady():
                self.pause()
```

threadedcomponent

threadedcomponent([queuelengths]) -> new threadedcomponent

Base class for a version of an Axon component that runs main() in a separate thread (meaning it can, for example, block). Subclass to make your own.

Internal queues buffer data between the thread and the Axon inboxes and outboxes of the component. Set the default queue length at initialisation (default=1000).

A simple example:

```
class IncrementByN(Axon.ThreadedComponent.threadedcomponent):

    Inboxes = { "inbox" : "Send numbers here",
```

```
            "control" : "NOT USED",
          }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
                value = value + self.n
                self.send(value,"outbox")

            if not self.anyReady():
                self.pause()
```

Axon.background

Running an Axon system in a separate thread

The background class makes it easy to run an Axon system in a separate thread (in effect: in the background).

This simplifies integration of Axon/Kamaelia code into other python code. See also Axon.Handle for a simple way to wrap a component in a thread safe way to access its inboxes and outboxes.

Example Usage

At its simplest, you could run a Kamaelia task independently in the background - such as a simple network connection, that dumps received data into a thread safe queue, after de-chunking it into lines of text.

NOTE: This example can be achieved more simply by using Axon.Handle. See the documentation of Axon.Handle to find out more.

1. We implement a simple component to collect the data:

```
from Axon.background import background
from Axon.Component import component

class Receiver(component):
    def __init__(self, queue):
        super(Bucket,self).__init__()
```

```
        self.q = queue

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                self.q.put(self.recv("inbox"))
            self.pause()
            yield 1
```

2. Then we create a background object and call its start() method:

```
from Axon.background import background

background().start()
```

3. Finally, we create and activate our Kamaelia pipeline of components, including the receiver component we've just written, passing it a thread-safe queue to put the data into:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Internet.TCPClient import TCPClient
from Kamaelia.Visualisation.PhysicsGraph import chunks_to_lines
from Queue import Queue

queue = Queue()

Pipeline(
    TCPClient("my.server.com", 1234),
    chunks_to_lines(),
    Receiver(queue)
).activate()
```

We can now fetch items of data, from the queue when they arrive:

```
received_line = queue.get()
```

Behavour

Create one of these and start it running by calling its start() method.

After that, any components you activate will default to using this scheduler.

Only one instance can be used within a given python interpreter.

The background thread is set as a "daemon" thread. This means that if your program exits, this background thread will be killed too. If it were not a daemon, then it would prevent the python interpreter terminating until the components running in it had all terminated too.

Other

background

A python thread which runs the Axon Scheduler. Takes the same arguments at creation that Axon.Scheduler.scheduler.run.runThreads accepts.

Create one of these and start it running by calling its start() method.

After that, any components you activate will default to using this scheduler.

Only one instance can be used within a given python interpreter.

Axon.debug

Internal debugging support - debug output logging

Provides a way to generate debugging (logging) output on standard output that can be filtered to just what is needed at the time.

- Some Axon classes create/use write debug output using an instance of debug()
- debug uses debugConfigFile.readConfig() to read a configuration for what should and should not be output

What debugging output actually gets output (and what is filtered out) is controlled by two things: what *section* the debugging output is under and the *level* of detail of a given piece of output.

Each call to output debugging information specifies what section it belongs to and the level of detail it represents.

The filtering of this is configured from a configuration file (see Axon.deugConfigFile for information on the format) which lists each expected section and the maximum level of detail that will be output for that section.

How to use it

Create a debug object:

```
debugger = Axon.debug.debug()
```

Specify the configuration to use, either specifying a file, or letting the debugger choose its own defaults:

```
debugger.useConfig(filename="my_debug_config_file")
```

Any subsequent debug objects you create will use the same configuration when you call their useConfig() method - irrespective of whether you specify a filename or not!

Call the note() method whenever you potentially want debugging output; specifying the "section name" and minimum debug level under which it should be reported:

```
while 1:
    ...
    assert self.debugger.note("MyObject.main", 10, "loop begins")
    ...
    if something_happened():
        ...
        assert self.debugger.note("MyObject.main", 5, "received ", msg)
        ...
```

- Using different section names for different parts of your debugging output allow you to select which bits you are interested in.
- Use the 'level' number to indicate the level of detail of any given piece of debugging output.

The note() method always returns True, meaning you can wrap it in an assert statement. If you then use python's "-O" command line flag, assert statements will be ignored, completely removing any performance overhead the due to the debugging output.

Adjusting the configuration of individual debugger objects

All debug objects share the same initial configuration - when you call their useConfig() method, all pick up the same configuration file that was specified on the first call.

However, after the useConfig() call you can customise the configuration of individual debug objects.

You can increase or decrease the maximum level of detail that will be output for a given section:

```
debugger.increaseDebug("MyObject.main")
debugger.decreaseDebug("MyObject.main")
```

You can add (or replace) the configuration for individual debugging sections -ie. (re)specify what the maximum level of detail will be for a given section:

```
debugger.addDebugSections()
```

Or you can replace the entire set:

```
replacementSections = { "MyObject.main" : 10,
                        "MyObject.init" : 5,
                        ...
                      }

debugger.addDebug(**replacementSections)
```

Other

debug

debug([assertBadDebug]) -> new debug object.

Object for outputting debugging output, filtered as required. Only outputs debugging data for section names it recognises - as specified in a debug config file.

Keyword arguments:

- assertBadDebug -- Optional. If evaluates to true, then any debug output for an unrecognised section (as defined in the configuration) causes an exception (default=1)

Axon.debugConfigDefaults

Default debugging configuration

This file defines default configuration used by Axon.debug.debug when it cannot find a configuration file to load.

The defaultConfig() method returns the default configuration.

Other

defaultConfig

Returns a default debugging configuration - a dictionary mapping section names to (level, location) tuples

Axon.debugConfigFile

Reading debugging configuration files

The readConfig() method reads debugging configuration from the specified file.

- Axon.debug.debug uses this to read configuration for itself.

Each call to output debugging information specifies what section it belongs to and the level of detail it represents. A configuration file specifies what sections should be expected and what maximum level of detail should be output for each.

Debugging configuration file format

Debugging configuration files are simple text files.

- **Comments** are lines beginning with a hash '#' character

- **Blank lines** are permitted
- **Configuration lines** are a space (not tab) separated triple of *section* name, *level* of detail, and *location*

For example:

```
# The following tags are for debugging the debug system
#

debugTestClass.even         5  default
debugTestClass.triple       10 default
debugTestClass.run          1  default
debugTestClass.__init__     5  default
debugTestClass.randomChange 10 default
```

For a given *section*, the *level* number specifies the maximum level of detail that you want outputted. Any calls to output debugging information for that section but with a higher level number will be filtered out.

The final *location* field is currently not used. It is recommended to specify "default" for the moment.

Other

debugConfig

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

readConfig

> Reads debug configuration from the specified file.
>
> Returns a dictionary mapping debugging section names to maximum levels of detail to be output for that section.

Axon.idGen

Unique ID generation

The methods of the idGen class are used to generate unique IDs in various forms (numbers, strings, etc) which are used to give microprocesses and other Axon objects a unique identifier and name.

- Every Axon.Microprocess.microprocess gets a unique ID

- Axon.ThreadedComponent.threadedcomponent uses unique IDs to identify threads

Generating a new unique ID

Do not use the idGen class defined in this module directly. Instead, use any of these module methods to obtain a unique ID:

- **Axon.idGen.newId(thing)** - returns a unique identifier as a string based on the class name of the object provided
- **Axon.idGen.strId(thing)** - returns a unique identifier as a string based on the class name of the object provided
- **Axon.idGen.numId()** - returns a unique identifier as a number
- **Axon.idGen.tupleId(thing)** - returns both the numeric and string versions of a new unique id as a tuple (where the string version is based on the class name of the object provided)

Calling tupleId(thing) is *not* equivalent to calling numId() then strId(thing) because doing that would return two different id values!

Examples:

```
>>> x=Component.component()
>>> idGen.newId(x)
'Component.component_4'
>>> idGen.strId(x)
'Component.component_5'
>>> idGen.numId()
6
>>> idGen.tupleId(x)
(7, 'Component.component_7')
```

Other

Debug

Output a debug message.

Specify the 'section' the debug message should come under. The user will have specified the maximum 'level' to be outputted for that section.

- Use higher level numbers for more detailed debugging output.
- Use different section names for different parts of your code to allow the user to select which sections they want output for

Always returns True, so can be used as argument to an assert statement. This means you can then disable debugging output (and any associated performance overhead) by using python's "-O" command line flag.

Keyword arguments:

- section -- the section you want this debugging output classified under
- level -- the level of detail of this debugging output (number)
- *message -- object(s) to print as the debugging output

debugger

debug([assertBadDebug]) -> new debug object.

Object for outputting debugging output, filtered as required. Only outputs debugging data for section names it recognises - as specified in a debug config file.

Keyword arguments:

- assertBadDebug -- Optional. If evaluates to true, then any debug output for an unrecognised section (as defined in the configuration) causes an exception (default=1)

idGen

Unique ID creator.

Use numId(), strId(), and tupleId() methods to obtain unique IDs.

newId

Allocates & returns the next available id combined with the object's class name, in string form

numId

Allocates & returns the next available id

strId

Allocates & returns the next available id combined with the object's class name, in string form

tupleId

Allocates the next available id and returns it both as a tuple (num,str) containing both the numeric version and a string version where it is combined with the object's class name.

Axon.util

General utility functions & common includes

Other

Finality

Used for implementing try...finally... inside a generator.

axonRaise

> Raises the supplied exception with the supplied arguments *if* Axon.util.production is set to True.

listSubset

Returns true if the requiredList is a subset of the suppliedList.

logError

> Currently does nothing but can be rewritten to log ignored errors if the production value is true.

next

next(iterator[, default])

Return the next item from the iterator. If default is given and the iterator is exhausted, it is returned instead of raising StopIteration.

production

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

removeAll

Very simplistic method of removing all occurances of y in list xs.

safeList

Returns the list version of arg, otherwise returns an empty list.

testInterface

Look for a minimal match interface for the component. The interface should be a tuple of lists, i.e. ([inboxes],[outboxes]).

vrange

range(stop) -> range object range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1. start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3. These are exactly the valid indices for a list of 4 elements. When step is given, it specifies the increment (or decrement).

- Kamaelia ----------pygame 2.1.2 (SDL 2.0.20, Python 3.10.12) Hello from the pygame community. https://www.pygame.org/contribute.html

Kamaelia.Apps.CL.CollabViewer.CollabParsing

CollabParser: Parse collaboration data between organizations received as dictionary

Parse collaboration data between organizations received as dictionary, and then send out TopologyViewer commands

Example Usage

A simple file driven collaboration parser and draw them with 3D topology viewer:

```
Pipeline( ReadFileAdaptor('Data/collab.json'),
          JSONDecoder(),
          CollabParser(),
          TopologyViewer3DWithParams(),
          ConsoleEchoer(),
        ).run()
```

How does it work?

The input format: The input is a dictionary, {'orgData' : {'org1' : ['staff1',...],...}, 'collabData' : {'collab1' : ['staff1',...],...}} e.g., {'orgData' : {'BBC' : ['Beckham', 'Bell', 'Betty', 'Bill', 'Brad', 'Britney'], 'Google' : ['Geoff', 'Gerard', 'Gordon', 'George', 'Georgia', 'Grant'], 'Manchester' : ['Michael', 'Matt', 'Madonna', 'Mark', 'Morgon', 'Mandela'], 'Leeds' : ['Leo', 'Lorri', 'Louis', 'Lampard', 'Lily', 'Linda'], 'Sheffield' : ['Sylvain', 'Sugar', 'Sophie', 'Susan', 'Scarlet', 'Scot']}, 'collabData' : {'Audio' : ['Beckham', 'Bell', 'Geoff', 'Gerard', 'Gordon', 'Leo'], 'Video' : ['Michael',

'Matt', 'Sophie', 'Susan'], 'Internet' : ['Sylvain', 'Sugar', 'Beckham', 'Mandela'], 'XML' : ['Lampard', 'Lily', 'Linda', 'Geoff', 'Scot'], 'Visualisation' : ['Leo', 'Lorri', 'Susan', 'Britney']} }

The output is TopologyViewer commands.

Typically, it receives inputs from JSONDecoder and send output to TopologyViewer3D. After the data are drawn by TopologyViewer3D, double-click nodes to show all people involved in the collaboration or belonging to the organization.

CollabWithViewParser: Parse collaboration data between organizations received as dictionary with view support

Parse collaboration data between organizations received as dictionary, and then send out a dictionary of TopologyViewer commands

Example Usage

A simple file driven collaboration parser, and then send output to DictChooser and at last draw them with 3D topology viewer:

```
Graphline(
    READER = ReadFileAdaptor('Data/collab.json'),
    JSONDECODER = JSONDecoder(),
    CONSOLEECHOER = ConsoleEchoer(),
    COLLABPARSER = CollabWithViewParser(),
    BUTTONORG = Button(caption="orgView", msg="orgView", position=(-10,8,-20)),
    BUTTONSTAFF = Button(caption="staffView", msg="staffView", position=(-8,8,-20)),
    DICTCHOOSER = DictChooser(),
    VIEWER = TopologyViewer3DWithParams(laws=laws),
linkages = {
    ("READER","outbox") : ("JSONDECODER","inbox"),
    ("JSONDECODER","outbox")  : ("COLLABPARSER","inbox"),
    ("COLLABPARSER","outbox")  : ("DICTCHOOSER","option"),
    ("BUTTONORG","outbox")  : ("DICTCHOOSER","inbox"),
    ("BUTTONSTAFF","outbox")  : ("DICTCHOOSER","inbox"),
    ("DICTCHOOSER","outbox")  : ("VIEWER","inbox"),
    ("VIEWER","outbox")  : ("CONSOLEECHOER","inbox"),
}
```

).run()

How does it work?

The input format: Same as CollabParser.

The output is a dictionary of TopologyViewer commands. The format is {'orgView' : [...], 'staffView' : [...]}

Typically, it receives inputs from JSONDecoder and sends output to the option box of DictChooser first, and then DictChooser sends data to TopologyViewer3D.

After the data are drawn by TopologyViewer3D, double-click nodes to show all people involved in the collaboration or belonging to the organization. Click button to swith between different views.

Components

CollabParser

CollabParser(...) -> new CollabParser component.

Kamaelia component to parse collaboration data between organizations received as dictionary.

CollabWithViewParser

CollabWithViewParser(...) -> new CollabWithViewParser component.

Kamaelia component to parse collaboration data between organizations received as dictionary into different views' TopologyViewer commands.

Kamaelia.Apps.CL.DictChooser

Dictionary Chooser

The DictChooser component chooses option from its dictionary options according to what received in its 'inbox' and sends the result to its 'oubox'. Dictionary options can either be created in the component's initialisation or created/ extended from its 'option' box at real time.

Example Usage

A simple picture show:

```
imageOptions = { "image1" : "image1.png", "image2" : "image2.png", "image3" : "image3.png" }

Graphline( CHOOSER  = DictChooser(options=imageOptions),
           image1  = Button(position=(300,16), msg="image1", caption="image1"),
           image2 = Button(position=(16,16),  msg="image2", caption="image2"),
           image3 = Button(position=(16,16),  msg="image3", caption="image3"),
           DISPLAY  = Image(position=(16,64), size=(640,480)),
           linkages = { ("image1" ,"outbox") : ("CHOOSER","inbox"),
                        ("image2","outbox") : ("CHOOSER","inbox"),
                        ("image3","outbox") : ("CHOOSER","inbox"),
```

```
                    ("CHOOSER" ,"outbox") : ("DISPLAY","inbox"),
                }
    ).run()
```

The chooser is driven by the 'image1', 'image2' and 'image3' Button components. Chooser then sends filenames to an Image component to display them.

How does it work?

When creating it, optionally pass the component a dictionary of options to choose from.

DictChooser will only accept finite length data dictionary.

If 'allowDefault' is enabled, it will send a arbitrary option chosen from its existing options to its "outbox" outbox before receiving anything from its 'inbox'. Otherwise, it does nothing. If no existing options, it sends nothing.

Send the index of dictionary options to DictChooser's "inbox" inbox to choose the option. If the index is one index of DictChooser's dictionary options, then it sends the corresponding option (a list of data) to its 'outbox' one by one.

Dictionary options can either be created in the component's initialisation or created/ extended from its 'option' box at real time. If the index is the same, new option will replace old one.

Format of dictionary options:

{index1 : data list1, index2 : data list2, ...}

If Chooser or InfiniteChooser receive a shutdownMicroprocess message on the "control" inbox, they will pass it on out of the "signal" outbox. The component will then terminate.

Components

DictChooser

Use dictionary to choose different options instead of list in Chooser

Kamaelia.Apps.CL.JSON

JSON serialisation codec

This component encode data (python object) to serialisable JSON format and decode serialised JSON data.

JSONEncoder: JSON serialisation encoder

Encode data (python object) to serialisable JSON format

Example Usage

A simple DataSource driven JSON serialisation encoder:

```
Pipeline( DataSource([['foo', {'bar': ('baz', None, 1.0, 2)}]]),
          JSONEncoder(),
          SimpleFileWriterWithOutput('Data/collab.json'),
          ConsoleEchoer(),
        ).run()
```

How does it work?

Whenever it receives data (python object) from its inbox, it encode the data using cjson and then send the serialised data (JSON string) to its outbox.

JSONDecoder: JSON serialisation decoder

Decode serialised JSON data to its original format

Example Usage

A simple DataSource driven JSON serialisation encoder:

```
Pipeline( ReadFileAdaptor('Data/collab.json'),
          JSONDecoder(),
          ConsoleEchoer(),
        ).run()
```

How does it work?

Whenever it receives data (JSON string) from its inbox, it decodes the data using cjson to its original format and then send the decoded data to its outbox.

Components

JSONEncoder

JSONEncoder(...) -> new JSONEncoder component.

Kamaelia component to encode data using JSON coding.

JSONDecoder

JSONDecoder(...) -> new JSONDecoder component.

Kamaelia component to decode data encoded by JSON coding.

Kamaelia.Apps.CL.SimpleFileWriterWithOutput

Components

SimpleFileWriterWithOutput

SimpleFileWriter(filename) -> component that writes data to the file

Writes any data sent to its inbox to the specified file.

Send the filename to its outbox.

Kamaelia.Apps.Compose.BuildViewer

Other

BuildViewer

NO DOCS

ComponentParticle

Version of Physics.Particle designed to represent components in a simple pipeline

Kamaelia.Apps.Compose.CodeGen

Other

CodeGen

NO DOCS

Kamaelia.Apps.Compose.GUI

Kamaelia.Apps.Compose.GUI.ArgumentsPanel

Other

ArgumentsPanel

NO DOCS

Kamaelia.Apps.Compose.GUI.BuilderControlsGUI

Other

BuilderControlsGUI

NO DOCS

Kamaelia.Apps.Compose.GUI.TextOutputGUI

Other

TextOutputGUI

NO DOCS

Kamaelia.Apps.Compose.PipeBuild

Other

PipeBuild

This component takes messages instructing it to add/remove entities from a pipeline, and outputs the messages needed to update a topology viewer, and also a full enumeration of the pipeline

Accepts: ("ADD", id, name, data, afterid) Add item to pipeline, with id, name, and data, immediately after the element with id 'afterid'. Or on the end if afterid == None.

("DEL", id) Remove item from the pipeline with id

Emits: Topology msgs: ("DEL", "ALL") ("ADD", "NODE", ...) ("ADD", "LINK", ...) ("DEL", "NODE", ...) ("DEL", "LINK", ...)

Pipeline enumeration messages: ("PIPELINE", [<pipeline data items>])

Kamaelia.Apps.Compose.PipelineWriter

Other

PipelineWriter

Writes a Kamaelia pipeline based on instructions received.

Accepts: ("PIPELINE", [pipelineelements]) pipelineelements = dictionary containing: name : name of class/factory function module : module containing it instantiation : string of the arguments to be passed (the bit that goes inside the brackets)

Emits: Strings of python source code followed by 'None' to terminate

Kamaelia.Apps.Europython09.BB.Authenticator

Other

Authenticator

NO DOCS

Kamaelia.Apps.Europython09.BB.Exceptions

Other

GotShutdownMessage

NO DOCS

Kamaelia.Apps.Europython09.BB.LineOrientedInputBuffer

Other

LineOrientedInputBuffer

NO DOCS

Kamaelia.Apps.Europython09.BB.MessageBoardUI

Other

MessageBoardUI

NO DOCS

Kamaelia.Apps.Europython09.BB.RequestResponseComponent

Other

RequestResponseComponent

NO DOCS

Kamaelia.Apps.Europython09.BB.Support

Other

Folder

NO DOCS

readUsers

NO DOCS

Kamaelia.Apps.Europython09.BB.UserStatePersistence

Other

StateSaverLogout

NO DOCS

UserRetriever

NO DOCS

Kamaelia.Apps.Europython09.FTP

Other

Uploader

NO DOCS

Kamaelia.Apps.Europython09.Options

Other

checkArgs

NO DOCS

needToShowUsage

NO DOCS

parseargs

NO DOCS

readJSONConfig

NO DOCS

showHelp

NO DOCS

showUsageBasedOnHowUsed

NO DOCS

Kamaelia.Apps.Europython09.Util

Other

Find

NO DOCS

Grep

NO DOCS

Sort

NO DOCS

TwoWayBalancer

NO DOCS

Kamaelia.Apps.Europython09.VideoCaptureSource

Other

VideoCaptureSource

NO DOCS

threadedcomponent

threadedcomponent([queuelengths]) -> new threadedcomponent

Base class for a version of an Axon component that runs main() in a separate thread (meaning it can, for example, block). Subclass to make your own.

Internal queues buffer data between the thread and the Axon inboxes and outboxes of the component. Set the default queue length at initialisation (default=1000).

A simple example:

```
class IncrementByN(Axon.ThreadedComponent.threadedcomponent):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
                value = value + self.n
                self.send(value,"outbox")

            if not self.anyReady():
                self.pause()
```

Kamaelia.Apps.Europython09.VideoFileTranscode

Other

Transcoder

NO DOCS

Kamaelia.Apps.GSOCPaint.Button

Pygame Button Widget

A button widget for pygame display surfaces. Sends a message when clicked.

Uses the Pygame Display service.

Example Usage

Three buttons that output messages to the console:

```
button1 = Button(caption="Press SPACE or click",key=K_SPACE).activate()
button2 = Button(caption="Reverse colours",fgcolour=(255,255,255),bgcolour=(0,0,0)).activate
button3 = Button(caption="Mary...",msg="Mary had a little lamb", position=(200,100)).activat
```

```
ce = ConsoleEchoer().activate()
button1.link( (button1,"outbox"), (ce,"inbox") )
button2.link( (button2,"outbox"), (ce,"inbox") )
button3.link( (button3,"outbox"), (ce,"inbox") )
```

How does it work?

The component requests a display surface from the Pygame Display service component. This is used as the surface of the button. It also binds event listeners to the service, as appropriate.

Arguments to the constructor configure the appearance and behaviour of the button component:

- If an output "msg" is not specified, the default is a tuple ("CLICK", id) where id is the self.id attribute of the component.
- A pygame keycode can be specified that will also trigger the button as if it had been clicked
- you can set the text label, colour, margin size and position of the button
- the button can have a transparent background
- you can specify a size as width,height. If specified, the margin size is ignored and the text label will be centred within the button

If a producerFinished or shutdownMicroprocess message is received on its "control" inbox. It is passed on out of its "signal" outbox and the component terminates.

Upon termination, this component does *not* unbind itself from the Pygame Display service. It does not deregister event handlers and does not relinquish the display surface it requested.

Components

ImageButton

NO DOCS

Kamaelia.Apps.GSOCPaint.ColourSelector

Colour Selector Widget

A Colour Selector tool, which represents all colours in the RGB spectrum.
One surface is used to plot the colours for selection and a slider used to manipulate the other colour, buttons used to change with colours are plotted.
Example Usage

Colour Selector Widget

Create a ColourSelector at 10,170 (from top-left corner) of size 255,255
this provides best colour display.

```
from Kamaelia.Apps.GSOCPaint.ColourSelector import
ColourSelector

colSel = ColourSelector(position = (10,170), size =
(255,255)).activate()

self.link( (colSel,"outbox"), (self,"outbox"), passthrough = 2 )
```

How Does it Work?

The Component requests a surface and by default plots the red colours
against green. The border is also drawn, this is here to represent what colour
is currently selected. As the user moves around on the surface selecting various
colours, the border updates to represent this.
The colours are plotted only once to reduce CPU usage, if I were to have a marker
at the point of the selected colour a "crosshair" perhaps I would need to have the
background constantly redrawing increasing CPU usgae greatly.
The pygame slider code is used to control the colour which isn't being plotted.
(Try it out, you'll see what I mean)
***
Other

ColourSelector

XYPad([bouncingPuck, position, bgcolour, fgcolour, positionMsg,

collisionMsg, size]) -> new XYPad component.

Create an XY pad widget using the Pygame Display service.
Sends messages

for position and direction changes out of its "outbox" outbox.

Keyword arguments (all optional):

Colour Selector Widget

bouncingPuck -- whether the puck will continue to move after it has been

dragged (default=True)

position -- (x,y) position of top left corner in pixels

bgcolour -- (r,g,b) fill colour (default=(255,255,255))

fgcolor -- (r, g, b) colour of the puck and border

messagePrefix -- string to be prepended to all messages

positionMsg -- sent as the first element of a (positionMsg, 1) tuple when

the puck moves

collisionMsg -- (t, r, b, l) sent as the first element of a

(collisionMsg[i], 1) tuple when the puck hits a side

(default = ("top", "right", "bottom", "left"))

size -- (w,h) in pixels (default=(100, 100))

##############################
Kamaelia.Apps.GSOCPaint.Slider

==========
Slider Widget

An interface widget to give a visual bar where the user can drag a line up and down to represent a value they're selecting for example a "Volume" bar.

Example Usage

Create a slider bar at 10, 460 with the messagePrefix as "Size" and a default value of 9.

from Kamaelia.Apps.GSOCPaint.Slider import Slider

SizeSlider = Slider(size=(255, 50), messagePrefix = "Size", position

= (10, 460), default = 9).activate() self.link((SizeSlider,"outbox"), (self,"outbox"), passthrough = 2)

How Does it Work?

The slider returns a tuple that looks like this (messagePrefix, value)

Other

Slider

XYPad([bouncingPuck, position, bgcolour, fgcolour, positionMsg, collisionMsg, size]) -> new XYPad component.

Create an XY pad widget using the Pygame Display service. Sends messages for position and direction changes out of its "outbox" outbox.

Keyword arguments (all optional): bouncingPuck -- whether the puck will continue to move after it has been dragged (default=True) position -- (x,y) position of top left corner in pixels bgcolour -- (r,g,b) fill colour (default=(255,255,255)) fgcolor -- (r, g, b) colour of the puck and border messagePrefix -- string to be prepended to all messages positionMsg -- sent as the first element of a (positionMsg, 1) tuple when the puck moves collisionMsg -- (t, r, b, l) sent as the first element of a (collisionMsg[i], 1) tuple when the puck hits a side (default = ("top", "right", "bottom", "left")) size -- (w,h) in pixels (default=(100, 100))

Kamaelia.Apps.GSOCPaint.ToolBox

Paint ToolBox.

This is the "Tool box" window for Kamaelia: Paint, it brings together
all the widgets used by the paint app and places them on a surface.
Much like the one in Gimp or Photoshop (but less complex and more fun :))
***
Other

ToolBox

NO DOCS
##################################
Kamaelia.Apps.Games4Kids.BasicSprite

Paint ToolBox.

***
Other

BasicSprite

NO DOCS
##
Kamaelia.Apps.Games4Kids.MyGamesEventsComponent

***
Other

MyGamesEventsComponent

NO DOCS
##
Kamaelia.Apps.Games4Kids.SpriteScheduler

***
Other

SpriteScheduler

NO DOCS
##
Kamaelia.Apps.Grey.ConcreteMailHandler

==========
Concrete Mail Core

This code enforces the basic statemachine that SMTP expects, switching between the various commands and finally results in forwarding on the SMTP command

to the appropriate SMTP server. By itself, this can be used as a simple SMTP Proxy server.

This class is a subclass of MailHandler, and as such largely consists of methods overriding methods from MailHandler which are designed to be overridden.

Furthermore, it expects to forward any mail it accepts to another SMTP mail server, as transparently as possible. Thus this concrete mail core effectively forms the core of an SMTP proxy.

Note

As it stands however, by default this mail proxy will *not* forward any mails to the internal server. In order to change this, you would need to subclass this server and replace the method "shouldWeAcceptMail" since that defaults to returning False

Example Usage

As noted, you are not expected to use this ConcreteMailHandler directly, but if you did, you would use it like this:

```
ServerCore(protocol=ConcreteMailHandler, port=1025)
```

At minimum, you would need to do this:: class SpamMeHandler(ConcreteMailHandler): def sh
return True

ServerCore(protocol=SpamMeHandler, port=1025)

You could alternatively do this:: class SpamMeMailServer(ServerCore): class protcol(Concretel
return True

ServerCore(port=1025)

How does it work?

As noted this overrides all the methods relating to handling SMTP commands, and enforces the state machine that SMTP requires. It's particularly strict about this, in breach of Postel's law for two reasons -

- It helps eradicate spam
- because most spam systems are generally lax these days and most non-spam systems are generally strict

It was also primarily written in the context of a greylisting server.

Some core values it tracks with regard to a mail -

- a list of recipients
- the (claimed) sender
- the (claimed) remote/client name
- the actual client & local port/ip addresses

Once the client has finished sending the data for an email, the proxy forwards the mail to the local real SMTP server. Fundamentally this happens by making a connection to the real server using the TCPClient component, and then replaying all the lines the original server sent us to the local server.

(ie an inbox_log is built up with all data recieved from inbox "inbox" and then the contents of this are replayed when being sent to the local (real) SMTP mail server)

Configuration

This class provides a large number of configuration options. You can either change this through subclassing or by providing as named arguments to the ___init___ function. The options you have -

- servername - The name this server will choose to use to identify itself
- serverid - The string this server will use to identify itself (in terms of software in use)
- smtp_ip - the ip address of the server you're proxying for/to
- smtp_port - this is the port the server you're proxying for/to is listening on

The following attributes get set when a client connects -

- peer - the IP address of the client
- peerport - the port which the peer is connected from
- local - the IP address the client has connected to
- localport - the port which they're connected to

Components

ConcreteMailHandler

NO DOCS

Kamaelia.Apps.Grey.GreyListingPolicy

Greylisting Policy For/Subclass Of Concrete Mail Handler

This component implements a greylisting SMTP proxy protocol, by subclassing ConcreteMailHandler and overriding the appropriate methods (primarily the shouldWeAcceptMail method).

For more detail, please see http://www.kamaelia.org/KamaeliaGrey

Example Usage

You use this as follows (at minimum):

```
ServerCore(protocol=GreyListingPolicy, port=25)
```

If you want to have a hardcoded/configured greylisting server you could do this:

```
class GreyLister(ServerCore):
    class protocol(GreyListingPolicy):
        allowed_senders = []
        allowed_sender_nets = []
        allowed_domains = [ ]

GreyLister(port=25)
```

How does it work?

Primarily it override the method shouldWeAcceptMail, and implements the following logic:

```
if self.sentFromAllowedIPAddress():  return True # Allowed hosts can always send to anywhere
if self.sentFromAllowedNetwork():    return True # People on truste networks can always do t
if self.sentToADomainWeForwardFor():
    try:
        for recipient in self.recipients:
            if self.whiteListed(recipient):
                return True
            if not self.isGreylisted(recipient):
                return False
    except Exception, e:
        pass
    return True # Anyone can always send to hosts we own
```

Clearly AllowedIPAddress, AllowedNetwork, whiteListed, and DomainWeForwardFor are fairly clear concepts, so for more details on those please look at the implementation.

isGreylisted by comparison is slightly more complex. Fundamentally this works on the basis of saying this:

- have we seen the triple (ip, sender, recipient) before ?
- if we have, then allow the message through
- otherwise, defer the message

Now there is a little more subtlty here, based on the following conditions:

- If greylisted, and not been there too long, allow through
- If grey too long, refuse (restarting the greylisting for that combo)
- If not seen this triplet before, defer and note triplet
- If triplet retrying waaay too soon, reset their timer & defer
- If triplet retrying too soon generally speaking just defer
- If triplet hasn't been seen in aaaages, defer
- Otherwise, allow through & greylist them

Components

GreyListingPolicy

NO DOCS

Kamaelia.Apps.Grey.MailHandler

Abstract SMTP Mailer Core

This component effectively forms the skeleton of an SMTP server. It expects an SMTP client to connect and send various SMTP requests to it. This basic SMTP Mailer Core however, does not actually do anything in response to any of the SMTP commands it expects.

Each SMTP command is actually given a dummy callback which more customised SMTP protocol handlers are expected to override. Beyond this, this component is expected to be used as a protocol handler for ServerCore.

Fundamentally, this component handles the command/response structure of SMTP fairly directly, but expects the brains of the protocol to be implemented by a more intelligent subclass.

Example Usage

Whilst this will work to a minimal extent:

```
ServerCore(protocol=MailHandler, port=1025)
```

This will not actually form a very interesting SMTP, nor SMTP compliant, server since whilst it will tell you commands it doesn't understand, it will not do anything interesting.

You are as noted expected to subclass MailHandler. For a better example of how to subclass MailHandler you are suggested to look at Kamaelia.Apps.ConcreteMailHandler.ConcreteMailHandler

Note

This component is not complete - you are expected to subclass it to finish it off as you need. Specifically it does not implement the following:

- It does not enforce "this command followed by that command"
- It does not actually do anything with any DATA a client sends you
- It neither performs local mail delivery nor proxying - you'd need to implement this yourself.

How does it work?

The component is expected to be connected to a client TCP connection by ServerCore, such that messages from the network arrive on inbox "inbox", and outgoing messages get sent to outbox "outbox"

The component will terminate if any of these is true:

- The client breaks the connection
- One of the methods sets self.breakConnection to True.
- If a "socketShutdown" message arrives on inbox "control"

The main() method divides the connection into effectively two main states:

- accepting random commands prior to getting a DATA command
- accepting the email during a DATA command

SMTP commands are specifically dispatched to a particular handler for that command. In this component none of the handlers do anything interesting.

Configuration

The abstract mailer supports some basic config settings:

- logfile - path/filename where requests should get logged
- debuglogfile - path/filename to where the debug log file should do.

Methods you are expected to override

Whilst you are probably better off subclassing ConcreteMailHandler, you will probably need to override the following methods in a subclass if you subclass MailHandler directly.

- handleConnect(self)
- handleHelo(self,command)
- handleEhlo(self,command)
- handleMail(self,command)
- handleRcpt(self,command)
- handleData(self,command)
- handleQuit(self,command)
- handleRset(self,command)
- handleNoop(self,command)
- handleVrfy(self,command)
- handleHelp(self,command)
- logResult(self)
- handleDisconnect(self)

Components

MailHandler

NO DOCS

Kamaelia.Apps.Grey.PeriodicWakeup

Components

PeriodicWakeup

NO DOCS

Kamaelia.Apps.Grey.Support

Other

SlurpFile

NO DOCS

openConfig

NO DOCS

parseConfigFile

NO DOCS

Kamaelia.Apps.Grey.WakeableIntrospector

On Demand/Wakeable Introspector

This component grabs a list of all running/runnable components whenever it receives a message on its inbox "inbox". This list is then sorted, and noted to a logfile.

Example Usage

This component is intended to be used with PeriodicWakeup, as follows:

```
Pipeline(
     PeriodicWakeup(interval=20),
     WakeableIntrospector(logfile="/tmp/trace"),
)
```

How does it work?

This component uses the fact that we can ask the scheduler for a list of running componenents, takes this, sorts it and dumps the result to a logfile.

It then sits quietly waking for a message (any message) on the inbox "inbox".

Termination

This component will shutdown if any message is sent to its control inbox.

TODO

In retrospect, it may've been nicer to split the introspection from the logging. Better termination/shutdown would be a good idea.

Components

WakeableIntrospector

NO DOCS

Other

LoggingWakeableIntrospector

NO DOCS

Kamaelia.Apps.IRCLogger.Support

Other

AppendingFileWriter

appends to instead of overwrites logs

HTMLOutformat

each formatted line becomes a line on a table.

Kamaelia

This is a doc string, will it be of use?

LoggerWriter

> puts an html header in front of every file it opens. Does not make well-formed HTML, as files are closed without closing the HTML tags. However, most browsers don't have a problem with this. =D

TimedOutformat

prepends a timestamp onto formatted data and ignores all privmsgs prefixed by "[off]"

cannedResponse

NO DOCS

cannedYesTheyreAround

NO DOCS

currentDateString

returns the current date in YYYY-MM-DD format

currentTimeString

returns the current time in hour:minute:second format

getFilenames

returns tuple (logname, infoname) according to the parameters given

lastlog

Convert a string or number to a floating point number, if possible.

logging

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

outformat

each formatted line becomes a line on a table.

respondToQueries

Takes a BasicLogger as its first argument. If this function recognizes "msg" as a command, then it sends back the appropriate response to IRC

Kamaelia.Apps.JMB.Common.ConfigFile

NOTE: This module may be useful outside of the context it's currently being used. However, there are a few caveats that may not make it unsuitable for general usage, thus it is in Kamaelia.Apps.

This module is used for reading config files. A config file is a standard .ini file that looks something like this:

[section] option1: blah option2: blahblah option3: blahblahblah

ConfigFileReader

The base of the set is the ConfigFileReader component, which will read the config file and then send each individual section of it out on its outbox as a tuple consisting of the section name, and a list of tuples that represents the options.

Thus, for the example given above, the ConfigFileReader will output something like:

('section', [('option1', 'blah'), ('option2', 'blahblah'), ('option3', 'blahblah-blah')

]

)

The formatters

The ConfigFileReader was designed to use a set of formatters to format the results into a usable piece of data. FormatterBase defines how these formatters should work.

Using this method, each formatter may be a link in the chain or they may also be the producer of the end result. You may get the end result out of the last component by calling its getResults method. This will simply return self.results (which is where the results should be stored by the formatter).

DictFormatter

This formatter is used to format the results into a dictionary. Please note that the value stored into self.results will be different from the compiled results of listening to its outbox. Reading self.results will give a dictionary of dictionaries while the results passed on the outbox will be a tuple containing the section name and a dictionary that contains the options. For example, if a DictFormatter was connected to a ConfigFileReader that read the example config file posted above, you would get the following if you read self.results:

{'section' : { 'option1' : 'blah', 'option2' : 'blahblah', 'option3' : 'blahblah-blah',

}}

However, if you were to read the DictFormatter's outbox, you would get:

('section', { 'option1' : 'blah', 'option2' : 'blahblah', 'option3' : 'blahblahblah',

})

This is done in case any listening formatters depend upon the order of the sections.

ParseConfigFile

NOTE: This function was made to be used before any other components have been activated. Using this function when other components are activated will result in undefined behavior.

This function is the main way to read a config file and format it. You pass it a filename to read and a set of formatters.

As an example, this will read a file named by the string filename and print the resulting dict out:

```
print ParseConfigFile(filename, [DictFormatter()])
```

Other

ConfigFileReader

NO DOCS

DictFormatter

NO DOCS

FormatterBase

NO DOCS

ParseConfigFile

NO DOCS

ParseException

NO DOCS

Kamaelia.Apps.JMB.Common.Console

This is just a source file for miscellaneous console IO functions. There is also an interface for using python's logging system for console IO.

Other

critical

Print a critical message to the screen.

debug

Print a debug message to the screen

error

Print an error to the screen

info

Print info to the screen

logging

Logging package for Python. Based on PEP 282 and comments thereto in comp.lang.python.

To use, simply 'import logging' and log away!

prompt_corrupt

This is really just a convenience method for prompt_yesno.

prompt_yesno

> Just a generic function to determine if the user wants to continue or not. Will repeat if input is unrecognizable.

setConsoleName

> This sets the name of the python logger that represents the console.

warning

Print a warning to the screen

Kamaelia.Apps.JMB.Common.ServerSetup

This module includes various useful functions for setting a webserver up.

Other

initializeLogger

This sets up the logging system.

killLoggers

Shuts down the logging system and flushes input.

normalizeUrlList

Add necessary default entries that the user did not enter.

normalizeWsgiVars

Put WSGI config data in a state that the server expects.

processPyPath

Use ServerConfig to add to the python path.

Kamaelia.Apps.JMB.Common.Structs

This is a collection of configuration objects that will allow for object-oriented access of configuration data rather than having to use a dictionary. This data is intended to come from an ini file, but may come from anywhere.

Other

ConfigObject

NO DOCS

ServerConfigObject

NO DOCS

StaticConfigObject

NO DOCS

XMPPConfigObject

NO DOCS

Kamaelia.Apps.JMB.Common.UrlConfig

This module contains the necessary parts to adapt the Kamaelia Config file reader to read a Wsgi Config file as used in Kamaelia Publish. A UrlListFormatter is included, but you probably don't want to use it directly unless you want to make another formatter to parse its output.

Instead, you should call the convenience function ParseUrlFile. This will process all of the formatting for you.

Example

Suppose you had the following urls file:

[static_files] regex: static import_path: Kamaelia.Apps.JMB.WSGI.Apps.Static app_object: static_app static_path: ~/www index_file: index.html

[simple_app] regex: simple import_path: Kamaelia.Apps.JMB.WSGI.Apps.Simple app_object: simple_app

[error_404] import_path: Kamaelia.Apps.JMB.WSGI.Apps.ErrorHandler app_object: application

A call to ParseUrlFile would produce the following output:

```
[ { 'kp.regex' : 'static', 'kp.import_path' : 'Kamaelia.Apps.JMB.WSGI.Apps.Static',
          'kp.app_object' : 'static_app', 'kp.static_path' : '~/www',
          'kp.index_file' : 'index.html',

    }

          'kp.regex' : 'simple', 'kp.import_path' : 'Kamaelia.Apps.JMB.WSGI.Apps.Simple',
          'kp.app_object' : 'simple_app',

    }

          'kp.regex' : '.*', 'kp.import_path' : 'Kamaelia.Apps.JMB.WSGI.Apps.ErrorHandler',
          'kp.app_object' : 'application',

    }

]
```

Changes made

The Formatter will make the following changes:

* Each option will be prepended with 'kp.' if it does not already have an identifier at the beginning. Thus, 'foo.bar' will stay 'foo.bar' and will not be prepended with 'kp.', while 'regex' will be converted to 'kp.regex'

* It will raise an exception if there is not an error_404 section or the error_404 section contains a regex.

- It will add '.*' as the regex for error_404.

Other

ParseUrlFile

NO DOCS

UrlListFormatter

This component expects to be linked to a DictFormatter

Kamaelia.Apps.JMB.Common.autoinstall

This module contains the functionality for autoinstall of necessary config files.

FIXME: Allow user to override the default install location.

Other

autoinstall

This function essentially just takes a tar file from the data file within a zip executable and expands it into the users home directory.

Kamaelia.Apps.JMB.WSGI.Apps.ErrorHandler

This is an error serving application. It currently only supports serving 404 pages, but may be adapted to serve other kinds of error pages.

This app requires no custom environment variables or dependencies.

Other

application

This is just a plain old error page serving application.

Kamaelia.Apps.JMB.WSGI.Apps.Simple

This is just a simple WSGI app that will call start_response, write to the write callable, write a message to the log, print out every environ entry, and then print the text of the wsgi.input entry.

This app requires no custom environ entries or dependencies.

Other

simple_app

Simplest possible application object

Kamaelia.Apps.JMB.WSGI.Apps.Test

This is just a collection of WSGI apps that can be used for testing purposes.

Other

BlockingApp

> This application will run an infinite loop and produce no output. This can be useful to test how blocking applications can affect the server and to give you time to test what will happen if an event happens before a WSGI app has begun transmitting its output.

Kamaelia.Apps.JMB.WSGI._WSGIHandler

WSGI Handler

NOTE: This is experimental software.

This is the WSGI handler for ServerCore. It will wait on the HTTPParser to transmit the body in full before proceeding. Thus, it is probably not a good idea to use any WSGI apps requiring a lot of large file uploads (although it could theoretically function fairly well for that purpose as long as the concurrency level is relatively low).

For more information on WSGI, what it is, and to get a general overview of what this component is intended to adapt the ServerCore to do, see one of the following links:

- http://www.python.org/dev/peps/pep-0333/ (PEP 333)
- http://www.wsgi.org/wsgi/ (WsgiStart wiki)
- http://en.wikipedia.org/wiki/Web_Server_Gateway_Interface (Wikipedia article on WSGI)

Dependencies

This component depends on the wsgiref module, which is included with python 2.5. Thus if you're using an older version, you will need to install it before using this component.

The easiest way to install wsgiref is to use easy_install, which may be downloaded from http://peak.telecommunity.com/DevCenter/EasyInstall . You may then install wsgiref using the command "sudo easy_install wsgiref" (without the quotes of course).

Please note that Kamaelia Publish includes wsgiref.

How do I use this component?

The easiest way to use this component is to use the WsgiHandler factory function that is included in Factory.py in this package. That method has URL handling that will route a URL to the proper place. There is also a SimpleWsgiHandler that may be used if you only want to support one application object. For more information on how to use these functions, please see Factory.py. Also please note that both of these factory functions are made to work with ServerCore/SimpleServer. Here is an example of how to create a simple WSGI server:

from Kamaelia.Protocol.HTTP import HTTPProtocol from Kamaelia.Experimental.Wsgi.Factory import WsgiFactory # FIXME: Docs are broken :-(

WsgiConfig = { 'wsgi_ver' : (1, 0), 'server_admin' : 'Jason Baker', 'server_software' : 'Kamaelia Publish'

}

url_list = [#Note that this is a list of dictionaries. Order is important. {
'kp.regex' : 'simple', 'kp.import_path' : 'Kamaelia.Apps.Wsgi.Apps.Simple', 'kp.app_obj' : 'simple_app',

}

'kp.regex' : '.', *#The .* means that this is a 404 handler
'kp.import_path' : 'Kamaelia.Apps.Wsgi.Apps.ErrorHandler',
'kp.app_obj' : 'application',

}

]

routing = [['/', WsgiFactory(WsgiConfig, url_list)]]

ServerCore(protocol=HTTPProtocol(routing), port=8080, socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)).run()

Internal overview

request object Note that certain WSGI applications will require configuration data from the urls file. If you use the WsgiFactory to run this handler, all options specified in the urls file will be put into the environment variable with a kp. in front of them.

For example, the 'regex' entry in a urls file would go into the environ dictionary like this if it was set to 'simple':

{ ... 'kp.regex' : 'simple', ...

}

wsgi.input PEP 333 requires that the WSGI environ dictionary also contain a file-like object that holds the body of the request. Currently, the WsgiHandler will wait for the full request before starting the application (which is not optimal behavior). If the method is not PUT or POST, the handler will use a pre-made null-file object that will always return empty data. This is an optimization to lower peak memory usage and to speed things up.

WsgiConfig The WsgiHandler requires a WsgiConfig dictonary for general configuration info. The following items are required to be defined:

- wsgi_ver: the WSGI version as a Tuple. You want to use (1, 0)
- server_admin: the name and/or email address of the server's administrator
- server_software: The software and/or software version that runs the server

FIXME: It would be nice if the WsgiConfig were made into an object rather than a dictionary.

Other

ErrorLogger

This is the file-like object intended to be used for wsgi.errors.

NullFileLike

This is a file-like object that is meant to represent an empty file.

WsgiAppError

This is an exception that is used if a Wsgi application does something it shouldnt.

WsgiError

This is used to indicate an internal error of some kind. It is thrown if the write() callable is called without start_response being called.

Kamaelia.Apps.JMB.WSGI._factory

This module is what you use to create a WSGI Handler.

Other

PopWsgiURI

This is a function to pop a level from the PATH_INFO key into the SCRIPT_NAME key of a WSGI-like dictionary.

request - a WSGI-like dictionary

FIXME: Rest needs fixing

SimpleWSGIFactory

Creates a WSGI Handler that can handle only one WSGI Application.

WSGIConfig - A WSGIConfig object app_object - The WSGI application object to run error_log - The file to store errors in logger_name - The name of the python logger to log errors to

WSGIFactory

Creates a WSGI Handler using url routing.

WSGIConfig - A WSGIConfig object url_list - A URL list to look up App objects. It must contain three keys: kp.regex - the regex to match the uri against (will only match the first section) kp.import_path - The path to import the WSGI application object from kp.app_object - the attribute of the module named in kp.import_path that names the WSGI application object error_log - The file to store errors in logger_name - The name of the python logger to log errors to

WSGIImportError

This exception is to indicate that there was an error in importing a WSGI app.

Kamaelia.Apps.MH.Profiling

Other

FormattedProfiler

NO DOCS

Profiler

Profiler([samplingrate][,outputrate]) -> new Profiler component.

Basic code profiler for Axon/Kamaelia systems. Measures the amount of time different microproceses are running.

Keyword arguments: - samplingrate -- samples of state taken per second (default=1.0) - outputrate -- times statistics are output per second (default=1.0)

ProfilerOutputFormatter

NO DOCS

Kamaelia.Apps.MPS.TPipe

What it does

XXX TODO: Module level docs

Much like unix "tee", copies data from inboxes to outboxes but also to a subcomponent. Copying to the subcomponent is conditional on the data, and also

Example Usage

What this shows followed by double colon:: def func(): print "really really simple minimal code fragment"

Indicate any runtime user input with a python prompt:: »> func() really really simple minimal code fragment

Optional comment on any particularly important thing to note about the above example.

How does it work?

Statements, written in the present tense, describing in more detail what the component does.

Explicitly refer to "named" inbox an "named" outbox to avoid ambiguity.

Does the component terminate? What are the conditions?

If the 'xxxx' argument is set to yyy during initialization, then something different happens.

A subheading for a subtopic

Lists of important items might be needed, such as commands: the item
: A description of the item, what it is and what it does, and maybe consequences of that.

another item
: A description of the item, what it is and what it does, and maybe consequences of that.

You may also want bullet lists:

- first item
- second item

Optional extra topics

May be necessary to describe something separately, eg. a complex data structure the component expects to receive, or the GUI interface it provides.

Other

TPipe

TPipe([condition=<callback>][,action=<Component>],[,sink=<Bool>],[,source=<Bool>]) -> new TPipe component.

This component takes a single component as an argument. It runs this argument and sets it up to be able to send data to it. (and optionally recieve data from it) The subcomponent can obviously be any component, for example another pipeline, graphline, threaded-component or backplane.

This component is designed to sit in a Pipeline, where there is interesting data flowing through. It always forwards whatever messages it recieves on its inboxes "inbox,control" to its outboxs "outbox,signal", *UNLESS* the flag "demux" is set.

Additionally, it runs the condition provided against each piece of data. If the condition returns True, then the data is forwarded to the subcomponent, otherwise it isn't.

If the argument "mode" is "route", then any piece of data goes either to the subcomponent inboxes or out the component's outboxes. If the argument "mode" is "split", it can go to both. The default is "split".

Finally it takes two boolean arguments "sink" and "source. sink defaults to true and means that we should send data to the subcomponent. source defaults to false. If source is True however, any data coming out the subcomponents outboxes "outbox, signal" is passed through to TPipes outboxes "outbox, signal" (respectively).

Keyword arguments:

- condition -- Applied to data on inbox, forwards to subcomponent
- action -- The subcomponent to be activated and recieve a copy of data
- sink - If True (default) condition will be applied to data if true, data forwarded to action
- source - If True (default=False), any data from action will go out our outbox. In a pipeline this results in data injection.
- mode - "split" (default) or "route". "split" means the each piece of data can go to both the component's outboxes and the subcomponent. "route" means the data goes to one or the other.

Kamaelia.Apps.SA.Chassis

This file contains some utility classes which are used by both the client and server components of the port tester application.

Other

TTL

This "Time To Live" component is designed to wrap another existing component. The TTL starts an embedded SingleTick component which waits for "delay" seconds and then the TTL progressivly becomes more aggressive in its attempts to shutdown the wrapped component. Ideally this component should not be needed, but it is handy for components that do not have their own timeout functionality.

TTL(comp, delay)

Kamaelia.Apps.SA.DSL

This file contains some utility classes which are used by both the client and server components of the port tester application.

Other

DataSink

This is a limited use component. It is sort of the inverse of the DataSource component. It takes messages in the inbox and sticks them in a list and passes them on to its outbox. The list is not thread-safe, so do not access it until the component has shutdown.

DataSink(outlist, limit=0, pass_thru=True)

Kamaelia.Apps.SA.Time

This file contains some utility classes which are used by both the client and server components of the port tester application.

Other

PeriodicTick

This threaded component will periodically (every "delay" seconds) send True out it's outbox. You can specify an optional "check_interval" which will cause the component to more frequently check it's control inbox for termination signals.

PeriodicTick(delay, check_interval=delay, tick_mesg=True)

SingleTick

This threaded component will wait "delay" seconds then send True out it's outbox and shutdown. You can specify an optional "check_interval" which will cause the component to periodically check it's control inbox for early termination signals.

SingleTick(delay, check_interval=delay, tick_mesg=True)

Kamaelia.Apps.Show.GraphSlides

Components

GraphSlideXMLComponent

NO DOCS

Other

GraphSlideXMLParser

NO DOCS

onDemandGraphFileParser_Prefab

NO DOCS

Kamaelia.Apps.SocialBookmarks.BBCProgrammes

Interface to BBC /programmes JSON etc - Identifies PID of current programme on a chosen channel - Provides output for PIDs in a chosen format (XML, RDF etc) - Identifies currently playing tracks on radio channels TODO

Other

GMT

NO DOCS

NowPlaying

NO DOCS

WhatsOn

NO DOCS

Kamaelia.Apps.SocialBookmarks.ComponentBoxTracer

Other

ComponentBoxTracer

NO DOCS

Kamaelia.Apps.SocialBookmarks.Print

Other

Print

NO DOCS

PrintOnFile

Class to make printing of messages conditional

doPrint

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

Kamaelia.Apps.SocialBookmarks.PureFilter

Other

PureFilter

NO DOCS

Kamaelia.Apps.SocialBookmarks.StopOnFile

Other

StopOnFile

NO DOCS

Kamaelia.Apps.SocialBookmarks.TwitterStream

Interface to Twitter streaming API - Grabs JSON data based on chosen keywords - Also relays the PIDs related to the JSON responses to ensure they match up TODO: Add watching for in-stream rate limiting / error messages - Doesn't currently honour tweet deletion messages (TODO)

Other

HTTPClientRequest

NO DOCS

HTTPClientResponseHandler

NO DOCS

HTTPDataStreamingClient

NO DOCS

LineFilter

NO DOCS

ShutdownNow

NO DOCS

TwitterStream

NO DOCS

addTrace

NO DOCS

http__basic__auth__header

NO DOCS

parse__url

NO DOCS

Kamaelia.Apps.SocialBookmarks.WithSSLProxySupport

Other

ConnectRequest

NO DOCS

FailingComponent

NO DOCS

GeneralFail

NO DOCS

HandleConnectRequest

NO DOCS

Linebuffer

NO DOCS

Pauser

NO DOCS

ShutdownNow

NO DOCS

Sink

NO DOCS

Tagger

NO DOCS

With

NO DOCS

Kamaelia.Apps.SpeakNWrite.Gestures.Analyser

Other

Analyse

Takes a quantised stroke and attempts to match it against letter shapes

DIVERGENCE_CURVE_IDEAL

Convert a string or number to a floating point number, if possible.

MIN_SIZE_THRESHOLD

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

Kamaelia.Apps.SpeakNWrite.Gestures.Grammar

Other

StrokeGrammar

> Takes output from stroke recogniser and applies grammar rules to combine strokes (taking into account spatial position) into output symbols.

Kamaelia.Apps.SpeakNWrite.Gestures.GrammarRules

Other

BCK

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

grammar

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Kamaelia.Apps.SpeakNWrite.Gestures.Patterns

Other

BCK

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

patterns

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Kamaelia.Apps.SpeakNWrite.Gestures.Pen

Other

Pen

NO DOCS

Kamaelia.Apps.SpeakNWrite.Gestures.PreProcessing

Other

DOT_SEPARATION_THRESHOLD

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

MAX_STROKE_POINTS

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

Normalise

Takes a path and normalises it to a bounding box width and height 1, and also notes the aspect ratio.

QuantiseStroke

Quantises a stroke into a reduced set of points

SegmentStroke

Takes a quantised stroke and breaks it into line segments

angleMappings

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

nearest45DegreeStep

Returns (in degrees) the nearest 45 degree angle match to the supplied vector.

> Returned values are one of 0, 45, 90, 135, 180, 225, 270, 315. If the supplied vector is (0,0), the returned angle is 0.

WARNING: Python Imaging Library Not available, defaulting to bmp only mode

Kamaelia.Apps.SpeakNWrite.Gestures.StrokeRecogniser

Other

StrokeRecogniser

NO DOCS

Kamaelia.Apps.Whiteboard.Canvas

Other

Canvas

> Canvas component - pygame surface that accepts drawing instructions

Kamaelia.Apps.Whiteboard.CheckpointSequencer

Other

CheckpointSequencer

NO DOCS

Kamaelia.Apps.Whiteboard.CommandConsole

Other

CommandConsole

NO DOCS

CommandParser

NO DOCS

parseCommands

NO DOCS

Kamaelia.Apps.Whiteboard.Decks

Other

Decks

NO DOCS

Kamaelia.Apps.Whiteboard.Email

Other

Email

NO DOCS

Kamaelia.Apps.Whiteboard.Entuple

/

Entuple data

Receives data on its "inbox" inbox; wraps that data inside a tuple, and outputs that tuple from its "outbox" outbox.

Example Usage

Taking console input and sandwiching it in a tuple between the strings ("You" and "said") and ("just" and now"):

```
Pipeline( ConsoleReader(),
          Entuple(prefix=["You","said"], postfix=["just","now"]),
          ConsoleEchoer(),
        ).run()
```

At runtime:: »> Hello there! ('You', 'said', 'Hello there!', 'just', 'now')

How does it work?

At initialisation specify a list of items to be placed at the front (prefix) and back (postfix) of the tuples that are output.

When an item of data is received at the "inbox" inbox; it is placed inside a tuple, after the prefixes and before the postfixes. It is then immediately sent out of the "outbox" outbox.

For example: if the prefix is [1,2,3] and the postfix is ['a','b'] and the item of data that arrives is 'flurble' then (1,2,3,'flurble','a','b') will be sent to the "outbox" outbox.

If Entuple receives a shutdownMicroprocess message on its "control" inbox, it will pass it on out of the "signal" outbox. The component will then terminate.

Other

Entuple

Entuple([prefix][,postfix]) -> new Entuple component.

Component that takes data received on its "inbox" inbox and wraps it inside of a custom tuple; sending it out of its "outbox" outbox.

Keyword arguments: - prefix -- list of items to go at the front of the tuple (default=[]) - postfix -- list of items to go at the back of the tuple (default=[])

Kamaelia.Apps.Whiteboard.Options

Other

parseOptions

NO DOCS

Kamaelia.Apps.Whiteboard.Painter

Other

Painter

Painter() -> new Painter component.

Kamaelia.Apps.Whiteboard.Palette

Other

buildPalette

NO DOCS

colours

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Kamaelia.Apps.Whiteboard.Play

Other

AlsaPlayer

NO DOCS

SimpleReader

NO DOCS

parseargs

NO DOCS

Kamaelia.Apps.Whiteboard.ProperSurfaceDisplayer

Other

ProperSurfaceDisplayer

NO DOCS

Kamaelia.Apps.Whiteboard.Record

Other

AlsaRecorder

NO DOCS

parseargs

NO DOCS

Kamaelia.Apps.Whiteboard.Router

Other

Router

Router([(rule,dest)][,(rule,dest)]...) -> new Router component.

Component that routes incoming messages to destination outboxes according to whether or not they pass the specified rules.

TwoWaySplitter

NO DOCS

Kamaelia.Apps.Whiteboard.Routers

Other

ConditionalSplitter

NO DOCS

Router

Router([(rule,dest)][,(rule,dest)]...) -> new Router component.

Component that routes incoming messages to destination outboxes according to whether or not they pass the specified rules.

TwoWaySplitter

NO DOCS

Kamaelia.Apps.Whiteboard.SingleShot

Other

OneShot

NO DOCS

Kamaelia.Apps.Whiteboard.SmartBoard

Other

SmartBoard

NO DOCS

usb

PyUSB - Easy USB access in Python

This package exports the following modules and subpackages:

> core - the main USB implementation legacy - the compatibility layer with 0.x version backend - the support for backend implementations. control - USB standard control requests. libloader - helper module for backend library loading.

Since version 1.0, main PyUSB implementation lives in the 'usb.core' module. New applications are encouraged to use it.

Kamaelia.Apps.Whiteboard.TagFiltering

Other

FilterAndTagWrapper

> Returns a component that wraps a target component, tagging all traffic going into its inbox; and filtering outany traffic coming out of its outbox with the same unique id.

FilterAndTagWrapperKeepingTag

NO DOCS

FilterButKeepTag

NO DOCS

FilterTag

NO DOCS

TagAndFilterWrapper

> Returns a component that wraps a target component, tagging all traffic coming from its outbox; and filtering outany traffic coming into its inbox with the same unique id.

TagAndFilterWrapperKeepingTag

NO DOCS

UidTagger

NO DOCS

Kamaelia.Apps.Whiteboard.Tokenisation

Other

EscapedListMarshalling

NO DOCS

lines_to_tokenlists

NO DOCS

substitutions

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

tokenlists_to_lines

NO DOCS

Kamaelia.Apps.Whiteboard.TwoWaySplitter

Other

TwoWaySplitter

NO DOCS

Kamaelia.Apps.Whiteboard.UI

Other

ClearPage

NO DOCS

ClearScribbles

NO DOCS

Delete

NO DOCS

Eraser

NO DOCS

LoadDeck

NO DOCS

LocalPagingControls

NO DOCS

PagingControls

NO DOCS

Quit

NO DOCS

SaveDeck

NO DOCS

Kamaelia.Apps.Whiteboard.Webcam

Other

ProperSurfaceDisplayer

NO DOCS

PygameDisplay

PygameDisplay(...) -> new PygameDisplay component

Use PygameDisplay.getDisplayService(...) in preference as it returns an existing instance, or automatically creates a new one.

Or create your own and register it with setDisplayService(...)

Keyword arguments (all optional):

- width -- pixels width (default=800)
- height -- pixels height (default=600)
- background_colour -- (r,g,b) background colour (default=(255,255,255))
- fullscreen -- set to True to start up fullscreen, not windowed (default=False)

VideoCaptureSource

NO DOCS

WaitComplete

WaitComplete(generator) -> new WaitComplete object.

Message to ask the scheduler to temporarily suspect this microprocess and run a new one instead based on the generator provided; resuming the original when the new one completes.

Use within a microprocess by yielding one back to the scheduler.

Arguments:

- the generator to be run as the separate microprocess

WebcamManager

NO DOCS

component

Base class for an Axon component. Subclass to make your own.

A simple example:

```
class IncrementByN(Axon.Component.component):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
```

```
while self.dataReady("inbox"):
    value = self.recv("inbox")
    value = value + self.n
    self.send(value,"outbox")

if not self.anyReady():
    self.pause()

yield 1
```

producerFinished

producerFinished([caller][,message]) -> new producerFinished ipc message.

Message to indicate that the producer has completed its work and will produce no more output. The receiver may wish to shutdown.

Keyword arguments:

- caller -- Optional. None, or the producer who has finished. Assigned to self.caller
- message -- Optional. None, or a message giving any relevant info. Assigned to self.message

shutdownMicroprocess

shutdownMicroprocess(*microprocesses) -> new shutdownMicroprocess ipc message.

Message used to indicate that the component recieving it should shutdown. Or to indicate to the scheduler a shutdown knockon from a terminating microprocess.

Arguments:

- the microprocesses to be shut down (when used as a knockon)

threadedcomponent

threadedcomponent([queuelengths]) -> new threadedcomponent

Base class for a version of an Axon component that runs main() in a separate thread (meaning it can, for example, block). Subclass to make your own.

Internal queues buffer data between the thread and the Axon inboxes and outboxes of the component. Set the default queue length at initialisation (default=1000).

A simple example:

```
class IncrementByN(Axon.ThreadedComponent.threadedcomponent):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
                value = value + self.n
                self.send(value,"outbox")

            if not self.anyReady():
                self.pause()
```

Kamaelia.Audio.Filtering

Components

LPF

Low pass butterworth filter for 8Hz data. One pole, -3dB at 2KHz

Other

convert

NO DOCS

convertback

NO DOCS

Kamaelia.Audio.RawAudioMixer

Multi-source Raw Audio Mixer

A component that mixes raw audio data from an unknown number of sources, that can change at any time. Audio data from each source is buffered until a minimum threshold amount, before it is included in the mix. The mixing operation is a simple addition. Values are not scaled down.

Example Usage

Mixing up to 3 sources of audio (sometimes a source is active, sometimes it isn't):

```
Graphline(
    MIXER = RawAudioMixer( sample_rate=8000,
                           channels=1,
                           format="S16_LE",
                           readThreshold=1.0,
                           bufferingLimit=2.0,
                           readInterval=0.1),
                         ),
    A = pipeline( SometimesOn_RawAudioSource(), Entuple(prefix="A") ),
    B = pipeline( SometimesOn_RawAudioSource(), Entuple(prefix="B") ),
    C = pipeline( SometimesOn_RawAudioSource(), Entuple(prefix="C") ),

    OUTPUT = RawSoundOutput( sample_rate=8000,
                             channels=1
                             format="S16_LE",
                           ),
           linkages = {
               (A, "outbox") : (MIXER, "inbox"),
               (B, "outbox") : (MIXER, "inbox"),
               (C, "outbox") : (MIXER, "inbox"),

               (MIXER, "outbox") : (OUTPUT, "inbox"),
           },
         ).run()
```

Each source is buffered for 1 second before it is output. If more than 2 seconds of audio are buffered, then samples are dropped.

How does it work?

Send (id, raw-audio) tuples to RawAudioMixer's inbox. Where 'id' is any value that uniquely distinguishes each source of audio.

RawAudioMixer buffers each source of audio, and mixes them together additively,

outputting the resulting stream of audio data.

Constructor arguments:

- sample_rate, channels, format The format of audio to be mixed. The only format understood at the moment is "S16_LE"
- readThreshold number of seconds of audio that will be buffered before RawAudioMixer starts mixing it into its output.
- bufferingLimit maximum number of seconds of audio that will be buffered. If more piles up then some audio will be lost.
- readInterval number of seconds between each time RawAudioMixer outputs a chunk of audio data.

RawAudioMixer buffers each source of audio separately. If the amount of audio in any buffer exceeds the 'buffering limit' then the oldest samples buffered will be lost.

When one or more buffered sources reaches the 'read threshold' then they are mixed together and output. How often audio is output is determined by setting the 'read Interval'.

Mixing is done additively and is *not* scaled down (ie. it is a sum() function, not an average()). Therefore, ensure that the sum of the sources being mixed does not exceed the range of values that samples can take.

Why the buffering, thresholds, and read intervals? It is done this way so that RawAudioMixer can mix without needing to know what sources of audio there are, and whether they are running or stopped. It also enables RawAudioMixer to cope with audio data arriving from different sources at different times.

You may introduce new audio sources at any time - simply send audio data tagged with a new, unique identifier.

You may stop an audio source at any time too - simply stop sending audio data. The existing buffered data will be output, until there is not left.

If there is not enough audio in any of the buffers (or perhaps there are no sources of audio) then RawAudioMixer will not output anything, not even 'silence'.

If a shutdownMicroprocess or producerFinished message is received on this component's "control" inbox this component will cease reading in data from any audio sources. If it is currently outputting audio from any of its buffers, it will continue to do so until these are empty. The component will then forward on the shutdown message it was sent, out of its "signal" outbox and immediately terminate.

TODO:

- Needs a timeout mechanism to discard very old data (otherwise this is effectively a memory leak!)

– If an audio source sends less than the readThreshold amount of audio data, then stops; then this data never gets flushed out.

Components

RawAudioMixer

RawAudioMixer([sample_rate][,channels][,format][,readThreshold][,bufferingLimit][,readInterval]) -> new RawAudioMixer component.

Mixes raw audio data from an unknown number of sources, that can change at any time. Audio data from each source is buffered until a minimum threshold amount, before it is included in the mix. The mixing operation is a simple addition. Values are not scaled down.

Send (uniqueSourceIdentifier, audioData) tuples to the "inbox" inbox and mixed audio data will be sent out of the "outbox" outbox.

Keyword arguments:

- sample_rate -- The sample rate of the audio in Hz (default=8000)
- channels -- Number of channels in the audio (default=1)
- format -- Sample format of the audio (default="S16_LE")
- readThreshold -- Duration to buffer audio before it starts being used in seconds (default=1.0)
- bufferingLimit -- Maximum buffer size for each audio source in seconds (default=2.0)
- readInterval -- Time between each output chunk in seconds (default=0.1)

Other

AudioBuffer

AudioBuffer(activationThreshold, sizeLimit) -> new AudioBuffer component.

Doesn't 'activate' until threshold amount of data arrives. Until it does, attempts to read data will just return nothing.

Keyword arguments:

-- activationThreshold - Point at which the buffer is deemed activated
-- sizeLimit - Filling the buffer beyond this causes samples to be dropped

Kamaelia.Automata.Behaviours

Simple behaviours

A collection of components that send to their "outbox" outbox, values according to simple behaviours - such as constant value, bouncing, looping etc.

Example Usage

Generate values that bounce up and down between 0 and 1 in steps of 0.05:

```
bouncingFloat(scale_speed=0.05*10)
```

Generate (x,y) coordinates, starting at (50,50) that bounce within a 200x100 box with a 10 unit inside margin:

```
cartesianPingPong(point=(50,50), width=200, height=100, border=10)
```

Generate the angles for the seconds hand on an analog watch:

```
loopingCounter(increment=360/60, modulo=360)
```

Constantly generate the number 7:

```
continuousIdentity(original=7)
```

Constantly generate the string "hello":

```
continuousIdentity(original="hello")
```

Constantly generate the value 0:

```
continuousZero()
```

Constantly generate the value 1:

```
continuousOne()
```

More detail

All components start emitting values as soon as they are activated. They then emit values as fast as they can (there is no throttling/rate control).

All components will terminate if they receive the string "shutdown" on their "control" inbox. They also then send "shutdown" to their "signal" outbox.

All components will pause and stop emitting values if they receive the string "pause" on their "control" inbox. They will resume from where they left off if they receive the string "unpause" on the same inbox.

Components

bouncingFloat

bouncingFloat(scale_speed) -> new bouncingFloat component

A component that emits a value that constantly bounces between 0 and 1.

scale_speed scales the rate at which the value changes. 1.0 = tenths, 0.5 = twentieths, etc.

cartesianPingPong

cartesianPingPong(point,width,height,border) -> new cartesianPing-Pong component

A component that emits (x,y) values that bounce around within the specified bounds.

Keyword arguments:

- point -- starting (x,y) coordinates
- width, height -- bounds of the area
- border -- distance in from bounds at which bouncing happens

loopingCounter

loopingCounter(increment[,modulo]) -> new loopingCounter component

Emits an always incrementing value, that wraps back to zero when it reaches the specified limit.

Keyword arguments: - increment -- increment step size - modulo -- counter wrap back to zero before reaching this value (default=360)

continuousIdentity

continuousIdentity(original) -> new continuousIdentity component

A component that constantly emits the original value.

continuousZero

continuousZero() -> new continuousZero component

A component that constantly emits the value 0.

continuousOne

continuousOne() -> new continuousOne component

A component that constantly emits the value 1.

Other

send_one_component

Base class for an Axon component. Subclass to make your own.

A simple example:

```
class IncrementByN(Axon.Component.component):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
                value = value + self.n
                self.send(value,"outbox")

            if not self.anyReady():
                self.pause()

            yield 1
```

Kamaelia.BaseIPC

Base IPC class. Subclass it to create your own IPC classes.

When doing so, make sure you set the following:

- Its doc string, so a string explanation can be generated for an instance of your subclass.
- 'Parameters' class attribute to a list of named parameters you accept at creation, prefixing optional parameters with "?", e.g. "?depth"

For example

A custom IPC class to report a theft taking place! :

```
class Theft(Kamaelia.BaseIPC.IPC):
    """Something has been stolen!"""

    Parameters = ["?who","what"]
```

So what happens when we use it? :

```
>>> ipc = Theft(who="Sam", what="sweeties")
>>> ipc.__doc__
'Something has been stolen!'
>>> ipc.who
'Sam'
>>> ipc.what
'sweeties'
```

Other

IPC

explanation %(foo)s did %(bar)s

Kamaelia.Chassis.Carousel

Component Carousel Chassis

This component lets you create and wire up another component. You can then swap it for a new one by sending it a message. The message contents is used by a factory function to create the new replacement component.

The component that is created is a child contained within Carousel. Wire up to Carousel's "inbox" inbox and "outbox" outbox to send and receive messages from the child.

Example Usage

A reusable file reader:

```
def makeFileReader(filename):
    return ReadFileAdapter(filename = filename, ...other args... )

reusableFileReader = Carousel(componentFactory = makeFileReader)
```

Whenever you send a filename to the "next" inbox of the reusableFileReader component, it will read that file. You can do this as many times as you wish. The data read from the file comes out of the carousel's outbox.

Putting this re-usable file reader to use: the following simple example lets the user enter the names of files to read:

```
Graphline(
   FILENAME_INPUT = ConsoleReader(prompt="enter a filename>"),
   FILE_READER = Carousel(componentFactory = makeFileReader),
   OUTPUT = ConsoleEchoer(),
   linkages = {
       ("FILENAME_INPUT", "outbox") : ("FILE_READER", "next"),
       ("FILE_READER", "outbox") : ("OUTPUT", "inbox"),
   }).run()
```

The user input causes the Carousel to replace the current file reader component (if it has not already terminated) with a new one. The output from this file reader is sent straight back to the console.

Why is this useful?

This chassis component is for making a carousel of components. It gets its name from a broadcast carousel - where a programme (or set of programmes) is broadcast one after another, often on a loop. Alternatively, think of public information screens which display a looping carousel of slides of information.

You gain reusability from things that are not directly reusable and normally come to a halt. For example, make a carousel of file reader components, and you can read from more than one file, one after another. The carousel will make a new file reader component every time you make new request.

The Carousel automatically sends a "NEXT" message when a component finishes, to prompt you make a new request.

How does it work?

The carousel chassis creates and encapsulates (as a child) the component you want it to, and lets it get on with it.

Anything sent to the carousel's "inbox" inbox is passed onto the child component. Anything the child sends out appears at the carousel's "outbox" outbox.

If the child terminates, then the carousel unwires it and sends a "NEXT" message out of its "requestNext" outbox (unless of course it has been told to shutdown).

Another component, such as a Chooser, can respond to this message by sending the new set of arguments (for the factory function) to the carousel's "next" inbox. The carousel then uses your factory function to create a new child component. This way, a sequence of operations can be automatically chained together.

If the argument source needs to receive a "NEXT" message before sending its first set of arguments, then set the argument make1stRequest=True when creating the carousel.

You can actually send new orders to the "next" inbox at any time, not just in response to requests from the Carousel. The carousel will immediately ask that

child to terminate; then as soon as it has done so, it will create the new one and wire it in in its place.

If Carousel receives an Axon.Ipc.producerFinished message on its "control" inbox then it will finish handling any pending messages on its "next" inbox (in the way described above) then when there are none left, it will ask the child component to shut down by sending on the producerFinished message to the child. As soon as the child has terminated, the Carousel will terminate and send on the producerFinished message out of its own "signal" outbox.

If Carousel receives an Axon.Ipc.shutdownMicroprocess message on its "control" inbox then it will immediately send it on to its child component to ask it to terminate. As soon as the child has termianted, the Carousel will terminate and send on the shutdownMicroprocess message out of its own "signal" outbox.

Of course, if the Carousel has no child at the time either shutdown request is received, it will immediately terminate and send on the message out of its "signal" outbox.

Components

Carousel

Carousel(componentFactory,[make1stRequest]) -> new Carousel component

Create a Carousel component that makes child components one at a time (in carousel fashion) using the supplied factory function.

Keyword arguments:

- componentFactory -- function that takes a single argument and returns a component
- make1stRequest -- if True, Carousel will send an initial "NEXT" request. (default=False)

Kamaelia.Chassis.ConnectedServer

Connected Servers

These 'chassis' style components are used to implementing connected servers. The most common example of this is a server which runs on top of the TCP. Examples include: a web server, email server, imap server, game protocol server, etc.

At present, there are two variants of this: *ServerCore* and *SimpleServer* (You are generally recommended to use ServerCore)

Both of these revolve around building TCP based servers. They handle the mechanics of creating the listening component, and when new connections

come in, creating instances of your protocol handler components to handle the connections.

As a result, the primary arguments are the port to listen on and a function call or class name that when called returns a component for handling this connection.

Your protocol handler then receives data from a specific client on its inbox "inbox" and sends data to that same client on its outbox "outbox".

ServerCore passes additional information about the connection to the function that creates the protocol handler. You are not required to do anything with that information if you don't need to.

Aside from that, ServerCore & SimpleServer are used in the same way. (ServerCore is just an extension, and rationalisation of the older simple server code).

There is more information here: http://www.kamaelia.org/Cookbook/TCPSystems

Example Usage

A server using a simple echo protocol, that just echoes back anything sent by the client. Becase the protocol has no need to know any details of the connection, the SimpleServer component is used:

```
import Axon
from Kamaelia.Chassis.ConnectedServer import SimpleServer

PORTNUMBER = 12345
class EchoProtocol(Axon.Component.component):

    def main(self):
        while not self.shutdown():
            yield 1
            if self.dataReady("inbox"):
                data = self.recv("inbox")
                self.send(data, "outbox")

    def shutdown(self):
        if self.dataReady("control"):
            msg = self.recv("control")
            return isinstance(msg, Axon.Ipc.producerFinished)

simpleServer = SimpleServer( protocol = EchoProtocol, port = PORTNUMBER )
simpleServer.run()
```

Try connecting to this server using the telnet command, and it will echo back to you every character you type.

A more complex server might need to inform the protocol of the IP address and port of the client that connects, or the ip address and port at this (the server end) to which the client has connected. For this, ServerCore is used:

```
import Axon
from Axon.Ipc import shutdownMicroprocess
from Kamaelia.Chassis.ConnectedServer import ServerCore

PORTNUMBER = 12345
class CleverEchoProtocol(Axon.Component.component):

    def main(self):
        welcomeMessage =                 "Welcome! You have connected to %s on port %d from

        self.send(welcomeMessage, "outbox")
        while not self.shutdown():
            yield 1
            if self.dataReady("inbox"):
                data = self.recv("inbox")
                self.send(data, "outbox")

    def shutdown(self):
        if self.dataReady("control"):
            msg = self.recv("control")
            return isinstance(msg, Axon.Ipc.producerFinished)

myServer = ServerCore( protocol = CleverEchoProtocol, port = PORTNUMBER )
myServer.run()
```

Example output when telnetting to this more complex server, assuming both server and telnet session are running on the same host, and the server is listening to port number 8081:

```
$ telnet localhost 8081
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome! You have connected to 127.0.0.1 on port 8081 from 127.0.0.1 on port 47316
```

Why is this useful?

Provides a framework for creating generic protocol handlers to deal with information coming in on a single port (and a single port only). This however covers a large array of server types.

A protocol handler is simply a component that can receive and send data (as byte strings) in a particular format and with a particular behaviour - ie. conforming to a particular protocol.

Provide this chassis with a factory function to create a component to handle the protocol. Whenever a client connects a handler component will then be created to handle communications with that client.

Data received from the client will be sent to the protocol handler component's "inbox" inbox. To send data back to the client, the protocol handler component should send it out of its "outbox" outbox.

For the SingleServer component, the factory function takes no arguments. It should simply return the component that will be used to handle the protocol, for example:

```
def makeNewProtocolHandler():
    return MyProtocolComponent()
```

For the ServerCore component, the factory function must accept the following arguments (with these names):

- peer -- the address of the remote endpoint (the client's address)
- peerport -- the port number of the remote endpoint (the port number from which the client connection originated)
- localip -- the address of the local endpoint (this end of the connection)
- localport -- the port number of the local endpoint (this end of the connection)

For example:

```
def makeNewProtocolHandler(peer, peerport, localip, localport):
    print "Debugging: client at address "+peer+" on port "+str(peerport)
    print " ... has connected to address "+localip+" on port "+str(localport)
    return MyProtocolComponent()
```

Do not activate the component. SingleServer or ServerCore will do this once the component is wired up.

Writing a protocol handler

A protocol handler component should use its standard inboxes ("inbox" and "control") and outboxes ("outbox" and "signal") to communicate with client it is connected to.

- Bytes received from the client will be sent to the "inbox" inbox as a string.
- Send a string out of the "outbox" outbox to send bytes back to the client.

If the connection is closed, a Kamaelia.IPC.socketShutdown message will arrive at the protocol handler's "control" inbox. If this happens then the connection should be assumed to have already closed. Any more messages sent will not be sent to the client. The protocol handler should react by terminating as soon as possible.

To cause the connection to close, send a producerFinished or shutdownMicroprocess message out of the protocol handler's "signal" outbox. As soon as this has been done, it can be assumed that the connection will be closed as soon as is practical. The protocol handler will probably also want to terminate at this point.

How does it work?

SimpleServer is based on ServerCore. It simply contains a wrapper around the protocol handler function that throws away the connection information instead of passing it in as arguments.

At initialisation the component registers a TCPServer component to listen for new connections on the specified port.

You supply a factory function that takes no arguments and returns a new protocol handler component.

When it receives a 'newCSA' message from the TCPServer (via the "_socketactivity" inbox), the factory function is called to create a new protocol handler. The protocol handler's "inbox" inbox and "outbox" outbox are wired to the ConnectedSocketAdapter (CSA) component handling that socket connection, so it can receive and send data.

If a 'shutdownCSA' message is received (via "_socketactivity") then a Kamaelia.IPC.socketShutdown message is sent to the protocol handler's "control" inbox, and both it and the CSA are unwired.

This component does not terminate. It ignores any messages sent to its "control" inbox.

In practice, this component provides no external connectors for your use.

History

This code is based on the code used for testing the Internet Connection abstraction layer.

To do

This component currently lacks an inbox and corresponding code to allow it to be shut down (in a controlled fashion). Needs a "control" inbox that responds to shutdownMicroprocess messages.

Components

ServerCore

ServerCore(protocol[,port]) -> new Simple protocol server component

A simple single port, multiple connection server, that instantiates a protocol handler component to handle each connection. The function that creates the protocol must access arguments providing information about the connection.

Keyword arguments:

- protocol -- function that returns a protocol handler component
- port -- Port number to listen on for connections (default=1601)

SimpleServer

SimpleServer(protocol[,port]) -> new Simple protocol server component

A simple single port, multiple connection server, that instantiates a protocol handler component to handle each connection.

Keyword arguments:

- protocol -- function that returns a protocol handler component
- port -- Port number to listen on for connections (default=1601)

FastRestartServer

NO DOCS

Other

MoreComplexServer

ServerCore(protocol[,port]) -> new Simple protocol server component

A simple single port, multiple connection server, that instantiates a protocol handler component to handle each connection. The function that creates the protocol must access arguments providing information about the connection.

Keyword arguments:

- protocol -- function that returns a protocol handler component
- port -- Port number to listen on for connections (default=1601)

Kamaelia.Chassis.Graphline

Wiring up components in a topology

The Graphline component wires up a set of components and encapsulates them as a single component. They are wired up to each other using the 'graph' of linkages that you specify.

Example Usage

Joining a PromtedFileReader and a rate control component to make a file reader that reads at a given rate:

```
return Graphline(RC  = ByteRate_RequestControl(**rateargs),
                 RFA = PromptedFileReader(filename, readmode),
                 linkages = { ("RC",  "outbox")  : ("RFA", "inbox"),
                              ("RFA", "outbox")  : ("self", "outbox"),
                              ("RFA", "signal")  : ("RC",  "control"),
                              ("RC",  "signal")  : ("self", "signal"),
                              ("self", "control") : ("RFA", "control")
                              }
```

The references to 'self' create linkages that passes through a named inbox on the graphline to a named inbox of one of the child components. Similarly a child's outbox is pass-through to a named outbox on the graphline.

Shutdown Examples

In this example:

- Pinger is a component that sends the messages from "tosend" after with a brief delay between messages. It sends the messages out of the stated outbox.
- Waiter is a component that starts up, and then waits for any message sent to its inbox "control"
- Whinger is a component that complains that it is running periodically, but will shutdown if it receives any message on its inbox "control"

As a result, this example creates 3 components inside a graphline that wait for shutdown. The Pinger sends a message, which is duplicated to all the subcomponents, at which point in time, they shutdown, causing the system to shutdown:

```
Pipeline(
    Pinger(tosend=[Axon.Ipc.producerFinished()],box="signal"),
    Graphline(
        TO_SHUTDOWN1 = Waiter(),
        TO_SHUTDOWN2 = Waiter(),
        TO_SHUTDOWN3 = Waiter(),
        linkages = {}
    ),
    Whinger(),
).run()
```

Note: the shutdown message propogates all the way through the system to the whinger, which then also shuts down.

Full code for this is in ./Examples/UsingChassis/Graphline/DemoShutdown.py

You can also still have shutdown links between components. If you do, then the Graphline doesn't interfere with them:

```
Pipeline(
    Pinger(tosend=[Axon.Ipc.producerFinished()],box="signal"),
    Graphline(
        TO_SHUTDOWN1 = Waiter(),
        TO_SHUTDOWN2 = Waiter(),
        TO_SHUTDOWN3 = Waiter(),
        linkages = {
            ("TO_SHUTDOWN1","signal"):("TO_SHUTDOWN2","control"),
            ("TO_SHUTDOWN2","signal"):("TO_SHUTDOWN3","control"),
        }
    ),
    Whinger(),
).run()
```

Full code for this is in ./Examples/UsingChassis/Graphline/LinkedShutdown.py

How does it work?

A Graphline component gives you a way of wiring up a system of components and then encapsulating th ewhole as a single component, with its own inboxes and outboxes.

The components you specify are registered as children of the Graphline component. When you activate the component, all the child components are activated, and the linkages you specified are created between them.

When specifying linkages, the component 'name' is the string version of the argument name you used to refer to the component. In the example above, the components are therefore referred to as "RC" and "RFA".

If the name you specify is not one of the components you specify, then it is assumed you must be referring to the Graphline component itself. In the above example, "self" is used to make this clear. This gives you a way of passing data in and out of the system of components you have specified.

In these cases, it is assumed you wish to create a pass-through linkage - you want the Graphline component to forward the named inbox to a child's inbox, or to forward a child's outbox to a named outbox of the Graphline. For example:

```
Graphline( child = MyComponent(...),
           linkages = { ...
                        ("self", "inbox") : ("child", "bar"),
                        ... }
         )
```

... is interpreted as meaning you want to forward the "inbox" inbox of the Graphline to the "bar" inbox of the component referred to as "child". Similarly:

```
Graphline( child = MyComponent(...),
           linkages = { ...
                        ("child", "fwibble") : ("self", "outbox"),
                        ... }
         )
```

...is interpreted as wishing to forward the "fwibble" outbox of the component referred to as "child" to the "outbox" outbox of the Graphline component.

Any inbox or outbox you name on the Graphline component is created if it does not already exist. For example, you might want the Graphline to have a "video" and an "audio" inbox:

```
Graphline( videoHandler = MyVideoComponent(),
           audioHandler = MyAudioComponent(),
           linkages = { ...
                        ("self", "video") : ("videoHandler", "inbox"),
                        ("self", "audio") : ("audioHandler", "inbox"),
                        ...
                      }
         )
```

The Graphline component will always have inboxes "inbox" and "control" and outboxes "outbox" and "signal", even if you do not specify any linkages to them.

During runtime, the Graphline component monitors the child components. It will terminate if, and only if, *all* the child components have also terminated.

NOTE that if your child components create additional components themselves, the Graphline component will not know about them. It only monitors the components it was originally told about.

Graphline does not GENERALLY intercept any of its inboxes or outboxes. It ignores whatever traffic flows through them. If you have specified linkages from them to components inside the graphline, then the data automatically flows to/from them as you specified.

Shutdown Handling

There is however an exception: shutdown handling, where the difference is light touch, which is this:

```
while not self.childrenDone():
     always pass on messages from our control to appropriate sub-component's control
     if message is shutdown, set shutdown flag

# then after loop

if no component-has-linkage-to-graphline's signal
     if shutdown flag set:
```

```
        pass on shutdownMicroprocess
    else:
        pass on producerFinished
```

If the user has wired up the graphline's control box to pass through to one of their components, then that request is honoured, and the user then becomes wholly responsible for shutdown.

Components

Graphline

Graphline(linkages,**components) -> new Graphline component

Encapsulates the specified set of components and wires them up with the specified linkages.

Keyword arguments:

- linkages -- dictionary mapping ("componentname","boxname") to ("componentname","boxname")
- components -- dictionary mapping names to component instances (default is nothing)

Other

component

Base class for an Axon component. Subclass to make your own.

A simple example:

```
class IncrementByN(Axon.Component.component):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
```

```
            value = value + self.n
            self.send(value,"outbox")

        if not self.anyReady():
            self.pause()

        yield 1
```

Kamaelia.Chassis.PAR

Running components in parallel conveniently, shared output

The PAR component activates all the subcomponents listed to run in parallel - hence the name - from Occam. Shutdown messages are passed to all subcomponents. Their shutdown messages propogate out the PAR component's signal outbox.

Future work will include the ability to define input policies regarding what to do with messages from the main inbox. (not yet implemented)

For more complex topologies, see the Graphline component.

Example Usage

One example initially. This:

```
Pipeline(
   PAR(
       Button(caption="Next", msg="NEXT", position=(72,8)),
       Button(caption="Previous", msg="PREV",position=(8,8)),
       Button(caption="First", msg="FIRST",position=(256,8)),
       Button(caption="Last", msg="LAST",position=(320,8)),
   ),
   Chooser(items = files),
   Image(size=(800,600), position=(8,48)),
).run()
```

Is equivalent to this:

```
Graphline(
     NEXT = Button(caption="Next", msg="NEXT", position=(72,8)),
     PREVIOUS = Button(caption="Previous", msg="PREV",position=(8,8)),
     FIRST = Button(caption="First", msg="FIRST",position=(256,8)),
     LAST = Button(caption="Last", msg="LAST",position=(320,8)),

     CHOOSER = Chooser(items = files),
     IMAGE = Image(size=(800,600), position=(8,48)),
     linkages = {
```

```
        ("NEXT","outbox") : ("CHOOSER","inbox"),
        ("PREVIOUS","outbox") : ("CHOOSER","inbox"),
        ("FIRST","outbox") : ("CHOOSER","inbox"),
        ("LAST","outbox") : ("CHOOSER","inbox"),

        ("CHOOSER","outbox") : ("IMAGE","inbox"),
     }
).run()
```

Shutdown Examples

This component is well behaved with regard to shutdown. It has the following behaviour: if it recieves a shutdownMicroprocess message on it's control box, it forwards a copy of this to all the subcomponents. When they exit, this component exits. This enables it to be used to shutdown entire systems cleanly. For example, the above example using PAR can be shutdown after 15 seconds using this code:

```
class timedShutdown(Axon.ThreadedComponent.threadedcomponent):
    TTL = 1
    def main(self):
        time.sleep(self.TTL)
        self.send(Axon.Ipc.shutdownMicroprocess(), "signal")

Pipeline(
    timedShutdown(TTL=5),
    Pipeline(
        PAR(
            Button(caption="Next",     msg="NEXT", position=(72,8)),
            Button(caption="Previous", msg="PREV", position=(8,8)),
            Button(caption="First",    msg="FIRST" ,position=(256,8)),
            Button(caption="Last",     msg="LAST", position=(320,8)),
        ),
        Chooser(items = files),
        Image(size=(800,600), position=(8,48), maxpect=(800,600)),
    ),
).run()
```

For a larger example, the following is also a well behaved with regard to shutdown from PAR components - despite containing a ticker, buttons, 2 different video playback subsystems, two "talker text" panes, and a presentation tool:

```
Pipeline(
        timedShutdown(TTL=15),
        PAR(
            Pipeline(
                     ReadFileAdaptor(file, readmode="bitrate",
```

```
                                    bitrate = 300000*8/5),
                        DiracDecoder(),
                        MessageRateLimit(framerate),
                        VideoOverlay(position=(260,48), size=(200,300)),
                ),
                Pipeline( ReadFileAdaptor(file, readmode="bitrate", bitrate = 2280960*8),
                          DiracDecoder(),
                          ToRGB_interleaved(),
                          VideoSurface(size=(200, 300), position=(600,48)),
                ),
                Pipeline(
                    PAR(
                        Button(caption="Next",     msg="NEXT", position=(72,8)),
                        Button(caption="Previous", msg="PREV", position=(8,8)),
                        Button(caption="First",    msg="FIRST" ,position=(256,8)),
                        Button(caption="Last",     msg="LAST", position=(320,8)),
                    ),
                    Chooser(items = files),
                    Image(size=(200,300), position=(8,48), maxpect=(200,300)),
                ),
                Pipeline(
                    Textbox(size=(200,300), position=(8,360)),
                    TextDisplayer(size=(200,300), position=(228,360)),
                ),
                Ticker(size=(200,300), position=(450,360)),
            ),
).run()
```

How does it work?

As present this is a relatively simple container component. It links all the outboxes from all it's subcomponents to it's own outboxes. It then activates them all, and pauses. When awoken by a message on the control inbox, this is forwared to the child components in order to shutdown.

Policies

To be written. The idea behind policies is to allow someone to override the default behaviour regarding inbox data. This potentially enables the creation of things like threadpools, splitters, and general workers.

Components

PAR

PAR(inputpolicy=None, outputpolicy=None, *components) -> new PAR component

Activates all the components contained inside in parallel (Hence the name - from Occam).

Inputs to inboxes can be controlled by passing in a policy. The default policy is this:

```
messages to "control" are forwarded to all children

if a control is a shutdownMicroprocess, shutdown

when all children exit, exit.

messages to "inbox" are forwarded to all components by default.
```

See the module docs on writing a policy function.

Outputs from all outboxes are sent to the graphline's corresponding outbox. At present supported outboxes replicated are: "outbox", and "signal".

For more complex wiring/policies you probably ought to use a Graphline component.

Keyword arguments:

- policy -- policy function regarding input mapping.
- components -- list of components to be activated.

Kamaelia.Chassis.Pipeline

Wiring up components in a Pipeline

The Pipeline component wires up a set of components in a linear chain (a Pipeline) and encapsulates them as a single component.

Example Usage

A simple pipeline of 4 components:

```
Pipeline(MyDataSource(...),
         MyFirstStageOfProcessing(...),
         MySecondStageOfProcessing(...),
         MyDestination(...),
        ).run()
```

How does it work?

A Pipeline component gives you a way of wiring up a system of components in a chain and then encapsulating the whole as a single component. The inboxes of this component pass through to the inboxes of the first component in the

Pipeline, and the outboxes of the last component pass through to the outboxes of the Pipeline component.

The components you specify are registered as children of the Pipeline component. When Pipeline is activate, all children are wired up and activated.

For the components in the Pipeline, "outbox" outboxes are wired to "inbox" inboxes, and "signal" outboxes are wired to "control" inboxes. They are wired up in the order in which you specify them - data will flow through the chain from first component to last.

The "inbox" and "control" inboxes of the Pipeline component are wired to pass-through to the "inbox" and "control" inboxes (respectively) of the first component in the Pipeline chain.

The "outbox" and "signal" outboxes of the last component in the Pipeline chain are wired to pass-through to the "outbox" and "signal" outboxes (respectively) of the Pipeline component.

During runtime, the Pipeline component monitors the child components. It will terminate if, and only if, *all* the child components have also terminated.

NOTE that if your child components create additional components themselves, the Pipeline component will not know about them. It only monitors the components it was originally told about.

Pipeline does not intercept any of its inboxes or outboxes. It ignores whatever traffic flows through them.

Components

Pipeline

Pipeline(*components) -> new Pipeline component.

Encapsulates the specified set of components and wires them up in a chain (a Pipeline) in the order you provided them.

Keyword arguments:

- components -- the components you want, in the order you want them wired up

Other

component

Base class for an Axon component. Subclass to make your own.

A simple example:

```
class IncrementByN(Axon.Component.component):

    Inboxes = { "inbox" : "Send numbers here",
                "control" : "NOT USED",
              }
    Outboxes = { "outbox" : "Incremented numbers come out here",
                 "signal" : "NOT USED",
               }

    def __init__(self, N):
        super(IncrementByN,self).__init__()
        self.n = N

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                value = self.recv("inbox")
                value = value + self.n
                self.send(value,"outbox")

            if not self.anyReady():
                self.pause()

            yield 1
```

pipeline

NO DOCS

Kamaelia.Chassis.Prefab

Pre-fabrication function chassis

This is a collection of functions that link up components standardised ways.

They take a collection of components as arguments, and then wire them up in a particular fashion. These components are children inside the prefab.

JoinChooserToCarousel

Automated "what arguments should I use for my next reusable component?"

Take a Carousel that makes components on request from a set of arguments. Take a Chooser that responds to request for the 'next' set of arguments.

This pre-fab is a component that wires them together. When the Carousel requests the arguments for the next component, the Chooser can respond with them.

For example, you could wire up a playlist to something reusable that reads files at a given rate. Alternatively, it could be a list of videos or pictures passed to a reusable media viewer. It could even be a list of shell commands passed to a reusable shell/system caller.

Example Usage Reading from a playlist of files:

```
def makeFileReader(filename):
    return ReadFileAdapter(filename = filename, ...other args... )

reusableFileReader = Carousel componentFactory = makeFileReader)
playlist = Chooser(["file1","file2" ... ])

playlistreader = JoinChooserToCarousel(playlist, reusableFileReader)
playlistreader.activate()
```

More detail Any component can be used that has the expected inboxes and outboxes, and which behaves in a relevant manner.

Chooser must have inboxes "inbox" and "control" and outboxes "outbox" and "signal".

Carousel must have inboxes "inbox", "control" and "next" and outboxes "outbox", "signal" and "requestNext".

The Chooser and Carousel are encapsulated within this prefab component as children.

"inbox", "outbox" and "signal" of the Carousel are "inbox", "outbox" and "signal" of this prefab.

Messages sent to this prefab's "control" inbox go to the Chooser, which should then pass it onto the Carousel, allowing shutdown.

To do This prefab needs a better name - it currently describes its design, not what its for.

Other

JoinChooserToCarousel

> JoinChooserToCarousel(chooser, carousel) -> component containing both wired up
>
> Wires up a Chooser and a Carousel, so when the carousel requests the next item, the Chooser supplies it.
>
> Keyword arguments:
>
> - chooser -- A Chooser component, or one with similar interfaces

- carousel -- A Carousel component, or one with similar interfaces

Kamaelia.Chassis.Seq

Run components one after the other (in sequence)

A Seq component runs components one after the other in sequence, waiting until one terminates before starting the next.

Strings can also be put in the sequence. They'll be printed to the console

Example Usage

Run several OneShot components running one after the other:

```
Pipeline( Seq( "BEGIN SEQUENCE",
               OneShot("Hello\n"),
               OneShot("Doctor\n"),
               OneShot("Name\n"),
               OneShot("Continue\n"),
               OneShot("Yesterday\n"),
               OneShot("Tomorrow\n"),
               "END SEQUENCE",
             ),
          ConsoleEchoer(),
        ).run()
```

Running this generates the following output:

```
BEGIN SEQUENCE
Hello
Doctor
Name
Continue
Yesterday
Tomorrow
END SEQUENCE
```

Behaviour

Each component in the sequence is activated as a child component and is wired up so that the "inbox" inbox and "outbox" outbox are forwarded to the "inbox" inbox and "outbox" outbox of the Seq component itself.

When the child component terminates it is replaced with the next in the sequence.

If a string is listed instead of a component then it is printed on the console and Seq immediately moves onto the next in the sequence.

Any messages sent out of the child component's "signal" outbox are dropped - this is so that if you Pipeline a Seq component to another, it does not cause it to terminate when the Seq component switches to a new child.

This component ignores any messages sent to its "control" inbox.

When the end of the sequence is reached, a producerFinished() message is sent out of the "signal" outbox and the component terminates.

Components

Seq

Seq(*sequence) -> new Seq component.

Runs a set of components in sequence, one after the other. Their "inbox" inbox and "outbox" outbox are forwarded to the "inbox" inbox and "outbox" outbox of the Seq component.

Keyword arguments:

- *sequence -- Components that will be run, in sequence. Can also include strings that will be output to the console.

Kamaelia.Codec.RawYUVFramer

Raw YUV video data framer

This component takes a raw stream of YUV video data and breaks it into invidual frames. It sends them out one at a time, tagged with relevant data such as the frame size.

Many components that expect uncompressed video require it to be structured into frames in this way, rather than as a raw stream of continuous data. This component fulfills that requirement.

Example Usage

Reading and encoding raw video:

```
imagesize = (352, 288)        # "CIF" size video

Pipeline(ReadFileAdapter("raw352x288video.yuv", ...other args...),
         RawYUVFramer(imagesize),
         DiracEncoder(preset="CIF"),
        ).activate()
```

More Detail

Receives raw yuv video data, as strings on its "inbox" inbox.

Sends out individual frames packaged in a dictionary:

```
{
  "yuv" : (y_data, u_data, v_data),  # a tuple of strings
  "size" : (width, height),          # in pixels
  "pixformat" : "YUV420_planar",     # raw video data format
}
```

The component will terminate if it receives a shutdownMicroprocess or producerFinished message on its "control" inbox. The message is passed on out of the "signal" outbox.

Components

RawYUVFramer

RawYUVFramer(size,pixformat) -> raw yuv video data framing component

Creates a component that frames a raw stream of YUV video data into frames.

Keyword arguments:

- size -- (width,height) size of a video frame in pixels
- pixformat -- raw video data format (default="YUV420_Planar")

Kamaelia.Codec.WAV

Reading and writing simple WAV audio files

Read and write WAV file format audio data using the WAVParser and WAVWriter components, respectively.

Example Usage

Playing a WAV file, where we don't know the format until we play it:

```
from Kamaelia.Audio.PyMedia.Output import Output
from Kamaelia.File.Reading import RateControlledFileReader
from Kamaelia.Chassis.Graphline import Graphline
from Kamaelia.Chassis.Carousel import Carousel

def makeAudioOutput(format_info):
    return Output( sample_rate = format_info['sample_rate'],
                   format      = format_info['sample_format'],
```

```
                  channels    = format_info['channels']
                )

Graphline(
    SRC = RateControlledFileReader("test.wav",readmode="bytes",rate=44100*4),
    WAV = WAVParser(),
    DST = Carousel(makeAudioOutput),
    linkages = {
        ("SRC","outbox") : ("WAV","inbox"),
        ("SRC","signal") : ("WAV","control"),
        ("WAV","outbox") : ("DST","inbox"),
        ("WAV","signal") : ("DST","control"),
        ("WAV","all_meta") : ("DST","next"),
    }
    ).run()
```

Capturing audio and writing it to a WAV file:

```
from Kamaelia.Audio.PyMedia.Input import Input
from Kamaelia.File.Writing import SimpleFileWriter
from Kamaelia.Chassis.Pipeline import Pipeline

Pipeline( Input(sample_rate=44100, channels=2, format="S16_LE"),
          WAVWriter(sample_rate=44100, channels=2, format="S16_LE"),
          SimpleFileWriter("captured_audio.wav"),
        ).run()
```

WAVParser behaviour

Send binary data as strings containing a WAV file to the "inbox" inbox.

As soon as the format of the audio data is determined (from the headers) it is sent out the "all_meta" outbox as a dictionary, for example:

```
{ "sample_format" : "S16_LE",
  "channels"      : 2,
  "sample_rate"   : 44100,
}
```

The individual components are also sent out the "sample_format", "channels" and "sample_rate" outboxes.

The raw audio data from the incoming WAV data is sent out of the "outbox" outbox, until the end of the WAV file is reached. If the WAV headers specify an audio size of zero, then it is assumed to be of indefinite length, otherwise the value is assumed to be the actual size, and this component will terminate and send out a producerFinished() message when it thinks it has reached the end.

This component supports sending the raw audio data to a size limited inbox. If

the size limited inbox is full, this component will pause until it is able to send out the data.

If a producerFinished message is received on the "control" inbox, this component will complete parsing any data pending in its inbox, and finish sending any resulting data to its outbox. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete processing, or sending on any pending data.

WAVWriter behaviour

Initialise this component, specifying the format the audio data will be in.

Send raw audio data (in the format you specified!) as binary strings to the "inbox" inbox, and this component will write it out as WAV file format data out of the "outbox" outbox.

The WAV format headers will immediately be sent out of the "outbox" outbox as soon as this component is initialised and activated (ie. before you even start sending it audio data to write out). The size of the audio data is set to zero as the component has no way of knowing the duration of the audio.

This component supports sending data out of its outbox to a size limited inbox. If the size limited inbox is full, this component will pause until it is able to send out the data.

If a producerFinished message is received on the "control" inbox, this component will complete parsing any data pending in its inbox, and finish sending any resulting data to its outbox. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete processing, or sending on any pending data.

Development history

WAVWriter is based on code by Ryn Lothian developed during summer 2006.

Components

WAVParser

> WAVParser() -> new WAVParser component.
>
> Send WAV format audio file data to its "inbox" inbox, and the raw audio data will be sent out of the "outbox" outbox as binary strings.

The format of the audio data is also sent out of other outboxes as soon as it is determined (before the data starts to flow).

WAVWriter

WAVWriter(channels, sample_format, sample_rate) -> new WAVWriter component.

Send raw audio data as binary strings to the "inbox" inbox and WAV format audio data will be sent out of the "outbox" outbox as binary strings.

Kamaelia.Codec.YUV4MPEG

Parsing and Creation of YUV4MPEG format files

YUV4MPEGToFrame parses YUV4MPEG format data sent to its "inbox" inbox and sends video fram data structures to its "outbox" outbox.

FrameToYUV4MPEG does the reverse - taking frame data structures sent to its "inbox" inbox and outputting YUV4MPEG format data to its "outbox" outbox."

The YUV4MPEG file format is supported by many tools, such as mjpegtools, mplayer/mencoder, and ffmpeg.

Example Usage

Playback a YUV4MPEG format file:

```
Pipeline( RateControlledFileReader("video.yuv4mpeg",readmode="bytes", ...),
          YUV4MPEGToFrame(),
          VideoOverlay()
        ).run()
```

Decode a dirac encoded video file to a YUV4MPEG format file:

```
Pipeline( RateControlledFileReader("video.dirac",readmode="bytes", ...),
          DiracDecoder(),
          FrameToYUV4MPEG(),
          SimpleFileWriter("output.yuv4mpeg")
        ).run()
```

YUV4MPEGToFrame Behaviour

Send binary data as strings containing YUV4MPEG format data to the "inbox" inbox and frame data structures will be sent out of the "outbox" outbox as soon as they are parsed.

See below for a description of the uncompressed frame data structure format.

This component supports sending data out of its outbox to a size limited inbox. If the size limited inbox is full, this component will pause until it is able to send out the data. Data will not be consumed from the inbox if this component is waiting to send to the outbox.

If a producerFinished message is received on the "control" inbox, this component will complete parsing any data pending in its inbox, and finish sending any resulting data to its outbox. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete processing, or sending on any pending data.

FrameToYUV4MPEG Behaviour

Send frame data structures to the "inbox" inbox of this component. YUV4MPEG format binary string data will be sent out of the "outbox" outbox.

See below for a description of the uncompressed frame data structure format.

The header data for the YUV4MPEG file is determined from the first frame.

All frames sent to this component must therefore be in the same pixel format and size, otherwise the output data will not be valid YUV4MPEG.

This component supports sending data out of its outbox to a size limited inbox. If the size limited inbox is full, this component will pause until it is able to send out the data. Data will not be consumed from the inbox if this component is waiting to send to the outbox.

If a producerFinished message is received on the "control" inbox, this component will complete parsing any data pending in its inbox, and finish sending any resulting data to its outbox. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete processing, or sending on any pending data.

UNCOMPRESSED FRAME FORMAT

A frame is a dictionary data structure. It must, at minimum contain the first 3 ("yuv", "size" and "pixformat"):

```
{
  "yuv" : (y_data, u_data, v_data)  # a tuple of strings
  "size" : (width, height)          # in pixels
  "pixformat" :  pixelformat        # format of raw video data
  "frame_rate" : fps                # frames per second
```

```
  "interlaced" : 0 or not 0         # non-zero if the frame is two interlaced fields
  "topfieldfirst" : 0 or not 0      # non-zero the first field comes first in the data
  "pixel_aspect" : fraction         # aspect ratio of pixels
  "sequence_meta" : metadata        # string containing extended metadata
                                    # (no whitespace or control characters)
}
```

All other fields are optional when providing frames to FrameToYUV4MPEG.

YUV4MPEGToFrame only guarantees to fill inthe YUV data itself. All other fields will be filled in if the relevant header data is detected in the file.

The pixel formats recognised (and therefore supported) are:

```
"YUV420_planar"
"YUV411_planar"
"YUV422_planar"
"YUV444_planar"
"YUV4444_planar"
"Y_planar"
```

Components

YUV4MPEGToFrame

YUV4MPEGToFrame() -> new YUV4MPEGToFrame component.

Parses YUV4MPEG format binarydata, sent as strings to its "inbox" inbox and outputs uncompressed video frame data structures to its "outbox" outbox.

FrameToYUV4MPEG

FrameToYUV4MPEG() -> new FrameToYUV4MPEG component.

Parses uncompressed video frame data structures sent to its "inbox" inbox and writes YUV4MPEG format binary data as strings to its "outbox" outbox.

Other

parse_frame_tags

Parses YUV4MPEG frame tags.

parse_seq_tags

Parses YUV4MPEG header tags

Kamaelia.Device.DVB.NowNext

Processing Simplified Now & Next Event Information

These components filter or process simplified events, derived from Event Information Table data containing now and next information.

Convert a parsed EIT table to simplified individual events using the Kamaelia.Devices.DVB.Parse.ParseEventInformationTable.SimplifyEIT component.

NowNextServiceFilter selects information relating to only particular services (channels) in the data.

NowNextProgrammeJunctionDetect detects the point at which one programme ends and another begins - known as the "programme junction". It distinguishes between programme junctions and ammendments to a programme's details.

Example Usage Tuning to a particular broadcast multiplex and detecting when a new programme starts on service 4164, outputting the information about the new programme:

```
frequency = 505833330.0/1000000.0
feparams = {
    "inversion" : dvb3.frontend.INVERSION_AUTO,
    "constellation" : dvb3.frontend.QAM_16,
    "code_rate_HP" : dvb3.frontend.FEC_3_4,
    "code_rate_LP" : dvb3.frontend.FEC_3_4,
}

EIT_PID = 0x12
BBC_ONE = 4164       # the 'service id' on this particular multiplex

Pipeline( DVB_Multiplex(505833330.0/1000000.0, [EIT_PID], feparams),
          DVB_Demuxer({ EIT_PID:["outbox"]}),
          ReassemblePSITables(),
          ParseEventInformationTable_Subset(True,False,False,False), # now and next for this
          SimplifyEIT(),
          NowNextProgrammeJunctionDetect(),
          NowNextServiceFilter(BBC_ONE),
          ConsoleEchoer(),
```

The above code receives the broadcast multiplex, reconstructs and parses the Event Information Table in it, then simplifies it to a stream of events. These events are then filtered and processed.

NowNextServiceFilter NowNextServiceFilter selects information relating to only particular services (channels) in the data.

Behaviour

At initialisation, specify the service id's of the services to be detected as arguments.

Send the parsed and simplified events to this component's "inbox" inbox. Those which match the service id's specified at initialisation will immediately be sent on out of the "outbox" outbox. Those which do not match are discarded.

If a shutdownMicroprocess or producerFinished message is sent to this component's "control" inbox, it will immediately be sent on out of the "signal" outbox. The component will then immediately terminate.

NowNextProgrammeJunctionDetect NowNextProgrammeJunctionDetect detects the point at which one programme ends and another begins - known as the "programme junction". It distinguishes between programme junctions and ammendments to a programme's details.

Behaviour

Send the parsed and simplified events to this component's "inbox" inbox. This component then distinguishes between ammendments to a programme (such as (a change to to how long it is, or its description) and actual programme junctions (the end of one programme and the start of the next).

A single NowNextProgrammeJunctionDetect instance can handle the events for an unlimited number of services concurrently.

When a programme junction is detected, the event describing the programme that has just started is sent out of the "outbox" and "now" outboxes. Any event describing the 'next' programme that will follow it is sent out the "next" outbox.

If the details of a programme have just been ammended (it is not a junction), then the new event information is sent out of the "now_update" outbox if it relates to the current programme on air; or the "next_update" outbox if it relates to the programme that will follow it.

NowNextProgrammeJunctionDetect only handles 'now' and 'next' events. Events for schedule (electronic programme guide) details are ignored.

If a shutdownMicroprocess or producerFinished message is sent to this component's "control" inbox, it will immediately be sent on out of the "signal" outbox. The component will then immediately terminate.

How does it work?

NowNextProgrammeJunction detect keeps a record of the ids of the 'now' and 'next' events each service.

When an event is received, it is looked up in this table. If the event id matches, then it is assumed to be an ammendment of details. If it does not then it is assumed that a programme junction must be taking place.

Components

NowNextServiceFilter

NowNextServiceFilter(*services) -> new NowNextServiceFilter component.

Filters simplified events from Event Information Tables, only letting through those that match the service ids specified.

Argument list is a list of service id's to be let through by the filter.

NowNextProgrammeJunctionDetect

NowNextProgrammeJunctionDetect() -> new NowNextJunctionDetect component.

Takes simplified events derived from parsed Event Information Table data and sorts them according to whether they simply ammend/correct details or whether they represent the start of a new programme (a junction).

Kamaelia.Device.DVB.PSITables

Processing Parsed DVB PSI Tables

Components for filtering and processing parsed Programme Status Information (PSI) tables - that is the output from components in Kamaelia.Device.DVB.Parse

Selecting 'currently' valid tables FilterOutNotCurrent takes in parsed DVB PSI tables, but only outputs the ones that are marked as being currently-valid. Tables that are not yet valid are simply dropped.

NOTE: whether a table is currently-valid or not is *different* from concepts such as present-following (now & next) used for event/programme information. See DVB specification documents for a more detailed explanation.

Example Usage

Tuning to a particular broadcast multiplex and displaying the current selection of services (channels) in the multiplex (as opposed to any future descriptions of services that may be appearing later):

```
frequency = 505833330.0/1000000.0
feparams = {
    "inversion" : dvb3.frontend.INVERSION_AUTO,
    "constellation" : dvb3.frontend.QAM_16,
    "code_rate_HP" : dvb3.frontend.FEC_3_4,
    "code_rate_LP" : dvb3.frontend.FEC_3_4,
}

PAT_PID=0

Pipeline( DVB_Multiplex([PAT_PID], feparams),
          DVB_Demuxer({ PAT_PID:["outbox"]}),
          ReassemblePSITables(),
          ParseProgramAssociationTable(),
          FilterOutNotCurrent(),
          PrettifyProgramAssociationTable(),
          ConsoleEchoer(),
        ).run()
```

Behaviour

Send parsed DVB PSI tables to this component's "inbox" inbox. If the table is a currently-valid one it will immediately be sent on out of the "outbox" outbox.

Tables that are not-yet-valid will be ignored.

If a shutdownMicroprocess or producerFinished message is received on this component's "control" inbox, it will be immediately sent on out of the "signal" outbox and the component will terminate.

How does it work?

The parsed tables you send to this component are dictionaries. This component simply checks the value of the 'current' key in the dictionary.

Components

FilterOutNotCurrent

Filters out any parsed tables not labelled as currently valid

Kamaelia.Device.DVB.Parse.PrettifyTables

Pretty printing of parsed DVB PSI tables

A selection of components for creating human readable strings of the output of the various components in Kamaelia.Device.DVB.Parse that parse data tables

in DVB MPEG Transport Streams.

Example Usage

Pretty printing of a Program Association Table (PAT):

```
FREQUENCY = 505.833330
feparams = {
    "inversion" : dvb3.frontend.INVERSION_AUTO,
    "constellation" : dvb3.frontend.QAM_16,
    "code_rate_HP" : dvb3.frontend.FEC_3_4,
    "code_rate_LP" : dvb3.frontend.FEC_3_4,
}

PAT_PID = 0x0

Pipeline( DVB_Multiplex(FREQUENCY, [PAT_PID], feparams),
          DVB_Demuxer({ PAT_PID:["outbox"]}),
          ReassemblePSITables(),
          ParseProgramAssociationTable(),
          PrettifyProgramAssociationTable(),
          ConsoleEchoer(),
        ).run()
```

Example output:

```
PAT received:
    Table ID           : 0
    Table is valid for : CURRENT (valid)
    NIT is in PID      : 16
    For transport stream id : 4100
        For service 4228 : PMT is in PID 4228
        For service 4351 : PMT is in PID 4351
        For service 4479 : PMT is in PID 4479
        For service 4164 : PMT is in PID 4164
        For service 4415 : PMT is in PID 4415
        For service 4671 : PMT is in PID 4671
```

This data came from an instantaneous snapshot of the PAT for Crystal Palace MUX 1 transmission (505.8MHz) in the UK on 20th Dec 2006.

Behaviour

The components available are:

```
PrettifyProgramAssociationTable
PrettifyNetworkInformationTable
PrettifyProgramMapTable
PrettifyServiceDescriptionTable
```

```
PrettifyEventInformationTable
PrettifyTimeAndDateTable
PrettifyTimeOffsetTable
```

Send to the "inbox" inbox of any of these components the relevant parsed table, and a string will be sent out the "outbox" outbox containing a 'prettified' human readable equivalent of the table data.

If a shutdownMicroprocess or producerFinished message is received on the "control" inbox, then it will immediately be sent on out of the "signal" outbox and the component will then immediately terminate.

Components

PrettifyProgramAssociationTable

PrettifyProgramAssociationTable() -> new PrettifyProgramAssociationTable component.

Send parsed program association tables to the "inbox" inbox and a human readable string version will be sent out the "outbox" outbox.

PrettifyNetworkInformationTable

PrettifyNetworkInformationTable() -> new PrettifyNetworkInformationTable component.

Send parsed network information tables to the "inbox" inbox and a human readable string version will be sent out the "outbox" outbox.

PrettifyProgramMapTable

PrettifyProgramMapTable() -> new PrettifyProgramMapTable component.

Send parsed program map tables to the "inbox" inbox and a human readable string version will be sent out the "outbox" outbox.

PrettifyServiceDescriptionTable

PrettifyServiceDescriptionTable() -> new PrettifyServiceDescriptionTable component.

Send parsed service description tables to the "inbox" inbox and a human readable string version will be sent out the "outbox" outbox.

PrettifyEventInformationTable

PrettifyEventInformationTable() -> new PrettifyEventInformationTable component.

Send parsed event information tables to the "inbox" inbox and a human readable string version will be sent out the "outbox" outbox.

PrettifyTimeAndDateTable

PrettifyTimeAndDateTable() -> new PrettifyTimeAndDateTable component.

Send parsed time and date tables to the "inbox" inbox and a human readable string version will be sent out the "outbox" outbox.

PrettifyTimeOffsetTable

PrettifyTimeOffsetTable() -> new PrettifyTimeOffsetTable component.

Send parsed time offset tables to the "inbox" inbox and a human readable string version will be sent out the "outbox" outbox.

Other

formatDescriptors

Little wrapper around pretty printing of fields in a descriptor, to print the keyname for each descriptor item and sort the indent

iif

NO DOCS

pformat

NO DOCS

Kamaelia.Device.DVB.Parse.ReassemblePSITables

Reassembly of DVB PSI Tables

Components that take a stream of MPEG Transport stream packets containing Programme Status Information (PSI) tables and reassembles the table sections, ready for parsing of the data within them.

ReassemblePSITables can do this for a stream of packets containing a single table.

ReassemblePSITablesService provides a full service capable of reassembling multiple tables from a multiplexed stream of packets, and distributing them to subscribers.

Example Usage A simple pipeline to receive, parse and display the Event Information Table in a multiplex:

```
FREQUENCY = 505.833330
feparams = {
    "inversion" : dvb3.frontend.INVERSION_AUTO,
    "constellation" : dvb3.frontend.QAM_16,
    "code_rate_HP" : dvb3.frontend.FEC_3_4,
    "code_rate_LP" : dvb3.frontend.FEC_3_4,
}

EIT_PID = 0x12

Pipeline( OneShot( msg=["ADD", [0x2000] ] ),    # take all packets of all PIDs
          Tuner(FREQUENCY, feparams),
          DVB_SoftDemuxer( { EIT_PID : ["outbox"] } ),
          ReassemblePSITables(),
          ParseEventInformationTable(),
          PrettifyEventInformationTable(),
          ConsoleEchoer(),
        ).run()
```

Set up a dvb tuner and demultiplexer as a service; then set up a PSI tables service (that subscribes to the demuxer); then finally subscribe to the PSI tables service to get Event Information Tables and parse and display them:

```
RegisterService( Receiver( FREQUENCY, FE_PARAMS, 0 ),
                 {"DEMUXER":"inbox"},
            ).activate()

RegisterService(         Graphline( PSI     = ReassemblePSITablesService(),
               DEMUXER = ToService("DEMUXER"),
               linkages = {
                   ("PSI", "pid_request") : ("DEMUXER", "inbox"),
                   ("",    "request")     : ("PSI",     "request"),
               }
             ),
    {"PSI_Tables":"request"}
).activate()

Pipeline( Subscribe("PSI", [EIT_PID]),
          ParseEventInformationTable(),
          PrettifyEventInformationTable(),
          ConsoleEchoer(),
        ).run()
```

In the above example, the final pipeline subscribes to the 'PSI' service, requesting the PSI tables in MPEG Transport Stream packets with packet id 0x12.

The ReassemblePSITablesService service uses a ToService component to send its own requests to the 'DEMUXER' service to ask for MPEG Transport Stream packets with the packet ids it needs.

ReassemblePSITables ReassemblePSITables reassembles one PSI table at a time from a stream of MPEG transport stream packets containing that table.

Behaviour

Send individual MPEG Transport Stream packets to the "inbox" inbox containing fragments of a particular PSI table.

ReassemblePSITables will reconstruct the table sections. As soon as a section is complete, it will be sent, as a raw binary string, out of the "outbox" outbox. The process repeats indefinitely.

If a shutdownMicroprocess or producerFinished message is received on the "control" inbox, then it will immediately be sent on out of the "signal" outbox and the component will then immediately terminate.

ReassemblePSITablesService ReassemblePSITablesService provides a full service capable of reassembling multiple tables from a multiplexed stream of packets, and distributing them to subscribers.

Behaviour

ReassemblePSITablesServices takes individual MPEG Transport Stream packets sent to its "inbox" inbox and reconstructs table sections, distributing them to clients/subscribers that have requested them.

To be a client you can wrap ReassemblePSITablesService into a named service by using a Kamaelia.Experiment.Services.RegisterService component, and then subscribe to it using a Kamaelia.Experiment.Services.SubscribeTo component.

Alternatively, send a 'ADD' or 'REMOVE' message to its "request" inbox, requesting to be sent (or no longer be sent) tables from packets of particular PIDs, and specifying the inbox to which you want the packets to be sent. The format of these requests is:

```
("ADD",    [pid, pid, ...], (dest_component, dest_inboxname))
("REMOVE", [pid, pid, ...], (dest_component, dest_inboxname))
```

ReassemblePSITablesService will automatically do the wiring or unwiring needed to ensure the packets you have requested get sent to the inbox you specified.

Send an 'ADD' request, and you will immediately start receiving tables in those PIDs. Send a 'REMOVE' request and you will shortly no longer receive tables in the PIDs you specify. Note that you may still receive some tables after your 'REMOVE' request.

ReassemblePSITablesService will also send its own requests (in the same format) out of its "pid_request" outbox. You can wire this up to the source of transport stream packets, so that ReassemblePSITablesService can tell that source what PIDs it needs. Alternatively, simply ensure that your source is already sending all the PIDs your ReassemblePSITablesService component will need.

If a shutdownMicroprocess or producerFinished message is received on the "control" inbox, then it will immediately be sent on out of the "signal" outbox and the component will then immediately terminate.

How does it work?

ReassemblePSITablesService creates an outbox for each subscriber destination, and wires from it to the destination.

For each PID that needs to be processed, a ReassemblePSITables component is created to handle reconstruction of that particular table.Transport Stream packets arriving at the "inbox" inbox are sent to the relevant ReassemblePSITables component for table reconstruction. Reconstructed tables coming back from each ReassemblePSITables component are forwarded to all destinations that have subscribed to it.

When ReassemblePSITablesServices starts or stops using packets of a given PID, an 'add' or 'remove' message is also sent out of the "pid_request" outbox:

```
("ADD",    [pid], (self, "inbox"))
("REMOVE", [pid], (self, "inbox"))
```

This can be wired up to the source of transport stream packets, so that ReassemblePSITablesService can tell that source what PIDs it needs.

Components

ReassemblePSITables

Takes DVB Transport stream packets for a given PID and reconstructs the PSI packets from within the stream.

Will only handle stream from a single PID.

ReassemblePSITablesService

ReassemblePSITablesService() -> new ReassemblePSITablesService component.

Subscribe to PSI packets by sending ("ADD", (component,inbox), [PIDs]) to "request" Unsubscribe by sending ("REMOVE", (component,inbox), [PIDs]) to "request"

Kamaelia.Exceptions

General Kamaelia Exceptions

This module defines a set of standard exceptions generally useful in Kamaelia. They are all based on the Axon.AxonExceptions.AxonException base class.

The exceptions

- **BadRequest(request, innerexception)** - signalling that a request caused an exception`self.request` is the original request and `self.exception` is the exception that it caused to be thrown
- **socketSendFailure()** - signalling that a socket failed trying to send
- **connectionClosedown()** - singalling that a connection closed down
- **connectionDied()** - signalling that a connection died
  - connectionDiedSending()
  - connectionDiedReceiving()
  - connectionServerShutdown()

Other

BadRequest

Thrown when parsing a request fails

connectionClosedown

NO DOCS

connectionDied

NO DOCS

connectionDiedReceiving

NO DOCS

connectionDiedSending

NO DOCS

connectionServerShutdown

NO DOCS

socketSendFailure

NO DOCS

Kamaelia.Experimental.Chassis

Inbox size limiting Pipelines, Graphlines and Carousels

Extended versions of Kamaelia.Chassis.Pipeline, Kamaelia.Chassis.Graphline and Kamaelia.Chassis.Carousel that add the ability to specify size limits for inboxes of components.

Example Usages

A pipeline with inbox size limits on 3 of the components' "inbox" inboxes:

```
Pipeline( 5,  MyComponent(),      # 'inbox' inbox limited to 5 items
          2,  MyComponent(),      # 'inbox' inbox limited to 2 items
              MyComponent(),      # 'inbox' inbox unlimited
          28, MyComponent()       # 'inbox' inbox limited to 28 items
        )
```

A graphline where component 'A' has a size limit of 5 on its "inbox" inbox; and component 'C' has a size limit of 17 on its "control" inbox:

```
Graphline( A = MyComponent(),
           B = MyComponent(),
           C = MyComponent(),
           linkages = { ... },
           boxsizes = {
               ("A","inbox") : 5,
               ("C","control") : 17
           }
         )
```

A Carousel, where the child component will have a size limit of 5 on its "inbox" inbox:

```
Carousel( MyComponent(), boxsize=5 )
```

Decoding a Dirac video file and saving each frame in a separate file:

```
Pipeline(
    RateControlledFileReader("video.dirac", ... ),
    DiracDecoder(),
    TagWithSequenceNumber(),
    InboxControlledCarousel(
        lambda (seqnum, frame) :
            Pipeline( OneShot(frame),
                      FrameToYUV4MPEG(),
                      SimpleFileWriter("%08d.yuv4mpeg" % seqnum),
                    )
        ),
    )
```

More details

The behaviour of these three components/prefabs is identical to their original counterparts (Kamaelia.Chassis.Pipeline, Kamaelia.Chassis.Graphline and Kamaelia.Chassis.Carousel).

For Pipelines, if you want to size limit the "inbox" inbox of a particular component in the pipeline, then put the size limit as an integer before it. Any component without an integer before it is left with the default of an unlimited "inbox" inbox.

The behaviour therefore reduces back to be identical to that of the normal Pipeline component.

For Graphlines, if you want to size limit particular inboxes, supply the "boxsizes" argument with a dictionary that maps (componentName, boxName) keys to the size limit for that box.

Again, if you don't specify a "boxsizes" argument, then behaviour is identical to that of the normal Graphline component.

For Carousels, if you want a size limit on the "inbox" inbox of the child component (created by the factory function), then specify it using the "boxsizes" argument.

Again, if you don't specify a "boxsizes" argument, then behaviour is identical to that of the normal Carousel component.

InboxControlledCarousel behaves identically to Carousel.

The "inbox" inbox is equivalent to the "next" inbox of Carousel. The "data_inbox" inbox is equivalent to the "inbox" inbox of Carousel.

Other

Carousel

> Carousel(componentFactory[,make1stRequest][,boxSize]) -> new Carousel component
>
> Create a Carousel component that makes child components one at a time (in carousel fashion) using the supplied factory function.
>
> Keyword arguments:
>
> - componentFactory -- function that takes a single argument and returns a component
> - make1stRequest -- if True, Carousel will send an initial "NEXT" request. (default=False)
> - boxsize -- size limit for "inbox" inbox of the created child component

Graphline

Graphline([linkages][,boxsizes],**components) -> new Graphline component

Encapsulates the specified set of components and wires them up with the specified linkages.

Keyword arguments:

- linkages -- dictionary mapping ("componentname","boxname") to ("componentname","boxname")
- boxsizes -- dictionary mapping ("componentname","boxname") to size limit for inbox
- components -- dictionary mapping names to component instances (default is nothing)

InboxControlledCarousel

InboxControlledCarousel(componentFactory[,make1stRequest][,boxSize]) -> new Carousel component

Create an InboxControlledCarousel component that makes child components one at a time (in carousel fashion) using the supplied factory function.

Keyword arguments:

- componentFactory -- function that takes a single argument and returns a component
- make1stRequest -- if True, Carousel will send an initial "NEXT" request. (default=False)
- boxsize -- size limit for "inbox" inbox of the created child component

Pipeline

Pipeline(*components) -> new Pipeline component.

Encapsulates the specified set of components and wires them up in a chain (a Pipeline) in the order you provided them.

Keyword arguments:

- components -- the components you want, in the order you want them wired up. Any Integers set the "inbox" inbox size limit for the component that follows them.

Kamaelia.Experimental.ERParsing

Parser components for Entity-Relationship data

ERParser parses and buffers lines of text containing entity-relationship data. Once a shutdown message is received, it emits the parsed data as a list of entities and relationships.

ERModel2Visualiser transforms parsed entity-relationship data into textual commands for the TopologyViewer component to produce a visualisation. The TopologyViewer must be configured with suitable particle types - such as Kamaelia.Visualisation.ER.ERVisualiserServer.ERVisualiser

See: Kamaelia.Visualisation.PhysicsGraph.TopologyViewer.TopologyViewer

Example Usage:

A simple pipeline that reads in entity-relationship data from a file and writes out commands, suitable for a topology visualiser, to the console:

```
from Kamaelia.File.ReadFileAdaptor import ReadFileAdaptor
from Kamaelia.Util.PureTransformer import PureTransformer
from Kamaelia.Util.Console import ConsoleEchoer

Pipeline(
    ReadFileAdaptor(entity_relationship_data_file),
    ERParser(),
    ERModel2Visualiser(),
    PureTransformer(lambda x: pprint.pformat(x)+"\n"),
    ConsoleEchoer(),
).run()
```

Provide the following file of entity relationship data:

```
#
# entity relationship data in this file!
#

entity Artist:
    simpleattributes artisticname genre

entity Manager:
    simpleattributes ID name1 telephone

entity ContractInfo:
    simpleattributes contractID data_from data_to duration1

entity MasterTrack:
    simpleattributes trackID working_title duration2
```

```
entity SoundEngineer:
    simpleattributes sound_eng_ID name2

entity FinishedTrack:
    simpleattributes version final_duration released_title

entity Album:
    simpleattributes album_ID title

relation ManagedBy(Artist,Manager)
relation HasContract(Artist,ContractInfo)
relation RecordedBy(MasterTrack,Artist)
relation EditedBy(SoundEngineer,MasterTrack)
relation OriginatesFrom(FinishedTrack,MasterTrack)
relation GroupedOn(FinishedTrack,Album)
relation CreatedBy(Album,Artist)
```

Once the ReadFileAdaptor component has finished reading and terminates, the ERParser component sends a message, containing the following, out of its "outbox" outbox:

```
[['entity', {'name': 'Artist', 'simpleattributes': ['artisticname', 'genre']}],
 ['entity',
  {'name': 'Manager', 'simpleattributes': ['ID', 'name1', 'telephone']}],
 ['entity',
  {'name': 'ContractInfo',
   'simpleattributes': ['contractID', 'data_from', 'data_to', 'duration1']}],
 ['entity',
  {'name': 'MasterTrack',
   'simpleattributes': ['trackID', 'working_title', 'duration2']}],
 ['entity',
  {'name': 'SoundEngineer', 'simpleattributes': ['sound_eng_ID', 'name2']}],
 ['entity',
  {'name': 'FinishedTrack',
   'simpleattributes': ['version', 'final_duration', 'released_title']}],
 ['entity', {'name': 'Album', 'simpleattributes': ['album_ID', 'title']}],
 ['relation', {'entities': ['Artist', 'Manager'], 'name': 'ManagedBy'}],
 ['relation', {'entities': ['Artist', 'ContractInfo'], 'name': 'HasContract'}],
 ['relation', {'entities': ['MasterTrack', 'Artist'], 'name': 'RecordedBy'}],
 ['relation',
  {'entities': ['SoundEngineer', 'MasterTrack'], 'name': 'EditedBy'}],
 ['relation',
  {'entities': ['FinishedTrack', 'MasterTrack'], 'name': 'OriginatesFrom'}],
 ['relation', {'entities': ['FinishedTrack', 'Album'], 'name': 'GroupedOn'}],
 ['relation', {'entities': ['Album', 'Artist'], 'name': 'CreatedBy'}]]
```

And the following is output from the console:

```
'ADD NODE Artist Artist auto entity'
'ADD NODE artisticname artisticname auto attribute'
'ADD NODE genre genre auto attribute'
'ADD NODE Manager Manager auto entity'
'ADD NODE ID ID auto attribute'
'ADD NODE name1 name1 auto attribute'
'ADD NODE telephone telephone auto attribute'
'ADD NODE ContractInfo ContractInfo auto entity'
'ADD NODE contractID contractID auto attribute'
'ADD NODE data_from data_from auto attribute'
'ADD NODE data_to data_to auto attribute'
'ADD NODE duration1 duration1 auto attribute'
'ADD NODE MasterTrack MasterTrack auto entity'
'ADD NODE trackID trackID auto attribute'
'ADD NODE working_title working_title auto attribute'
'ADD NODE duration2 duration2 auto attribute'
'ADD NODE SoundEngineer SoundEngineer auto entity'
'ADD NODE sound_eng_ID sound_eng_ID auto attribute'
'ADD NODE name2 name2 auto attribute'
'ADD NODE FinishedTrack FinishedTrack auto entity'
'ADD NODE version version auto attribute'
'ADD NODE final_duration final_duration auto attribute'
'ADD NODE released_title released_title auto attribute'
'ADD NODE Album Album auto entity'
'ADD NODE album_ID album_ID auto attribute'
'ADD NODE title title auto attribute'
'ADD NODE ManagedBy ManagedBy auto relation'
'ADD NODE HasContract HasContract auto relation'
'ADD NODE RecordedBy RecordedBy auto relation'
'ADD NODE EditedBy EditedBy auto relation'
'ADD NODE OriginatesFrom OriginatesFrom auto relation'
'ADD NODE GroupedOn GroupedOn auto relation'
'ADD NODE CreatedBy CreatedBy auto relation'
'ADD LINK Artist artisticname'
'ADD LINK Artist genre'
'ADD LINK Manager ID'
'ADD LINK Manager name1'
'ADD LINK Manager telephone'
'ADD LINK ContractInfo contractID'
'ADD LINK ContractInfo data_from'
'ADD LINK ContractInfo data_to'
'ADD LINK ContractInfo duration1'
'ADD LINK MasterTrack trackID'
'ADD LINK MasterTrack working_title'
```

```
'ADD LINK MasterTrack duration2'
'ADD LINK SoundEngineer sound_eng_ID'
'ADD LINK SoundEngineer name2'
'ADD LINK FinishedTrack version'
'ADD LINK FinishedTrack final_duration'
'ADD LINK FinishedTrack released_title'
'ADD LINK Album album_ID'
'ADD LINK Album title'
'ADD LINK Artist ManagedBy'
'ADD LINK Manager ManagedBy'
'ADD LINK Artist HasContract'
'ADD LINK ContractInfo HasContract'
'ADD LINK MasterTrack RecordedBy'
'ADD LINK Artist RecordedBy'
'ADD LINK SoundEngineer EditedBy'
'ADD LINK MasterTrack EditedBy'
'ADD LINK FinishedTrack OriginatesFrom'
'ADD LINK MasterTrack OriginatesFrom'
'ADD LINK FinishedTrack GroupedOn'
'ADD LINK Album GroupedOn'
'ADD LINK Album CreatedBy'
'ADD LINK Artist CreatedBy'
```

ERParser Behaviour

Send entity-relationship textual description data to the "inbox" inbox as individual strings, one line per string.

When a producerFinished or shutdownMicroprocess is sent to the "control" inbox this component sends out a single message containing, as a list, the entities and relationships parsed from the data.

ERParser then immediately terminates and sends out the shudown message it received out of its "signal" outbox.

See description of "Entity-Relationship textual description format" and "Parsed Entity-Relationship data".

ERModel2Visualiser Behaviour

Send parsed entity-relationship data to the "inbox" inbox.

When a producerFinished or shutdownMicroprocess is sent to the "control" inbox this component transforms it into a set of textual (string) commands suitable for a TopologyViewer component and sends it out of its "outbox" outbox in two messages. The first contains the commands to create the required nodes and the second contains the commands to create the linkages between them.

ERModel2Visualiser then immediately terminates and sends out the shudown message it received out of its "signal" outbox.

The TopologyViewer component that receives the commands must be configured to know the following node types:

- entity -- represents an entity
- relation -- represents the mid-point and name label of a relation
- attribute -- represents an attribute of an entity
- isa -- represents the mid-point and label of a subtype-supertype relation

See description of "Parsed Entity-Relationship data".

Entity-Relationship textual description format

Entity relationship data is expressed as a text file. Blank lines or lines beginning with a '#' character are ignored.

Define entities by writing entity NAME: on its own line, without indentation. To define attributes for an entity, write, indented, on the next line, simpleattributes followed by a space separated list of attributes.

To make an entity a subtype of another entity, begin the entity declaration instead with entity NAME(SUPERTYPE):.

Example entities:

```
entity person:
    simpleattributes female name=sarah

# the following entities are subtypes of the 'person' entity

entity mum(person):

entity daughter(person):

entity son(person):
```

To define relations, write relation RELATION_NAME(ENTITY,ENTITY,...) on its own line without indentation. A relation must involve two or more entities. For example:

```
relation RelatedTo(mum,daughter,son)
relation siblings(son,daughter)
```

Parsed Entity-Relationship data

The parsed data takes the form of a list of entities and relations.

An entity is represented as a list beginning with the string "entity", followed by a dictionary defining attributes. The key "name" maps to the name of the entity:

["entity", { "name": <name>, "simpleattributes": [<attribute>, <attribute>, ...], "subtype": <supertype-name>]

The key "simpleattributes", if present, maps to a list containing, as strings each attribute.

Some may also contain a "subtype" key which maps to the name of the entity that this one is a subtype of.

For example:

```
[ "entity",
  { "name"              : "person",
    "simpleattributes" : [ "female", "name=sarah" ]
  }
]

[ "entity",
  { "name" : "mum",
    "subtype" : "person"
  }
]
```

A relation is represented as a list beginning with the string "relation", followed by a dictionary defining attributes:

["relation", { "name": <name>, "entities": [<entity-name>, <entity-name>, ...]

The key "name" maps to the name of the entity. The key "entities" maps to a list containing, as strings, the names of the entities involved in the relation. For example:

```
[ "relation",
  { "name"     : "IsA",
    "entities" : [ "mum", "person" ]
  }
]
```

Components

ERParser

ERParser() -> new ERParser component.

Parses lines of Entity-Relationship data, send to the "inbox" inbox as strings. Once shutdown, sends out a list of parsed entity and relationship data.

ERModel2Visualiser

ERModel2Visualiser() -> new ERModel2Visualiser component.

Send parsed entity-relationship data as lists of entities and relations to the "inbox" inbox. Once shutdown, sends out commands to drive a TopologyViewer component to produce a visualisation of the described entities and relations.

Other

ParseError

NO DOCS

parseEntityLine

NO DOCS

parseMultilineEntity

NO DOCS

parseRelationLine

NO DOCS

parseSimpleAttributes

NO DOCS

parse_model

NO DOCS

parser

NO DOCS

Kamaelia.Experimental.PythonInterpreter

PythonInterpreter Component

Initial version of an interactive console in Kamaelia. Provides a nice way of playing with Kamaelia components and examining running systems.

Example Usage

Show a pygame Textbox for reading user input, and a TextDisplayer, showing the interpreter's response.

```
from Kamaelia.UI.Pygame.Text import Textbox, TextDisplayer
Pipeline( Textbox(size = (800, 300), position = (100,380)), InterpreterTransformer(), TextDisplayer(size = (800, 300), position = (100,40)), ).run()
```

This then operates as a normal python interpreter which happens to run in a pygame window.

How to use it

This component takes python commands from its inbox "inbox", and spits out normal python interpreter repsonses out its outbox "outbox".

There are more examples in the examples directory.

Warning

This component may well change its location in the Kamaelia namespace.

TODO

More detailed examples.

Other

InterpreterTransformer

NO DOCS

StandaloneInterpreter

NO DOCS

Kamaelia.Experimental.Services

Components to help build Services (EXPERIMENTAL)

These components make it easier to build and use public services, registered with the Coordinating Assistant Tracker.

Note: These components are EXPERIMENTAL and are likely to under go substantial change

Register Service

A function that registers specified inboxes on a component as named services with the Coordinating Assistant Tracker (CAT). Returns the component, so can be dropped in where you would ordinarily use a component.

Example Usage: Create and activate MyComponent instance, registering its "inbox" inbox with the CAT as a service called "MyService":

```
RegisterService( MyComponent(), {"MyService":"inbox"} ).activate()
```

How does it work? This method registers the component you provide with the CAT. It register the inboxes on it that you specify using the names that you specify.

Subscibe To Service

A component that connects to a public service and sends a fixed format message to it requesting to subscribe to a set of 'things' it provides.

Example Usage: Subscribe to a (fictional) "TV Channels Service", asking for three channels. The tv channel data is then recorded:

```
pipeline(
    SubscribeTo( "TV Channels Service", ["BBC ONE", "BBC TWO", "ITV"] ),
    RecordChannels(),
    ).run()
```

The message sent to the "TV Channels Service" will be:

```
("ADD", ["BBC ONE", "BBC TWO", "ITV"], ( <theSubscribeToComponent>, "inbox" ) )
```

How does it work? Describe more detail here

Connect To Service

A component that connects to a public service. Any data you send to its inbox gets sent to the service.

Example Usage

```
pipeline( MyComponentThatSendMessagesToService(),
        ConnectTo("Name of service"),
        ).run()
```

How does it work? Describe in more detail here.

Components

Subscribe

Subscribes to a service, and forwards what it receives to its outbox. Also forwards anything that arrives at its inbox to its outbox.

Unsubscribes when shutdown.

ToService

NO DOCS

Other

RegisterService

NO DOCS

Kamaelia.File.Append

Append

This component accepts data from it's inbox "inbox" and appends the data to the end of the given file.

It takes four arguments, with these default values:

```
filename = None
forwarder = True
blat_file = False
hold_open = True
```

filename should be clear. If you don't supply this, it'll break.

forwarder - this component defaults to passing on a copy of the data it's appending to the file. This makes this component useful for dropping in between other components for logging/debugging what's going on.

blat_file - if this is true, the file is zapped before we start appending data.

hold_open - This determines if the file is closed between instances of data arriving.

Components

Append

Appender() -> component that incrementally append data to the end of a file (think logging)

Uses the following keyword argyments:

```
* filename - File to append to (required)
* forwarder - copy to outbox (default: True)
* blat_file - write empty file (default: False)
* hold_open - keep file open (default: True)
```

Other

Axon

Axon - the core concurrency system for Kamaelia

Axon is a component concurrency framework. With it you can create software "components" that can run concurrently with each other. Components have "inboxes" and "outboxes" through with they communicate with other components.

A component may send a message to one of its outboxes. If a linkage has been created from that outbox to another component's inbox; then that message will arrive in the inbox of the other component. In this way, components can send and receive data - allowing you to create systems by linking many components together.

Each component is a microprocess - rather like a thread of execution. A scheduler takes care of making sure all microprocesses (and therefore all components) get regularly executed. It also looks after putting microprocesses to sleep (when they ask to be) and waking them up (for example, when something arrives in one of their inboxes).

Base classes for building your own components

- **Axon.Component**
  - defines the basic component. Subclass it to write your own components.
- **Axon.AdaptiveCommsComponent**
  - like a basic component but with facilties to let you add and remove inboxes and outboxes during runtime.
- **Axon.ThreadedComponent**
  - like ordinary components, but which truly run in a separate thread - meaning they can perform blocking tasks (since they don't have to yield control to the scheduler for other components to continue executing)

Underlying concurrency system

- **Axon.Microprocess**
  - Turns a python generator into a schedulable microprocess - something that can be started, paused, reawoken and stopped. Subclass it to make your own.
- **Axon.Scheduler**

  - Runs the microprocesses. Manages the starting, stopping, pausing and waking of them. Is also a microprocess itself!

Services, statistics, Instrospection

- **Axon.CoordinatingAssistantTracker**
  - provides mechanisms for components to advertising and discover services they can provide for each other.
  - acts as a repository for collecting statistics from components in the system
- **Axon.Introspector**
  - outputs live topology data describing what components there are in a running axon system and how they are linked together.

Exceptions, Messages and Misc

- **Axon.Base**
  - base metaclass for key Axon classes
- **Axon.AxonExceptions**
  - classes defining various exceptions in Axon.
- **Axon.Ipc**
  - classes defining various IPC messages in Axon used for signalling shutdown, errors, notifications, etc...
- **Axon.idGen**
  - unique id value generation
- **Axon.util**
  - various miscellaneous support utility methods

Integration with other systems

- **Axon.background**
  - use Axon components within other python programs by wrapping them in a scheduler running in a separate thread
- **Axon.Handle**
  - a Handle for getting data into and out of the standard inboxes and outboxes of a component from a non Axon based piece of code. Useful in combination with Axon.background

Internals for implementing inboxes, outboxes and linkages

- **Axon.Box**
  - The base implementation of inboxes and outboxes.
- **Axon.Postoffice**
  - All components have one of these for creating, destroying and tracking linkages.
- **Axon.Linkage**
  - handles used to describe linkages from one postbox to another

What, no Postman? Optimisations made to Axon have dropped the Postman. Inboxes and outboxes handle the delivery of messages themselves now.

Debugging support

- **Axon.debug**
  - defines a debugging output object.
- **Axon.debugConfigFile**
  - defines a method for loading a debugging configuration file that determines what debugging output gets displayed and what gets filtered out.
- **Axon.debugConfigDefaults**
  - defines a method that supplies a default debugging configuration.

Kamaelia.File.BetterReading

Intelligent File Reader

This component reads the filename specified at its creation and outputs it as several messages. When a certain number of messages in its outbox have not yet been delivered it will pause to reduce memory and CPU usage. To wake it, ideally Axon should unpause it when the outbox has less than a certain number of messages (i.e. when some are delivered) but for now you can send it an arbitrary message (to "inbox") which will wake the component.

System Requirements

This module requires a UNIX system to run currently.

Components

IntelligentFileReader

> IntelligentFileReader(filename, chunksize, maxqueue) -> file reading component
>
> Creates a file reader component. Reads a chunk of chunksize bytes, using the Selector to avoid having to block, pausing when the length of its send-queue exceeds maxqueue chunks.

Kamaelia.File.MaxSpeedFileReader

Reading a file as fast as possible

MaxSpeedFileReader reads a file in bytes mode as fast as it can; limited only by any size limit on the inbox it is sending the data to.

This component is therefore useful for building systems that are self rate limiting - systems that are just trying to process data as fast as they can and are limited by the speed of the slowest part of the chain.

Example Usage

Read "myfile" in in chunks of 1024 bytes. The rate is limited by the rate at which the consumer component can consume the chunks, since its inbox has a size limit of 5 items of data:

```
consumer = Consumer()
consumer.inboxes["inbox"].setSize(5)

Pipeline( MaxSpeedFileReader("myfile", chunksize=1024),
          consumer,
        ).run()
```

More details

Specify a filename and chunksize and MaxSpeedFileReader will read bytes from the file in the chunksize you specified and send them out of its "outbox" outbox.

If the destination inbox it is sending chunks to is size limited, then MaxSpeedFileReader will pause until space becomes available. This is how the speed at which the file is ingested is regulated - by the rate at which it is consumed.

When the whole file has been read, this component will terminate and send a producerFinished() message out of its "signal" outbox.

If a producerFinished message is received on the "control" inbox, this component will complete sending any data that may be waiting. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete sending on any pending data.

Components

MaxSpeedFileReader

> MaxSpeedFileReader(filename[,chunksize]) -> new MaxSpeedFileReader component.
>
> Reads the contents of a file in bytes mode; sending it out as fast as it can in chunks from the "outbox" outbox. The rate of reading is only limited by any size limit of the destination inbox to which the data is being sent.
>
> Keyword arguments:

- filename -- The filename of the file to read
- chunksize -- Optional. The maximum number of bytes in each chunk of data read from the file and sent out of the "outbox" outbox (default=32768)

Kamaelia.File.ReadFileAdaptor

ReadFileAdaptor Component

A simple component that reads data from a file, line by line, or in chunks of bytes, at a constant rate that you specify.

Example Usage

Read one line at a time from a text file every 0.2 seconds, outputting to standard output:

```
Pipeline( ReadFileAdaptor("myfile.text", readmode='line', readsize=1, steptime=0.2),
          ConsoleEchoer(),
        ).run()
```

Read from a file at 1 Kilobits per second, in chunks of 256 bytes and send it to a server over a TCP network socket:

```
Pipeline( ReadFileAdaptor("myfile.data", readmode='bitrate', readsize=1, bitrate=1024*10),
          TCPClient("remoteServer.provider.com", port=1500),
        ).run()
```

How to use it

This component takes input from the outside world, and makes it available on it's outbox. It can take input from the following data sources:

- A specific file
- The output of a command
- stdin

In all cases, it can make the data available in the following modes:

- On a line by line basis
- On a "block by block" basis.
- At an attempted (not guaranteed) bit rate.
- If at a bit rate, also a "frame rate". (eg if you want 1Mbit/s, do you want that as 1x1Mbit block, 4x250Kbit blocks, 25x 40Kbit blocks, or what, each second?
- A "step time" to define how often to read 0 == as often/fast as possible 0.1 == every 0.1 seconds, 0.5 = every 0.5 seconds, 2 = every 2 seconds etc.

Clearly some of these modes are mutually exclusive!

Once ReadFileAdaptor has finished reading from the file, it finishes by sending a producerFinished() message out of its "signal" outbox, then immediately terminates.

There is no way to tell ReadFileAdaptor to prematurely stop. It ignores all messages sent to its "control" inbox.

To do

The default standalone behaviour is to read in from stdin, and dump to stdout in a teletype fashion the data it recieves. It _may gain command line parsing at some point, which would be wacky. (And probably a good way of initialising components - useful standalone & externally)

XXX TODO: Signal EOF on an external output to allow clients to destroy us. XXX Implement the closeDown method - ideally add to the component XXX framework.

Components

ReadFileAdaptor

An instance of this class is a read file adaptor component. It's constructor arguments are all optional. If no arguments are provided, then the default is to read from stdin, one line at a time, as fast as possible. Note that this will cause the outbox to fill at the same rate as stdin can provide data. (Be wary of memory constraints this will cause!)

Arguments & meaning:

- filename="filename" - the name of the file to read. If you want stdin, do not provide a filename! If you want the output from a command, also leave this blank...
- command="command" - the name of the command you want the output from. Leave the filename blank if you use this!
- **readmode - possible values:**
  - "bitrate" - read at a specified bitrate.
  - "line" - read on a line by line basis
  - "block" - read the file on a block by block basis.
- If bitrate mode is set, you should set bitrate= to your desired bitrate (unless you want 64Kbit/s), and chunkrate= to your desired chunkrate (unless you want 24 fps). You are expected to be able to handle the bit rate you request!
- If block mode is set then you should set readsize (size of the block in bytes), and steptime (how often you want bytes).

If steptime is set to zero, you will read blocks at the speed the source device can provide them. (be wary of memory constraints)

After setting the ReadFileAdaptor in motion, you can then hook it into your linkages like any other component.

Other

EOF

NO DOCS

Kamaelia.File.Reading

Components for reading from files

These components provide various ways to read from files, such as manual control of the rate at which the file is read, or reusing a file reader to read from multiple files.

Key to this is a file reader component that reads data only when asked to. Control over when data flows is therefore up to another component - be that a simple component that requests data at a constant rate, or something else that only requests data when required.

PromptedFileReader

This component reads bytes or lines from the specified file when prompted.

Send the number of bytes/lines to the "inbox" inbox, and that data will be read and sent to the "outbox" outbox.

Example Usage

Reading 1000 bytes per second in 10 byte chunks from 'myfile':

```
Pipeline(ByteRate_RequestControl(rate=1000,chunksize=10)
         PromptedFileReader("myfile", readmode="bytes")
        ).activate()
```

More detail

The component will terminate if it receives a shutdownMicroprocess message on its "control" inbox. It will pass the message on out of its "signal" outbox.

If unable to read all N bytes/lines requested (perhaps because we are nearly at the end of the file) then those bytes/lines that were read successfully are still output.

When the end of the file is reached, a producerFinished message is sent to the "signal" outbox.

The file is opened only when the component is activated (enters its main loop).

The file is closed when the component shuts down.

SimpleReader

This is a simplified file reader that simply reads the given file and spits it out "outbox". It also handles maximum pipewidths, enabling rate limiting to be handled by a piped component.

Example Usage

Usage is the obvious:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.File.Reading import SimpleReader
from Kamaelia.Util.Console import ConsoleEchoer

Pipeline(
    SimpleReader("/etc/fstab"),
    ConsoleEchoer(),
).run()
```

More detail

This component will terminate if it receives a shutdownMicroprocess message on its "control" inbox. It will pass the message on out of its "signal" outbox.

If unable to send the message to the recipient (due to the recipient enforcing pipewidths) then the reader pauses until the recipient is ready and resends (or a shutdown message is recieved).

The file is opened only when the component is activated (enters its main loop).

The file is closed when the component shuts down.

RateControlledFileReader

This component reads bytes/lines from a file at a specified rate. It is performs the same task as the ReadFileAdapter component.

You can configure the rate, and the chunk size or frequency.

Example Usage

Read 10 lines per second, in 2 chunks of 5 lines, and output them to the console:

```
Pipeline(RateControlledFileReader("myfile", "lines", rate=10, chunksize=5),
         ConsoleEchoer()
        ).activate()
```

More detail

This component is a composition of a PromptedFileReader component and a ByteRate_RequestControl component.

The component will shut down after all data is read from the file, emitting a producerFinished message from its "signal" outbox.

The component will terminate if it receives a shutdownMicroprocess message on its "control" inbox. It will pass the message on out of its "signal" outbox.

The inbox "inbox" is not wired and therefore does nothing.

ReusableFileReader

A reusable PromptedFileReader component, based on a Carousel component. Send it a new filename and it will start reading from that file. Do this as many times as you like.

Send it the number of bytes/lines to read and it will output that much data, read from the file.

Example Usage

Read data from a sequence of files, at 1024 bytes/second in 16 byte chunks:

```
playlist = Chooser(["file1","file2","file3", ...]
rate = ByteRate_RequestControl(rate=1024,chunksize=16)
reader = ReusableFileReader("bytes")

playlist.link( (reader, "requestNext"), (playlist,"inbox") )
playlist.link( (playlist,"outbox"), (reader, "next") )

Pipeline(ratecontrol, reader).activate()
```

Or, with the Control-Signal path linked up properly, using the JoinChooserToCarousel prefab:

```
playlist = Chooser(["file1","file2","file3", ...]
rate = ByteRate_RequestControl(rate=1024,chunksize=16)
reader = ReusableFileReader("bytes")

playlistreader = JoinChooserToCarousel(playlist, reader)

Pipeline(ratecontrol, playlistreader).activate()
```

More detail

Bytes or lines are read from the file on request. Send the number of bytes/lines to the "inbox" inbox, and that data will be read and sent to the "outbox" outbox.

This component will terminate if it receives a shutdownMicroprocess or producerFinished message on its "control" inbox. The message will be passed on out of its "signal" outbox.

No producerFinished or shutdownMicroprocess type messages are sent by this component between one file and the next.

RateControlledReusableFileReader

A reusable file reader component, based on a Carousel component. Send it a filename and the rate you want it to run at, and it will start reading from that file at that rate. Do this as many times as you like.

Example Usage

Read data from a sequence of files, at different rates:

```
playlist = Chooser([ ("file1",{"rate":1024}),
                     ("file2",{"rate":16}), ...])
reader = RateControlledReusableFileReader("bytes")

playlist.link( (reader, "requestNext"), (playlist,"inbox") )
playlist.link( (playlist,"outbox"), (reader, "next") )

reader.activate()
playlist.activate()
```

Or, with the Control-Signal path linked up properly, using the JoinChooserToCarousel prefab:

```
playlist = Chooser([ ("file1",{"rate":1024}),
                     ("file2",{"rate":16}), ...])
reader = RateControlledReusableFileReader("bytes")

playlistreader = JoinChooserToCarousel(playlist, reader).activate()
```

More detail

The rate control is performed by a ByteRate_RequestControl component. The rate arguments should be those that are accepted by this component.

This component will terminate if it receives a shutdownMicroprocess or producerFinished message on its "control" inbox. The message will be passed on out of its "signal" outbox.

No producerFinished or shutdownMicroprocess type messages are sent by this component between one file and the next.

FixedRateControlledReusableFileReader

A reusable file reader component that reads data from files at a fixed rate. It is based on a Carousel component.

Send it a new filename and it will start reading from that file. Do this as many times as you like.

Example Usage

Read data from a sequence of files, at 10 lines a second:

```
playlist = Chooser(["file1", "file2", "file3", ... ])
reader = FixedRateControlledReusableFileReader("lines", rate=10, chunksize=1)

playlist.link( (reader, "requestNext"), (playlist,"inbox") )
playlist.link( (playlist,"outbox"), (reader, "next") )

reader.activate()
playlist.activate()
```

Or, with the Control-Signal path linked up properly, using the JoinChooserToCarousel prefab:

```
playlist = Chooser(["file1", "file2", "file3", ... ])
reader = FixedRateControlledReusableFileReader("lines", rate=10, chunksize=1)

playlistreader = JoinChooserToCarousel(playlist, reader).activate()
```

More detail

The rate control is performed by a ByteRate_RequestControl component. The rate arguments should be those that are accepted by this component.

This component will terminate if it receives a shutdownMicroprocess or producerFinished message on its "control" inbox. The message will be passed on out of its "signal" outbox.

No producerFinished or shutdownMicroprocess type messages are sent by this component between one file and the next.

Development history

PromptedFileReader - developed as an alternative to ReadFileAdapter - prototyped in /Sketches/filereading/ReadFileAdapter.py

Components

PromptedFileReader

PromptedFileReader(filename[,readmode]) -> file reading component

Creates a file reader component. Reads N bytes/lines from the file when N is sent to its inbox.

Keyword arguments:

- readmode -- "bytes" or "lines"

SimpleReader

SimpleReader(filename[,mode][,buffering]) -> simple file reader

Creates a "SimpleReader" component.

Arguments:

- filename -- Name of the file to read
- mode -- This is the python readmode. Defaults to "r". (you may way "rb" occasionally)
- buffering -- The python buffer size. Defaults to 1. (see http://www.python.org/doc/2.5.2/lib/built-in-funcs.html)

Other

EOF

NO DOCS

FixedRateControlledReusableFileReader

FixedRateControlledReusableFileReader(readmode, rateargs) -> reusable file reader component

A file reading component that can be reused. Based on a carousel - send a filename to the "next" or "inbox" inboxes to start reading from that file.

Data is read at the specified rate.

Keyword arguments: - readmode = "bytes" or "lines" - rateargs = arguments for ByteRate_RequestControl component constructor

RateControlledFileReader

RateControlledFileReader(filename[,readmode][,**rateargs]) -> constant rate file reader

Creates a PromptedFileReader already linked to a ByteRate_RequestControl, to control the rate of file reading.

Keyword arguments:

- readmode -- "bytes" or "lines"

- rateargs -- arguments for ByteRate_RequestControl component constructor

RateControlledReusableFileReader

RateControlledReusableFileReader(readmode) -> rate controlled reusable file reader component.

A file reading component that can be reused. Based on a Carousel - send (filename, rateargs) to the "next" inbox to start reading from that file at the specified rate.

- rateargs are the arguments for a ByteRate_RequestControl component.

Keyword arguments: - readmode = "bytes" or "lines"

ReusableFileReader

ReusableFileReader(readmode) -> reusable file reader component.

A file reading component that can be reused. Based on a Carousel - send a filename to the "next" inbox to start reading from that file.

Must be prompted by another component - send the number of bytes/lines to read to the "inbox" inbox.

Keyword arguments: - readmode = "bytes" or "lines"

Kamaelia.File.TriggeredFileReader

Triggered File Reader

This component accepts a filepath as an "inbox" message, and outputs the contents of that file to "outbox". All requests are processed sequentially.

This component does not terminate.

Drawback - this component uses blocking file reading calls but does not run on its own thread.

Components

TriggeredFileReader

TriggeredFileReader() -> component that reads arbitrary files

Kamaelia.File.UnixPipe

This is a deprecation stub, due for later removal.

Other

Pipethrough

NO DOCS

Kamaelia.File.UnixProcess

UnixProcess

Launch another unix process and communicate with it via its standard input and output, by using the "inbox" and "outbox" of this component.

Example Usage

The purpose behind this component is to allow the following to occur:

```
Pipeline(
  dataSource(),
  UnixProcess("command", *args),
  dataSink(),
).run()
```

How to use it

More specificaly, the longer term interface of this component will be:

UnixProcess:

- inbox - data recieved here is sent to the program's stdin
- outbox - data sent here is from the program's stdout
- control - at some point we'll define a mechanism for describing control messages - these will largely map to SIG* messages though. We also need to signal how we close our writing pipe. This can happen using the normal producerFinished message.
- signal - this will be caused by things like SIGPIPE messages. What this will look like is yet to be defined. (Let's see what works first.

Python and platform compatibility

This code is only really tested on Linux.

Initially this will be python 2.4 only, but it would be nice to support older versions of python (eg 2.2.2 - for Nokia mobiles).

For the moment I'm going to send STDERR to dev null, however things won't stay that way.

Components

UnixProcess

NO DOCS

Other

Chargen

NO DOCS

ChargenComponent

NO DOCS

Pipethrough

NO DOCS

makeNonBlocking

NO DOCS

run_command

NO DOCS

Kamaelia.File.UnixProcess2

Unix sub processes with communication through pipes

UnixProcess2 allows you to start a separate process and send data to it and receive data from it using the standard input/output/error pipes and optional additional named pipes.

This component works on *nix platforms only. It is almost certainly not Windows compatible. Tested only under Linux.

How is this different to UnixProcess?

UnixProcess2 differs from UnixProcess in the following ways:

- UnixProcess2 does not drop data if there is a backlog.
- UnixProcess2 allows you to set up extra named input and output pipes.
- UnixProcess2 supports sending data to size limited inboxes. The subprocess will be blocked if its output is going into an inbox that is full. This enables you to use size limited inboxes to regulate the subprocess.
- UnixProcess2 does not take data from its inboxes until it is able to deliver it to the subprocess. If the inboxes are set to be size limiting, this can therefore be used to limit the rate of execution of upstream components.

Example Usage

Using the 'wc' word count GNU util to count the number of lines in some data:

```
Pipeline( RateControlledFileReader(filename, ... ),
          UnixProcess2("wc -l"),
          ConsoleEchoer(),
        ).run()
```

Feeding separate audio and video streams to ffmpeg, and taking the encoded output:

```
Graphline(
    ENCODER = UnixProcess2( "ffmpeg -i audpipe -i vidpipe -",
                            inpipes = { "audpipe":"audio",
                                        "vidpipe":"video",
                                      },
                            boxsizes = { "audio":2, "video":2 }
                          ),
    VIDSOURCE = MaxSpeedFileReader(...),
    AUDSOURCE = MaxSpeFileReader(...),
    SINK = SimpleFileWriter("encodedvideo"),
    linkages = {
        ("VIDSOURCE","outbox") : ("ENCODER","video"),
        ("AUDSOURCE","outbox") : ("ENCODER","audio"),
        ("ENCODER","outbox") : ("SINK", "inbox"),
        }
    ).run()
```

Behaviour

At initialisation, specify:

- the command to invoke the sub process
- the size limit for internal buffers
- additional named input and output pipes
- box size limits for any input pipe's inbox, including "inbox" for STDIN

Named input pipes must all use different inbox names. They must not use "inbox" or "control". Named output pipes may use any outbox name they wish. More than one named ouput pipe can use the same outbox, including "outbox".

The pipe files needed for named pipes are created automatically at activation and are deleted at termination.

Activate UnixProcess2 and the sub process will be started. Use the inboxes and outboxes of UnixProcess2 to communicate with the sub process. For example:

```
UnixProcess2( "ffmpeg -i /tmp/inpipe -f wav /tmp/outpipe",
```

```
            inpipes  = { "/tmp/inpipe" :"in"  },
            outpipes = { "/tmp/outpipe":"out" },
          )

       ------------------------------------------------
      |               UnixProcess2                     |
      |             ------------------------           |
      |            |   "ffmpeg -i ..."     |           |
---> "inbox" ---> STDIN                  STDOUT ---> "outbox" --->
      |            |                     STDERR ---> "error"  --->
      |            |                        |           |
---> "in"    ---> "/tmp/inpipe"  "/tmp/outpipe" ---> "out"    --->
      |            |________________________|           |
      |________________________________________________|
```

Send binary string data to the "inbox" inbox and it will be sent to STDIN of the proces.

Binary string data from STDOUT and STDERR is sent out of the "outbox" and "error" outboxes respectively.

To send to a named pipe, send binary string data to the respective inbox you specified.

Data written by the sub process to a named output pipe will be sent out of the respective outbox.

The specified buffering size sets a maximum limit on the amount of data that can be buffered on the inputs and outputs to and from the sub process. It also determines the chunk size in which data coming from the sub process will emerge. Note therefore that data may languish in an output buffer indefinitely until the process terminates. Do not assume that data coming from a sub process will emerge the moment it is generated.

UnixProcess2 will leave data in its inboxes until it is able to send it to the required pipe. If a destination box is full (noSpaceInBox exception) then UnixProces will wait and retry until it succeeds in sending the data.

If the sub process closes its pipes (STDIN, STDOUT, STDERR) then UnixProcess2 will close its named input and output pipes too, and will send a producerFinished message out of its "signal" outbox then immediately terminate.

If a producerFinished message is received on the "control" inbox, UnixProcess2 will finish passing any pending data waiting in inboxes into the sub process and will finish passing on any pending data waiting to come from the sub process onto destination outboxes. Once this is complete, UnixProcess2 will close all pipes and send a producerFinished message out of its "signal" outbox and immediately terminate.

If a shutdownMicroprocess message is received on the "control" inbox, UnixProcess2 will close all pipes as soon as possible and send the message on out of its

"signal" outbox and immediately terminate.

How does it work?

The UnixProcess2 component itself is primarily just the initiator of the sub process and a container for other child components that handle the actual I/O with pipes. It uses _ToFileHandle and _FromFileHandle components for each input and output pipe respectively.

For each specified named pipe, the specified pipe file is created if required (using mkfifo).

The shutdown signalling boxes of all child components are daisy chained. Shutdown messages sent to the "control" inbox of UnixProcess2 are routed to the "control" inbox of the component handling STDIN. The shutdown message is then propagated to named output pipes and then named input pipes.

If STDOUT close it causes STDERR to close. If STDERR closes then the shutdown message is propagated to STDIN and then onto named pipes as described above.

When the process exits, it is assumed the STDIN, STDOUT and STDERR will close by themselves in due course. However an explicit shutdown message is sent to the named pipes.

XXX Fix Me

If UnixProcess2 is terminated by receiving shutdown messages, it doesn't currently explicitly terminate the sub process.

Components

UnixProcess2

> UnixProcess2(command[,buffersize][,outpipes][,inpipes][,boxsizes]) -> new UnixProcess2 component.
>
> Starts the specified command as a separate process. Data can be sent to stdin and received from stdout. Named pipes can also be created for extra channels to get data to and from the process.
>
> Keyword arguments:

```
- command     -- command line string that will invoke the subprocess
- buffersize  -- bytes size of buffers on the pipes to and from the process (default=32
- outpipes    -- dict mapping named-pipe-filenames to outbox names (default={})
- inpipes     -- dict mapping named-pipe-filenames to inbox names (default={})
- boxsizes    -- dict mapping inbox names to box sizes (default={})
```

Other

makeNonBlocking

Set a file handle to non blocking behaviour on read & write

Kamaelia.File.WholeFileWriter

Whole File Writer

This component accepts file creation jobs and signals the completion of each jobs. Creation jobs consist of a list [filename, contents] added to "inbox". Completion signals consist of the string "done" being sent to "outbox".

All jobs are processed sequentially.

This component does not terminate.

Components

WholeFileWriter

> WholeFileWriter() -> component that creates and writes files
>
> Uses [filename, contents] structure to file creation messages in "inbox"

Kamaelia.File.Writing

Simple File Writer

This component writes any data it receives to a file.

Example Usage

Copying a file:

```
from Kamaelia.File.Writing import SimpleFileWriter

Pipeline(RateControlledFileReader("sourcefile",rate=1000000),
         SimpleFileWriter("destinationfile")
        ).activate()
```

More detail

Any data sent to this component's inbox is written to the specified file. Any existing file with the same name is overwritten.

The file is opened for writing when the component is activated, and is closed when it shuts down.

This component terminates, closing the file, if it receives a shutdownMicroprocess or producerFinished message on its "control" inbox. The message is passed on out of its "signal" outbox.

Development history

SimpleFileWriter - prototyped in /Sketches/filereading/WriteFileAdapter.py

Components

SimpleFileWriter

SimpleFileWriter(filename) -> component that writes data to the file

Writes any data sent to its inbox to the specified file.

Kamaelia.IPC

Other

newCSA

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

newExceptional

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

newReader

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

newServer

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

newWriter

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

removeExceptional

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

removeReader

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

removeWriter

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

serverShutdown

Message to indicate that the server should shutdown

shutdownCSA

Helper class to notify of new CSAs as they are created. newCSA.object will return the CSA.

socketShutdown

Message to indicate that the network connection has been closed.

Kamaelia.Internet.ConnectedSocketAdapter

Talking to network sockets

A Connected Socket Adapter (CSA) component talks to a network server socket. Data is sent to and received from the socket via this component's inboxes and outboxes. A CSA is effectively a wrapper for a socket.

Most components should not need to create CSAs themselves. Instead, use components such as TCPClient to make an outgoing connection, or TCPServer or SimpleServer to be a server that responds to incoming connections.

Example Usage

See source code for TCPClient to see how Connected Socket Adapters can be used.

See also

- TCPClient -- for making a connection to a server
- TCPServer --
- SimpleServer -- a prefab chassis for building a server

How does it work?

A CSA is usually created either by a component such as TCPClient that wants to establish a connection to a server; or by a primary listener socket - a component acting as a server - listening for incoming connections from clients.

The socket should be set up and passed to the constructor to make the CSA.

Incoming data, read by the CSA, is sent out of its "outbox" outbox as strings containing the received binary data. Send data by sending it, as strings, to the "inbox" outbox.

The CSA expects to be wired to a component that will notify it when new data has arrived at its socket (by sending an Axon.Ipc.status message to its "ReadReady" inbox. This is to allow the CSA to sleep rather than busy-wait or blocking when waiting for new data to arrive. Typically this is the Selector component.

This component will terminate (and close its socket) if it receives a producerFinished or shutdownMicroprocess message on its "control" inbox.

When this component terminates, it sends a socketShutdown(socket) message out of its "CreatorFeedback" outbox and a shutdownCSA((selfCSA,self.socket)) message out of its "signal" outbox.

The message sent to "CreatorFeedback" is to notify the original creator that the socket is now closed and that this component should be unwired.

The message sent to the "signal" outbox serves to notify any other component involved - such as the one feeding notifications to the "ReadReady" inbox (eg. the Selector component).

Components

ConnectedSocketAdapter

> ConnectedSocketAdapter(socket) -> new CSA component wrapping specified socket
>
> Component for communicating with a socket. Send to its "inbox" inbox to send data, and receive data from its "outbox" outbox.
>
> "ReadReady" inbox must be wired to something that will notify it when new data has arrived at the socket.

Other

SSLSocket

NO DOCS

crashAndBurn

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

whinge

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Kamaelia.Internet.Multicast_receiver

Simple multicast receiver

A simple component for receiving packets in the specified multicast group.

Remember that multicast is an unreliable connection - packets may be lost, duplicated or reordered.

Example Usage

Receiving multicast packets from group address 1.2.3.4 port 1000 and displaying them on the console:

```
Pipeline( Multicast_receiver("1.2.3.4", 1000),
          ConsoleEchoer()
        ).activate()
```

The data emitted by Multicast_receiver (and displayed by ConsoleEchoer) is of the form (source_address, data).

More detail

Data received from the multicast group is emitted as a tuple: (source_addr, data) where data is a string of the received data.

This component ignores anything received on its "control" inbox. It is not yet possible to ask it to shut down. It does not terminate.

Multicast groups do not 'shut down', so this component never emits any signals on its "signal" outbox.

Components

Multicast_receiver

Multicast_receiver(address, port) -> component that receives multicast traffic.

Creates a component that receives multicast packets in the given multicast group and sends it out of its "outbox" outbox.

Keyword arguments:

- address -- address of multicast group (string)
- port -- port number

Other

tests

NO DOCS

Kamaelia.Internet.Multicast_sender

Simple multicast sender

A simple component for sending data to a multicast group.

Remember that multicast is an unreliable connection - packets may be lost, duplicated or reordered.

Example Usage

Multicasting a file to group address 1.2.3.4 on port 1000 (local address 0.0.0.0 port 0):

```
Pipeline( RateControlledFileReader("myfile", rate=100000),
          Multicast_sender("0.0.0.0", 0, "1.2.3.4", 1000),
        ).activate()
```

More detail

Data sent to the component's "inbox" inbox is sent to the multicast group.

This component ignores anything received on its "control" inbox. It is not yet possible to ask it to shut down. It does not terminate.

This component never emits any signals on its "signal" outbox.

Components

Multicast_sender

Multicast_sender(local_addr, local_port, remote_addr, remote_port) -> component that sends to a multicast group.

Creates a component that sends data received on its "inbox" inbox to the specified multicast group.

Keyword arguments:

- local_addr -- local address (interface) to send from (string)
- local_port -- local port number
- remote_addr -- address of multicast group to send to (string)
- remote_port -- port number

Other

tests

NO DOCS

Kamaelia.Internet.Multicast_transceiver

Simple multicast transceiver

A simple component for transmitting and receiving multicast packets.

Remember that multicast is an unreliable connection - packets may be lost, duplicated or reordered.

Example Usage

Send a file to, and receive data from multicast group address 1.2.3.4 port 1000:

```
Pipeline( RateControlledFileReader("myfile", rate=100000),
          Multicast_transceiver("0.0.0.0", 0, "1.2.3.4", 1000),
        ).activate()

Pipeline( Multicast_transceiver("0.0.0.0", 1000, "1.2.3.4", 0)
          ConsoleEchoer()
        ).activate()
```

Or:

```
Pipeline( RateControlledFileReader("myfile", rate=100000),
          Multicast_transceiver("0.0.0.0", 1000, "1.2.3.4", 1000),
          ConsoleEchoer()
        ).activate()
```

The data emitted by Multicast_transciever (and displayed by ConsoleEchoer) is of the form (source_address, data).

More detail

Data sent to the component's "inbox" inbox is sent to the multicast group.

Data received from the multicast group is emitted as a tuple: (source_addr, data) where data is a string of the received data.

This component ignores anything received on its "control" inbox. It is not yet possible to ask it to shut down. It does not terminate.

Multicast groups do not 'shut down', so this component never emits any signals on its "signal" outbox.

Why a transciever component?

Listens for packets in the given multicast group. Any data received is sent to the receiver's outbox. The logic here is likely to be not quite ideal. When complete though, this will be preferable over the sender and receiver components since it models what multicast really is rather than what people tend to think it is.

Components

Multicast_transceiver

Multicast_transciever(local_addr, local_port, remote_addr, remote_port) -> component that send and receives data to/from a multicast group.

Creates a component that sends data received on its "inbox" inbox to the specified multicast group; and sends to its "outbox" outbox tuples of the form (src_addr, data) containing data received.

Keyword arguments:

- local_addr -- local address (interface) to send from (string)
- local_port -- port number
- remote_addr -- address of multicast group (string)
- remote_port -- port number

Other

tests

NO DOCS

Kamaelia.Internet.Selector

NOTIFICATION OF SOCKET AND FILE EVENTS

The Selector component listens for events on sockets and sends out notifications. It is effectively a wrapper around the unix 'select' statement. Components request that the Selector component notify them when a supplied socket or file object is ready.

The selectorComponent is a service that registers with the Coordinating Assistant Tracker (CAT).

NOTE: The behaviour and API of this component changed in Kamaelia 0.4 and is likely to change again in the near future.

Example Usage

See the source code for TCPClient for an example of how the Selector component can be used.

How does it work?

Selector is a service. Obtain it by calling the static method Selector.getSelectorService(...). Any existing instance will be returned, otherwise a new one is automatically created.

This component ignores anything sent to its "inbox" and "control" inboxes. This component does not terminate.

Register socket or file objects with the selector, to receive a one-shot notification when that file descriptor is ready. The file descriptor can be a python file object or socket object. The notification is one-shot - meaning you must resubmit your request every time you wish to receive a notification.

Ensure you deregister the file object when closing the file/socket. You may do this even if you have already received the notification. The Selector component will be unable to handle notifications for any other descriptors if it still has a registered descriptor that has closed.

Register for a notification by sending an one of the following messages to the "notify" inbox, as returned by Selector.getSelectorService():

- Kamaelia.KamaeliaIpc.newReader(caller, (component,inboxname), descriptor)
- Kamaelia.KamaeliaIpc.newWriter(caller, (component,inboxname), descriptor)

- Kamaelia.KamaeliaIpc.newExceptional(caller, (component,inboxname), descriptor)

Choose which as appropriate:

- a newReader() request will notify when there is data ready to be read on the descriptor
- a newWriter() request will notify when writing to the descriptor will not block.
- a newExceptional() request will notify when an exceptional event occurs on the specified descriptor.

Selector will notify the taret component by sending the file/socket descriptor object to the target inbox the component provided. It then automatically deregisters the descriptor, unlinking from the target component's inbox.

For a given descriptor for a given type of event (read/write/exceptional) only one notification will be sent when the event occurs. If multiple notification requests have been received, only the first is listened to; all others are ignored.

Of course, once the notification as happened, or someone has requested that descriptor be deregistered, then someone can register for it once again.

Deregister by sending on of the following messages to the "notify" inbox of Selector:

- Kamaelia.KamaeliaIpc.removeReader(caller, descriptor)
- Kamaelia.KamaeliaIpc.removeWriter(caller, descriptor)
- Kamaelia.KamaeliaIpc.removeExceptional(caller, descriptor)

It is advisable to send a deregister message when the corresponding file descriptor closes, in case you registered for a notification, but it has not occurred.

Components

Selector

Selector() -> new Selector component

Use Selector.getSelectorService(...) in preference as it returns an existing instance, or automatically creates a new one.

Other

EXCEPTIONALS

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.___int___(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

FAILHARD

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

READERS

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.___int___(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

WRITERS

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.___int___(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

timeout

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.___int___(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

Kamaelia.Internet.Simulate.BrokenNetwork

Broken Network Simulation

Components to simulate properties of an unreliable network connection. Specifically: out of order delivery, duplication, and loss of packets.

Original author: Tom Gibson (whilst at BBC)

Example Usage

Testing a forward-error correction scheme to cope with an unreliable network:

```
Pipeline( RateControlledFileReader("sourcefile",rate=1000000),
          MyForwardErrorCorrector(),
          Duplicate(),
          Throwaway(),
          Reorder(),
          MyErrorRecoverer(),
          SimpleFileWriter("receiveddata")
        ).activate()
```

Duplicate, Throwaway, Reorder

These three components all receive data and, respectively, randomly duplicate packets, re-order packets or throw some packets away.

They can be used to simulate the effects of multicast delivery over wireless or a WAN.

More details

These component all receive data on their "inbox" inbox and send it on to their "outbox" outbox. However, they will sometimes tamper with the data in the manners described!

None of these components terminate when sent shutdown messages.

History

This was used for the development of a simple recovery protocol. The actual version in use replaces the string2tuple and tuple2string code (in sketches in tomg.py, omitted here), with something more robust.

Components

Duplicate

Duplicate() -> new component.

This component passes on data it receives. Sometimes it randomly duplicates items.

Throwaway

Throwaway() -> new component.

This component passes on data it receives, but sometimes it doesn't!

Reorder

Reorder() -> new component

This component passes on data it receives, but will sometimes jumble it up (reordering it).

Kamaelia.Internet.SingleServer

This is a simpler server than the SimpleServer component. Specifically it only allows a single connection to occur at a time. Any data received on that connection is sent to the component's outbox, and any data received on its inbox is sent to the connection. When a connection closes, it sends a producerFinished signal.

TODO: If there is already a connection, then any new connections are shutdown. It would be better if they weren't accepted in the first place, but that requires changes to TCPServer.

FastRestartSingleServer is exactly the same as SingleServer, expect its default is to set appropriate socket options to allow the server to restart instantly. This isn't the traditional default because in kamaelia we prefer to follow the defaults of the OS, rather than local defaults.

Components

SingleServer

NO DOCS

FastRestartSingleServer

NO DOCS

echo

NO DOCS

Kamaelia.Internet.TCPClient

Simple TCP Client

This component is for making a TCP connection to a server. Send to its "inbox" inbox to send data to the server. Pick up data received from the server on its "outbox" outbox.

Example Usage

Sending the contents of a file to a server at address 1.2.3.4 on port 1000:

```
Pipeline( RateControlledFileReader("myfile", rate=100000),
          TCPClient("1.2.3.4", 1000),
        ).activate()
```

Example Usage - SSL

It is also possible to cause the TCPClient to switch into SSL mode. To do this you send it a message on its "makessl" inbox. It is necessary for a number of protocols to be able to switch between non-ssl and ssl, hence this approach rather than simply saying "ssl client" or "non-ssl client":

```
Graphline(
       MAKESSL = OneShot(" make ssl "),
       CONSOLE = ConsoleReader(),
       ECHO = ConsoleEchoer(),
       CONNECTION = TCPClient("kamaelia.svn.sourceforge.net", 443),
       linkages = {
           ("MAKESSL", "outbox"): ("CONNECTION", "makessl"),
           ("CONSOLE", "outbox"): ("CONNECTION", "inbox"),
           ("CONNECTION", "outbox"): ("ECHO", "inbox"),
       }
)
```

How does it work?

TCPClient opens a socket connection to the specified server on the specified port. Data received over the connection appears at the component's "outbox" outbox as strings. Data can be sent as strings by sending it to the "inbox" inbox.

An optional delay (between component activation and attempting to connect) can be specified. The default is no delay.

It creates a ConnectedSocketAdapter (CSA) to handle the socket connection and registers it with a selectorComponent so it is notified of incoming data. The selectorComponent is obtained by calling selectorComponent.getSelectorService(...) to look it up with the local Coordinating Assistant Tracker (CAT).

TCPClient wires itself to the "CreatorFeedback" outbox of the CSA. It also wires its "inbox" inbox to pass data straight through to the CSA's "inbox" inbox, and its "outbox" outbox to pass through data from the CSA's "outbox" outbox.

Socket errors (after the connection has been successfully established) may be sent to the "signal" outbox.

This component will terminate if the CSA sends a socketShutdown message to its "CreatorFeedback" outbox.

This component will terminate if a shutdownMicroprocess or producerFinished message is sent to its "control" inbox. This message is forwarded onto the CSA. TCPClient will then wait for the CSA to terminate. It then sends its own shutdownMicroprocess message out of the "signal" outbox.

Components

TCPClient

TCPClient(host,port[,delay]) -> component with a TCP connection to a server.

Establishes a TCP connection to the specified server.

Keyword arguments:

- host -- address of the server to connect to (string)
- port -- port number to connect on
- delay -- delay (seconds) after activation before connecting (default=0)

Kamaelia.Internet.TCPServer

TCP Socket Server

A building block for creating a TCP based network server. It accepts incoming connection requests and sets up a component to handle the socket which it then passes on.

This component does not handle the instantiation of components to handle an accepted connection request. Another component is needed that responds to this component and actually does something with the newly established connection. If you require a more complete implementation that does this, see Kamaelia.Internet.SingleServer or Kamaelia.Chassis.ConnectedServer.

Example Usage

See Kamaelia.Internet.SingleServer or Kamaelia.Chassis.ConnectedServer for examples of how this component can be used.

The process of using a TCPServer component can be summarised as: - Create a TCP Server - Wait for newCSA messages from the TCP Server's "protocolHandlerSignal" outbox - Send what you like to CSA's, ensure you recieve data from the CSAs - Send producerFinished to the CSA to shut it down.

How does it work?

This component creates a listener socket, bound to the specified port, and registers itself and the socket with a selectorComponent so it is notified of incoming connections. The selectorComponent is obtained by calling selectorComponent.getSelectorService(...) to look it up with the local Coordinating Assistant Tracker (CAT).

When the it recieves a new connection it performs an accept, and creates a ConnectedSocketAdapter (CSA) to handle the activity on that connection.

The CSA is passed in a newCSA(self,CSA) message to TCPServer's "protocolHandlerSignal" outbox.

The CSA is also registered with the selector service by sending it a newCSA(self,(CSA,sock)) message, to ensure the CSA is notified of incoming data on its socket.

The client component(s) using the TCPServer should handle the newly created CSA passed to it in whatever way it sees fit.

If a socketShutdown message is received on the "_feedbackFromCSA" inbox, then a shutdownCSA(self, CSA) message is sent to TCPServer's "protocolHandlerSignal" outbox to notify the client component that the connection has closed.

Also, a shutdownCSA(self, (CSA, sock)) message is sent to the selector service to deregister the CSA from receiving notifications.

This component does not terminate.

Components

TCPServer

> TCPServer(listenport) -> TCPServer component listening on the specified port.
>
> Creates a TCPServer component that accepts all connection requests on the specified port.

Kamaelia.Internet.ThreadedTCPClient

Simple Threaded TCP Client

This component is for making a TCP connection to a server. Send to its "inbox" inbox to send data to the server. Pick up data received from the server on its "outbox" outbox.

This component runs in its own separate thread so it can block on the socket connection. This was written because some platforms that don't support non-blocking calls to read/write data from sockets (eg. Python for Nokia Series-60).

Example Usage

Sending the contents of a file to a server at address 1.2.3.4 on port 1000:

```
Pipeline( RateControlledFileReader("myfile", rate=100000),
          ThreadedTCPClient("1.2.3.4", 1000),
        ).activate()
```

How does it work?

The component opens a socket connection to the specified server on the specified port. Data received over the connection appears at the component's "outbox" outbox as strings. Data can be sent as strings by sending it to the "inbox" inbox.

The component will shutdown in response to a producerFinished message arriving on its "control" inbox. The socket will be closed, and a socketShutdown message will be sent to the "signal" outbox.

All socket errors exceptions are passed on out of the "signal" outbox. This will always result in the socket being closed (if open) and a socketShutdown message also being sent to the "signal" outbox (after the exception).

It does not use a ConnectedSocketAdapter, instead handling all socket communications itself.

The compnent is based on Axon.ThreadedComponent.threadedcomponent

Components

ThreadedTCPClient

ThreadedTCPClient(host,port[,chargen][,initalsendmessage]) -> threaded component with a TCP connection to a server.

Establishes a TCP connection to the specified server.

Keyword arguments:

- host -- address of the server to connect to (string)

- port -- port number to connect on
- initialsendmessage -- to be send immediately after connection is established (default=None)

Kamaelia.Internet.TimeOutCSA

This module provides a set of components and convienience functions for making a CSA time out. To use it, simply call the function InactivityChassis with a timeout period and the CSA to wrap. This will send a producerFinished signal to the CSA if it does not send a message on its outbox or CreatorFeedback boxes before the timeout expires.

ResettableSender

This component is a simple way of makinga timeout event occur. If it receives nothing after 5 seconds, a "NEXT" message is sent.

Example Usage

A tcp client that connects to a given host and port and prints and expects to receive *something* at more frequent than 5 seconds intervals. If it does not, it prints a rude message the first time this happens:

```
Pipeline(
    TCPClient(HOST, PORT),
    ResettableSender(),
    PureTransformer(lambda x : "Oi, wake up. I have received nothing for 5 seconds!"),
    ConsoleEchoer(),
    ).run()
```

More detail

By default, ResettableSender is set up to send a timeout message "NEXT" out of its "outbox" outbox within 5 seconds. However, if it receives any message on its "inbox" inbox then the timer is reset.

Once the timeout has occurred this component terminates. It will therefore only ever generate one "NEXT" message, even if multiple timeouts occur.

Termination is silent - no messages to indicate shutdown are sent out of the "signal" outbox. This component ignores any input on its "control" inbox - including shutdown messages.

ActivityMonitor

ActivityMonitor monitors up to four streams of messages passing through it and, in response, sends messages out of its "observed" outbox to indicate that there has been activity.

This is intended to wrap a component such that it may be monitored by another component.

Example Usage

Type a line at the console, and your activity will be 'observed' - a 'RESET' message will appear as well as the line you have typed:

```
Graphline(
   MONITOR = ActivityMonitor(),
   OUTPUT  = ConsoleEchoer(),
   INPUT   = ConsoleReader(),
   linkages = {
       ("INPUT",   "outbox") : ("MONITOR", "inbox"),
       ("MONITOR", "outbox") : ("OUTPUT", "inbox"),

       ("MONITOR", "observed") : ("OUTPUT", "inbox"),
   }
).run()
```

Usage

To use it, connect any of the three inboxes to outboxes on the component to be monitored.

The messages that are sent to those inboxes will be forwarded to their respective outbox. For example, suppose "HELLO" was received on 'inbox2'. self.message ("RESET" by default) will be sent on the 'observed' outbox and "HELLO" will be sent out on 'outbox2'.

ActivityMonitor will shut down upon receiving a producerFinished or shutdownCSA message on its control inbox.

Please note that this can't wrap adaptive components properly as of yet.

More detail

Any message sent to the "inbox", "inbox2", "inbox3" or "control" inboxes are immediately forwarded on out of the "outbox", "outbox2", "outbox3" or "signal" outboxes respectively.

Whenever this happens, a "RESET" message (the default) is sent out of the "observed" outbox.

For every batch of messages waiting at the three inboxes that get forwarded on out of their respective outboxes, *only one* "RESET" message will be sent. This should therefore only be used as a general indication of activity, not as a means of counting every individual message passing through this component.

Any message sent to the "control" inbox is also checked to see if it is a shutdownCSA or producerFinished message. If it is one of these then it will still be forwarded on out of the "signal" outbox, but will also cause ActivityMonitor to immediately terminate.

Any messages waiting at any of the inboxes (including the "control" inbox) at the time the shutdown triggering message is received will be sent on before termination.

Other

ActivityMonitor

> This is intended to wrap a component such that it may be monitored by another component. To use it, connect any of the three inboxes to outboxes on the component to be monitored. The messages that are sent to those inboxes will be forwarded to their respective out box. For example, suppose "HELLO" was received on 'inbox2'. self.message ("RESET" by default) will be sent on the 'observed' outbox and "HELLO" will be sent out on 'outbox2'. An ActivityMonitor will shut down upon receiving a producerFinished or shutdownCSA message on its control inbox.
>
> Please note that this can't wrap adaptive components properly as of yet.

EchoServer

NO DOCS

ExtendedInactivity

> A factory function that will return a new function object that will create an InactivityChassis wrapped in someclass without storing the debug and timeout arguments.

InactivityChassis

> This convenience function will link a component up to an ActivityMonitor and a ResettableSender that will emit a producerFinished signal within timeout seconds if the component does not send any messages on its outbox or CreatorFeedback boxes. To link the specified component to an external component simply link it to the returned chassis's outbox or CreatorFeedback outboxes.

NoActivityTimeout

This is a factory function that will return a new function object that will produce an InactivityChassis with the given timeout and debug values. The values specified in timeout, debug, and someclass will be used in all future calls to the returned function object.

someclass - the class to wrap in an InactivityChassis timeout - the amount of time to wait before sending the shutdown signal debug - the debugger to use

PeriodicWakeup

This component is basically just a Cheap and cheerful clock that may be shut down. You may specify the interval and message to be sent. The component will sleep for self.interval seconds and then emit self.message before checking its control inbox for any shutdown messages and then going back to sleep.

ResettableSender

This component represents a simple way of making a timed event occur. By default, it is set up to send a timeout message "NEXT" within 5 seconds. If it receives a message on its inbox, the timer will be reset. This component will ignore any input on its control inbox.

WakeableIntrospector

This component serves to check if it is the only component in the scheduler other than a graphline and PeriodicWakeup. If it is, it will send out a producerFinished signal on its signal outbox. It will ignore any input on its inbox or control box, however it is useful to send it a message to its inbox to wake it up.

Kamaelia.Internet.UDP

Simple UDP components

These components provide simple support for sending and receiving UDP packets.

Note that this components are deemed somewhat experimental.

Example Usage

Send console input to port 1500 of myserver.com and receive packets locally on port 1501 displaying their contents (and where they came from) on the console:

```
from Kamaelia.Chassis.Pipeline import Pipeline
```

```
from Kamaelia.Util.Console import ConsoleEchoer
from Kamaelia.Util.Console import ConsoleReader
from Kamaelia.Internet.UDP import SimplePeer

Pipeline( ConsoleReader(),
          SimplePeer("127.0.0.1", 1501, "myserver.com", 1500),
          ConsoleEchoer(),
        ).run()
```

Sends data from a data source as UDP packets, changing between 3 different destinations, once per second:

```
class DestinationSelector(component):
    def main(self):
        while 1:
            for dest in [ ("server1.com",1500),
                          ("server2.com",1500),
                          ("server3.com",1500), ]:
                self.send(dest,"outbox")
            next=time.time()+1.0
            while time.time() < next:
                yield 1

Graphline(         SOURCE = MyDataSource(),
    SELECT = DestinationSelector(),
    UDP    = TargettedPeer(),
    linkages = {
        ("SOURCE", "outbox") : ("UDP", "inbox"),
        ("SELECT", "outbox") : ("UDP", "target"),
    }
).run()
```

Send UDP packets containing "hello" to several different servers, all on port 1500:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.DataSource import DataSource
from Kamaelia.Internet.UDP import PostboxPeer

Pipeline(
    DataSource( [ ("myserver1.com",1500,"hello"),
                  ("myserver2.com",1500,"hello"),
                  ("myserver3.com",1500,"hello"),
                ]
                ),
    PostboxPeer(),
).run()
```

Behaviour

When any of these components receive a UDP packet on the local address and port they are bound to; they send out a tuple (data,(host,port)) out of their "outbox" outboxes. 'data' is a string containing the payload of the packet. (host,port) is the address of the sender/originator of the packet.

SimplePeer is the simplest to use. Any data sent to its "inbox" inbox is sent as a UDP packet to the destination (receiver) specified at initialisation.

TargettedPeer behaves identically to SimplePeer; however the destination (receiver) it sends UDP packets to can be changed by sending a new (host,port) tuple to its "target" inbox.

PostboxPeer does not have a fixed destination (receiver) to which it sends UDP packets. Send (host,port,data) tuples to its "inbox" inbox to arrange for a UDP packet containing the specified data to be sent to the specified (host,port).

None of these components terminate. They ignore any messages sent to their "control" inbox and do not send anything out of their "signal" outbox.

Implementation Details

SimplePeer, TargettedPeer and PostboxPeer are all derived from the base class BasicPeer. BasicPeer provides some basic code for receiving from a socket.

Although technically BasicPeer is a component, it is not a usable one as it does not implement a main() method.

Components

SimplePeer

SimplePeer([localaddr][,localport][,receiver_addr][,receiver_port]) -> new SimplePeer component.

A simple component for receiving and transmitting UDP packets. It binds to the specified local address and port - from which it will receive packets and sends packets to a receiver on the specified address and port.

Arguments:

- localaddr -- Optional. The local addresss (interface) to bind to. (default="0.0.0.0")
- localport -- Optional. The local port to bind to. (default=0)
- receiver_addr -- Optional. The address the receiver is bound to - to which packets will be sent. (default="0.0.0.0")
- receiver_port -- Optional. The port the receiver is bound on - to which packets will be sent. (default=0)

TargettedPeer

TargettedPeer([localaddr][,localport][,receiver_addr][,receiver_port]) -> new TargettedPeer component.

A simple component for receiving and transmitting UDP packets. It binds to the specified local address and port - from which it will receive packets and sends packets to a receiver on the specified address and port.

Can change where it is sending to by sending the new (addr,port) receiver address to the "target" inbox.

Arguments:

- localaddr -- Optional. The local addresss (interface) to bind to. (default="0.0.0.0")
- localport -- Optional. The local port to bind to. (default=0)
- receiver_addr -- Optional. The address the receiver is bound to - to which packets will be sent. (default="0.0.0.0")
- receiver_port -- Optional. The port the receiver is bound on - to which packets will be sent. (default=0)

PostboxPeer

PostboxPeer([localaddr][,localport]) -> new PostboxPeer component.

A simple component for receiving and transmitting UDP packets. It binds to the specified local address and port - from which it will receive packets. Sends packets to individually specified destinations

Arguments:

- localaddr -- Optional. The local addresss (interface) to bind to. (default="0.0.0.0")
- localport -- Optional. The local port to bind to. (default=0)

Other

BasicPeer

BasicPeer() -> new BasicPeer component.

Base component from which others are derived in this module. Not properly functional on its own and so *should not be used* directly.

Kamaelia.Internet.UDP_ng

Simple UDP components

These components provide simple support for sending and receiving UDP packets.

NOTE: This set of components really an evolution of those in UDP.py, and is likely to replace those in future.

Example Usage

Send console input to port 1500 of myserver.com and receive packets locally on port 1501 displaying their contents (and where they came from) on the console:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Console import ConsoleEchoer
from Kamaelia.Util.Console import ConsoleReader
from Kamaelia.Internet.UDP import SimplePeer

Pipeline( ConsoleReader(),
          SimplePeer("127.0.0.1", 1501, "myserver.com", 1500),
          ConsoleEchoer(),
        ).run()
```

Sends data from a data source as UDP packets, changing between 3 different destinations, once per second:

```
class DestinationSelector(component):
    def main(self):
        while 1:
            for dest in [ ("server1.com",1500),
                          ("server2.com",1500),
                          ("server3.com",1500), ]:
                self.send(dest,"outbox")
            next=time.time()+1.0
            while time.time() < next:
                yield 1

Graphline(         SOURCE = MyDataSource(),
    SELECT = DestinationSelector(),
    UDP    = TargettedPeer(),
    linkages = {
        ("SOURCE", "outbox") : ("UDP", "inbox"),
        ("SELECT", "outbox") : ("UDP", "target"),
    }
).run()
```

Send UDP packets containing "hello" to several different servers, all on port 1500:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.DataSource import DataSource
from Kamaelia.Internet.UDP import PostboxPeer

Pipeline(
    DataSource( [ ("myserver1.com",1500,"hello"),
                  ("myserver2.com",1500,"hello"),
                  ("myserver3.com",1500,"hello"),
                ]
                ),
    PostboxPeer(),
).run()
```

Behaviour

When any of these components receive a UDP packet on the local address and port they are bound to; they send out a tuple (data,(host,port)) out of their "outbox" outboxes. 'data' is a string containing the payload of the packet. (host,port) is the address of the sender/originator of the packet.

SimplePeer is the simplest to use. Any data sent to its "inbox" inbox is sent as a UDP packet to the destination (receiver) specified at initialisation.

UDPSender and UDPReceiver duplicate the sending and receiving functionality of SimplePeer in seperate components.

TargettedPeer behaves identically to SimplePeer; however the destination (receiver) it sends UDP packets to can be changed by sending a new (host,port) tuple to its "target" inbox.

PostboxPeer does not have a fixed destination (receiver) to which it sends UDP packets. Send (host,port,data) tuples to its "inbox" inbox to arrange for a UDP packet containing the specified data to be sent to the specified (host,port).

All of the components shutdown upon receiving a ShutdownMicroprocess message on their "control" inbox. UDPSender also shuts down upon receiving a ProducerFinished message on its "control" inbox. In this case before it shuts down it sends any data in its send queue, and any data waiting on its inbox.

Implementation Details

All of the UDP components are all derived from the base class BasicPeer. BasicPeer provides some basic code for sending and receiving from a socket.

Although technically BasicPeer is a component, it is not a usable one as it does not implement a main() method.

Components

UDPSender

UDPSender([receiver_addr][,receiver_port]) -> new UDPSender component.

A simple component for transmitting UDP packets. It sends packets received from the "inbox" inbox to a receiver on the specified address and port.

Arguments:

- receiver_addr -- Optional. The address the receiver is bound to - to which packets will be sent. (default="0.0.0.0")
- receiver_port -- Optional. The port the receiver is bound on - to which packets will be sent. (default=0)

UDPReceiver

UDPReceiver([localaddr][,localport]) -> new UDPReceiver component.

A simple component for receiving UDP packets. It binds to the specified local address and port - from which it will receive packets. Packets received are sent to it "outbox" outbox.

Arguments:

- localaddr -- Optional. The local addresss (interface) to bind to. (default="0.0.0.0")
- localport -- Optional. The local port to bind to. (default=0)

SimplePeer

SimplePeer([localaddr][,localport][,receiver_addr][,receiver_port]) -> new SimplePeer component.

A simple component for receiving and transmitting UDP packets. It binds to the specified local address and port - from which it will receive packets and sends packets to a receiver on the specified address and port.

Arguments:

- localaddr -- Optional. The local addresss (interface) to bind to. (default="0.0.0.0")
- localport -- Optional. The local port to bind to. (default=0)
- receiver_addr -- Optional. The address the receiver is bound to - to which packets will be sent. (default="0.0.0.0")
- receiver_port -- Optional. The port the receiver is bound on - to which packets will be sent. (default=0)

TargettedPeer

TargettedPeer([localaddr][,localport][,receiver_addr][,receiver_port]) -> new TargettedPeer component.

A simple component for receiving and transmitting UDP packets. It binds to the specified local address and port - from which it will receive packets and sends packets to a receiver on the specified address and port.

Can change where it is sending to by sending the new (addr,port) receiver address to the "target" inbox.

Arguments:

- localaddr -- Optional. The local addresss (interface) to bind to. (default="0.0.0.0")
- localport -- Optional. The local port to bind to. (default=0)
- receiver_addr -- Optional. The address the receiver is bound to - to which packets will be sent. (default="0.0.0.0")
- receiver_port -- Optional. The port the receiver is bound on - to which packets will be sent. (default=0)

PostboxPeer

PostboxPeer([localaddr][,localport]) -> new PostboxPeer component.

A simple component for receiving and transmitting UDP packets. It binds to the specified local address and port - from which it will receive packets. Sends packets to individually specified destinations

Arguments:

- localaddr -- Optional. The local addresss (interface) to bind to. (default="0.0.0.0")
- localport -- Optional. The local port to bind to. (default=0)

Other

BasicPeer

BasicPeer() -> new BasicPeer component.

Base component from which others are derived in this module. Not properly functional on its own and so *should not be used* directly.

Kamaelia.KamaeliaExceptions

Other

BadRequest

Thrown when parsing a request fails

connectionClosedown

NO DOCS

connectionDied

NO DOCS

connectionDiedReceiving

NO DOCS

connectionDiedSending

NO DOCS

connectionServerShutdown

NO DOCS

socketSendFailure

NO DOCS

Kamaelia.KamaeliaIPC

This is a deprecation stub, due for later removal.

Other

newCSA

NO DOCS

newExceptional

NO DOCS

newReader

NO DOCS

newServer

NO DOCS

newWriter

NO DOCS

notify

NO DOCS

producerFinished

NO DOCS

removeExceptional

NO DOCS

removeReader

NO DOCS

removeWriter

NO DOCS

shutdownCSA

NO DOCS

socketShutdown

NO DOCS

Kamaelia.Protocol.AudioCookieProtocol

Simple "Audio fortune cookie" Protocol Handler

A simple protocol handler that simply sends the contents of a randomly chosen audio file to the client.

Example Usage

:: »> SimpleServer(protocol=AudioCookieProtocol, port=1500).run()

On Linux client: > netcat <server ip> 1500 | aplay -

How does it work?

AudioCookieProtocol creates a ReadFileAdapter and configures it to read the standard output result of running the afortune.pl script, at a fixed rate of 95.2kbit/s.

afortune.pl randomly selects a file and returns its contents.

The ReadFileAdapter's "outbox" outbox is directly wired to pass through to the "outbox" outbox of AudioCookieProtocol.

This component does not terminate.

No EOF/termination indication is given once the end of the file is reached.

Components

AudioCookieProtocol

AudioCookieProtocol([debug]) -> new AudioCookieProtocol component.

A protocol that spits out raw audio data from a randomly selected audio file.

Keyword arguments:

- debug -- Debugging output control (default=0)

Kamaelia.Protocol.EchoProtocol

Simple Echo Protocol

A simple protocol component that echoes back anything sent to it.

It simply copies its input to its output.

Example Usage

A simple server that accepts connections on port 1501, echoing back anything sent to it:

```
>>> SimpleServer(protocol=EchoProtocol, port=1501).run()
```

On a unix/linux client:

```
> telnet <server ip> 1501
Trying <server ip>...
Connected to <server ip>...
hello world, this will be echoed back when I press return (newline)
hello world, this will be echoed back when I press return (newline)
oooh, thats nice!
```

```
oooh, thats nice!
```

How does it work?

The component receives data on its "inbox" inbox and immediately copies it to its "outbox" outbox.

If any message is received on its "control" inbox, the component passes the message onto its "signal" outbox and terminates.

Components

EchoProtocol

> EchoProtocol() -> new EchoProtocol component
>
> Simple component that copies anything sent to its "inbox" inbox to its "outbox" outbox.

Kamaelia.Protocol.EchoProtocolComponent

This is a deprecation stub, due for later removal.

Other

EchoProtocol

NO DOCS

Kamaelia.Protocol.FortuneCookieProtocol

Simple Fortune Cookie Protocol

Simple fortune cookie protocol handler, that emits a fortune cookie (at 40 characters/second) then disconnects.

Example Usage

A simple server that accepts connections on port 1500, sending a fortune cookie to any client that connects:

```
>>> SimpleServer(protocol=FortuneCookieProtocol, port=1500).run()
```

On a unix/linux client:

```
> telnet <server ip> 1500
Trying <server ip>...
Connected to <server ip>...
```

```
"I love Saturday morning cartoons, what classic humour!  This is what
entertainment is all about ... Idiots, explosives and falling anvils."
                -- Calvin and Hobbes, Bill Watterson
Connection closed by foreign host.
>
```

How does it work?

The component gets it's data from the command "fortune -a" - the normal unix command which dumps out a random fortune cookie - read via a ReadFileAdaptor, reading from the command pipe at a constant bit rate. The ReadFileAdaptor is wired up so that data is passed through to the FortuneCookieProtocol's "outbox" outbox.

At the end of the cookie, the FortuneCookieProtocol component terminates, closing the connection (it receives and passes on the producerFinished message generated by the ReadFileAdapter component).

Components

FortuneCookieProtocol

FortuneCookieProtocol([debug]) -> new FortuneCookieProtocol component.

A protocol that spits out a random fortune cookie, then terminates.

Keyword arguments:

- debug -- Debugging output control (default=0)

Kamaelia.Protocol.Framing

Simple data Framing and chunking

A simple set of components for framing data and chunking it, and for reversing the process.

The Framer component frames messages as string, prefixed with a tag (eg. sequence number) and their length. The Chunker component inserts markers into the data stream to identify the start of chunks (eg. frames).

The DeChunker and DeFramer reverse the process.

Example Usage

Framing messages for transport over a stream based connection (eg, TCP):

```
Pipeline(MessageSource(...),   # emits message
         DataChunker(),
         TCPClient("<server ip>", 1500),
        ).activate()
```

And on the server:

```
Pipeline(SingleServer(1500),
         DataDeChunker(),
         MessageReceiver(...)
        ).activate()
```

Packing data for transport over a link that may loose packets:

```
Pipeline(DataSource(...),     # emits (sequence_number, data) pairs
         Framer(),
         Chunker(),
         UnreliableTransportMechanismSender(),
        ).activate()
```

At the receiver:

```
Pipeline(UnreliableTransportMechanismReceiver(),
         DeChunker(),
         DeFramer(),
         DataHandler()        # receives (sequence_number, data) pairs
        ).activate()
```

Note that this example doesn't attempt to fix errors in the stream, just detect them.

How does it work?

Framer / DeFramer

Framer/DeFramer frame and deframe data pairs of the form (tag,data). 'data' should be the main payload, and 'tag' is suitable for something like a frame sequence number.

Both tag and data are treated as strings. 'data' can contain any data. 'tag' should not contain newline or any whitespace character(s).

The framed data has the format "<tag> <length> <data>" where 'length' is the length of the 'data' string.

The SimpleFrame class performs the actual framing and deframing of the data.

These components terminate if they receive a producerFinished() message on their "control" inbox. They pass the message onto their "signal" outbox before terminating.

DataChunker / DataDeChunker

The DataChunker/DataDeChunker components chunk and dechunk the data by inserting 'sync' sequences of characters to delimit chunks of data. Each message received by DataChunker on its "inbox" inbox is considered a chunk.

DataChunker prefixes each chunk with the 'sync' message sequence and escapes any occurrences of that sequence within the data itself. The result is output on its "outbox" outbox.

DataDeChunker does the reverse process. If data is received without a preceeding 'sync' sequence then there is no way to tell if that chunk is complete (whole) and it will be discarded. Once the internal buffer contains a full chunk of data with a 'sync' sequence before and after it, that chunk is output from its "outbox" outbox. The 'sync' sequences are removed and any escaped occurrences of the 'sync' message within the data are un-escaped again.

Note that DataDeChunker buffers chunks until it knows they have been fully received. If a final chunk is not followed by a occurence of the 'sync' message then it will never be output.

However DataDeChunker can be told to flush the remaining contents of its buffer by sending any message to its "flush" inbox.

These components terminate if they receive a producerFinished() message on their "control" inbox. They pass the message onto their "signal" outbox before terminating.

Components

Framer

Framer() -> new Framer component.

Frames (tag, data) pairs into strings containing the same data.

DeFramer

DeFramer -> new DeFramer component.

Converts string that were framed using the Framer component back into (tag, data) pairs.

DataChunker

DataChunker([syncmessage]) -> new DataChunker component.

Delineates messages by prefixing them with a 'sync' sequence, allowing a receiver to synchronise to the chunks in the stream. Any occurrences of the sequence within the message itself are escaped to prevent misinterpretation.

Keyword arguments:

- syncmessage -- string to use as 'sync' sequence (default="XXXXXXXXXXXXXXXXXXXXXXXX")

DataDeChunker

DataDeChunker([syncmessage]) -> new DataDeChunker component.

Synchronises to a stream of string data, delimited into chunks by a 'sync' sequence. Chunks are buffered until the next 'sync sequence is received and are then passed on.

Keyword arguments:

- syncmessage -- string to use as 'sync' sequence (default="XXXXXXXXXXXXXXXXXXXXXXXX")

Other

COUNT

int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.___int___(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. »> int('0b100', base=0) 4

CorruptFrame

Data frame is corrupt.

IncompleteChunk

Chunk of data incomplete (or cannot guarantee is complete).

ShortFrame

Data frame too short.

SimpleFrame

SimpleFrame(tag,data) -> new SimpleFrame object.

Object that frames/deframes data, with a tag prefix (eg. a sequence number).

Kamaelia.Protocol.HTTP.ErrorPages

This page contains the default HTTP Error handling. There are two ways to call this code: either use getErrorPage to get the dictionary containing the error directly or by using an ErrorPageHandler to send the page out.

Other

ErrorPageHandler

This is the default error page handler. It is essentially the above function getErrorPage mapped to a resource handler for the HTTPServer.

getErrorPage

Get the HTML associated with an (integer) error code.

Kamaelia.Protocol.HTTP.HTTPClient

Single-Shot HTTP Client

This component is for downloading a single file from an HTTP server. Pick up data received from the server on its "outbox" outbox.

Generally you should use SimpleHTTPClient in preference to this.

Example Usage

How to use it:

```
Pipeline(
    SingleShotHTTPClient("http://www.google.co.uk/"),
    SomeComponentThatUnderstandsThoseMessageTypes()
).run()
```

If you want to use it directly, note that it doesn't output strings but ParsedHTTPHeader, ParsedHTTPBodyChunk and ParsedHTTPEnd like HTTPParser. This makes has the advantage of not buffering huge files in memory but outputting them as a stream of chunks. (with plain strings you would not know the contents of the headers or at what point that response had ended!)

How does it work?

SingleShotHTTPClient accepts a URL parameter at its creation (to ___init___). When activated it creates an HTTPParser instance and then connects to the webserver specified in the URL using a TCPClient component. It sends an HTTP request and then any response from the server is received by the HTTPParser.

HTTPParser processes the response and outputs it in parts as:

```
ParsedHTTPHeader,
ParsedHTTPBodyChunk,
ParsedHTTPBodyChunk,
...
ParsedHTTPBodyChunk,
ParsedHTTPEnd
```

If SingleShotHTTPClient detects that the requested URL is a redirect page (using the Location header) then it begins this cycle anew with the URL of the new page, otherwise the parts of the page output by HTTPParser are sent on to "outbox".

Simple HTTP Client

This component downloads the pages corresponding to HTTP URLs received on "inbox" and outputs their contents (file data) as a message, one per URL, to "outbox" in the order they were received.

Example Usage

Type URLs, and they will be downloaded and placed, back to back in "downloadedfile.txt":

```
Pipeline(
    ConsoleReader(">>> ", ""),
    SimpleHTTPClient(),
    SimpleFileWriter("downloadedfile.txt"),
).run()
```

How does it work?

SimpleHTTPClient uses the Carousel component to create a new SingleShotHTTPClient component for every URL requested. As URLs are handled sequentially, it has only one SSHC child at anyone time.

Components

SimpleHTTPClient

NO DOCS

SingleShotHTTPClient

> SingleShotHTTPClient() -> component that can download a file using HTTP by URL
>
> Arguments: - starturl -- the URL of the file to download - [postbody] -- data to POST to that URL - if set to None becomes an empty

body in to a POST (of PUT) request - [connectionclass] -- specify a class other than TCPClient to connect with - [method] -- the HTTP method for the request (default to GET normally or POST if postbody != ""

Other

AttachConsoleToDebug

NO DOCS

HTTPRequest

NO DOCS

ParsedHTTPRedirect

NO DOCS

intval

Convert a string to an integer, representing errors by None

makeSSHTTPClient

Creates a SingleShotHTTPClient for the given URL. Needed for Carousel.

removeTrailingCr

NO DOCS

Kamaelia.Protocol.HTTP.HTTPHelpers

Helper classes and components for HTTP

Components

HTTPMakePostRequest

HTTPMakePostRequest is used to turn messages into HTTP POST requests for SimpleHTTPClient.

Kamaelia.Protocol.HTTP.HTTPParser

HTTP Parser

This component is for transforming HTTP requests or responses into multiple easy-to-use dictionary objects.

Unless you are implementing a new HTTP component you should not use this component directly. Either SimpleHTTPClient, HTTPServer (in conjuncton with SimpleServer) or SingleShotHTTPClient will likely serve your needs.

If you want to use it directly, note that it doesn't output strings but ParsedHTTPHeader, ParsedHTTPBodyChunk and ParsedHTTPEnd objects.

Example Usage

If you want to play around with parsing HTTP responses: (like a client):

```
pipeline(
    ConsoleReader(),
    HTTPParser(mode="response"),
    ConsoleEchoer()
).run()
```

If you want to play around with parsing HTTP requests: (like a server):

```
pipeline(
    ConsoleReader(),
    HTTPParser(mode="response"),
    ConsoleEchoer()
).run()
```

Components

HTTPParser

Component that transforms HTTP requests or responses from a single TCP connection into multiple easy-to-use dictionary objects.

Other

ParsedHTTPBodyChunk

NO DOCS

ParsedHTTPEnd

NO DOCS

ParsedHTTPHeader

NO DOCS

removeTrailingCr

NO DOCS

splitUri

NO DOCS

Kamaelia.Protocol.HTTP.HTTPRequestHandler

HTTP Server

The fundamental parts of a webserver - an HTTP request parser and a request handler/response generator. One instance of this component can handle one TCP connection. Use a SimpleServer or similar component to allow several concurrent HTTP connections to the server.

Example Usage

> **def createhttpserver():** return HTTPServer(HTTPResourceGlue.createRequestHandler)
>
> SimpleServer(protocol=createhttpserver, port=80).run()

This defines a function which creates a HTTPServer instance with HTTPResourceGlue.createRequestHandler as the request handler component creator function. This function is then called by SimpleServer for every new TCP connection.

How does it work?

HTTPServer creates and links to a HTTPParser and HTTPRequestHandler component. Data received over TCP is forwarded to the HTTPParser and the output of HTTPRequestHandler forwarded to the TCP component's inbox for sending.

See HTTPParser (in HTTPParser.py) and HTTPRequestHandler (below) for details of how these components work.

HTTPServer accepts a single parameter - a request handler function which is passed onto and used by HTTPRequestHandler to generate request handler components. This allows different HTTP server setups to run on different ports serving completely different content.

HTTP Request Handler

HTTPRequestHandler accepts parsed HTTP requests (from HTTPParser) and outputs appropriate responses to those requests.

How does it work?

HTTPServer creates 2 subcomponents - HTTPParser and HTTPRequestHandler which handle the processing of requests and the creation of responses respectively.

Both requests and responses are handled in a stepwise manner (as opposed to processing a whole request or response in one go) to reduce latency and cope well with bottlenecks.

One request handler (self.handler) component is used per request - the particular component instance (including parameters, component state) is picked by a function called createRequestHandler - a function specified by the user. A suitable definition of this function is available in HTTPResourceGlue.py.

Generally you will have a handler spawned for each new request, terminating after completing the sending of the response. However, it is also possible to use a 'persistent' component if you do the required jiggery-pokery to make sure that at any one time this component is not servicing more than one request simultaenously ('cause it wouldn't work).

What does it support?

Components as request handlers (hurrah!).

3 different ways in which the response data (body) can be terminated:

Chunked transfer encoding

This is the most complex of the 3 ways and was introduced in HTTP/1.1. Its performance is slightly worse that the other 2 as multiple length-lines have to be added to the data stream. It is recommended for responses whose size is not known in advance as it allows keep-alive connections (more than one HTTP request per TCP connection).

Explicit length

This is the easiest of the 3 ways but requires the length of the response to be known before it is sent. It uses a header 'Content-Length' to indicate this value. This method is prefered for any response whose length is known in advance.

Connection: close

This method closes (or half-closes) the TCP connection when the response is complete. This is highly inefficient when the client wishes to download several resources as a new TCP connection must be created and destroyed for each resource. This method is retained for HTTP/1.0 compatibility. It is however

preferred for responses that do not have a true end, e.g. a continuous stream over HTTP as the alternative, chunked transfer encoding, has poorer performance.

The choice of these three methods is determined at runtime by the characteristics of the first response part produced by the request handler and the version of HTTP that the client supports (chunked requires 1.1 or higher).

What may need work?

- HTTP standards-compliance (e.g. handling of version numbers for a start)
- **Requests for byte ranges, cache control (though these may be better implemented** in each request handler)
- Performance tuning (also in HTTPParser)
- **Prevent many MBs of data being queued up because TCPClient finds it has a slow** upload to the remote host

Components

HTTPRequestHandler

HTTPRequestHandler() -> new HTTPRequestHandler component capable of fulfilling the requests received over a single connection after they have been parsed by HTTPParser

Other

MapStatusCodeToText

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

currentTimeHTTP

Get the current date and time in the format specified by HTTP/1.1

Kamaelia.Protocol.HTTP.HTTPResourceGlue

HTTP Resource Glue

What does it do?

It picks the appropriate resource handler for a request using any of the request's attributes (e.g. uri, accepted encoding, language, source etc.)

Its basic setup is to match prefixes of the request URI each of which have their own predetermined request handler class (a component class).

HTTPResourceGlue also creates an instance of the handler component, allowing complete control over its ___init___ parameters. Feel free to write your own for your webserver configuration.

Other

Kamaelia

This is a doc string, will it be of use?

URLHandlers

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

createRequestHandler

NO DOCS

Kamaelia.Protocol.HTTP.HTTPServer

HTTP Server

The fundamental parts of a webserver - an HTTP request parser and a request handler/response generator. One instance of this component can handle one TCP connection. Use a SimpleServer or similar component to allow several concurrent HTTP connections to the server.

Example Usage

> **def createhttpserver():** return HTTPServer(HTTPResourceGlue.createRequestHandler)
>
> SimpleServer(protocol=createhttpserver, port=80).run()

This defines a function which creates a HTTPServer instance with HTTPResourceGlue.createRequestHandler as the request handler component creator function. This function is then called by SimpleServer for every new TCP connection.

How does it work?

HTTPServer creates and links to a HTTPParser and HTTPRequestHandler component. Data received over TCP is forwarded to the HTTPParser and the

output of HTTPRequestHandler forwarded to the TCP component's inbox for sending.

See HTTPParser (in HTTPParser.py) and HTTPRequestHandler (below) for details of how these components work.

HTTPServer accepts a single parameter - a request handler function which is passed onto and used by HTTPRequestHandler to generate request handler components. This allows different HTTP server setups to run on different ports serving completely different content.

HTTP Request Handler

HTTPRequestHandler accepts parsed HTTP requests (from HTTPParser) and outputs appropriate responses to those requests.

How does it work?

HTTPServer creates 2 subcomponents - HTTPParser and HTTPRequestHandler which handle the processing of requests and the creation of responses respectively.

Both requests and responses are handled in a stepwise manner (as opposed to processing a whole request or response in one go) to reduce latency and cope well with bottlenecks.

One request handler (self.handler) component is used per request - the particular component instance (including parameters, component state) is picked by a function called createRequestHandler - a function specified by the user. A suitable definition of this function is available in HTTPResourceGlue.py.

Generally you will have a handler spawned for each new request, terminating after completing the sending of the response. However, it is also possible to use a 'persistent' component if you do the required jiggery-pokery to make sure that at any one time this component is not servicing more than one request simultaenously ('cause it wouldn't work).

What does it support?

Components as request handlers (hurrah!).

3 different ways in which the response data (body) can be terminated:

Chunked transfer encoding

This is the most complex of the 3 ways and was introduced in HTTP/1.1. Its performance is slightly worse that the other 2 as multiple length-lines have to be added to the data stream. It is recommended for responses whose size is not known in advance as it allows keep-alive connections (more than one HTTP request per TCP connection).

Explicit length

This is the easiest of the 3 ways but requires the length of the response to be known before it is sent. It uses a header 'Content-Length' to indicate this value. This method is prefered for any response whose length is known in advance.

Connection: close

This method closes (or half-closes) the TCP connection when the response is complete. This is highly inefficient when the client wishes to download several resources as a new TCP connection must be created and destroyed for each resource. This method is retained for HTTP/1.0 compatibility. It is however preferred for responses that do not have a true end, e.g. a continuous stream over HTTP as the alternative, chunked transfer encoding, has poorer performance.

The choice of these three methods is determined at runtime by the characteristics of the first response part produced by the request handler and the version of HTTP that the client supports (chunked requires 1.1 or higher).

What may need work?

- HTTP standards-compliance (e.g. handling of version numbers for a start)
- **Requests for byte ranges, cache control (though these may be better implemented** in each request handler)
- Performance tuning (also in HTTPParser)
- **Prevent many MBs of data being queued up because TCPClient finds it has a slow** upload to the remote host

Components

HTTPShutdownLogicHandling

NO DOCS

Other

HTTPServer

HTTPServer() -> new HTTPServer component capable of handling a single connection

Arguments: -- createRequestHandler - a function required by HTTPRequestHandler that creates the appropriate request-handler component for each request, see HTTPResourceGlue

MapStatusCodeToText

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary

initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Kamaelia.Protocol.HTTP.Handlers.Minimal

Minimal

A simple HTTP request handler for HTTPServer. Minimal serves files within a given directory, guessing their MIME-type from their file extension.

Example Usage

See HTTPResourceGlue.py for how to use request handlers.

System Requirements

This component requires a UNIX system to be run currently.

Components

Minimal

> A simple HTTP request handler for HTTPServer which serves files within a given directory, guessing their MIME-type from their file extension.
>
> Arguments: -- request - the request dictionary object that spawned this component -- homedirectory - the path to prepend to paths requested -- indexfilename - if a directory is requested, this file is checked for inside it, and sent if found

Other

MinimalFactory

NO DOCS

sanitizeFilename

NO DOCS

sanitizePath

NO DOCS

Kamaelia.Protocol.HTTP.Handlers.SessionExample

Session Example

A simple persistent request handler component. Each time a URL that is handled by this component is requested, the page's 'hit counter' is incremented and shown to the user as text.

Components

SessionExample

NO DOCS

Other

Kamaelia

This is a doc string, will it be of use?

SessionExampleWrapper

NO DOCS

Sessions

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Kamaelia.Protocol.HTTP.Handlers.UploadTorrents

Upload Torrents

A session-based HTTP request handler for HTTPServer. UploadTorrents saves .torrent files that are uploaded to it as POST data and stores the number of .torrent files save to a file "meta.txt".

Components

UploadTorrents

NO DOCS

Other

Kamaelia

This is a doc string, will it be of use?

Sessions

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

UploadTorrentsWrapper

Returns an UploadTorrents component, manages that components lifetime and access.

Kamaelia.Protocol.HTTP.IcecastClient

Icecast/SHOUTcast MP3 streaming client

This component uses HTTP to stream MP3 audio from a SHOUTcast/Icecast server.

IcecastClient fetches the combined audio and metadata stream from the HTTP server hosting the stream. IcecastDemux separates the audio data from the metadata in stream and IcecastStreamWriter writes the audio data to disk (discarding metadata).

Example Usage

Receive an Icecast/SHOUTcast stream, demultiplex it, and write it to a file:

```
pipeline(
    IcecastClient("http://64.236.34.97:80/stream/1049"),
    IcecastDemux(),
    IcecastStreamWriter("stream.mp3"),
).run()
```

How does it work?

The SHOUTcast/Icecast protocol is virtually identical to HTTP. As such, IcecastClient subclasses SingleShotHTTPClient modifying the request slightly to ask for stream metadata(e.g. track name) to be included (by adding the icy-metadata header). It is otherwise identical to its parent class.

Components

IcecastDemux

Splits a raw Icecast stream into A/V data and metadata

IcecastClient

> IcecastClient(starturl) -> Icecast/SHOUTcast MP3 streaming component
>
> Arguments: - starturl -- the URL of the stream

IcecastStreamWriter

NO DOCS

Other

IceIPCDataChunk

An audio/video stream data chunk (typically MP3)

IceIPCDisconnected

Icecast stream disconnected

IceIPCHeader

Icecast header - content type: %(contenttype)s

IceIPCMetadata

Icecast stream metadata

IcecastStreamRemoveMetadata

NO DOCS

intval

NO DOCS

Kamaelia.Protocol.HTTP.MimeTypes

Mapping of common file extensions to their associated MIME types.

Other

extensionToMimeType

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

workoutMimeType

Determine the MIME type of a file from its file extension

Kamaelia.Protocol.IRC.IRCClient

Kamaelia IRC Interface

IRC_Client provides an IRC interface for Kamaelia components.

SimpleIRCClientPrefab is a handy prefab that links IRC_Client and TCPClient to each other and IRC_Client's "talk" and "heard" boxes to the prefab's "inbox" and "outbox" boxes, respectively. SimpleIRCClientPrefab does not terminate.

The functions informat, outformat, channelInformat, and channelOutformat can be used to format incoming and outgoing messages.

Example Usage

To link IRC_Client to the web with console input and output:

```
client = Graphline(irc = IRC_Client(),
             tcp = TCPClient(host, port),
             linkages = {("self", "inbox") : ("irc" , "talk"),
                         ("irc", "outbox") : ("tcp" , "inbox"),
                         ("tcp", "outbox") : ("irc", "inbox"),
                         ("irc", "heard") : ("self", "outbox"),
                         })
Pipeline(ConsoleReader(),
         PureTransformer(channelInformat("#kamtest")),
         client,
         PureTransformer(channelOutformat("#kamtest")),
         ConsoleEchoer(),
).run()
```

Note: The user needs to enter:

```
/nick aNickName
/user uname server host realname
```

into the console before doing anything else in the above example. Be quick before the connection times out.

Then try IRC commands preceded by a slash. Messages to the channel need not be preceded by anything:

```
>>> /join #kamtest
>>> /msg nickserv identify secretpassword
>>> /topic #kamtest Testing IRC client
>>> Hello everyone.
>>> /part #kamtest
>>> /quit
```

This example sends all plaintext to #kamtest by default. To send to another channel by default, change the arguments of channelInformat and channelOutformat to the name of a different channel. (E.g. "#python")

For a more comprehensive example, see Logger.py in Tools.

How does it work?

Sending messages over IRC IRC_Client accepts commands arriving at its "talk" inbox. A command is a list/tuple and is in the form ('cmd', [arg1] [,arg2] [,arg3...]). IRC_Client retransmits these as full-fledged IRC commands to its "outbox". Arguments following the command are per RFC 1459 and RFC 2812.

For example,

- ('NICK', 'Zorknpals')
- ('USER', 'jlei', 'nohost', 'noserver', 'Kamaelia IRC Client')
- ('JOIN', '#kamaelia')
- ('PRIVMSG', '#kamtest', 'hey, how's it going?')
- ('TOPIC', '#cheese', "Mozzerella vs. Parmesan")
- ('QUIT')
- ('QUIT', "Konversation terminated!")
- ('BERSERKER', "Lvl. 10")

Note that "BERSERKER" is not a recognized IRC command. IRC_Client will not complain about this, as it treats commands uniformly, but you might get an error 421, "ERR_UNKNOWNCOMMAND" back from the server.

Sending CTCP commands IRC_Client also handles a few CTCP commands:

ACTION ("ME", channel-or-user, the-action-that-you-do).

MSG If you use the outformat function defined here, 'MSG' commands are treated as 'PRIVMSGs'.

No other CTCP commands are implemented.

Receiving messages IRC_Client's "inbox" takes messages from an IRC server and retransmits them to its "heard" outbox in tuple format. Currently each tuple has fields (command, sender, receiver, rest). This method has worked well so far.

Example output:

('001', 'heinlein.freenode.net', 'jinnaslogbot', ' Welcome to the freenode IRC Network jinnaslogbot')
('NOTICE', '', 'jinnaslogbot', '***Checking ident')
('PRIVMSG', 'jlei', '#kamtest', 'stuff')
('PART', 'kambot', '#kamtest', 'see you later)
('ACTION', 'jinnaslogbot', '#kamtest', 'does the macarena')

To stop IRC_Client, send a shutdownMicroprocess or a producerFinished to its "control" box. The higher-level client must send a login itself and respond to pings. IRC_Client will not do this automatically.

Known Issues

The prefab does not terminate. (?) Sometimes messages from the server are split up. IRC_Client does not recognize these messages and flags them as errors.

Other

IRC_Client

IRC_Client() -> new IRC_Client component

SimpleIRCClientPrefab

SimpleIRCClientPrefab(...) -> IRC_Client connected to tcp via a Graphline. Routes its "inbox" to IRC_Client's "talk" and IRC_Client's "heard" to "outbox"

Keyword arguments: - host -- the server to connect to. Default irc.freenode.net - port -- the port to connect on. Default 6667.

Kamaelia.Protocol.MimeRequestComponent

Mime Request Component.

Takes a request of the form:

XXXXX <url> PROTO/Ver Key: value Key: value Content-Length: value Key: value »blank line« >body text<

And converts it into a python object that contains: requestMethod : string url : string Protocol : string Protocol Version : string (not parsed into a number) KeyValues : dict body : raw data

Has a default inbox, and a default outbox. Requests data comes in the inbox. MimeRequest objects come out the outbox.

Components

MimeRequestComponent

Component that accepts raw data, parses it into consituent parts of a MIME request. Attempts no interpretation of the request however.

Other

Kamaelia

This is a doc string, will it be of use?

Kamaelia.Protocol.Packetise

Components

MaxSizePacketiser

This is a simple class whose purpose is to take a data stream and convert it into packets of a maximum size.

The default packet size is 1000 bytes.

This component was created due to limitations of multicast meaning packets get discarded more easily over a certain size.

Example usage:

```
Pipeline(
    ReadFileAdaptor(file_to_stream, readmode="bitrate", bitrate=400000,
                    chunkrate=50),
    SRM_Sender(),
    blockise(), # Ensure chunks small enough for multicasting!
    Multicast_transceiver("0.0.0.0", 0, "224.168.2.9", 1600),
).activate()
```

This component acts as a simple filter - data is expected on inboxes and packets come out the outbox.

This component does not terminate.

Kamaelia.Protocol.RTP.NullPayloadPreFramer

Kamaelia.Protocol.RTP.NullPayloadRTP

Null Payload RTP Classes. Null Payload Pre-Framer. Null Payload RTP Packet Stuffer - Same thing.

This Null payload also assumes constant bit rate load.

Subcomponents functionality:

- **FileControl: - Only if RFA internal - isn't**
  - FileReader - only if internal - isn't
  - FileSelector - only if internal - isn't
- Format Decoding
- DataFramaing
- Command Interpreter (Likely to be component core code)

Components

NullPayloadPreFramer

Inboxes: control -> File select, file read control, framing control recvsrc -> Block/Chunks of raw disk data

Outboxes: activatesrc -> Control messages to the file reading subsystem output -> The framed data, payload ready

Other

packLengthAsString

NO DOCS

Kamaelia.Protocol.RTP.RTCPHeader

RTP Header

This class provides a representation of the fixed RTP Headers as per section 5.1 of RFC1889. The following attributes on an RTPHeader object represent the fields in the header:

version,padding, extension, CSRCCount, marker, payloadtype sequencenumber, timestamp, SSRC, CSRC

The order of the fields and sizes are defined in the variable "struct".

Other

AppPacket

RFC1889, 6.5, Page 37

ByePacket

RFC1889, 6.5, Page 37

CName

RFC1889, 6.4.1, Page 32

Email

RFC1889, 6.4.3, Page 34

Loc

RFC1889, 6.4.5, Page 35

Name

RFC1889, 6.4.2, Page 34

Note

RFC1889, 6.4.7, Page 35

Phone

RFC1889, 6.4.4, Page 34

Priv

RFC1889, 6.4.8, Page 36 Note Whilst having the same basic format, the fields here are redefined since the description portion is subdivided.

RTCPHeader

Abstract (note, *not* base) class for RTCP packet types

ReceiverReport

RFC1889, 6.3.2, Page 28

ReportBlock

RFC 1889, 6.3.1 page 25, used in SR & RR

SenderReport

RFC1889, 6.3.1, Page 23

SourceDescription

RFC1889, 6.4, Page 31

SourceDescription_chunk

RFC1889, 6.4, Page 31

SourceDescription_item

RFC1889, 6.4, Page 31

Tool

RFC1889, 6.4.6, Page 35

Kamaelia.Protocol.RTP.RTP

RTP Packet Framing and Deframing

Send a dict specifying what needs to go into the RTP packet and RTPFramer will output it as a RTP frame.

RTPDeframer parses an RTP packet in binary string format and outputs a (seqnum, packet) tuple containing a sequence number and a dict structure containing the payload and metadata of the RTP packet. The format is the same as that used by RTPFramer.

These components simply format the data into the RTP packet format or take it back out again. They do not understand the specifics of each payload type. You must determine for yourself the correct values for each field (eg. payload type, timestamps, CSRCS, etc).

See RFC 3550 and RFC 3551 for information on the RTP speecification and the meaning and formats of fields in RTP packets.

Example Usage

Read MPEG Transport Stream packets (188 bytes each) from a file in groups of 7 at a time (to fill an RTP packet) and send them in RTP packets over multicast to 224.168.2.9 on port 1600:

```
class PrePackage(Axon.Component.component):
    def main(self):
        SSRCID = random.randint(0,(2**32) - 1)      # random unique ID for this source
```

```
        while 1:
            while self.dataReady("inbox"):
                recvData = self.recv("inbox")
                self.send(
                  { 'payloadtype' : 33,             # type 33 for MPEG 2 TS
                    'payload'     : recvData,
                    'timestamp'   : time.time() * 90000,
                    'ssrc'        : SSRCID,
                  },
                  "outbox")
            yield 1


Pipeline( RateControlledFileReader("transportstream",chunksize=7*188),
          PrePackage(),
          RTPFramer(),
          Multicast_Transceiver(("0.0.0.0", 0, "224.168.2.9", 1600)
```

Timestamps for MPEG TS in RTP are integers at 90KHz resolution (hence the x90000 scaling factor). A random value is chosen for the unique source identifier (ssrc).

Save the payload from a stream of RTP packets being received from multicast address 224.168.2.9 on port 1600 down to a file:

```
Pipeline( Multicast_transceiver("0.0.0.0", 1600, "224.168.2.9", 0),
          SimpleDetupler(1),                      # discard the source address
          RTPDeframer(),
          RecoverOrder(bufsize=64, modulo=65536), # reorder packets
          SimpleDetupler(1),                      # discard the sequence number
          SimpleDetupler("payload"),
          SimpleFileWriter("received_stream"),
        )
```

RTPFramer behaviour

Send to RTPFramer's "inbox" inbox a dictionary. It must contain these fields:

```
{
    'payloadtype' : integer payload type
    'payload'     : binary string containing the payload
    'timestamp'   : integer timestamp (32 bit, unsigned)
    'ssrc'        : sync source identifier (32 bit, unsigned)
```

...and these fields are optional:

```
'csrcs'        : list of contributing source identifiers (default = [])
'bytespadding' : number of bytes of padding to be added to the payload (default=0)
'extension'    : binary string of any extension data (default = "")
```

```
'marker'       : True to set the marker bit, otherwise False (default=False)
}
```

RTPFramer automatically adds a randomised offset to the timestamp, and generates the RTP packet sequence numbers, as required in the specification (RFC 3550).

RTPFramer constructs an RTP packet matching the fields specified and sends it as a binary string out of the "outbox" outbox.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

RTPDeframer Behaviour

Send to RTPDeframer's "inbox" inbox a binary string of an RTP packet, and the packet will be parsed, resulting in a (seqnum, packet_contents) tuple being sent to the "outbox" outbox. It will have this structure:

```
( sequence_number,
  {
    'payloadtype'  : integer payload type
    'payload'      : binary string containing the payload
    'timestamp'    : integer timestamp (32 bit, unsigned)
    'ssrc'         : sync source identifier (32 bit, unsigned)
    'csrcs'        : list of contributing source identifiers, [] if empty
    'extension'    : binary string of any extension data, "" if none
    'marker'       : True if marker bit was set, otherwise False
  }
)
```

See RFC 3550 for an explanation of the precise purposes of these fields.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

RTPFramer

RTPFramer() -> new RTPFramer component.

Creates a complete RTP packet based on a dict structure describing the packet.

RTPDeframer

RTPDeframer() -> new RTPDeframer component.

Deconstructs an RTP packet, outputting (seqnum, dict) tuple where seqnum is for recovering the order of packets, and dict contains the fields from the RTP packet.

Kamaelia.Protocol.RTP.RTPHeader

RTP Header

This class provides a representation of the fixed RTP Headers as per section 5.1 of RFC1889. The following attributes on an RTPHeader object represent the fields in the header:

version,padding, extension, CSRCCount, marker, payloadtype sequencenumber, timestamp, SSRC, CSRC

The order of the fields and sizes are defined in the variable "struct".

Other

RTPHeader

RFC1889, 5.1, Page 10

RTPSource

NO DOCS

RawRTPPayload

NO DOCS

RawRTPPayloadHeader

No RFC, specific to this implementation

Kamaelia.Protocol.RTP.RtpPacker

RtpPacker Component

Takes data from a preframer:

- Creates an RTP Header Object
- Uses the timestamp & sample count to generate an RTP timestamp

Components

RtpPacker

NO DOCS

Kamaelia.Protocol.RecoverOrder

Recover Order of Sequence Numbered Items

Recovers the order of data tagged with sequence numbers. Designed to cope with sequence numbers that have to eventually wrap.

Send (seqnum, data) tuples to the "inbox" inbox and they will be sent out of the "outbox" outbox ordered by ascending sequence number.

Example Usage

Recovering the order of RTP packets received over multicast:

```
Pipeline( Multicast_transceiver("0.0.0.0", 1600, "224.168.2.9", 0),
          SimpleDetupler(1),              # discard the source address
          RTPDeframer(),
          RecoverOrder(bufsize=64, modulo=65536),
          SimpleDetupler(1),              # discard sequence numbers
        ).activate()
```

Behaviour

At initialisation, specify the size of buffer and the modulo (wrapping point) for sequence numbers.

Send (seqnum, data) tuples to the "inbox" inbox and they will be buffered. Once the buffer is full, for every item sent to the "inbox" inbox, one will be emitted from the "outbox" outbox. The ones that are emitted will have been reordered by their sequence number.

You must ensure you choose a sufficiently large buffer size for the expected amount of reordering required. If an item arrives too late, it RecoverOrder will not be able to place it in its correct position in the sequence. It will still be emitted, but out of order.

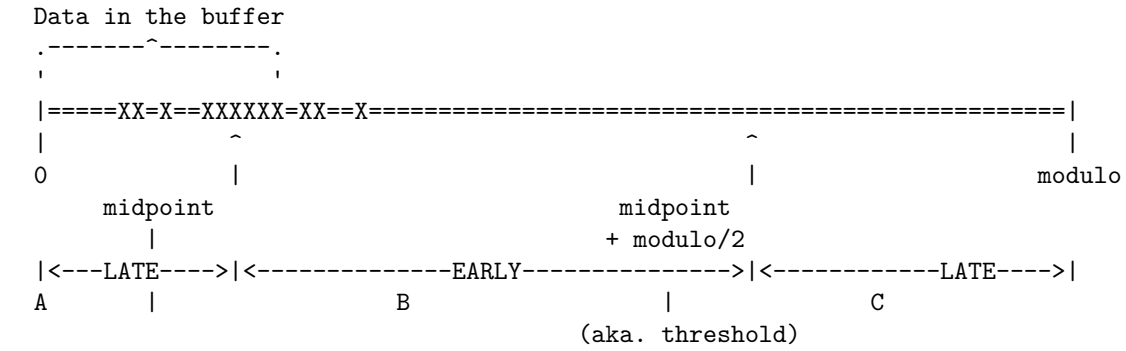
This component does not terminate. It ignores any messages sent to its "control" inbox.

Implementation details

The items are held in an internal buffer. The buffer is always in order - with the earliest sequence number at the front. Once the buffer is full, items are taken out from the front - thereby ensuring any delayed out-of-order items are given every possible chance to make it.

Since sequence numbers eventually wrap, a given sequence number could equally represent a data item that is very late, or very early.

This decision is made about a threshold - which is chosen to be the point in the sequence number line roughly furthest from the sequence numbers of the items in the buffer. This point is the sequence number of the middle item in the buffer, plus modulo/2:



Items with a sequence number after this threshold point are deemed to be late (rather than ridiculously early). An item arriving with sequence number B (marked above) has arrived early, and so should be appended to the end of the data items in the buffer. Conversely, items arriving with sequence numbers A or C (also marked above) must be late, so will be inserted at the front of the buffer.

This is implemented by adding modulo to all sequence numbers below the threshold when performing comparisons to determine where to insert the new sequence number into the buffer (the insertion point is found by doing a binary search). You can think of this as moving ranges A and B after range C.

Other

RecoverOrder

RecoverOrder([bufsize][,modulo]) -> new RecoverOrder component.

Receives and buffers (seqnum, data) pairs, and reorders them by ascending sequence number and emits them (when its internal buffer is full). This component can cope with the point at which sequence numbers wrap back to zero.

Keyword arguments:

- bufsize -- Size of the buffer for data items (default=30)
- modulo -- Sequence numbers run from 0 to modulo-1 then wrap back to 0 (default=2**32)

Kamaelia.Protocol.SDP

Session Description Protocol (SDP) Support

The SDPParser component parses Session Description Protocol (see RFC 4566) data sent to it as individual lines of text (not multiline strings) and outputs a dictionary containing the parsed session description.

Example Usage

Fetch SDP data from a URL, parse it, and display the output:

```
Pipeline( OneShot("http://www.mysite.com/sessiondescription.sdp"),
          SimpleHTTPClient(),
          chunks_to_lines(),
          SDPParser(),
          ConsoleEchoer(),
        ).run()
```

If the session description at the URL provided is this:

```
v=0
o=jdoe 2890844526 2890842807 IN IP4 10.47.16.5
s=SDP Seminar
i=A Seminar on the session description protocol
u=http://www.example.com/seminars/sdp.pdf
e=j.doe@example.com (Jane Doe)
c=IN IP4 224.2.17.12/127
t=2873397496 2873404696
a=recvonly
m=audio 49170 RTP/AVP 0
m=video 51372 RTP/AVP 99
a=rtpmap:99 h263-1998/90000
```

Then parsing will return this dictionary:

```
{ 'protocol_version': 0,
  'origin'     : ('jdoe', 2890844526, 2890842807, 'IN', 'IP4', '10.47.16.5'),
  'sessionname': 'SDP Seminar',
  'information': 'A Seminar on the session description protocol',
  'connection' : ('IN', 'IP4', '224.2.17.12', '127', 1),
  'time'       : [(2873397496L, 2873404696L, [])],
  'URI'        : 'http://www.example.com/seminars/sdp.pdf',
  'email'      : 'j.doe@example.com (Jane Doe)',
  'attribute'  : ['recvonly'],
  'media':
      [ { 'media'     : ('audio', 49170, 1, 'RTP/AVP', '0'),
          'connection': ('IN', 'IP4', '224.2.17.12', '127', 1)
        },
```

```
        { 'media'     : ('video', 51372, 1, 'RTP/AVP', '99'),
          'connection': ('IN', 'IP4', '224.2.17.12', '127', 1),
          'attribute' : ['rtpmap:99 h263-1998/90000']
        }
      ],
}
```

Behaviour

Send individual lines as strings to SDPParser's "inbox" inbox. SDPParser cannot handle multiple lines in the same string.

When SDPParser receives a producerFinished() message on its "control" inbox, or if it encounter another "v=" line then it knows it has reached the end of the SDP data and will output the parsed data as a dictionary to its "outbox" outbox.

The SDP format does *not* contain any kind of marker to signify the end of a session description - so SDPParser only deduces this by being told that the producer/data source has finished, or if it encounters a "v=" line indicating the start of another session description.

SDPParser can parse more than one session description, one after the other.

If the SDP data is malformed AssertionError, or other exceptions, may be raised. SDPParser does not rigorously test for exact compliance - it just complains if there are glaring problems, such as fields appearing in the wrong sections!

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox then, once any pending data at the "inbox" inbox has been processed, this component will terminate. It will send the message on out of its "signal" outbox.

Only if the message is a producerFinished message will it output the session description is has been parsing. A shutdownMicroprocess message will not result in it being output.

Format of parsed output

The result of parsing SDP data is a dictionary mapping descriptive names of types to values:

Session Description

Type	Dictionary key	Format of the value
v	"protocol_version"	version_number
o	"origin"	("user", session_id, session_version, "net_type", "addr_type", "addr")
s	"sessionname"	"session name"
t & r	"time"	(starttime, stoptime, [repeat,repeat, ...])

Session Description

where repeat = (interval,duration,[offset,offset, ...])

a "attribute" "value of attribute"
b "bandwidth" (mode, bitspersecond)
i "information" "value"
e "email" "email-address"
u "URI" "uri"
p "phone" "phone-number"
c "connection" ("net_type", "addr_type", "addr", ttl, groupsize)
z "timezone adjustments" [(adj-time,offset), (adj-time,offset), ...]
k "encryption" ("method","value")
m "media" [media-description, media-description, ...]
see next table for media description structure

Note that 't' and 'r' lines are combined in the dictionary into a single "time" key containing both the start and end times specified in the 't' line and a list of any repeats specified in any 'r' lines present.

The "media" key contains a list of media descriptions. Like for the overall session description, each is parsed into a dictionary, that will contain some or all of the following:

Media Descriptions

Type	Dictionary key	Format of the value
m	"media"	("media-type", port-number, number-of-ports, "protocol", "format")
c	"connection"	("net_type", "addr_type", "addr", ttl, groupsize)
b	"bandwidth"	(mode, bitspersecond)
i	"information"	"value"
k	"encryption"	("method","value")
a	"attribute"	"value of attribute"

Some lines are optional in SDP. If they are not included, then the parsed output will not contain the corresponding key.

The formats of values are left unchanged by the parsing. For example, integers representing times are simply converted to integers, but the units used remain unchanged (ie. they will not be converted to unix time units).

Components

SDPParser

SDPParser() -> new SDPParser component.

Parses Session Description Protocol data (see RFC 4566) sent to its "inbox" inbox as individual strings for each line of the SDP data. Outputs a dict containing the parsed data from its "outbox" outbox.

Other

AllDone

NO DOCS

ShutdownNow

NO DOCS

Kamaelia.Protocol.SimpleReliableMulticast

Simple Reliable Multicast

A pair of Pipelines for encoding (and decoding again) a stream of data such that is can be transported over an unreliable connection that may lose, duplicate or reorder data.

These components will ensure that data arrives in the right order and that duplicates are removed. However it cannot recover lost data.

Example Usage

Reliably transporting a file over multicast (assuming no packets are lost):

```
Pipeline(RateControlledFileReader("myfile"),
         SRM_Sender(),
         Multicast_transceiver("0.0.0.0", 0, "1.2.3.4", 1000),
        ).activate()
```

On the client:

```
class discardSeqnum(component):
    def main(self):
        while 1:
            if self.dataReady("inbox"):
                (_, data) = self.recv("inbox")
                self.send(data,"outbox")

Pipeline( Multicast_transceiver("0.0.0.0", 1000, "1.2.3.4", 0)
          SRM_Receiver(),
```

```
        discardSeqnum(),
        ConsoleEchoer()
    ).activate()
```

How does it work?

SRM_Sender is a Pipeline of three components:

- Annotator -- annotates a data stream with sequence numbers
- Framer -- frames the data
- DataChunker -- inserts markers between frames

SRM_Receiver is a Pipeline of three components:

- DataDeChunker -- recovers chunks based on markers
- DeFramer -- removes framing
- RecoverOrder -- sorts data by sequence numbers

These components will ensure that data arrives in the right order and that duplicates are removed. However it cannot recover lost data. But the final output is (seqnum,data) pairs - so there is enough information for the receiver to know that data has been lost.

The Annotator component receives data on its "inbox" inbox, and emits (seqnum, data) tuples on its "outbox" outbox. The sequence numbers start at 1 and increments by 1 for each item.

The Annotator component does not terminate and ignores messages arriving on its "control" inbox.

See documentation for the other components for details of their design and behaviour.

Components

Annotator

Annotator() -> new Annotator component.

Takes incoming data and outputs (n, data) where n is an incrementing sequence number, starting at 1.

RecoverOrder

RecoverOrder() -> new RecoverOrder component.

Receives and buffers (seqnum, data) pairs, and reorders them by ascending sequence number and emits them (when its internal buffer is full).

Other

SRM_Receiver

Simple Reliable Multicast receiver.

Dechunks, deframes and recovers the order of a data stream that has been encoded by SRM_Sender.

Final emitted data is (seqnum, data) pairs.

This is a Pipeline of components.

SRM_Sender

Simple Reliable Multicast sender.

Sequence numbers, frames and chunks a data stream, making it suitable for sending over an unreliable connection that may lose, reorder or duplicate data. Can be decoded by SRM_Receiver.

This is a Pipeline of components.

Kamaelia.Protocol.SimpleVideoCookieServer

Simple Video based fortune cookie server

To watch the video, on a linux box do this:

netcat <server ip> 1500 | plaympeg -2 -

Components

HelloServer

NO DOCS

Kamaelia.Protocol.Torrent.TorrentIPC

(Bit)Torrent IPC messages

Other

TIPCCloseTorrent

Close torrent %(torrentid)d

TIPCCreateNewTorrent

Create a new torrent

TIPCMakeTorrent

Create a .torrent file

TIPCNewTorrentCreated

New torrent %(torrentid)d created in %(savefolder)s

TIPCServiceAdd

Add a client to TorrentService

TIPCServicePassOn

Add a client to TorrentService

TIPCServiceRemove

Remove a client from TorrentService

TIPCTorrentAlreadyDownloading

That torrent is already downloading!

TIPCTorrentStartFail

Torrent failed to start!

TIPCTorrentStatusUpdate

Current status of a single torrent

Kamaelia.ReadFileAdaptor

This is a deprecation stub, due for later removal.

Other

ReadFileAdaptor

NO DOCS

Kamaelia.SampleTemplateComponent

Sample Template Component. Use this as the basis for your components!

Components

CallbackStyleComponent

NO DOCS

StandardStyleComponent

NO DOCS

Kamaelia.SimpleServerComponent

This is a deprecation stub, due for later removal.

Other

SimpleServer

NO DOCS

Kamaelia.SingleServer

This is a deprecation stub, due for later removal.

Other

SingleServer

NO DOCS

Kamaelia.Support.Data.Escape

Other

escape

NO DOCS

unescape

NO DOCS

Kamaelia.Support.Data.Experimental

Components

GraphSlideXMLComponent

NO DOCS

Other

GraphSlideXMLParser

NO DOCS

onDemandGraphFileParser_Prefab

NO DOCS

Kamaelia.Support.Data.ISO639_2

ISO 639-2 and 639-2T language code mappings

Other

code2name

NO DOCS

code2names

NO DOCS

mappings

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

name2code

NO DOCS

reverse_mappings

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

Kamaelia.Support.Data.MimeDict

Other

MimeDict

NO DOCS

Kamaelia.Support.Data.MimeObject

Other

mimeObject

Accepts a Mime header represented as a dictionary object, and a body as a string. Provides a way of handling as a coherant unit. ATTRIBUTES: header : dictionary. (keys == fields, values = field values) body : body of MIME object

Kamaelia.Support.Data.Rationals

Rational fraction conversion/handling

This set of functions assist in creating rational fractions (numbers represented by a fraction with an integer numerator and denominator).

Conversion from floating point to rational fraction

The rational(...) function converts a floating point value to a rational fraction.

It aims to generate as close an approximation as is reasonably possible, and to use as small (simple) a numerator and denominator as possible.

Examples

Conversion of a floating point number to a rational fraction:

```
>>> rational(0.75)
(3, 4)
```

Scale a rational's numerator and denominator to fit within limits:

```
>>> limit( (1500,2000), 80, -80)
(60, 80)
```

Find the greatest common divisor:

```
>>> gcd(18,42)
6
```

How does conversion work?

rational(...) uses the "continuous fractions" recursive approximation technique.

The algorithm effectively generates a continuous fractions up to a specified depth, and then multiplies them out to generate an integer numerator and denominator.

All depths are tried up to the maximum depth specified. The least deep one that yields an exact match is returned. This is also the simplest.

The numerator and denominator are simplified further by dividing them by their greatest common denominator.

For more information on continuous fractions try these: - http://mathworld.wolfram.com/ContinuedFraction.html - http://sites.google.com/site/christopher3reed/confracs - http://www.cut-the-knot.org/do_you_know/fraction.shtml - http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/cfINTRO.html#real

Other

gcd

gcd(a,b) -> greatest common divisor of a and b

limit

> limit((num,denom),poslimit,neglimit) -> (num,denom) scaled to fit within limits
>
> Scales the fraction (num,demon) so both the numerator and denominator are within the specified negative and positive bounds, inclusive.

rational

> rational(floatval[,maxdepth]) -> (numerator, denominator)
>
> Convert any floating point value to a best approximation fraction (rational)
>
> maxdepth -- the maximum recursion depth used by the algorithm (default=10)

Kamaelia.Support.Data.bitfieldrec

Bit Field Record Support

1. subclass bfrec
2. Define a class var "fields"

3. The value for this field should be a list of "field"s, created by calling the static method field.mkList. This takes a list of tuples, one tuple per field. (fieldname, bitwidth, None or list)

   See testBFR for an example.

Usage:

```
>> import bitfieldrec
>> bfrec,field = bitfieldrec.bfrec,bitfieldrec.field
>> reload(bitfieldrec)
```

Currently only supports packing. Does not support unpacking (yet).

Other

bfrec

NO DOCS

field

NO DOCS

Kamaelia.Support.Data.requestLine

Parsing for URI request lines

This object parses a URI request line, such as those used in HTTP to request data from a server.

Example

```
>>> r = requestLine("GET http://foo.bar.com/fwibble PROTO/3.3")
>>> print (parser.debug__str__())
METHOD          :GET
PROTOCOL        :PROTO
VERSION         :3.3
Req Type        :http
USER            :
PASSWORD        :
DOMAIN          :foo.bar.com
URL             :/fwibble
>>> print (r.domain)
foo.bar.com
```

Other

requestLine

requestLine(request) -> request object

The URI request is processed. The components of the request are placed in these attributes: - method - protocol - reqprotocol - domain - url - user - passwd - version

Kamaelia.Support.Deprecate

(this is stub documentation) FIXME: Documentation

Kamaelia Deprecation infrastructure

You can set a global environment variable - KAMAELIA_DEPRECATION_WARNINGS -to control debug warnings.

Possible values of KAMAELIA_DEPRECATION_WARNINGS:

Value	Action
(not set)	This causes the default debug level for warnings -- (QUIET)
QUIET	Supresses all deprecation warnings
WARN	Display warning only for first usage of each deprecated entity
VERBOSE	Displays warning for all deprecations, including traceback for each
CRASH	Raises exception causing the component (and probably the system) to crash - useful especially during testing

Other

GLOBAL_WARNING_LEVEL

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.___str___() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

Warning

Warning(message[, warninglevel]) -> new callable Warning object.

Implements warning mechanisms. When called, will output warnings according to the set warning level. The warning level used will be the greater of the specified one and the system wide level.

defaultWarningLevel

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.___str___() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

deprecationWarning

> deprecationWarning(message[, warninglevel]) -> display deprecation warning.
>
> Outputs a deprecation warning in accordance with the higher of the specified warning level and the system wide one.

makeClassStub

> makeClasStub(klass, message[, warninglevel]) -> stub for target klass.
>
> Returns a stub class for the specified target class. The stub will output deprecation warnings in accordance with the higher of the specified warning level and the system wide one.

makeFuncStub

> makeFuncStub(func, message[, warninglevel]) -> stub for target func.
>
> Returns a stub function for the specified target func. The stub will output deprecation warnings in accordance with the higher of the specified warning level and the system wide one.

Kamaelia.Support.Particles.MultipleLaws

Particle Physics Laws for multiple particles

A class implementing laws for interactions between multiple particle types, in discrete time simulations. Can be used with the physics ParticleSystem class. This implementation supports different parameters for interactions between different particle types.

You specify a mapping between pairs of types of particles and the set of laws to apply between them.

This class provides the same methods as the SimpleLaws class. It is a drop in replacement for when you wish to specialise a physics model to apply different laws depending on the types of particles involved.

Example Usage

For two types of particle "Entity" and "Attribute": - Entities only repel each other - Attributes bond at a distance of 200 units - Attributes bond to entities at a distance of 50 units :

```
mapping = { ("Entity","Entity") :SimpleLaws(maxBondForce=0, repulsionStrength=10),
            ("Attribute","Attribute") : SimpleLaws(bondLength=200),
            ("Entity","Attribute") : SimpleLaws(bondLength=50),
          }

laws = MultipleLaws( typesToLaws=mapping,
                   )
```

How does it work?

It provides the same method interface as the SimpleLaws class, but applies different sets of laws depending on the particle types passed when methods are called (SimpleLaws always applies the same rules irrespective).

The different laws provided are stored with the specified mappings. If you specify a mapping for (typeA,typeB), then it will also be applied to (typeB,typeA). You do not need to specify the mappings both ways round, though you may if you wish.

If you do not specify the complete set of mappings for the particle types to all of each other, then a default law (if specified) will be used to fill in the gaps.

Note that the default law does not get applied to particle types not mentioned when in the mappings you provide. For example, if your mappings only cover particle types 'A','B', and 'C', then interactions involving a new type 'D' will cause an exception to be raised.

The 'maximum interaction radius' for a given particle type is set to the maximum of the interaction radii for all the different interaction laws it is involved in.

Other

MultipleLaws

> MultipleLaws(typesToLaws[,defaultLaw]) -> new MultipleLaws object
>
> Computes forces between specified particle types at specified separation distances. Different forces are applied depending on whether they are bonded or unbonded and depending on the types of particle interacting.
>
> Keyword arguments:

- typesToLaws -- dictionary mapping pairs of particle type names (A,B) to object that will compute the laws acting between them
- defaultLaw -- law object applied to pairings missing from the mapping

Kamaelia.Support.Particles.Particle

Particle in a discrete time physics simulation

The Particle class provides the basis for particles in a ParticleSystem simulation. The particles handle their own physics interaction calculations. You can have as many, or few, spatial dimensions as you like.

Extend this base class to add extra functionality, such as the ability to render to a graphics display (see RenderingParticle for an example of this)

Example Usage

See ParticleSystem

How does it work?

Particle maintains lists of other particles it is bonded to. The bonds have direction, so the bonding information is stored in two lists - bondedTo and bondedFrom.

Bonds are made and broken by calling the makeBond(...), breakBond(...) and breakAllBonds(...) methods.

Particle calculates its interactions with other particles when the doInteractions(...) method is called. This must be supplied with an object containins the laws to apply, and another providing the ability to search for particles within a given distance of a point. See SimpleLaws/MultipleLaws and SpatialIndexer respectively. This updates the velocity of the particle but not its actual position.

The particle's position is only updated when the update(...) method is called.

A simulation system should calculate each simulation cycle as a two step process: First, for all particles, calling doInteractions(...). Second, for all particles, calling update(...).

A particle can be frozen in place by calling freeze() and unFreeze(). This forces the particle's velocity to zero, meaning it doesn't move because of interactions with other particles.

The simulation must have a 'tick' counter, whose value changes (increments) every simulation cycle. Particle stores the last tick value it was presented with so that, when interacting with other particles, it can see which others have already been processed in the current cycle. This way, it avoids accidentaly calculating some interactions twice.

Other

Particle

Particle within a physics system with an arbitrary number of dimensions.

> Represents a particle that interacts with other particles. One set of forces are applied for those particles that are unbonded. Interactions between bonded particles are controlled by another set of forces.
>
> Bonds are bi-directional. Establishing a bond from A to B, will also establish it back from B to A. Similarly, breaking the bond will do so in both directions too.

Kamaelia.Support.Particles.ParticleSystem

Discrete time particle physics simulation

A discrete time simulator of a system of bonded and unbonded particles, of multiple types.

The actual physics calculations are deferred to the particles themselves. You can have as many, or few, spatial dimensions as you like.

Example Usage

Create 3 particles, two of which are bonded and move noticeably closer after 5 cycles of simulation:

```
>>> laws = SimpleLaws(bondLength=5)
>>> sim = ParticleSystem(laws)
>>> sim.add( Particle(position=(10,10)) )
>>> sim.add( Particle(position=(10,20)) )
>>> sim.add( Particle(position=(30,40)) )
>>> sim.particles[0].makeBond(sim.particles, 1)   # bond 1st and 2nd particles
>>> for p in sim.particles: print p.getLoc()
...
(10, 10)
(10, 20)
(30, 40)
>>> sim.run(cycles=5)
>>> for p in sim.particles: print p.getLoc()
...
[10.0, 13.940067328]
[10.0, 16.059932671999999]
[30, 40]
>>>
```

How does it work?

Set up ParticleSystem by instantiating, specifying the laws to act between particles and an (optional) set of initial particles.

Particles should be derived from the Particle base class (or have equivalent functionality).

Particles can be added or removed from the system by reference, or removed by their ID.

ParticleSystem will work for particles in space with any number of dimensions -so long as all particles use the same!

Bonds between particles are up to the particles to manage for themselves.

The simulation runs in cycles when the run(...) method is called. Each cycle advances the 'tick' count by 1. The tick count starts at zero, unless otherwise specified during initialization.

The following attributes store the particles registered in ParticleSystem: - particles -- simple list - particleDict -- dictionary, indexed by particle.ID

ParticleSystem uses a SpatialIndexer object to speed up calculations. SpatialIndexer reduce the search space when determining what particles lie within a given region (radius of a point).

If your code changes the position of a particle, the simulator must be informed, so it can update its spatial indexing data, by calling updateLoc(...)

The actual interactions between particles are calculated by the particles themselves, *not* by ParticleSystem.

ParticleSystem calls the doInteractions(...) methods of all particles so they can influence each other. It then calls the update(...) methods of all particles so they can all update their positions and velocities ready for the next cycle.

This is a two stage process so that, in a given cycle, all particles see each other at the same positions, irrespective of which particle's doInteractions(...) method is called first. Particles should not apply their velocities to update their position until their update(...) method is called.

Other

ParticleSystem

> ParticleSystem(laws[,initialParticles][,initialTick]) -> new ParticleSystem object
>
> Discrete time simulator for a system of particles.
>
> Keyword arguments:

- initialParticles -- list of particles (default=[])
- initialTick -- start value of the time 'tick' count (default=0)

Kamaelia.Support.Particles.SimpleLaws

Simple Particle Physics Laws

A class implementing laws for interactions between particles in discrete time simulations. Can be used with the physics ParticleSystem class. Laws are based on the inverse square law.

Different laws are applied for 'bonded' and 'unbonded' particles. Unbonded particles repel. Repulsion and attraction forces balance for bonded particles at a specified "bond length".

There are a range of parameters that can be set at initialisation. All have sensible defaults.

Example usage

Laws for particles that bond at a distance of 200 units:

```
>>> laws = SimpleLaws(bondLength=200)
>>> laws.bonded("","", 200, 200**2)
0.0
>>> laws.bonded("","", 210, 210**2)
2.0
>>> laws.bonded("","", 195, 195**2)
1.0
```

Laws for particles that decelerate *fast*:

```
>>> laws = SimpleLaws(damp=0.5)
>>> velocity = [10.0, 5.0]
>>> laws.dampening("", velocity)
[5.0, 2.5]
```

Laws for particles that don't repel much but bond extra strongly:

```
>>> laws = SimpleLaws(repulsionStrength = 1.0, maxBondForce = 40.0)
>>> laws.unbonded("","", 50, 50**2)
-4.0
>>> laws.bonded("","", 50, 50**2)
-20.0
```

The physics model used

Instances of this class provide methods for calculating the force that acts between a pair of particles when bonded or unbonded. It can also calculate any reduction in velocity due to 'friction'.

Repulsion forces are calculated using the inverse square law (1 / distance squared).

Bonding attraction and repulsion forces are calculated using Hook's law for springs. However a cut-off is applied so the force is never greater than when the bond is stretched to twice its resting length.

There are also other cut-offs (described below) to help prevent a simulation becoming unstable, and to help speed up simulation.

How does it work?

You specify arguments to control the strengths of bonding and repulsion (unbonded) forces and the distances they act over.

You can also specify friction and cutoffs for maximum and minimum velocities. These latter arguments are 'fudge factors' that you can use to help stop a system spiralling out of control. Note that in a discrete time simulation, if particle velocities/accelerations are too great the simulation can become unstable and particles will fly everywhere!

You may also specify a 'maximum interaction radius' - the distance at which, for unbonded particles, a simulator need not bother calculating the forces acting (because they are too small to worry about). This is to allow simulators to run faster by reducing the number of calculations performed per cycle.

For unspecified arguments, their defaults are scaled appropriately for the bondLength you specify, such that behaviour will appear unchanged, just at a different scaling.

Other

SimpleLaws

SimpleLaws([bondlength][,maxRepelRadius][,repulsionStrength][,maxBondForce]
[,damp][,dampcutoff][,maxVelocity]) -> new SimpleLaws object

Computes forces between particle at specified separation distances. Different forces are applied depending on whether they are bonded or unbonded. The same forces are applied irrespective of particle type.

Keyword arguments:

- bondLength -- Length of stable bonds between particles (default=100)
- maxRepelRadius -- Maximum distance repulsion force acts at (default=200)
- repulsionStrength -- Strength of repulsion of unbonded particles at bondLength separation (default=3.2)

- maxBondForce -- Max force applied by bond at 0 or 2*bondLength separation (default=20.0)
- damp -- amount of friction 0.0 = none, 1.0 = no movement possible (default=0.8)
- dampcutoff -- velocity set to zero if below this value (default=0.4)
- maxVelocity -- maximum allowed velocity for particles (default=32)

For unspecified arguments, their defaults are scaled appropriately for the bondLength you specify, such that behaviour will appear unchanged, just at a different scaling.

Kamaelia.Support.Particles.SpatialIndexer

Fast spatial indexing of entities

A SpatialIndexer object is an index of entities that provides fast lookups of entities whose coordinates are within a specified radius of a specified point. You can have as many, or few, spatial dimensions as you like.

This is particularly useful for computationally intensive tasks such as calculating interactions between particles as performed, for example, by the Particle and ParticleSystem classes.

Example Usage

Creating and index and registering two entities with it at (1,2) and (12,34). We also tell the SpatialIndexer that the 'usual' radius we'll be searching over is 5 units:

```
>>> class Entity:
...   def __init__(self, coords):
...     self.coords = coords
...   def getLoc(self):
...     return self.coords
...
>>> index = SpatialIndexer(proxDist=5.0)
>>> a = Entity((1.0, 2.0))
>>> b = Entity((12.0, 34.0))
>>> index.add(a,b)
```

Only 'a' is within 10 units of (0,0):

```
>>> index.withinRadius((0,0), 10.0) == [(a,5.0)]
True
```

The returned tuples are of the form: (entity, distance-squared)

Neither point is within 1 unit of (0,0):

```
>>> index.withinRadius((0,0), 1.0)
[]
```

Both 'a' and 'b' are within 50 units of (0,0):

```
>>> index.withinRadius((0,0), 50.0) == [(a,5.0), (b,1300)]
True
```

We can ask the same, but request that 'a' be excluded:

```
>>> filter = lambda particle : particle != a
>>> index.withinRadius((0,0), 50.0, filter) == [(b,1300)]
True
```

If we remove 'a' then only 'b' will be found:

```
>>> index.remove(a)
>>> index.withinRadius((0,0), 50.0) == [(b, 1300.0)]
True
```

If we change the position of b we must *notify* the SpatialIndexer:

```
>>> index.withinRadius((0,0), 10.0) == []
True
>>> b.coords=(5.0,6.0)
>>> index.withinRadius((0,0), 10.0) == [(b, 61.0)]
False
>>> index.updateLoc(b)
>>> index.withinRadius((0,0), 10.0) == [(b, 61.0)]
True
```

How does it work?

SpatialIndexer stores entities in an associative data structure, indexed by their spatial location. Simply put, it breaks space into a grid of cells. The coordinates of that cell index into a dictionary. All particles that fall within a given cell are stored in a list in that dictionary entry.

It can then rapidly search for cells overlapping the area we want to search and return those entities that fall within that area.

The size of the cells is specified during initialisation. Choose a size roughly equal to the radius you'll most often be searching over. Too small a value will case SpatialIndexer to spend too long enumerating through cells. To big a cell size and far more entities will be searched that necessary.

Entities must provide a getLoc() method that returns a tuple of the coordinates of that entity.

Use the add(...) and remove(...) methods to register and deregister entities from the spatial index.

If you change the coordinates of an entity, the SpatialIndexer must be notified by calling its updateLoc(...) method.

Other

SpatialIndexer

> SpatialIndexer(proxDist) -> new SpatialIndexer object
>
> Creates an indexing object, capable of quickly finding entities within a given radius of a given point.
>
> Optimise by setting proxDist to the radius you'll most commonly use when wanting to find entities.

Kamaelia.Support.Protocol.HTTP

This is a module of utility functions to be used with the HTTP server.

FIXME: Needs lots of REST/Doc fixes :-(

Request Translators

The HTTPServer code provides you with an HTTP Request that looks something like this:

```
{'bad': False,
 'body': '',
 'headers': {'accept': 'text/xml,application/xml,...,q=0.9',
             'accept-charset': 'ISO-8859-1,utf-8;q=0.7,*;q=0.7',
             'accept-encoding': 'gzip,deflate',
             'accept-language': 'en-gb,en;q=0.5',
             'connection': 'keep-alive',
             'host': '127.0.0.1:8082',
             'keep-alive': '300',
             'user-agent': 'Mozilla/5.0 (X11; U;....0.0.16'},
 'localip': '127.0.0.1',
 'localport': 8082,
 'method': 'GET',
 'non-query-uri': '/hello',
 'peer': '127.0.0.1',
 'peerport': 40819,
 'protocol': 'HTTP',
 'query': '',
 'raw-uri': '/hello',
```

```
 'uri-prefix-trigger': '/hello',
 'uri-protocol': '',
 'uri-server': '127.0.0.1:8082',
 'uri-suffix': '',
 'version': '1.1'}
```

However, different sorts of handlers often need to build on top of this. As a result this module provides a number of translators that transform the above dictionary format into something else.

WSGILikeTranslator for example transforms the above information into the sort of things required by a WSGI application, as specified in PEP 333.

Specifically, with a request translator, you may have the format of a parsed HTTP request changed before it gets to your handler. You may use the function ReqTranslatorFactory to create a factory function that will create your handler using the request translator you specify automatically.

ReqTranslatorFactory This function will make a factory that can create handlers for the HTTP Server If this is used, the requests coming in to that handler will be formatted using the given translator:

```
* hand - a factory function that returns a handler to be used by the HTTP Server

* trans - a function that takes a request and returns a translated dictionary
  to be used by the handler.
```

FIXME: Parameter names aren't human readable, except by original author.

WSGILikeTranslator FIXME: Needs a rewrite to something readable.

This function will translate the HTTPParser's syntax into a more WSGI-like syntax. Pass it to the HTTPProtocol factory function and requests will be sent to your resource handler with a subset of a WSGI environ dictionary. You just need to supply more of the wsgi variables (like wsgi.input).

> request - the request to be translated

This function will return the translated dictionary.

ConvertHeaders FIXME: Again, needs a rewrite to something readable.

Converts environ variables to strings for wsgi compliance. Also puts the request headers into CGI variables.

> request - The request as formatted by the HTTP Server environ - the WSGI environ dict to contain the converted headers

Popping request dictionaries

FIXME: This needs a rewrite into something understandable.

A fairly common practice in dealing with HTTP dictionaries is to "pop" a URI. That is to say, to move one level down in the webserver's "Filesystem." The main function for doing this is PopURI, which requires you to manually specify the keys to use from the dictionary. Also provided are PopKamaeliaURI and PopWsgiURI.

PopKamaeliaURI FIXME: What does this actually do and why? (What does POP mean in this context?)

This is a function to pop a level from the PATH_INFO key into the SCRIPT_NAME key of a WSGI-like dictionary.

request - a WSGI-like dictionary

PopWsgiURI FIXME: What does this actually do and why? (What does POP mean in this context?)

This is a function to pop a level from the uri-suffix into the uri-prefix-trigger key of a request dictionary.

request - a request dictionary

HTTP Protocol

These functions are included to simplify using the HTTPServer.

Instead of instantiating an HTTPServer directly, you may wish to use the included HTTPProtocol factory function if you are using the HTTPServer with ServerCore. You may also use the requestHandlers function to generate a createRequestHandler function to pass to the HTTPServer.

To use HTTPProtocol with ServerCore, use the following:

```
from Kamaelia.Support.Protocol.HTTP import HTTPProtocol
from Kamaelia.Chassis.ConnectedServer import ServerCore
from Kamaelia.Protocol.Handlers.Minimal import MinimalFactory

routing = [['/', MinimalFactory('index.html', 'htdocs')]]
ServerCore(
    protocol=HTTPProtocol(routing),
    port = 8080,
    socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)).run()
```

HTTPProtocol FIXME: Needs rest fixes FIXME: Does this actually result in a component? If so it's a prefab and FIXME: should not be here.

This function will generate an HTTP Server that may be used with ServerCore.

routing - An iterable of iterables. Each item in the main iterabe may be thought of as an entry in the HTTPServer's "routing table." An entry's syntax is roughly as follows: [... [<URI prefix>, <Handler factory>] ...]

See above for example syntax.

errorPages - A component to create in the event of an error. That component's ___init___ function must accept two arguments: the error code (as an integer) and a message to be displayed (although it can ignore these if it so chooses)

requestHandlers FIXME: Needs rest fixes FIXME: Looks like this outputs components, meaning it's a prefab and should FIXME: sit inside the main tree, not support.

This function will generate a createRequestHandlers function for use with HTTPServer

routing - An iterable of iterables formatted the same as HTTPProtocol. errorPages - A component to create in the event of an error. That component's ___init___ function must accept two arguments: the error code (as an integer) and a message to be displayed (although it can ignore these if it so chooses)

Misc

These are various other functions:

CheckSlashes FIXME: Why is this here? Where is it used?

This function will make sure that a URI begins with a slash and does not end with a slash.

item - the uri to be checked sl_char - the character to be considered a 'slash' for the purposes of this function

Other

ConvertHeaders

Converts environ variables to strings for wsgi compliance. Also puts the request headers into CGI variables.

request - The request as formatted by the HTTP Server environ - the WSGI environ dict to contain the converted headers

FIXME: Rest needs fixing

HTTPProtocol

This function will generate an HTTP Server that may be used with ServerCore.

routing - An iterable of iterables. Each item in the main iterabe may be thought of as an entry in the HTTPServer's "routing table." An entry's syntax is roughly as follows: [... [<URI prefix>, <Handler factory>] ...]

See above for example syntax.

errorPages - A component to create in the event of an error. That component's ___init___ function must accept two arguments: the error code (as an integer) and a message to be displayed (although it can ignore these if it so chooses)

FIXME: Very probably in the wrong place because this looks like a prefab

PopKamaeliaURI

This is a function to pop a level from the uri-suffix into the uri-prefix-trigger key of a request dictionary.

request - a request dictionary

FIXME: Rest needs fixing

PopURI

This function is used to pop a directory from the PATH_INFO key to the SCRIPT_NAME key (named by pi_key and sn_key respectively). This is logically equivalent to moving down a level in the webserver's 'file system.'

You may also use the convenience functions PopWsgiURI (if the dictionary is formatted as a WSGI environ dict) and PopKamaeliaURI (if the dictionary is formatted as created by the HTTP Server)

request - the dictionary containing the keys to be manipulated sn_key - the key that the SCRIPT_NAME is referenced by in request pi_key - the key that the PATH_INFO is referenced by in request ru_key - the key that represents the full URI (without a query string)

FIXME: Rest needs fixing

PopWsgiURI

This is a function to pop a level from the PATH_INFO key into the SCRIPT_NAME key of a WSGI-like dictionary.

request - a WSGI-like dictionary

FIXME: Rest needs fixing

ReqTranslatorFactory

This function will make a factory that can create handlers for the HTTP Server If this is used, the requests coming in to that handler will be formatted using the given translator.

hand - a factory function that returns a handler to be used by the HTTP Server trans - a function that takes a request and returns a translated dictionary to be used by the handler.

FIXME: Rest needs fixing

WSGILikeTranslator

This function will translate the HTTPParser's syntax into a more WSGI-like syntax. Pass it to the HTTPProtocol factory function and requests will be sent to your resource handler with a subset of a WSGI environ dictionary. You just need to supply more of the wsgi variables (like wsgi.input).

request - the request to be translated

This function will return the translated dictionary. FIXME: Rest needs fixing

checkSlashes

This function will make sure that a URI begins with a slash and does not end with a slash.

item - the uri to be checked sl_char - the character to be considered a 'slash' for the purposes of this function

requestHandlers

This function will generate a createRequestHandlers function for use with HTTPServer

routing - An iterable of iterables formatted the same as HTTPProtocol. errorPages - A component to create in the event of an error. That component's ___init___ function must accept two arguments: the error code (as an integer) and a message to be displayed (although it can ignore these if it so chooses)

FIXME: Rest needs fixing FIXME: Very probably in the wrong place because this looks like a prefab

Kamaelia.Support.Protocol.IRC

Kamaelia IRC Support Code

This provides support for Kamaelia.Protocol.IRC.*

Specifically it provides 2 core functions and 2 utility methods.

Core functions

informat(text,defaultChannel='#kamtest')

Summary - Puts a string input into tuple format for IRC_Client. Understands irc commands preceded by a slash ("/"). All other text is formatted such that sending it would send the message to the default channel.

Detail - If the text starts with a "/" it is treated as a command. Informat understands some specific commands which it helps you format for sending to the IRCClient. The commands it understands are:

```
QUIT
PRIVMSG
MSG
NOTICE
KILL
TOPIC
SQUERY
KICK
USER
ME
```

For commands it doesn't recognise, it makes a guess at how to forward it.

If you send it text which does NOT start with "/", it is assumed to be badly formatted text, intended to be sent to the current default channel. It is then formatted appropriately for sending on to an IRC_Client component.

For an example of usage, see Examples/TCP_Systems/IRC/BasicDemo.py

outformat(data, defaultChannel='#kamtest')

Takes tuple output from IRC_Client and formats for easier reading If a plaintext is received, outformat treats it as a privmsg intended for defaultChannel (default "#kamtest").

Specific commands it understands and will make a an attempt to format appropriately are:

```
PRIVMSG
JOIN
PART
NICK
ACTION
TOPIC
QUIT
MODE
```

It will also identify certain types of errors.

For an example of usage, see Examples/TCP_Systems/IRC/BasicDemo.py

Utility functions

channelOutformat(channel)

Creates a customised outformat function with defaultChannel predefined to channel. (ie Returns a lambda)

channelInformat(channel)

Creates a customised informat function with defaultChannel predefined to channel. (ie Returns a lambda)

Open Issues

Should these really be components rather than helper functions?

Other

channelInformat

returns informat with the specified channel as the default channel

channelOutformat

returns outformat with the specified channel as the default channel

informat

> Puts a string input into tuple format for IRC_Client. Understands irc commands preceded by a slash ("/")

outformat

> Takes tuple output from IRC_Client and formats for easier reading. If a plaintext is received, outformat treats it as a privmsg intended for defaultChannel (default "#kamtest").

Kamaelia.Support.Tk.Scrolling

A couple of experimental classes to support some useful basic user interface elements One of these is a scrolling menu (!)

Other

ScrollingList

NO DOCS

ScrollingMenu

NO DOCS

Kamaelia.UI.GraphicDisplay

Other

have_opengl

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

have_pygame

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

Kamaelia.UI.MH.DragHandler

Pygame 'drag' event Handler

A class, to implement "click and hold" dragging operations in pygame. Hooks into the event dispatch mechanism provided by the PyGameApp component.

Subclass this class to implement your required dragging functionality.

Example Usage

A set of circles that can be dragged around the pygame window:

```
class Circle(object):
    def __init__(self, x, y, radius):
        self.x, self.y, self.radius = x, y, radius

    def draw(self, surface):
        pygame.draw.circle(surface, (255,128,128), (self.x, self.y), self.radius)


class CircleDragHandler(DragHandler):
    def __init__(self, event, app, theCircle):
        self.circle = theCircle
        super(CircleDragHandler,self).__init__(event, app)

    def detect(self, pos, button):
        if (pos[0] - self.circle.x)**2 + (pos[1] - self.circle.y)**2 < (self.circle.radius**
            return (self.circle.x, self.circle.y)
        return False

    def drag(self,newx,newy):
        self.circle.x = newx
        self.circle.y = newy

    def release(self,newx, newy):
        self.drag(newx, newy)


class DraggableCirclesApp(PyGameApp):

    def initialiseComponent(self):
        self.circles = []
        for i in range(100,200,20):
            circle = Circle(i, 2*i, 20)
            self.circles.append(circle)
            handler = lambda event, circle=circle : CircleDragHandler.handle(event, self, ci
            self.addHandler(MOUSEBUTTONDOWN, handler)


    def mainLoop(self):
        self.screen.fill( (255,255,255) )
        for circle in self.circles:
            circle.draw(self.screen)
        return 1
```

```
DraggableCirclesApp((800,600)).run()
```

How does it work?

Subclass DragHandler to use it, and (re)implement the ___init___(...), detect(...), drag(...) and release(...) methods.

Bind the handler(...) static method to the event (usually MOUSEBUTTONDOWN), providing the arguments for the initializer.

The DragHandler will instantiate upon the event and the detect(...) method will be called to determine whether a drag operation should begin.

The 'event' and 'app' attributes are set to the event that triggered this and the PyGameApp component concerned respectively.

Implement detect(...) so that it returns False to abort the drag operation, or (x,y) coordinates for the start of the drag operation. These co-ordinates don't have to be the same as the ones supplied - they are your opportunity to specify the origin for the drag.

During the drag, the DragHandler object will bind to the MOUSEMOTION and MOUSEBUTTONUP pygame events.

Whilst dragging, your drag(...) method will be called whenever the mouse moves and release(...) will be called when the mouse button(s) are finally released.

drag(...) and release(...) are passed updated x,y coordinates. These are the origin coordinates (returned by detect(...) method) plus motion since the drag began.

Other

DragHandler

> DragHandler(event,app) -> new DragHandler object
>
> Subclass this to implement mouse dragging operations in pygame. Bind the handle(...) class method to the MOUSEBUTTONDOWN pygame event to use it (via a lambda function or equivalent)
>
> Keyword Arguments:
>
> - event -- pygame event object cuasing this
> - app -- PyGameApp component this is happening in

Kamaelia.UI.MH.PyGameApp

Simple Pygame application framework

A component that sets up a pygame display surface and provides a main loop and simple event dispatch framework.

The rendering surface is requested from the Pygame Display service component, so this component can coexist with other components using pygame.

Example Usage

:: class SimpleApp1(PyGameApp):

def initialiseComponent(self): self.addHandler(MOUSEBUTTONDOWN, lambda event : self.mousedown(event))

def mainLoop(self): ... draw and do other stuff here... return 1

def mousedown(self, event): print ("Mouse down!")

app = SimpleApp1((800,600)).run()

How does it work?

Subclass this component to implement your own pygame 'app'. Replace the mainLoop() stub with your own code to redraw the display surface etc. This method will be called every cycle - do not incorporate your own loop!

The self.screen attribute is the pygame surface you should render to.

The component provides a simple event dispatch framework. Call addHandler and removeHandler to register and deregister handlers from events.

More than one handler can be registered for a given event. They are called in the order in which they were registered. If a handler returns True then the event is 'claimed' and no further handlers will be called.

The component will terminate if the user clicks the close button on the pygame display window, however your mainLoop() method will not be notified, and there is no specific 'quit' event handler.

Components

PyGameApp

PyGameApp(screensize[,caption][,transparency][,position]) -> new PyGameApp component.

Creates a PyGameApp component that obtains a pygame display surface and provides an internal pygame event dispatch mechanism.

Subclass to implement your own pygame "app".

Keyword arguments:

- screensize -- (width,height) of the display area (default = (800,600))
- caption -- Caption for the pygame window (default = "Topology Viewer")
- fullscreen -- True to start up in fullscreen mode (default = False)
- transparency -- None, or (r,g,b) colour to make transparent
- position -- None, or (left,top) position for surface within pygame window

Kamaelia.UI.OpenGL.ArrowButton

Simple Arrow Button component

A simple arrow shaped button without caption. Implements responsive button behavoir.

ArrowButton is a subclass of SimpleButton. It only overrides the draw() method, i.e. it only changes its appearance.

Example Usage

Two arrow buttons printing to the console:

```
Graphline(
    button1 = ArrowButton(size=(1,1,0.3), position=(-2,0,-10), msg="PINKY"),
    button2 = ArrowButton(size=(2,2,1), position=(5,0,-15), rotation=(0,0,90), msg="BRAIN"),
    echo = ConsoleEchoer(),
    linkages = {
        ("button1", "outbox") : ("echo", "inbox"),
        ("button2", "outbox") : ("echo", "inbox")
    }
).run()
```

Components

ArrowButton

ArrowButton(...) -> A new ArrowButton component.

A simple arrow shaped button without caption. Implements responsive button behavoir.

Kamaelia.UI.OpenGL.Button

OpenGL Button Widget

A button widget for the OpenGL display service. Sends a message when clicked or an assigned key is pressed.

This component is a subclass of OpenGLComponent and therefore uses the OpenGL display service.

Example Usage

4 Buttons which could be used for playback control (output to the console):

```
Graphline(
    BUTTON1 = Button(caption="<<", msg="Previous", position=(-3,0,-10)),
    BUTTON2 = Button(caption=">>", msg="Next", position=(3,0,-10)),
    BUTTON3 = Button(caption="Play", msg="Play", position=(-1,0,-10)),
    BUTTON4 = Button(caption="Stop", msg="Stop", position=(1,0,-10)),
    ECHO = ConsoleEchoer(),
    linkages = {
        (BUTTON1, "outbox") : (ECHO, "inbox"),
        (BUTTON2, "outbox") : (ECHO, "inbox"),
        (BUTTON3, "outbox") : (ECHO, "inbox"),
        (BUTTON4, "outbox") : (ECHO, "inbox"),
    }
).run()
```

How does it work?

This component is a subclass of OpenGLComponent (for OpenGLComponent functionality see its documentation). It overrides ___init___(), setup(), draw(), handleEvents() and frame().

In setup() it requests to receive mouse events and calls buildCaption() where the set caption is rendered on a pygame surface. This surface is then set as OpenGL texture.

In draw() a flat cuboid is drawn (if size is not specified) with the caption texture on both the front and the back surface.

In handleEvents() the component reacts to mouse events and, if a key is assigned, to the keydown event. If a mouse click happens when the mouse pointer is over the button it gets "grabbed", which is visualised by shrinking the button by a little amount, until the button is released again. Only if the mouse button is released over the button widget it gets activated. On activation the button gets rotated by 360 degrees around the X axis.

In frame() the button gets rotated when it has been activated.

Components

Button

Button(...) -> A new Button component.

A button widget for the OpenGL display service. Sends a message when clicked or an assigned key is pressed.

Keyword arguments:

- caption -- Button caption (default="Button")
- bgcolour -- Colour of surfaces behind caption (default=(244,244,244))
- fgcolour -- Colour of the caption text (default=(0,0,0)
- sidecolour -- Colour of side planes (default=(200,200,244))
- margin -- Margin size in pixels (default=8)
- key -- Key to activate button (default=None)
- fontsize -- Font size for caption text (default=50)
- pixelscaling -- Factor to convert pixels to units in 3d, ignored if size is specified (default=100)
- thickness -- Thickness of button widget, ignored if size is specified (default=0.3)
- msg -- Message which gets sent when button is activated (default="CLICK")

Kamaelia.UI.OpenGL.Container

Container component

A container to control several OpenGLComponents.

Example Usage

In the following example, three components are put into a container and get moved by a SimpleMover and rotated around the Y axis by a SimpleRotator:

```
o1 = SimpleButton(size=(1,1,1)).activate()
o2 = SimpleCube(size=(1,1,1)).activate()
o3 = ArrowButton(size=(1,1,1)).activate()

containercontents = {
    o1: {"position":(0,1,0)},
    o2: {"position":(1,-1,0)},
    o3: {"position":(-1,-1,0)},
}

Graphline(
```

```
    OBJ1=o1,
    OBJ2=o2,
    OBJ3=o3,
    CONTAINER=Container(contents=containercontents, position=(0,0,-10)),
    MOVER=SimpleMover(amount=(0.01,0.02,0.03)),
    ROTATOR=SimpleRotator(amount=(0,0.1,0)),
    linkages = {
        ("MOVER", "outbox") : ("CONTAINER","position"),
        ("ROTATOR", "outbox") : ("CONTAINER","rel_rotation")
    }
).run()
```

How does it work?

The Container component provides the same inboxes for absolute and relative movement as a OpenGLComponent. These are "position", "rotation", "scaling", "rel_position", "rel_rotation", "rel_scaling", their names are self explanatory. When the container receives a tuple in one of those inboxes, it does update its own transform and uses it to translate the movement to its content components. This is done in the method rearangeContents(). Currently only translation and scaling is supported. This means though components change their position with respect to the rotation of the container and their relative position, the components rotation does not change.

The contents have to be provided as constructor keyword in form of a nested dictionary of the following form:

```
{
    component1 : { "position":(x,y,z), "rotation":(x,y,z), "scaling":(x,y,z) },
    component2 : { "position":(x,y,z), "rotation":(x,y,z), "scaling":(x,y,z) },
    ...
}
```

Each of the "position", "rotation" and "scaling" arguments specify the amount relative to the container. They are all optional. As stated earlier, rotation is not supported yet so setting the rotation has no effect.

Container components terminate if a producerFinished or shutdownMicroprocess message is received on their "control" inbox. The received message is also forwarded to the "signal" outbox. Upon termination, this component does *not* unbind itself from the OpenGLDisplay service and does not free any requested resources.

Components

Container

Container(...) -> A new Container component.

A container to control several OpenGLComponents.

Keyword arguments:

- position -- Initial container position (default=(0,0,0)).
- rotation -- Initial container rotation (default=(0,0,0)).
- scaling -- Initial container scaling (default=(1,1,1)).
- contents -- Nested dictionary of contained components.

Kamaelia.UI.OpenGL.Interactor

General Interactor

This component implements the basic functionality of an Interactor. An Interactor listens to events of another component and tranlates them into movement which is applied to the target component. It provides methods to be overridden for adding functionality.

Example Usage

A very simple Interactor could look like this:

```
class VerySimpleInteractor(Interactor):
    def makeInteractorLinkages(self):
        self.link( (self,"outbox"), (self.target, "rel_rotation") )

    def setup(self):
        self.addListenEvents([pygame.MOUSEBUTTONDOWN])

    def handleEvents(self):
        while self.dataReady("events"):
            event = self.recv("events")
            if self.identifier in event.hitobjects:
                self.send((0,90,0))
```

For examples of how to create Interactors have a look at the files XXXInteractor.py.

A MatchedInteractor and a RotationInteractor each interacting with a SimpleCube:

```
CUBE1 = SimpleCube(size=(1,1,1), position=(1,0,0)).activate()
CUBE2 = SimpleCube(size=(1,1,1), position=(-1,0,0)).activate()
INTERACTOR1 = MatchedTranslationInteractor(target=CUBE1).activate()
INTERACTOR2 = SimpleRotationInteractor(target=CUBE2).activate()

Axon.Scheduler.scheduler.run.runThreads()
```

How does it work?

Interactor provides functionality for interaction with the OpenGL display service and OpenGL components. It is designed to be subclassed. The following methods are provided to be overridden:

- makeInteractorLinkages() -- make linkages to and from targets needed
- setup() -- set up the component
- handleEvents() -- handle input events ("events" inbox)
- frame() -- called every frame, to add additional functionality

Stubs method are provided, so missing these out does not result in broken code. The methods get called from the main method, the following code shows in which order:

```
def main(self):
    # create and send eventspy request
    ...
    # setup function from derived objects
    self.setup()
    ...
    while 1:
        yield 1
        # handle events function from derived objects
        self.handleEvents()
        # frame function from derived objects
        self.frame()
```

If you need to override the ___init___() method, e.g. to get initialisation parameters, make sure to pass on all keyword arguments to ___init___(...) of the superclass, e.g.:

```
def __init__(self, **argd):
    super(ClassName, self).__init__(**argd)
    # get an initialisation parameter
    myparam = argd.get("myparam", defaultvalue)
```

The following methods are provided to be used by inherited objects:

- addListenEvents(list of events) -- Request reception of a list of events
- removeListenEvents(list of events) -- Stop reveiving events

The are inteded to simplify component handling. For their functionality see their description.

The event identifier of the target component gets saved in self.identifier. Use this variable in event handling to determine if the target component has been hit.

Interactor components terminate if a producerFinished or shutdownMicroprocess message is received on their "control" inbox. The received message is also forwarded to the "signal" outbox. Upon termination, this component does *not*

unbind itself from the OpenGLDisplay service and does not free any requested resources.

Components

Interactor

Interactor(...) -> A new Interactor object (not very useful, designed to be subclassed)

This component implements the basic functionality of an Interactor. An Interactor listens to events of another component and tranlates them into movement which is applied to the target component. It provides methods to be overridden for adding functionality.

Keyword arguments:

- target -- OpenGL component to interact with
- nolink -- if True, no linkages are made (default=False)

Kamaelia.UI.OpenGL.Intersect

Intersect: provides a set of static 3D intersection functions

Other

Intersect

A collection of static intersection functions.

Kamaelia.UI.OpenGL.Label

OpenGL Label Widget

A Label widget for the OpenGL display service.

This component is a subclass of OpenGLComponent and therefore uses the OpenGL display service.

Example Usage

4 Labels with various sizes, colours, captions and positions:

```
Graphline(
    Label1 = Label(caption="That", size=(2,2,1), sidecolour=(0,200,0), position=(-3,0,-10)),
    Label2 = Label(caption="Boy", bgcolour=(200,100,0), position=(3,0,-10)),
    Label3 = Label(caption="Needs", margin=15, position=(-1,0,-10), rotation=(30,0,10)),
    Label4 = Label(caption="Therapy!", fontsize=20, size=(0.3,0.3,1), position=(1,0,-10)),
```

```
    ECHO = ConsoleEchoer(),
    linkages = {
        ("Label1", "outbox") : ("ECHO", "inbox"),
        ("Label2", "outbox") : ("ECHO", "inbox"),
        ("Label3", "outbox") : ("ECHO", "inbox"),
        ("Label4", "outbox") : ("ECHO", "inbox"),
    }
).run()
```

How does it work?

This component is a subclass of OpenGLComponent. It overrides ___init___(), setup(), draw(), handleEvents() and frame().

In setup() only buildCaption() gets called where the set caption is rendered on a pygame surface. This surface is then set as OpenGL texture.

In draw() a flat cuboid is drawn (if size is not specified) with the caption texture on both the front and the back surface.

Components

Label

Label(...) -> A new Label component.

A Label widget for the OpenGL display service.

Keyword arguments:

- caption -- Label caption (default="Label")
- bgcolour -- Colour of surfaces behind caption (default=(200,200,200))
- fgcolour -- Colour of the caption text (default=(0,0,0)
- sidecolour -- Colour of side planes (default=(200,200,244))
- margin -- Margin size in pixels (default=8)
- fontsize -- Font size for caption text (default=50)
- pixelscaling -- Factor to convert pixels to units in 3d, ignored if size is specified (default=100)
- thickness -- Thickness of Label widget, ignored if size is specified (default=0.3)

Kamaelia.UI.OpenGL.LiftTranslationInteractor

Lift Translation Interactor

An interactor for moving OpenGLComponents corresponding to mouse movement along the X,Y plane. When "grabbing" an object it is lifted by a specified amount.

LiftTranslationInteractor is a subclass of Interactor.

Example Usage

The following example shows four SimpleCubes which can be moved by dragging your mouse:

```
o1 = SimpleCube(position=(6, 0,-30), size=(1,1,1), name="center").activate()
i1 = LiftTranslationInteractor(target=o1).activate()

o2 = SimpleCube(position=(0, 0,-30), size=(1,1,1), name="center").activate()
i2 = LiftTranslationInteractor(target=o2).activate()

o3 = SimpleCube(position=(-3, 0,-30), size=(1,1,1), name="center").activate()
i3 = LiftTranslationInteractor(target=o3).activate()

o4 = SimpleCube(position=(15, 0,-30), size=(1,1,1), name="center").activate()
i4 = LiftTranslationInteractor(target=o4).activate()

Axon.Scheduler.scheduler.run.runThreads()
```

How does it work?

LiftTranslationInteractor is a subclass of Interactor. It overrides the ___ini___(), setup(), handleEvents() and frame() methods.

The matched movement works by using the position of the controlled object and determine its X,Y-aligned plane. The amount of mouse movement is then calculated as if it was on this plane. This is done by intersecting the direction vector which is included in the mouse event with the plane to get the point of intersection. Then the distance between the newly generated point and the last point is calculated. The result is the actual amount of movement along the X and the Y axis.

The interactor makes all the linkages it needs during initialisation. Because the interactor needs the actual position of the controlled component to be accurate all the time, it uses the components "position" outbox by default. If you don't want the interactor to make the linkages, you can set nolink=True as constructor argument. The following linkages are needed for the interactor to work (from the interactors point of view):

```
self.link( (self, "outbox"), (self.target, "rel_position") )
self.link( (self.target, "position"), (self, "inbox") )
```

Components

LiftTranslationInteractor

LiftTranslationInteractor(...) -> A new LiftTranslationInteractor component.

An interactor for moving OpenGLComponents corresponding to mouse movement along the X,Y plane. When "grabbing" an object it is lifted by a specified amount.

Keyword arguments:

- liftheight -- height by which the controlled object is lifted (default=2)

Kamaelia.UI.OpenGL.MatchedTranslationInteractor

Matched Translation Interactor

An interactor for moving OpenGLComponents corresponding to mouse movement along the X,Y plane.

MatchedTranslationInteractor is a subclass of Interactor.

Example Usage

The following example shows four SimpleCubes which can be moved by dragging your mouse:

```
o1 = SimpleCube(position=(6, 0,-30), size=(1,1,1)).activate()
i1 = MatchedTranslationInteractor(target=o1).activate()

o2 = SimpleCube(position=(0, 0,-20), size=(1,1,1)).activate()
i2 = MatchedTranslationInteractor(target=o2).activate()

o3 = SimpleCube(position=(-3, 0,-10), size=(1,1,1)).activate()
i3 = MatchedTranslationInteractor(target=o3).activate()

o4 = SimpleCube(position=(15, 0,-40), size=(1,1,1)).activate()
i4 = MatchedTranslationInteractor(target=o4).activate()

Axon.Scheduler.scheduler.run.runThreads()
```

How does it work?

MatchedTranslationInteractor is a subclass of Interactor. It overrides the ___init___(), setup(), handleEvents() and frame() methods.

The matched movement works by using the position of the controlled object and determine its X,Y-aligned plane. The amount of mouse movement is then

calculated as if it was on this plane. This is done by intersecting the direction vector which is included in the mouse event with the plane to get the point of intersection. Then the distance between the newly generated point and the last point is calculated. The result is the actual amount of movement along the X and the Y axis.

The interactor makes all the linkages it needs during initialisation. Because the interactor needs the actual position of the controlled component to be accurate all the time, it uses the components "position" outbox by default. If you don't want the interactor to make the linkages, you can set nolink=True as constructor argument. The following linkages are needed for the interactor to work (from the interactors point of view):

```
self.link( (self, "outbox"), (self.target, "rel_position") )
self.link( (self.target, "position"), (self, "inbox") )
```

Components

MatchedTranslationInteractor

MatchedTranslationInteractor(...) -> A new MatchedTranslationInteractor component.

An interactor for moving OpenGLComponents corresponding to mouse movement along the X,Y plane.

Kamaelia.UI.OpenGL.Movement

A collection of movement components and classes

Contained components:

- PathMover
- WheelMover
- SimpleRotator
- SimpleBuzzer

Contained classes:

- LinearPath

For a description of these classes have a look at their class documentation.

Example Usage

The following example show the usage of most of the components in this file (for an example how to use the WheelMover, see the TorrentOpenGLGUI example):

```
points = [(3,3,-20),
          (4,0,-20),
```

```
          (3,-3,-20),
          (0,-4,-20),
          (-3,-3,-20),
          (-4,0,-20),
          (-3,3,-20),
          (0,4,-20),
          (3,3,-20),
         ]
path = LinearPath(points, 1000)

cube1 = SimpleCube(size=(1,1,1)).activate()
pathmover = PathMover(path).activate()
pathmover.link((pathmover,"outbox"), (cube1,"position"))

cube2 = SimpleCube(size=(1,1,1)).activate()
simplemover = SimpleMover().activate()
simplemover.link((simplemover,"outbox"), (cube2,"position"))

cube3 = SimpleCube(size=(1,1,1), position=(-1,0,-15)).activate()
rotator = SimpleRotator().activate()
rotator.link((rotator,"outbox"), (cube3,"rel_rotation"))

cube4 = SimpleCube(size=(1,1,1), position=(1,0,-15)).activate()
buzzer = SimpleBuzzer().activate()
buzzer.link((buzzer,"outbox"), (cube4,"scaling"))

Axon.Scheduler.scheduler.run.runThreads()
```

Components

PathMover

PathMover(...) -> A new PathMover object.

PathMover can be used to move a 3d object along a path.

It can be controlled by sending commands to its inbox. These commands can be one of "Play", "Stop", "Next", "Previous", "Rewind", "Forward" and "Backward".

If the pathmover reaches the beginning or the end of a path it generates a status message which is sent to the "status" outbox. This message can be "Finish" or "Start".

Keyword arguments:

- path -- A path object (e.g. LinearPath) or a list of points

- repeat -- Boolean indication if the Pathmover should repeat the path if it reaches an end (default=True)

WheelMover

WheelMover(...) -> A new WheelMover component.

A component to arrange several OpenGlComponents in the style of a big wheel rotating around the X axis. Can be used to switch between components.

Components can be added and removed during operation using the "notify" inbox. Messages sent to it are expected to be a dictionary of the following form:

```
{
    "APPEND_CONTROL" :True,
    "objectid": id(object),
    "control": (object,"position")
}
```

for adding components and:

```
{
    "REMOVE_CONTROL" :True,
    "objectid": id(object),
}
```

for removing components.

If components are added when the wheel is already full (number of slots exhausted) they are simply ignored.

The whole wheel can be controlles by sending messages to the "switch" inbox. The commands can be either "NEXT" or "PREVIOUS".

Keyword arguments:

- steps -- number of steps the wheel is subdivided in (default=400)
- center -- center of the wheel (default=(0,0,-13))
- radius -- radius of the wheel (default=5)
- slots -- number of components which can be handled (default=20)

SimpleRotator

SimpleRotator(...) -> A new SimpleRotator component.

A simple rotator component mostly for testing. Rotates OpenGLComponents by the amount specified if connected to their "rel_rotation" boxes.

Keyword arguments:

- amount -- amount of relative rotation sent (default=(0.1,0.1,0.1))

SimpleMover

SimpleMover(...) -> A new SimpleMover component.

A simple mover component mostly for testing. Moves OpenGLComponents between the specified borders if connected to their "position" boxes. The amount of movement every frame and the origin can also be specified.

Keyword arguments:

- amount -- amount of movement every frame sent (default=(0.03,0.03,0.03))
- borders -- borders of every dimension (default=(5,5,5))
- origin -- origin of movement (default=(0,0,-20))

SimpleBuzzer

SimpleBuzzer(...) -> A new SimpleBuzzer component.

A simple buzzer component mostly for testing. Changes the scaling of OpenGLComponents it connected to their "scaling" boxes.

Other

LinearPath

LinearPath(...) -> A new LinearPath object.

LinearPath generates a linearly interpolated Path which can be used by the Pathmover component to control component movement.

It provides basic list functionality by providing a ___getitem___() as well as a ___len___() method for accessing the path elements.

Keyword arguments:

- points -- a list of points in the path
- steps -- number of steps to generate between the path endpoints (default=1000)

Kamaelia.UI.OpenGL.OpenGLComponent

General OpenGL component

This components implements the interaction with the OpenGLDisplay service that is needed to setup, draw and move an object using OpenGL.

It is recommended to use it as base class for new 3D components. It provides methods to be overridden for adding functionality.

Example Usage

One of the simplest possible reasonable component would like something like this:

```
class Point(OpenGLComponent):
    def draw(self):
        glBegin(GL_POINTS)
        glColor(1,0,0)
        glVertex(0,0,0)
        glEnd()
```

A more complex component that changes colour in response to messages sent to its "colour" inbox and reacts to mouse clicks by rotationg slightly:

```
class ChangingColourQuad(OpenGLComponent):
    def setup(self):
        self.colour = (0.5,1.0,0.5)
        self.addInbox("colour")
        self.addListenEvents([pygame.MOUSEBUTTONDOWN])

    def draw(self):
        glBegin(GL_QUADS)
        glColor(*self.colour)
        glVertex(-1, 1, 0)
        glVertex(1, 1, 0)
        glVertex(1, -1, 0)
        glVertex(-1, -1, 0)
        glEnd()

    def handleEvents(self):
        while self.dataReady("events"):
            event = self.recv("events")
            if event.type == pygame.MOUSEBUTTONDOWN and self.identifier in event.hitobjects:
                self.rotation += Vector(0,0,10)
                self.rotation %= 360

    def frame(self):
        while self.dataReady("colour"):
            self.colour = self.recv("colour")
            self.redraw()
```

How does it work?

OpenGLComponent provides functionality to display and move objects in OpenGL as well as to process events. The component registers at the OpenGL display service, draws its contens to a displaylist and applies its transformations to a Transform object. The display list id and the Transform objects are continuosly transfered to the display service when updated.

For movement several inboxes are provided. The messages sent to these boxes are collected and applied automatically. The inboxes and expected messages are:

- position -- position triples (x,y,z)
- rotation -- rotation triples (x,y,z)
- scaling -- scaling triples (x,y,z)
- rel_position -- relative position triples (x,y,z)
- rel_rotation -- relative rotation triples (x,y,z)
- rel_scaling -- relative scaling triples (x,y,z)

When an OpenGLComponent gets moved, it also provides feedback about its movement. This feedback is sent to the following outboxes:

- position -- position triples (x,y,z)
- rotation -- rotation triples (x,y,z)
- scaling -- scaling triples (x,y,z)

OpenGLComponent is designed to get subclassed by opengl components. Using it as base class has the advantage not having to worry about interaction with the OpenGl display service. To add functionality, the following methods are provided to be overridden:

- setup() -- set up the component
- draw() -- draw content using OpenGL
- handleEvents() -- handle input events ("events" inbox)
- frame() -- called every frame, to add additional functionality

Stubs method are provided, so missing these out does not result in broken code. The methods get called from the main method, the following code shows in which order:

```
def main(self):
    # create and send display request
    ...
    # setup function from derived objects
    self.setup()
    ...
    # inital apply trasformations
    self.applyTransforms() # generates and sends a Transform object
    # initial draw to display list
    self.redraw() # calls draw and saves it to a displaylist
```

```
...
while 1:
    yield 1
    self.applyTransforms()
    self.handleMovement()
    # handle events function from derived objects
    self.handleEvents()
    # frame function from derived objects
    self.frame()
```

As can be seen here, there is no invocation of draw in the main loop. It is only called once to generate a displaylist which then gets send to the display service. This is the normal situation with static 3D objects. If you want to create a dynamic object, e.g. which changes e.g. its geometry or colour (see second example above), you need to call the redraw() method whenever changes happen.

If you need to override the ___init___() method, e.g. to get initialisation parameters, make sure to pass on all keyword arguments to ___init___(...) of the superclass, e.g.:

```
def __init__(self, **argd):
    super(ClassName, self).__init__(**argd)
    # get an initialisation parameter
    myparam = argd.get("myparam", defaultvalue)
```

The following methods are provided to be used by inherited objects:

- redraw() -- Call draw() and save its actions to a displaylist. Send it as update request to the display service. *Don't call this method from within draw()!*
- addListenEvents(list of events) -- Request reception of a list of events
- removeListenEvents(list of events) -- Stop reveiving events

The are inteded to simplify component handling. For detailed description see their documentation.

Every OpenGLComponent has its own pygame Clock object. It is used to measure the time between frames. The value gets stored in self.frametime in seconds and can be used by derived components to make movement time-based rather than frame-based. For example to rotate 3 degrees per second you would do something like:

```
self.rotation.y += 3.0*self.frametime
```

OpenGLComponent components terminate if a producerFinished or shutdown-Microprocess message is received on their "control" inbox. The received message is also forwarded to the "signal" outbox. Upon termination, this component does *not* unbind itself from the OpenGLDisplay service and does not free any requested resources.

Components

OpenGLComponent

OpenGLComponent(...) -> create a new OpenGL component (not very useful though; it is rather designed to inherit from).

This components implements the interaction with the OpenGLDisplay service that is needed to setup, draw and move an object using OpenGL.

Keyword arguments:

- size -- three dimensional size of component (default=(0,0,0))
- rotation -- rotation of component around (x,y,z) axis (defaul=(0,0,0))
- scaling -- scaling along the (x,y,z) axis (default=(1,1,1))
- position -- three dimensional position (default=(0,0,0))
- name -- name of component (mostly for debugging, default="nameless")

Kamaelia.UI.OpenGL.OpenGLDisplay

OpenGL Display Service

This component provides an OpenGL window and manages input events, positioning and drawing of other components. It handles both OpenGL and Pygame components.

OpenGLDisplay is a service that registers with the Coordinating Assistant Tracker (CAT).

Example Usage

If you want to change some of the default parameters, like the viewport, you first have to create an OpenGLDisplay object and then register it. The following would show a simple cube from a slightly changed viewer position:

```
display = OpenGLDisplay(viewerposition=(0,-10,0), lookat=(0,0,-15)).activate()
OpenGLDisplay.setDisplayService(display)

SimpleCube(position=(0,0,-15)).activate()
```

If you want to use pygame components, you have to override the PygameDisplay service before creating any pygame components:

```
display = OpenGLDisplay.getDisplayService()
PygameDisplay.setDisplayService(display[0])
```

For examples of how components have to interfrere with OpenGLDisplay, please have a look at OpenGLComponent.py and Interactor.py.

How does it work?

OpenGLDisplay is a service. obtain it by calling the OpenGLDisplay.getDisplayService(...) static method. Any existing instance will be returned, otherwise a new one is automatically created.

Alternatively, if you wish to configure OpenGLDisplay with options other than the defaults, create your own instance, then register it as a service by calling the PygameDisplay.setDisplayService(...) static method. NOTE that it is only advisable to do this at the top level of your system, as other components may have already requested and created a OpenGLDisplay component!

When using only OpenGL components and no special display settings have to be made, you won't see OpenGLDisplay as it is registered automatically when it is first requested (by invoking the getDisplayService(...) static method).

You can also use an instance of OpenGLDisplay to override the PygameDisplay service as it implements most of the functionality of PygameDisplay. You will want to do this when you want to use Pygame components along with OpenGL components.

pygame only supports one display window at a time, you must not make more than one OpenGLDisplay component.

OpenGLDisplay listens for requests arriving at its "notify" inbox. A request can currently be to:

- register an OpenGL component (OGL_DISPLAYREQUEST)
- register a pygame component (DISPLAYREQUEST)
- register a pygame wrapper (WRAPPERREQUEST)
- register an eventspy (EVENTSPYREQUEST)
- listen or stop listening to events (ADDLISTENEVENT, REMOVELISTENEVENT)
- update the displaylist of an OpenGL component (UPDATE_DISPLAYLIST)
- update the transform of an OpenGL component (UPDATE_TRANSFORM)
- invoke a redraw of a pygame surface (REDRAW)

OpenGL components

OpenGL components get registered by an OGL_DISPLAYREQUEST. Such a request is a dictionary with the following keys:

```
{
    "OGL_DISPLAYREQUEST": True,     # OpenGL Display request
    "objectid" : id(object),            # id of requesting object (for identification)
    "callback" : (component,"inboxname"),   # to send the generated event id to
```

```
    "events" : (component, "inboxname"),    # to send event notification (optional)
    "size": (x,y,z),                # size of object (not yet used)
}
```

When OpenGLDisplay received such a request it generates an identifier and returns it to the box you specify by "callback". This identifier can later be used to determine if a mouse event "hit" the object.

It is important to note that OpenGL don't draw and transform themselves directly but only hand displaylists and Transform objects to the display service. After an OpenGL component has been registered, it can send displaylist- and transform-updates. These requests are dictionaries of the following form:

```
{
    "DISPLAYLIST_UPDATE": True, # update displaylist
    "objectid": id(object),     # id of requesting object
    "displaylist": displaylist  # new displaylist
}
```

If an object is static, i.e. does not change its geometry, it only needs to send this update one time. Dynamic objects can provide new displaylists as often as they need to.:

```
{
    "TRANSFORM_UPDATE": True,   # update transform
    "objectid": id(self),       # id of requesting object
    "transform": self.transform # new transform
}
```

A transform update should be sent every time the object transform changes, i.e. it is moved.

OpenGL components can also request listening to events. See "Listening to events" below.

It is generally recommended to use the class OpenGLComponent as base class for OpenGL components. It implements all the functionality required to create, draw, move OpenGL components and to handle events (see OpenGLComponent.py for the class and e.g. SimpleCube.py, Button.py and other components for examples).

Pygame components

OpenGLDisplay is designed to be compatible with PygameDisplay. After overriding the PygameDisplay service, pygame components can be created as usual. See the documentation of PygameDisplay (Kamaelia/UI/PygameDisplay.py) for how to do this.

NOTE: Overlays are not supported yet.

Pygame wrappers

It is possibly, by sending a WRAPPERREQUEST, to wrap an already registered pygame component by a OpenGL component. The surface of the pygame component is then excluded from normal drawing and this responsibility is handed to the requesting component by giving it the texture name corresponding to the surface. The event processing of mouse events is then also relinked to be done by the wrapper.

The wrapper request is a dictionary with the following keys:

```
{
    "WRAPPERREQUEST" : True,                    # wrap a pygame component
    "wrapcallback" : (object, "inboxname"),     # send response here
    "eventrequests" : (object, "inboxname"),    # to receive event requests by the wrapped c
    "wrap_objectid": id(wrapped_component)      # object id of the component to be wrapped
}
```

When a WRAPPERREQUEST is received for a component which is not registered yet, it is stored until the component to be wrapped gets registered.

When a wrapper request was received, the OpenGL display service returns a dictionary to the box specified by "wrapcallback" containing the following keys:

```
{
    "texname": texname,             # OpenGL texture name
    "texsize": (width, height),     # texture coordinate size
    "size": (width, height)         # size of pygame surface in pixels
}
```

See PygameWrapperPlane.py for an example implementation of a wrapper.

Listening to events

Once your component has been registered, it can request to be notified of specific pygame events. The same requests are used for Pygame and OpenGL components, only the keys are slightly different.

To request to listen to a given event, send a dictionary to the "notify" inbox, containing the following:

```
{
    "ADDLISTENEVENT" : pygame_eventtype,    # example: pygame.KEYDOWN
    "surface" : your_surface,               # for pygame components
    "objectid" : id(object),                # for OpenGL components
}
```

To unsubscribe from a given event, send a dictionary containing:

```
{
    "REMOVELISTENEVENT" : pygame_eventtype,
    "surface" : your_surface,               # for pygame components
    "objectid" : id(object),                # for OpenGL components
}
```

Events will be sent to the inbox specified in the "events" key of the "DISPLAYREQUEST" or "OGL_DISPLAYREQUEST" message. They arrive as a list of pygame event objects.

The events objects of type Bunch with the following variables:

- type -- Pygame event type

  For events of type pygame.KEYDOWN, pygame.KEYUP:

- key -- Pressed or released key

  For events of type pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP:

- pos -- Mouse position
- button -- Pressed or released mouse button number

  For events of type pygame.MOUSEMOTION:

- rel -- Relative mouse motion.
- buttons -- Buttons pressed while mousemotion

  For events of type pygame.MOUSEBUTTONDOWN, pygame.MOUSEBUTTONUP, pygame.MOUSEMOTION when sent to OpenGL components:

- viewerposition -- Position of viewer
- dir -- Direction vector of generated from mouse position
- hitobjects -- List of hit objects

NOTE: If the event is MOUSEMOTION, MOUSEBUTTONUP or MOUSEBUTTONDOWN then you will instead receive a replacement object, with the same attributes as the pygame event. But for pygame components, the 'pos' attribute adjusted so that (0,0) is the top left corner of *your* surface. For OpenGL components the origin and direction of the intersection vector determined using the mouse position and viewport will be added as well as a list of identfiers of objects that has been hit.

If a component has requested reception of an event type, it gets every event that happens of that type, regardless if it is of any concern to the component. In the case of mouse events there is a list of hit objects included which are determined by using OpenGL picking.

Eventspies

Eventspies are components that basically listen to events for other components. They are registered by sending an EVENSPYREQUEST:

```
{
    "EVENTSPYREQUEST" : True,
    "objectid" : id(object),           # id of requesting object
    "target": id(target),              # id of object to be spied
    "callback" : (object,"inboxname"), # for sending event identifier
    "events" : (object, "inboxname")   # for reception of events
```

```
}
```

In return you get the identifier of the target component that can be used to determine if the target component has been hit. An evenspy can request reception of event types like usual (using ADDLISTENEVENT and REMOVELISTENEVENT). When events are spied this does not affect normal event processing.

Shutdown

Upon reception of a pygame.QUIT event, OpenGLDisplay sends an Axon.Ipc.shutdownMicroprocess object out of its signal outbox. The service itself does not terminate.

Components

OpenGLDisplay

OpenGLDisplay(...) -> new OpenGLDisplay component

Use OpenGLDisplay.getDisplayService(...) in preference as it returns an existing instance, or automatically creates a new one.

Or create your own and register it with setDisplayService(...)

Keyword arguments (all optional):

- title -- caption of window (default=http://kamaelia.sourceforge.net)
- width -- pixels width (default=800)
- height -- pixels height (default=600)
- background_colour -- (r,g,b) background colour (default=(255,255,255))
- fullscreen -- set to True to start up fullscreen, not windowed (default=False)
- show_fps -- show frames per second in window title (default=True)
- **limit_fps -- maximum frame rate (default=60)**
  Projection parameters
- near -- distance to near plane (default=1.0)
- far -- distance to far plane (default=100.0)
- **perspective -- perspective angle (default=45.0)** Viewer position and orientation
- viewerposition -- position of viewer (default=(0,0,0))
- lookat -- look at point (default= (0,0,-self.farPlaneDist))

- **up -- up vector (default(0,1,0))** Fog
- fog -- tuple of fog distances (start, end). if not set, fog is disabled (default)
- fog_colour -- (r,g,b) fog colour (default=(255,255,255))
- **fog_density -- fog density (default=0.35)** Event processing
- hitall -- boolean, if false, only the nearest object under the cursor gets activated (default=False)

Other

Bunch

NO DOCS

Kamaelia.UI.OpenGL.ProgressBar

Progress Bar

A progress bar widget for the OpenGL display service.

This component is a subclass of OpenGLComponent and therefore uses the OpenGL display service.

Example Usage

A progress bar with changing value:

```
Graphline(
    BOUNCE = bouncingFloat(0.5),
    PROGRESS = ProgressBar(size = (3, 0.5, 0.2), position=(0,0,-10), progress=0.5),
    linkages = {
        ("BOUNCE", "outbox"):("PROGRESS", "progress"),
    }
).run()
```

How does it work?

ProgressBar is a subclass of OpenGLComponent (for OpenGLComponent functionality see its documentation). It overrides ___init___(), draw(), handleEvents() and frame().

This component basically draws a cuboid shaped cage and inside of it a transparent bar indicating a percentage.

The percentage values are received at the "progress" inbox. The values must be in the range [0,1]. If the value is 0.0, the bar is not drawn at all, if 1.0 the bar has its maximum length. Received values which lie outside of this range are clamped to it.

Because the progress bar is transparent, the widget has to be drawn in a special order. First, the cage is drawn normally. Then the depth buffer is made read only and the transparent bar is drawn.

Components

ProgressBar

ProgressBar(...) -> new ProgressBar component.

Create a progress bar widget using the OpenGLDisplay service. Shows a tranparent bar indicating a percentage.

Keyword arguments:

- cagecolour -- Cage colour (default=(0,0,0))
- barcolour -- Bar colour (default=(200,200,244))
- progress -- Initial progress value (default=0.0)

Kamaelia.UI.OpenGL.PygameWrapper

Wrapper for pygame components

A wrapper for two dimensional pygame components that allows to display them on a Plane in 3D using OpenGL.

This component is a subclass of OpenGLComponent and therefore uses the OpenGL display service.

Example Usage

The following example shows a wrapped Ticker and MagnaDoodle component:

```
# override pygame display service
ogl_display = OpenGLDisplay.getDisplayService()
PygameDisplay.setDisplayService(ogl_display[0])

TICKER = Ticker(size = (150, 150)).activate()
TICKER_WRAPPER = PygameWrapper(wrap=TICKER, position=(4, 1,-10), rotation=(-20,15,3)).activa
MAGNADOODLE = MagnaDoodle(size=(200,200)).activate()
MAGNADOODLEWRAPPER = PygameWrapper(wrap=MAGNADOODLE, position=(-2, -2,-10), rotation=(20,10,
READER = ConsoleReader().activate()

READER.link( (READER,"outbox"), (TICKER, "inbox") )
```

```
Axon.Scheduler.scheduler.run.runThreads()
```

How does it work?

This component is a subclass of OpenGLComponent. It overrides ___init___(), setup(), draw(), handleEvents() and frame().

In setup() first the needed additional mailboxes are created. These are the "eventrequest" and "wrapcallback" inboxes and the "wrapped_events" outbox:

- "eventrequest" is used for the reception of ADDLISTENEVENT and REMOVELISTENEVENT requests of the wrapped component.
- "wrapcallback" is used to receive the response from the display service.
- "wrapped_events" is where the input events get sent to.

Additionally, a WRAPPERREQUEST is sent to the OpenGL display service. It contains the objectid of the wrapped component as well as the comms for callback and eventrequests.

In frame(), it is waited for the response on the WRAPPERREQUEST. The response should contain the OpenGL texture name, the texture size and the size of the wrapped component. The wanted events are stored and the "wrapped_events" outbox is linked to the wrapped components "events" inbox. If the size of the wrapper is not set, it is calculated using the wrapped component pixel size multiplied by the pixelscaling factor.

To handle event requests by the wrapped component, the method handleEventRequests() gets called.

In handleEvents() received mouse events get translated into the 2d space of the wrapped component and sent to it if requested. This is done by using ray/polygon intersection to determine the point of intersection in 3d. The 2d coordinates are then calculated by using the dot product between the point of intersection relative to the top left corner and the edge vectors.

In draw() a cuboid gets drawn with the texture of the pygame component on its front plane. If the z component of the size is set to zero, only the front plane is drawn.

Components

PygameWrapper

> PygameWrapper(...) -> A new PygameWrapper component.
>
> A wrapper for two dimensional pygame components that allows to display them on a Plane in 3D using OpenGL.
>
> Keyword arguments:

- wrap -- Pygame component to wrap
- pixelscaling -- Factor to convert pixels to units in 3d, ignored if size is specified (default=100)
- sidecolour -- Colour of side and back planes (default=(200,200,244))
- thickness -- Thickness of wrapper, ignored if size is specified (default=0.3)

Kamaelia.UI.OpenGL.SimpleButton

Simple Button component

A simple cuboid shaped button without caption. Implements responsive button behavoir.

Could be used to subclass differently shaped buttons from. The colours of the front/back and the side faces can be specified.

Example Usage

Two simple buttons which send messages to the console:

```
Graphline(
    button1 = SimpleButton(size=(1,1,0.3), position=(-2,0,-10), msg="PINKY"),
    button2 = SimpleButton(size=(2,2,1), position=(5,0,-15), msg="BRAIN"),
    echo = ConsoleEchoer(),
    linkages = {
        ("button1", "outbox") : ("echo", "inbox"),
        ("button2", "outbox") : ("echo", "inbox")
    }
).run()
```

How does it work?

This component is a subclass of OpenGLComponent (for OpenGLComponent functionality see its documentation). It overrides ___init___(), setup(), draw() and handleEvents().

It draws a simple cuboid. It is activated on mouse button release over the object and on key down if a key is assigned. On mouse button down it is shrunk by a small amount until the button is released.

Components

SimpleButton

SimpleButton(...) -> A new SimpleButton component.

A simple cuboid shaped button without caption. Implements responsive button behavoir.

Keyword arguments:

- bgcolour -- Background colour (default=(244,244,244))
- sidecolour -- Colour of side planes (default=(200,200,244))
- key -- Activation key, pygame identifier (optional)
- msg -- Message that gets sent to the outbox when the button is activated (default="CLICK")

Kamaelia.UI.OpenGL.SimpleCube

Simple Cube component

A simple cube for the OpenGL display service.

This component is a subclass of OpenGLComponent and therefore uses the OpenGL display service.

Example Usage

Three cubes in different positions with various rotation and sizes:

```
Graphline(
    CUBEC = SimpleCube(position=(0, 0,-12), rotation=(40,90,0), size=(1,1,1)).activate(),
    CUBER = SimpleCube(position=(4,0,-22), size=(2,2,2)).activate(),
    CUBEB = SimpleCube(position=(0,-4,-18), rotation=(0,180,20), size=(1,3,2)).activate(),
    linkages = {}
).run()
```

How does it work?

SimpleButton is a subclass of OpenGLComponent (for OpenGLComponent functionality see its documentation). It overrides draw().

In draw() a simple cube made of 6 quads with different colours is drawn.

Components

SimpleCube

SimpleCube(...) -> new SimpleCube component.

A simple cube for the OpenGL display service.

Kamaelia.UI.OpenGL.SimpleRotationInteractor

Simple Rotation Interactor

A simple interactor for rotating OpenGLComponents around the X,Y axes.

SimpleRotationInteractor is a subclass of Interactor.

Example Usage

The following example shows four SimpleCubes which can be rotated by dragging your mouse over them:

```
o1 = SimpleCube(position=(6, 0,-30), size=(1,1,1)).activate()
i1 = SimpleRotationInteractor(target=o1).activate()

o2 = SimpleCube(position=(0, 0,-20), size=(1,1,1)).activate()
i2 = SimpleRotationInteractor(target=o2).activate()

o3 = SimpleCube(position=(-3, 0,-10), size=(1,1,1)).activate()
i3 = SimpleRotationInteractor(target=o3).activate()

o4 = SimpleCube(position=(15, 0,-40), size=(1,1,1)).activate()
i4 = SimpleRotationInteractor(target=o4).activate()

Axon.Scheduler.scheduler.run.runThreads()
```

How does it work?

SimpleTranslationInteractor is a subclass of Interactor (for Interactor functionality see its documentation). It overrides the ___init___(), setup() and handleEvents() methods.

The amount of rotation is determined using the relative 2d movement which is included in every mouse event and multiplying it by a factor. This factor must be specified on creation of the component.

Components

SimpleRotationInteractor

> SimpleRotationInteractor(...) -> A new SimpleRotationInteractor component.
>
> A simple interactor for rotating OpenGLComponents around the X,Y axes.
>
> Keyword arguments:

- rotationfactor -- factor to translate between 2d movment and 3d rotation (default=10.0)

Kamaelia.UI.OpenGL.SimpleTranslationInteractor

Simple Translation Interactor

A simple interactor for moving OpenGLComponents along th X,Y plane.

SimpleTranslationInteractor is a subclass of Interactor.

Example Usage

The following example shows four SimpleCubes which can be moved by dragging your mouse over them:

```
o1 = SimpleCube(position=(6, 0,-30), size=(1,1,1)).activate()
i1 = SimpleTranslationInteractor(target=o1).activate()

o2 = SimpleCube(position=(0, 0,-20), size=(1,1,1)).activate()
i2 = SimpleTranslationInteractor(target=o2).activate()

o3 = SimpleCube(position=(-3, 0,-10), size=(1,1,1)).activate()
i3 = SimpleTranslationInteractor(target=o3).activate()

o4 = SimpleCube(position=(15, 0,-40), size=(1,1,1)).activate()
i4 = SimpleTranslationInteractor(target=o4).activate()

Axon.Scheduler.scheduler.run.runThreads()
```

How does it work?

SimpleTranslationInteractor is a subclass of Interactor (for Interactor functionality see its documentation). It overrides the ___init___(), setup() and handleEvents() methods.

The amount of movement is determined using the relative 2d movement which is included in every mouse event and multiplying it by a factor. This factor must be specified on creation of the component.

Components

SimpleTranslationInteractor

SimpleTranslationInteractor(...) -> A new SimpleTranslationInteractor component.

A simple interactor for moving OpenGLComponents along th X,Y plane.

Keyword arguments:

- translationfactor -- factor to translate between 2d and 3d movement (default=10.0)

Kamaelia.UI.OpenGL.SkyGrassBackground

Sky & Grass background

A very simple component showing a plane with the upper half coloured light blue and the lower half green. Can be used for a background.

This component is a subclass of OpenGLComponent and therefore uses the OpenGL display service.

Example Usage

Only a background:

```
SkyGrassBackground(size=(5000,5000,0), position=(0,0,-100)).activate()
Axon.Scheduler.scheduler.run.runThreads()
```

Components

SkyGrassBackground

SkyGrassBackground(...) -> A new SkyGrassBackground component.

A very simple component showing a plane with the upper half coloured light blue and the lower half green. Can be used for a background.

Kamaelia.UI.OpenGL.TexPlane

Textured Plane

A plane showing a texture loaded from an image file.

This component is a subclass of OpenGLComponent and therefore uses the OpenGL display service.

Example Usage

A plane showing an image loaded from the file "nemo.jpeg":

```
PLANE = TexPlane(position=(0, 0,-6), texture="nemo.jpeg").activate()

Axon.Scheduler.scheduler.run.runThreads()
```

How does it work?

This component is a subclass of OpenGLComponent (for OpenGLComponent functionality see its documentation). It overrides ___init___(), setup(), draw().

In setup() the method loadTexture() get called which loads the texure from the image file specified. If the image in the file has dimensions which are not equal a power of two, the texture dimensions get enlarged (this is needed because of OpenGL texturing limitations).

In draw() a simple plane is drawn whith the loaded texture on it.

Components

TexPlane

TexPlane(...) -> A new TexPlane component.

A plane showing a texture loaded from an image file.

Keyword arguments:

- tex -- image file name
- pixelscaling -- factor for translation from pixels to units in 3D space (default=100.0)

Kamaelia.UI.OpenGL.Transform

3D Transform class

A class containing a transformation matrix and providing several methods to use/alter it. Transform uses Vector objects for most of its methods arguments.

Other

Transform

Transform([matrix]) -> new Transform object.

Keyword arguments:

- m -- A matrix containing values to be initially set

Kamaelia.UI.OpenGL.Vector

3D Vector class

A class for 3 dimensional vectors providing several methods for common vector operations.

Other

Vector

Vector([x][,y][,z]) -> A new Vector object.

Keyword arguments:

- x,y,z -- Initial values.

Kamaelia.UI.Pygame.Button

Pygame Button Widget

A button widget for pygame display surfaces. Sends a message when clicked.

Uses the Pygame Display service.

Example Usage

Three buttons that output messages to the console:

```
button1 = Button(caption="Press SPACE or click",key=K_SPACE).activate()
button2 = Button(caption="Reverse colours",fgcolour=(255,255,255),bgcolour=(0,0,0)).activate
button3 = Button(caption="Mary...",msg="Mary had a little lamb", position=(200,100)).activat

ce = ConsoleEchoer().activate()
button1.link( (button1,"outbox"), (ce,"inbox") )
button2.link( (button2,"outbox"), (ce,"inbox") )
button3.link( (button3,"outbox"), (ce,"inbox") )
```

How does it work?

The component requests a display surface from the Pygame Display service component. This is used as the surface of the button. It also binds event listeners to the service, as appropriate.

Arguments to the constructor configure the appearance and behaviour of the button component:

- If an output "msg" is not specified, the default is a tuple ("CLICK", id) where id is the self.id attribute of the component.

- A pygame keycode can be specified that will also trigger the button as if it had been clicked
- you can set the text label, colour, margin size and position of the button
- the button can have a transparent background
- you can specify a size as width,height. If specified, the margin size is ignored and the text label will be centred within the button

If a producerFinished or shutdownMicroprocess message is received on its "control" inbox. It is passed on out of its "signal" outbox and the component terminates.

Upon termination, this component does *not* unbind itself from the Pygame Display service. It does not deregister event handlers and does not relinquish the display surface it requested.

Components

Button

Button(...) -> new Button component.

Create a button widget in pygame, using the Pygame Display service. Sends a message out of its outbox when clicked.

Keyword arguments (all optional):

- caption -- text (default="Button <component id>")
- position -- (x,y) position of top left corner in pixels
- margin -- pixels margin between caption and button edge (default=8)
- bgcolour -- (r,g,b) fill colour (default=(224,224,224))
- fgcolour -- (r,g,b) text colour (default=(0,0,0))
- msg -- sent when clicked (default=("CLICK",self.id))
- key -- if not None, pygame keycode to trigger click (default=None)
- transparent -- draw background transparent if True (default=False)
- size -- None or (w,h) in pixels (default=None)

Kamaelia.UI.Pygame.Display

Pygame Display Access

This component provides a pygame window. Other components can request to be notified of events, or ask for a pygame surface or video overlay that will be rendered onto the display.

Pygame Display is a service that registers with the Coordinating Assistant Tracker (CAT).

Example Usage

See the Button component or VideoOverlay component for examples of how Pygame Display can be used.

How does it work?

Pygame Display is a service. obtain it by calling the PygameDisplay.getDisplayService(...) static method. Any existing instance will be returned, otherwise a new one is automatically created.

Alternatively, if you wish to configure Pygame Display with options other than the defaults, create your own instance, then register it as a service by calling the PygameDisplay.setDisplayService(...) static method. NOTE that it is only advisable to do this at the top level of your system, as other components may have already requested and created a Pygame Display component!

pygame only supports one display window at a time, you must not make more than one Pygame Display component.

Pygame Display listens for requests arriving at its "notify" inbox. A request can be to: - create or destroy a surface, - listen or stop listening to events (you must have already requested a surface) - move an existing surface - create a video overlay - notify of ne to redraw

The requests are described in more detail below.

Once your component has been given the requested surface, it is free to render onto it whenever it wishes. It should then immediately send a "REDRAW" request to notify Pygame Display that the window needs redrawing.

NOTE that you must set the alpha value of the surface before rendering and restore its previous value before yielding. This is because Pygame Display uses the alpha value to control the transparency with which it renders the surface.

Overlays work differently: instead of being given something to render to, you must provide, in your initial request, an outbox to which you will send raw yuv (video) data, whenever you want to change the image on the overlay.

Pygame Display instantiates a private, threaded component to listen for pygame events. These are then forwarded onto Pygame Display.

Pygame Display's main loop continuously renders the surfaces and video overlays onto the display, and dispatches any pygame events to listeners. The rendering order is as follows: - background fill (default=white) - surfaces (in the order they were requested and created) - video overlays (in the order they were requested and created)

In summary, to use a surface, your component should: 1. Obtain and wire up to the "notify" inbox of the Pygame Display service 2. Request a surface 3. Render onto that surface in its main loop

And to use overlays, your component should: 1. Obtain and wire up to the "notify" inbox of the Pygame Display service 2. Request an overlay, providing an outbox 3. Send yuv data to the outbox

This component does not terminate. It ignores any messages arriving at its "control" inbox and does not send anything out of its "outbox" or "signal" outboxes.

Surfaces

To request a surface, send a dictionary to the "notify" inbox. The following keys are mandatory:

```
{
    "DISPLAYREQUEST" : True,               # this is a 'new surface' request
    "size" : (width,height),               # pixels size for the new surface
    "callback" : (component, "inboxname")  # to send the new surface object to
}
```

These keys are optional:

```
{
    "position" : (left,top)                # location of the new surface in the window (defa
    "alpha" : 0 to 255,                    # alpha of the surface (255=opaque) (default=255)
    "transparency" : (r,g,b),              # colour that will appear transparent (default=No
    "events" : (component, "inboxname"),   # to send event notification to (default=None)
}
```

To deregister your surface, send a producerFinished(surface) message to the "notify" inbox. Where 'surface' is your surface. This will remove your surface and deregister any events you were listening to.

To change the position your surface is rendered at, send a dictionary to the "notify" inbox containing the folling keys:

```
{
    "CHANGEDISPLAYGEO" : True,             # this is a 'change geometry' request
    "surface" : surface,                   # the surface to affect
    "position" : (left,top)                # new location for the surface in the window
}
```

The "surface" and "position" keys are optional. However if either are not specified then there will be no effect!

Listening to events

Once your component has obtained a surface, it can request to be notified of specific pygame events.

To request to listen to a given event, send a dictionary to the "notify" inbox, containing the following:

```
{
    "ADDLISTENEVENT" : pygame_eventtype,     # example: pygame.KEYDOWN
```

```
    "surface" : your_surface,
}
```

To unsubscribe from a given event, send a dictionary containing:

```
{
    "REMOVELISTENEVENT" : pygame_eventtype,
    "surface" : your_surface,
}
```

Events will be sent to the inbox specified in the "events" key of the "DISPLAYREQUEST" message. They arrive as a list of pygame event objects.

NOTE: If the event is MOUSEMOTION, MOUSEBUTTONUP or MOUSEBUTTONDOWN then you will instead receive a replacement object, with the same attributes as the pygame event, but with the 'pos' attribute adjusted so that (0,0) is the top left corner of *your* surface.

Video Overlays

To request an overlay, send a dictionary to the "notify" inbox. The following keys are mandatory:

```
{
    "OVERLAYREQUEST" : True,                      # this is a 'new overlay' request
    "size" : (width,height),                      # pixels size of the overlay
    "pixformat" : pygame_pixformat,               # example: pygame.IYUV_OVERLAY
}
```

These keys are optional:

```
{
    "position" : (left,top),                      # location of the overlay (default=(0,0))
    "yuv" : (ydata,udata,vdata),                  # first frame of yuv data
    "yuvservice" : (component,"outboxname"),      # source of future frames of yuv data
    "positionservice" : (component,"outboxname"), # source of changes to the overlay positio
}
```

"yuv" enables you to provide the first frame of video data. It should be 3 strings, containing the yuv data for a whole frame.

If you have supplied a (component,outbox) pair as a "yuvservice" then any (y,u,v) data sent to that outbox will update the video overlay. Again the data should be 3 strings, containing the yuv data for a *whole frame*.

If you have supplied a "positionservice", then sending (x,y) pairs to the outbox you specified will update the position of the overlay.

There is currently no mechanism to destroy an overlay.

Redraw requests

To notify Pygame Display that it needs to redraw the display, send a dictionary containing the following keys to the "notify" inbox:

```
{
    "REDRAW" : True,             # this is a redraw request
    "surface" : surface          # surface that has been changed
}
```

Implementation Details

You may notice that this module also contains a _PygameEventSource component. PygameDisplay uses this separate threaded component to notify it when pygame events occur - so that it can sleep quiescently when it has nothing to do.

Unfortunately event handling itself cannot be done in the thread since pygame on many platforms (particularly win32) does not work properly if event handling and display creation is not done in the main thread of the program.

Components

PygameDisplay

PygameDisplay(...) -> new PygameDisplay component

Use PygameDisplay.getDisplayService(...) in preference as it returns an existing instance, or automatically creates a new one.

Or create your own and register it with setDisplayService(...)

Keyword arguments (all optional):

- width -- pixels width (default=800)
- height -- pixels height (default=600)
- background_colour -- (r,g,b) background colour (default=(255,255,255))
- fullscreen -- set to True to start up fullscreen, not windowed (default=False)

Other

Bunch

NO DOCS

Kamaelia.UI.Pygame.EventHandler

Pygame event handling

A simple framework for handling pygame events. Reimplement the appropriate stub method to handle a given event.

Example Usage

Detecting key presses and mouse button depressions and releases:: class MyEventHandler(Even
super(MyEventHandler,self).___init___() self.target = target

def keydown(self, key, mod, where): print ("Keypress '"+key+"' detected by "+where)

def mousebuttondown(self, pos, button, where): print ("Mouse button depressed")

def mousebuttonup(self, pos, button, where): print ("Mouse button released")

How does it work?

Implement your event handler by subclassing EventHandler and reimplementing the stub methods for the particular events you wish to handle.

The code that reads the events from pygame should pass them one at a time to EventHandler by calling the dispatch(...) method.

The optional 'trace' argument to the initialiser, when non-zero, causes the existing stub handlers to print messages to standard out, notifying you that the given event has taken place.

Other

EventHandler

EventHandler([trace]) -> new EventHandler object.

Pygame event dispatcher. Subclass and reimplement the stub methods to handle events. Pass in events to be handled by calling the dispatch(...) method.

Keyword arguments:

- trace -- Cause existing stub handlers to print messages to standard output.

Kamaelia.UI.Pygame.Image

Pygame image display

Component for displaying an image on a pygame display. Uses the Pygame Display service component.

The image can be changed at any time.

Example Usage

Display that rotates rapidly through a set of images:

```
imagefiles = [ "imagefile1", "imagefile2", ... ]

class ChangeImage(Axon.Component.component):
    def __init__(self, images):
        super(ChangeImage,self).__init__()
        self.images = images

    def main(self):
        while 1:
            for image in self.images:
                self.send( image, "outbox")
                print "boing",image
                for i in range(0,100):
                    yield 1

image = Image(image=None, bgcolour=(0,192,0))
ic    = ChangeImage(imagefiles)

Pipeline(ic, image).run()
```

How does it work?

This component requests a display surface from the Pygame Display service component and renders the specified image to it.

The image, and other properties can be changed later by sending messages to its "inbox", "bgcolour" and "alphacontrol" inboxes.

Note that the size of display area is fixed after initialisation. If an initial size, or image is specified then the size is set to that, otherwise a default value is used.

Change the image at any time by sending a new filename to the "inbox" inbox. If the image is larger than the 'size', then it will appear cropped. If it is smaller, then the Image component's 'background colour' will show through behind it. The image is always rendered aligned to the top left corner.

If this component receives a shutdownMicroprocess or producerFinished message on its "control" inbox, then this will be forwarded out of its "signal" outbox and the component will then terminate.

Components

Image

Image([image][,position][,bgcolour][,size][,displayExtra][,maxpect]) -> new Image component

Pygame image display component. Image, and other properties can be changed at runtime.

Keyword arguments:

- image -- Filename of image (default=None)
- position -- (x,y) pixels position of top left corner (default=(0,0))
- bgcolour -- (r,g,b) background colour (behind the image if size>image size)
- size -- (width,height) pixels size of the area to render the iamge in (default=image size or (240,192) if no image specified)
- displayExtra -- dictionary of any additional args to pass in request to Pygame Display service
- maxpect -- (xscale,yscale) scaling to apply to image (default=no scaling)

Kamaelia.UI.Pygame.KeyEvent

Pygame keypress event handler

A component that registers with a Pygame Display service component to receive key-up and key-down events from Pygame. You can set up this component to send out different messages from different outboxes depending on what key is pressed.

Example Usage

Capture keypresses in pygame for numbers 1,2,3 and letters a,b,c:

```
import pygame

Graphline( output = ConsoleEchoer(),
           keys = KeyEvent( key_events={ pygame.K_1 : (1,"numbers"),
                                         pygame.K_2 : (2,"numbers"),
                                         pygame.K_3 : (3,"numbers"),
                                         pygame.K_a : ("A", "letters"),
                                         pygame.K_b : ("B", "letters"),
```

```
                                    pygame.K_c : ("C", "letters"),
                                  },
                        outboxes={ "numbers" : "numbers between 1 and 3",
                                   "letters" : "letters between A and C",
                                 }
                      ),
    linkages = { ("keys","numbers"):("output","inbox"),
                 ("keys","letters"):("output","inbox")
               }
).run()
```

How does it work?

This component requests a zero sized display surface from the Pygame Display service component and registers to receive events from pygame.

Whenever a KEYDOWN event is received, the pygame keycode is looked up in the mapping you specified. If it is there, then the specified message is sent out of the specified outbox.

In addition, if the allKeys flag was set to True during initialisation, then any KEYDOWN or KEYUP event will result in a ("DOWN",keycode) or ("UP",keycode) message being sent to the "allkeys" outbox.

If you have specified a message to send for a particular key, then both that message and the 'all-keys' message will be sent when the KEYDOWN event occurs.

If this component receives a shutdownMicroprocess or producerFinished message on its "control" inbox, then this will be forwarded out of its "signal" outbox and the component will then terminate.

Components

KeyEvent

KeyEvent([allkeys][,key_events][,outboxes]) -> new KeyEvent component.

Component that sends out messages in response to pygame keypress events.

Keyword arguments:

- allkeys -- if True, all keystrokes send messages out of "allkeys" outbox (default=False)
- key_events -- dict mapping pygame keycodes to (msg,"outboxname") pairs (default=None)
- outboxes -- dict of "outboxname":"description" key:value pairs (default={})

Kamaelia.UI.Pygame.MagnaDoodle

Simple Pygame drawing board

A simple drawing board for the pygame display service.

Use your left mouse button to draw to the board and the right to erase your artwork.

Components

MagnaDoodle

MagnaDoodle(...) -> A new MagnaDoodle component.

A simple drawing board for the pygame display service.

(this component and its documentation is heaviliy based on Kamaelia.UI.Pygame.Button)

Keyword arguments:

- position -- (x,y) position of top left corner in pixels
- margin -- pixels margin between caption and button edge (default=8)
- bgcolour -- (r,g,b) fill colour (default=(224,224,224))
- fgcolour -- (r,g,b) text colour (default=(0,0,0))
- transparent -- draw background transparent if True (default=False)
- size -- None or (w,h) in pixels (default=None)

Kamaelia.UI.Pygame.Multiclick

Pygame Multi-click Button Widget

A button widget for pygame display surfaces. Sends a message when clicked. The message can be different for each mouse button.

Uses the Pygame Display service.

Example Usage

Three buttons that output messages to the console:

```
msgs = [ "button 1", "button 2", "button 3", "button 4", "button 5" ]
button1 = Button(caption="Click different mouse buttons!",msgs=msgs).activate()

ce = ConsoleEchoer().activate()
button1.link( (button1,"outbox"), (ce,"inbox") )
```

How does it work?

The component requests a display surface from the Pygame Display service component. This is used as the surface of the button. It also binds event listeners to the service, as appropriate.

Arguments to the constructor configure the appearance and behaviour of the button component:

- If "msgs" is specified, then a different message can be specified for each mouse button. If it is not specified, then "msg" is used instead, for all buttons.
- If an output "msg" is not specified, the default is a tuple ("CLICK", id) where id is the self.id attribute of the component.
- you can set the text label, colour, margin size, size and position of the button
- if you do not specify the size yourself, the size will default to fit the caption of the button.
- the button can have a transparent background

If a producerFinished or shutdownMicroprocess message is received on its "control" inbox. It is passed on out of its "signal" outbox and the component terminates.

Upon termination, this component does *not* unbind itself from the Pygame Display service. It does not deregister event handlers and does not relinquish the display surface it requested.

Components

Multiclick

Multiclick(...) -> new Multiclick component.

Create a button widget in pygame, using the Pygame Display service. Sends a message out of its outbox when clicked.

Keyword arguments (all optional):

- caption -- text (default="Button <component id>")
- position -- (x,y) position of top left corner in pixels
- margin -- pixels margin between caption and button edge (default=8)
- bgcolour -- (r,g,b) fill colour (default=(224,224,224))
- fgcolour -- (r,g,b) text colour (default=(0,0,0))
- msg -- sent when clicked (default=("CLICK",self.id)) of msgs is not specified
- msgs -- list of messages. msgs[x] is sent when button X is clicked (default=None)

- transparent -- draw background transparent if True (default=False)
- size -- (width,height) pixels size of the button (default=scaled to fit caption)

Kamaelia.UI.Pygame.Text

Pygame components for text input and display

TextDisplayer displays any data it receives on a Pygame surface. Every new piece of data is displayed on its own line, and lines wrap automatically.

Textbox displays user input while the user types, and sends its string buffer to its 'outbox' when it receives a ' '.

Example Usage

To take user input in Textbox and display it in TextDisplayer:

```
Pipeline(Textbox(size = (800, 300),
                 position = (0,0)),
         TextDisplayer(size = (800, 300),
                       position = (0,340))
         ).run()
```

How does it work?

TextDisplayer requests a display from the Pygame Display service and requests that Pygame Display send all keypresses to it. Every time TextDisplayer receives a keypress, it updates its string buffer and the display.

If it receives a newline, or if text must wrap, it moves the existing text upwards and blits the new line onto the bottom.

Known issues

The line wrapping length is specified by the width of the display divided by the width of the letter 'a' in the displayed font, so lines may wrap too far off the edge of the screen if the user types very narrow text (i.e. just spaces with no other charachters), or too far inside the edge of the screen (usually).

Components

TextDisplayer

TextDisplayer(...) -> new TextDisplayer Pygame component.

Keyword arguments:

- **size -- (w, h) size of the TextDisplayer surface, in pixels.**
  Default (500, 300).
- text_height -- font size. Default 18.
- **bgcolour -- tuple containing RGB values for the background color.**
  Default is a pale yellow.
- **fgcolour -- tuple containing RGB values for the text color.**
  Default is black.
- **position -- tuple containing x,y coordinates of the surface's**
  upper left corner in relation to the Pygame window. Default (0,0)

Textbox

Textbox(...) -> New Pygame Textbox component

Keyword Arguments:

- Textbox inherits its keyword arguments from TextDisplayer. Please see TextDisplayer docs.

Reads keyboard input and updates it on the screen. Flushes string buffer and sends it to outbox when a newline is encountered.

Kamaelia.UI.Pygame.Ticker

Pygame text 'Ticker'

Displays text in pygame a word at a time as a 'ticker'.

NOTE: This component is very much a work in progress. Its capabilities and API is likely to change substantially in the near future.

Example Usage

Ticker displaying text from a file:

```
Pipeline( RateControlledFileReader("textfile","lines",rate=1000),
          Ticker(position=(100,100))
        ).run()
```

How does it work?

The component requests a display surface from the Pygame Display service component. This is used as the ticker.

Send strings containing *lines of text* to the Ticker component. Do not send strings with words split between one string and the next. It displays the words

as a 'ticker' one word at a time. Text is automatically wrapped from one line to the next. Once the bottom of the ticker is reached, the text automatically jump-scrolls up a line to make more room.

The text is normalised by the ticker. Multiple spaces between words are collapsed to a single space. Linefeeds are ignored.

NOTE: 2 consecutive linefeeds currently results in a special message being sent out of the "_displaysignal" outbox. This is work-in-progress aimed at new features. It is only documented here for completeness and should not be relied upon.

You can set the text size, colour and line spacing. You can also set the background colour, outline (border) colour and width. You can also specify the size and position of the ticker

NOTE: Specifying the outline width currently does not work for any value other than 1.

NOTE: Specify the size of the ticker with the render_right and render_bottom arguments. Specifying render_left and render_top arguments with values other than 1 results in parts of the ticker being obscured.

The ticker displays words at a constant rate - it self regulates its display speed.

Whilst it is running, sending any message to the "pausebox" inbox will pause the Ticker. It will continue to buffer incoming text. Any message sent to the "unpausebox" inbox will cause the Ticker to resume.

Whilst running, you can change the transparency of the ticker by sending a value to the "alphacontrol" inbox between 0 (fully transparent) and 255 (fully opaque) inclusive.

If a producerFinished message is received on the "control" inbox, this component will send its own producerFinished message to the "signal" outbox and will terminate.

However, if the ticker is paused (message sent to "pausebox" inbox) then the component will ignore messages on its "control" inbox until it is unpaused by sending a message to its "unpausebox" inbox.

Components

Ticker

Ticker(...) -> new Ticker component.

A pygame based component that displays incoming text as a ticker.

Keyword arguments (all optional):

- text_height -- Font size in points (default=39)

- line_spacing -- (default=text_height/7)
- background_colour -- (r,g,b) background colour of the ticker (default=(128,48,128))
- text_colour -- (r,g,b) colour of text (default=(232,232,48))
- outline_colour -- (r,g,b) colour of the outline border (default=background_colour)
- outline_width -- pixels width of the border (default=1)
- position -- (x,y) pixels location of the top left corner
- render_left -- pixels distance of left of text from left edge (default=1)
- render_top -- pixels distance of top of text from top edge (default=1)
- render_right -- pixels width of ticker (default=399)
- render_bottom -- pixels height of ticker (default=299)

NOTE: render_left and render_top currently behave incorrectly if not set to 1

Other

GotShutdownException

NO DOCS

Kamaelia.UI.Pygame.VideoSurface

Pygame Video Surface

Displays uncompressed RGB video data on a pygame surface using the Pygame Display service.

Example Usage

Read raw YUV data from a file, convert it to interleaved RGB and display it using VideoSurface:

```
imagesize = (352, 288)        # "CIF" size video
fps = 15                      # framerate of video

Pipeline(ReadFileAdapter("raw352x288video.yuv", ...other args...),
         RawYUVFramer(imagesize),
         MessageRateLimit(messages_per_second=fps, buffer=fps*2),
         ToRGB_interleaved(),
         VideoSurface(),
        ).activate()
```

RawYUVFramer is needed to frame raw YUV data into individual video frames. ToRGB_interleaved is needed to convert the 3 planes of Y, U and V data to a single plane containing RGB data interleaved (R, G, B, R, G, B, R, G, B, ...)

How does it work?

The component waits to receive uncompressed video frames from its "inbox" inbox.

The frames must be encoded as dictionary objects in the format described below.

When the first frame is received, the component notes the size and pixel format of the video data and requests an appropriate surface from the Pygame Display service component, to which video can be rendered.

NOTE: Currently the only supported pixelformat is "RGB_interleaved".

When subsequent frames of video are received the rgb data is rendered to the surface and the Pygame Display service is notified that the surface needs redrawing.

At present, VideoSurface cannot cope with a change of pixel format or video size mid sequence.

UNCOMPRESSED FRAME FORMAT

Uncompresed video frames must be encoded as dictionaries. VideoSurface requires the following entries:

```
{
  "rgb" : rgbdata                     # a string containing RGB video data
  "size" : (width, height)            # in pixels
  "pixformat" : "RGB_interleaved"     # format of raw video data
}
```

Components

VideoSurface

VideoSurface([position]) -> new VideoSurface component

Displays a pygame surface using the Pygame Display service component, for displaying RGB video frames sent to its "inbox" inbox.

The surface is sized and configured by the first frame of (uncompressed) video data is receives.

Keyword arguments:

- position -- (x,y) pixels position of top left corner (default=(0,0))

Kamaelia.UI.PygameDisplay

This is a deprecation stub, due for later removal.

Other

PygameDisplay

NO DOCS

Kamaelia.UI.Tk.TkWindow

Simple Tk Window base class

A simple component providing a framework for having a Tk window as a component.

TkInvisibleWindow is a simple implementation of an invisible (hidden) Tk window, useful if you want none of the visible windows to be the Tk root window.

Example Usage

Three Tk windows, one with "Hello world" written in it:: class MyWindow(TkWindow): def __
self.title = title self.text = text super(MyWindow,self).___init___()

def setupWindow(self): self.label = Tkinter.Label(self.window, text=self.text)

self.window.title(self.title)

self.label.grid(row=0, column=0, sticky=Tkinter.N+Tkinter.E+Tkinter.W+Tkinter.S) self.window.rowconfigure(0, weight=1) self.window.columnconfigure(0, weight=1)

root = TkWindow().activate() # first window created is the root win2 = MyWindow("MyWindow","Hello world!").activate()

scheduler.run.runThreads(slowmo=0)

How does it work?

This component provides basic integration for Tk. It creates and sets up a Tk window widget, and then provides a kamaelia main loop that ensures Tk's own event processing loop is regularly called.

self.window contains the Tk window widget.

To set up your own widgets and event handling bindings for the window, reimplement the setupWindow() method.

NOTE: Do not bind the "<Destroy>" event as this is already handled. Instead, reimplement the destroyHandler() method. This is guaranteed to only be called if the destroy event is for this specific window.

The first window instantiated is the Tk "root" window. Note that closing this window will result in Tk trying to close down. To avoid this style of behaviour, create a TkInvisibleWindow as the root.

The existing main() method ensures Tk's event processing loop is regularly called.

You can reimplement main(). However, you must ensure you include the existing functionality: - regularly calling tkupdate() to ensure Tk gets to process its own events - calls self.window.destroy() method to destroy the window upon shutdown. - finishes if self.isDestroyed() returns True

The existing main() method will cause the component to terminate if a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It sends the message on out of the "signal" outbox and calls self.window.destroy() to ensure the window is destroyed.

NOTE: main() must not ask to be paused as it calls the Tk event loop. If the Tk event loop is not called, then a Tk app will freeze and be unable to respond to events.

NOTE: Event bindings are called from within Tk event handling. If, for example, there are two (or more) TkWindow instances, then a given event handler could be called whilst the thread of execution is actually within the other TkWindow component's main() method. This is a bit messy. It will not cause problems in a single threaded system, but may be an issue once Axon/Kamaelia is able to distribute across multiple processors.

Development History

Started as a first hash attempt at some components to incorporate Tkinter into Kamaelia in cvs:/Sketches/tk/tkInterComponents.py

Turned out to be remarkably resilient so far, so migrated into the main codebase.

Components

TkWindow

> TkWindow() -> new TkWindow component
>
> A component providing a Tk window. The first TkWindow created will be the "root" window.
>
> Subclass to implement your own widgets and functionality on the window.

tkInvisibleWindow

tkInvisibleWindow() -> new tkInvisibleWindow component.

An invisible, empty tk window. Can use as a 'root' window, thereby ensuring closing any (visible) window doesn't terminate Tk (closing the root does).

Kamaelia.Util.Backplane

Publishing and Subscribing with Backplanes

Backplanes provide a way to 'publish' data under a name, enabling other parts of the system to 'subscribe' to it on the fly, without having to know about the actual component(s) the data is coming from.

It is a quick and easy way to distribute or share data. Think of them like backplane circuit boards - where other circuit boards can plug in to send or receive any signals they need.

Example usage

A system where several producers publish data, for consumers to pick up:

```
Pipeline( Producer1(),
          PublishTo("DATA")
        ).activate()

Pipeline( Producer2(),
          PublishTo("DATA")
        ).activate()

Pipeline( SubscribeTo("DATA"),
          Consumer1(),
        ).activate()

Pipeline( SubscribeTo("DATA"),
          Consumer2(),
        ).activate()

Backplane("DATA").run()
```

A server where multiple clients can connect and they all get sent the same data at the same time:

```
Pipeline( Producer(),
          PublishTo("DATA")
        ).activate()
```

```
SimpleServer(protocol=SubscribeTo("DATA"), port=1500).activate()

Backplane("DATA").run()
```

More detail

The Backplane component collects data from publishers and sends it out to subscribers.

You can have as many backplanes as you like in a running system - provided they all register under different names.

A backplane can have multiple subscribers and multiple publishers. Publishers and subscribers can be created and destroyed on the fly.

To shut down a PublishTo() component, send a producerFinished() or shutdownMicroprocess() message to its "control" inbox. This does *not* propagate and therefore does *not* cause the Backplane or any subscribers to terminate.

To shut down a SubscribeTo() component, send a producerFinished() or shutdownMicroprocess() message to its "control" inbox. It will then immediately forward the mesage on out of its "signal" outbox and terminate.

To shut down the Backplane itself, send a producerFinished() or shutdownMicroprocess() message to its "control" inbox. It will then immediately terminate and also propagate this message onto any subscribers (SubscribeTo components), causing them to also terminate.

Implementation details

Backplane is actually based on a Kamaelia.Util.Splitter.PlugSplitter component, and the SubscribeTo component is a wrapper around a Kamaelia.Util.Splitter.Plug.

The Backplane registers itself with the coordinating assistant tracker.

- Its "inbox" inbox is registered under the name "Backplane_I<name>"
- Its "configuration" inbox is registered under the name "Backplane_O<name>"

PublishTo components look up the "Backplane_I<name>" service and simply forward data sent to their "inbox" inboxes direct to the "inbox" inbox of the PlugSplitter - causing it to be distributed to all subscribers.

SubscribeTo components look up the "Backplane_O<name>" service and request to have their "inbox" and "control" inboxes connected to the PlugSplitter. SubscribeTo then forwards on any messages it receives out of its "outbox" and "signal" outboxes respectively.

The PlugSplitter component's "control" inbox and "signal" outbox are not advertised as services. To shut down a Backplane you must therefore send a

shutdownMicroprocess() or producerFinished() message directly to its "control" inbox. When this happens, the shutdown message will be forwarded on to all subscribers - causing SubscribeTo components to also shut down.

Components

Backplane

Backplane(name) -> new Backplane component.

A named backplane to which data can be published for subscribers to pick up.

- Use PublishTo components to publish data to a Backplane.
- Use SubscribeTo components to receive data published to a Backplane.

Keyword arguments:

- name -- The name for the backplane. publishers and subscribers connect to this by using the same name.

PublishTo

PublishTo(destination) -> new PublishTo component

Publishes data to a named Backplane. Any data sent to the "inbox" inbox is sent to all (any) subscribers to the same named Backplane.

Keyword arguments:

- destination -- the name of the Backplane to publish data to

SubscribeTo

SubscribeTo(source) -> new SubscribeTo component

Subscribes to a named Backplane. Receives any data published to that backplane and sends it on out of its "outbox" outbox.

Keyword arguments:

- source -- the name of the Backplane to subscribe to for data

Other

publishTo

NO DOCS

subscribeTo

NO DOCS

Kamaelia.Util.Chargen

Simple Character Generator

This component is intended as a simple 'stream of characters' generator.

At the moment, it continually sends the string "Hello world" as fast as it can, indefinitely out of its "outbox" outbox.

Example Usage

:: »> Pipeline(Chargen(), ConsoleEchoer()).run() Hello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHel lo WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello Wor ldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHello WorldH ello WorldHello WorldHello WorldHello WorldHello WorldHello WorldHell

... you get the idea!

How does it work?

This component, once activated repeatedly emits the string "Hello World" from its "outbox" outbox. It is emitted as a single string. It does this continuously forever. It is not rate limited in any way, and so emits as fast as it can.

This component does not terminate, and ignores messages arriving at any of its inboxes. It does not output anything from its "signal" outbox.

Components

Chargen

Chargen() -> new Chargen component.

Component that emits a continuous stream of the string "Hello World" from its "outbox" outbox as fast as it can.

Other

tests

NO DOCS

Kamaelia.Util.Chooser

Iterating Over A Predefined List

The Chooser component iterates (steps) forwards and backwards through a list of items. Request the next or previous item and Chooser will return it.

The ForwardIteratingChooser component only steps forwards, but can therefore handle more than just lists - for example: infinite sequences.

Example Usage

A simple slideshow:

```
items=[ "image1.png", "image2.png", "image3.png", ... ]

Graphline( CHOOSER  = Chooser(items=imagefiles),
           FORWARD  = Button(position=(300,16), msg="NEXT", caption="Next"),
           BACKWARD = Button(position=(16,16),  msg="PREV", caption="Previous"),
           DISPLAY  = Image(position=(16,64), size=(640,480)),
           linkages = { ("FORWARD" ,"outbox") : ("CHOOSER","inbox"),
                        ("BACKWARD","outbox") : ("CHOOSER","inbox"),
                        ("CHOOSER" ,"outbox") : ("DISPLAY","inbox"),
                      }
         ).run()
```

The chooser is driven by the 'next' and 'previous' Button components. Chooser then sends filenames to an Image component to display them.

Another example: a forever looping carousel of files, read at 1MBit/s:

```
def filenames():
    while 1:
        yield "file 1"
        yield "file 2"
        yield "file 3"

JoinChooserToCarousel( chooser = InfiniteChooser(items=filenames),
                       carousel = FixedRateControlledReusableFilereader("byte",rate=131072,c
                     )
```

How does it work?

When creating it, pass the component a set of items for it to iterate over.

Chooser will only accept finite length datasets. InfiniteChooser will accept any interable sequence, even one that never ends from a generator.

Once activated, the component will emit the first item from the list from its "outbox" outbox.

If the list/sequence is empty, then nothing is emitted, even in response to messages sent to the "inbox" inbox described now.

Send commands to the "inbox" inbox to move onto another item of data and cause it to be emitted. This behaviour is very much like a database cursor or file pointer - you are issuing commands to step through a dataset.

Send "SAME" and the component will emit the same item of data that was last emitted last time. Both Chooser and InfiniteChooser respond to this request.

Send "NEXT" and the component will emit the next item from the list or sequence. If there is no 'next' item (becuase we are already at the end of the list/sequence) then nothing is emitted. Both Chooser and InfiniteChooser respond to this request.

With InfiniteChooser, if there is not 'next' item then, additionally, a producerFinished message will be sent out of its "signal" outbox to signal that the end of the sequence has been reached. The component will then terminate.

All requests described from now are only supported by the Chooser component. InfiniteChooser will ignore them.

Send "PREV" and the previous item from the list or sequence will be emitted. If there is no previous item (because we are already at the front of the list/sequence) then nothing is emitted.

Send "FIRST" or "LAST" and the first or last item from the list or sequence will be emitted, respectively. The item will be emitted even if we are already at the first/last item.

If Chooser or InfiniteChooser receive a shutdownMicroprocess message on the "control" inbox, they will pass it on out of the "signal" outbox. The component will then terminate.

Components

Chooser

Chooser([items], [loop]) -> new Chooser component.

Iterates through a finite list of items. Step by sending "NEXT", "PREV", "FIRST" or "LAST" messages to its "inbox" inbox.

If loop is provided and true, then the iterates round the given items.

Keyword arguments:

- items -- list of items to be chosen from, must be type 'list' (default=[])

ForwardIteratingChooser

Chooser([items]) -> new Chooser component.

Iterates through an iterable set of items. Step by sending "NEXT" messages to its "inbox" inbox.

Keyword arguments: - items -- iterable source of items to be chosen from (default=[])

Kamaelia.Util.ChunkNamer

Chunk Namer

A component that labels each message with a unique filename for that message. e.g. "A" ... "B" ... --> ["chunk1", "A"] ... ["chunk2", "B"] ...

Example Usage

Save each line entered to the console to a separate file:

```
pipeline(
    ConsoleReader(),
    ChunkNamer("test", ".txt"),
    WholeFileWriter()
).run()
```

Components

ChunkNamer

ChunkNamer() -> new ChunkNamer component.

Gives a filename to the chunk and sends it in the form [filename, contents], e.g. to a WholeFileWriter component.

Keyword arguments: -- basepath - the prefix to apply to the filename -- suffix - the suffix to apply to the filename

Kamaelia.Util.Chunkifier

Chunkifier

A component that fixes the message size of an input stream to a given value, outputting blocks of that size when sufficient input has accumulated. This component's input is stream orientated - all messages received are concatenated to the interal buffer without divisions.

Example Usage

Chunkifying a console reader:

```
pipeline(
    ConsoleReader(eol=""),
    Chunkifier(20),
    ConsoleEchoer()
).run()
```

How does it work?

Messages received on the "inbox" are buffered until at least N bytes have been collected. A message containing those first N bytes is sent out "outbox". A CharacterFIFO object is used to do this in linear time.

The usual sending of a producerFinished/shutdown to the "control" inbox will shut it down.

Components

Chunkifier

> Chunkifier([chunksize]) -> new Chunkifier component.
>
> Flow controller - collects incoming data and outputs it only as quanta of a given length in bytes (chunksize), unless the input stream ends (producerFinished).
>
> Keyword arguments: - chunksize -- Chunk size in bytes - nodelay -- if set to True, partial chunks will be output rather than buffering up data while waiting for more to arrive.

Other

CharacterFIFO

An efficient character queue type (designed to work in O(n) time for n characters).

Kamaelia.Util.Clock

Cheap And Cheerful Clock

Outputs the message True repeatedly. The interval between messages is the parameter "interval" specified at the creation of the component.

This component is useful because it allows another component to sleep, not using any CPU time, but waking periodically (components are unpaused when they are sent a message).

Why is it "cheap and cheerful"?

...Because it uses a thread just for itself. All clocks could share a single thread if some services kung-fu was pulled. Opening lots of threads is a bad thing - they have much greater overhead than normal generator-based components. However, the one-thread-per-clock approach used here is many times shorter and simpler than one using services.

Components

CheapAndCheerfulClock

Outputs the message True every interval seconds

Kamaelia.Util.Collate

Collate everything received into a single message

Buffers all data sent to it. When shut down, sends all data it has received as collated as a list in a single message.

Example Usage

Read a file, in small chunks, then collate them into a single chunk:

```
Pipeline( RateControlledFileReader("big_file", ... ),
          Collate(),
          ...
        )
```

Behaviour

Send data items to its "inbox" inbox to be collated.

Send a producerFinished or shutdownMicroprocess message to the "control" inbox to terminate this component.

All collated data items will be sent out of the "outbox" outbox as a list in a single message. The items are collated in the same order they first arrived.

The component will then send on the shutdown message to its "signal" outbox and immediately terminate.

Components

Collate

Collate() -> new Collate component.

Buffers all data sent to it. When shut down, sends all data it has received as a single message.

Kamaelia.Util.Comparator

Comparing two data sources

The Comparator component tests two incoming streams to see if the items they contain match (pass an equality test).

Example Usage

Compares contents of two files and prints "MISMATCH!" whenever one is found:

```
class DetectFalse(component):
    def main(self):
        while 1:
            yield 1
            if self.dataReady("inbox"):
                if not self.recv("inbox"):
                    print ("MISMATCH!")

Graphline( file1   = RateControlledFileReader(filename="file 1", ...),
           file2   = RateControlledFileReader(filename="file 2", ...),
           compare = Comparator(),
           fdetect = DetectFalse(),
           output  = ConsoleEchoer(),
           linkages = {
               ("file1","outbox") : ("compare","inA"),
               ("file2","outbox") : ("compare","inB"),
               ("compare", "outbox") : ("fdetect", "inbox"),
               ("fdetect", "outbox") : ("output", "inbox"),
           },
         ).run()
```

How does it work?

The component simply waits until there is data ready on both its "inA" and "inB" inboxes, then takes an item from each and compares them. The result of the comparison is sent to the "outbox" outbox.

If data is available at neither, or only one, of the two inboxes, then the component will wait indefinitely until data is available on both.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox, then a producerFinished message is sent out of the "signal" outbox and the component terminates.

The comparison is done by the combine() method. This method returns the result of a simple equality test of the two arguments.

You could always subclass this component and reimplement the combine() method to perform different functions (for example, an 'adder').

Components

Comparator

> Comparator() -> new Comparator component.
>
> Compares items received on "inA" inbox with items received on "inB" inbox. For each pair, outputs True if items compare equal, otherwise False.

Other

comparator

NO DOCS

Kamaelia.Util.Console

Console Input/Output

The ConsoleEchoer component outputs whatever it receives to the console.

The ConsoleReader component outputs whatever is typed at the console, a line at a time.

Example Usage

Whatever it typed is echoed back, a line at a time:

```
Pipeline( ConsoleReader(),
          ConsoleEchoer()
        ).run()
```

How does it work?

ConsoleReader is a threaded component. It provides a 'prompt' at which you can type. Your input is taken, a line a a time, and output to its "outbox" outbox, with the specified end-of-line character(s) suffixed onto it.

The ConsoleReader component ignores any input on its "inbox" and "control" inboxes. It does not output anything from its "signal" outbox.

The ConsoleReader component does not terminate.

The ConsoleEchoer component receives data on its "inbox" inbox. Anything it receives this way is displayed on standard output. All items are passed through the str() builtin function to convert them to strings suitable for display.

However, if the 'use_repr' argument is set to True during initialization, then repr() will be used instead of str(). Similarly if a "tag" is provided it's prepended before the data.

If the 'forwarder' argument is set to True during initialisation, then whatever is received is not only displayed, but also set on to the "outbox" outbox (unchanged).

If a producerFinished or shutdownMicroprocess message is received on the ConsoleEchoer component's "control" inbox, then it is sent on to the "signal" outbox and the component then terminates.

Components

ConsoleReader

ConsoleReader([prompt][,eol]) -> new ConsoleReader component.

Component that provides a console for typing in stuff. Each line is output from the "outbox" outbox one at a time.

Keyword arguments:

- prompt -- Command prompt (default="»> ")
- eol -- End of line character(s) to put on end of every line outputted (default is newline)

ConsoleEchoer

ConsoleEchoer([forwarder][,use_repr][,tag]) -> new ConsoleEchoer component.

A component that outputs anything it is sent to standard output (the console).

Keyword arguments:

- forwarder -- incoming data is also forwarded to "outbox" outbox if True (default=False)
- use_repr -- use repr() instead of str() if True (default=False)
- tag -- Pre-pend this text tag before the data to emit

Kamaelia.Util.ConsoleEcho

This is a deprecation stub for later removal.

Other

consoleEchoer

NO DOCS

Kamaelia.Util.DataSource

Data Source component

This component outputs messages specified at its creation one after another.

Example Usage

To output "hello" then "world":

```
pipeline(
    DataSource(["hello", "world"]),
    ConsoleEchoer()
).run()
```

Triggered Source component

Whenever this component receives a message on inbox, it outputs a certain message.

Example Usage

To output "wibble" each time a line is entered to the console:

```
pipeline(
    ConsoleReader(),
    TriggeredSource("wibble"),
    ConsoleEchoer()
).run()
```

Components

DataSource

NO DOCS

Other

TriggeredSource

NO DOCS

Kamaelia.Util.Detuple

Components

SimpleDetupler

This class expects to recieve tuples (or more accurately indexable objects) on its inboxes. It extracts the item tuple[index] from the tuple (or indexable object) and passes this out its outbox.

This component does not terminate.

This component was originally created for use with the multicast component. (It could however be used for extracting a single field from a dictionary like object).

Example usage:

```
Pipeline(
    Multicast_transceiver("0.0.0.0", 1600, "224.168.2.9", 0),
    detuple(1), # Extract data, through away sender
    SRM_Receiver(),
    detuple(1),
    VorbisDecode(),
    AOAudioPlaybackAdaptor(),
).run()
```

Kamaelia.Util.Fanout

Sending an output to many places

This component copies data sent to its inbox to multiple, specified outboxes. This allows you to 'fan out' a data source to several predetermined destinations.

Example Usage

Output data source both to a file and to the console:

```
Graphline( source  = MyDataSource(...),
           split   = Fanout(["toConsole","toFile"]),
           file    = SimpleFileWriter(filename="outfile"),
           console = ConsoleEchoer(),
           linkages = {
             ("source","outbox")   : ("split","inbox"),
             ("split","toConsole") : ("console","inbox"),
```

```
      ("split","toFile")    : ("file","inbox"),
    }
).run()
```

How does it work?

At initialization, specify a list of names for outboxes. Once the component is activated, any data sent to its "inbox" inbox will be replicated out to the list of outboxes you specified.

In effect, data sent to the "inbox" inbox is 'fanned out' to the specified set of destinations.

Nothing is sent to the "outbox" outbox.

If a shutdownMicroprocess or producerFinished message is received on the "control" inbox, then it is sent on to the "signal" outbox and the component terminates.

There is no corresponding 'Fanout' of data flowing into the "control" inbox.

Components

Fanout

Fanout(boxnames) -> new Fanout component.

A component that copies anything received on its "inbox" inbox to the named list of outboxes.

Keyword arguments:

- boxnames -- list of names for the outboxes any input will be fanned out to.

Other

fanout

NO DOCS

Kamaelia.Util.Filter

Simple framework for filtering data

A framework for filtering a stream of data. Write an object providing a filter(...) method and plug it into a Filter component.

Example Usage

Filters any non-strings from a stream of data:

```
class StringFilter(object):
    def filter(self, input):
        if type(input) == type(""):
            return input
        else:
            return None            # indicates nothing to be output

myfilter = Filter(filter = StringFilter).activate()
```

How does it work?

Initialize a Filter component, providing an object with a filter(...) method.

The method should take a single argument - the data to be filtered. It should return the result of the filtering/processing. If that result is None then the component outputs nothing, otherwise it outputs whatever the value is that was returned.

Data received on the component's "inbox" inbox is passed to the filter(...) method of the object you provided. The result is output on the "outbox" outbox.

If a producerFinished message is received on the "control" inbox then it is sent on out of the "signal" outbox. The component will then terminate.

However, before terminating it will repeatedly call your object's filter(...) method, passing it an empty string ("") until the result returned is None. If not None, then whatever value the filter(...) method returned is output. This is to give your object a chance to flush any data it may have been buffering.

Irrespective of whether your filtering object buffers any data from one call to the next, you must ensure that (eventually) calling it with an empty string ("") will result in None being returned.

Components

Filter

Filter([filter]) -> new Filter component.

Component that can modify and filter data passing through it. Plug your own 'filter' into it.

Keyword arguments:

- filter -- an object implementing a filter(data) method (default=NullFilter instance)

Other

FilterComponent

NO DOCS

NullFilter

A filter class that filters nothing. This is the null default for the Filter.

Kamaelia.Util.FilterComponent

This is a deprecation stub, due for later removal.

Other

FilterComponent

NO DOCS

Kamaelia.Util.FirstOnly

Pass on the first item only

The first item sent to FirstOnly will be passed on. All other items are ignored.

Example Usage

Displaying the frame rate, just once, from video when it is decoded:

```
Pipeline( ...
          DiracDecoder(),
          FirstOnly(),
          SimpleDetupler("frame_rate"),
          ConsoleEchoer(),
        )
```

Behaviour

The first data item sent to FirstOnly's "inbox" inbox is immediately sent on out of its "outbox" outbox.

Any subsequent data sent to its "inbox" inbox is discarded.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

FirstOnly

FirstOnly() -> new FirstOnly component.

Passes on the first item sent to it, and discards everything else.

Kamaelia.Util.Graphline

This is a deprecation stub, due for later removal.

Other

Graphline

NO DOCS

Kamaelia.Util.Introspector

Detecting the topology of a running Kamaelia system

The Introspector component introspects the current local topology of a Kamaelia system - that is what components there are and how they are wired up.

It continually outputs any changes that occur to the topology.

Example Usage

Introspect and display whats going on inside the system:

```
MyComplexSystem().activate()

Pipeline( Introspector(),
          text_to_token_lists()
          AxonVisualiser(),
        )
```

How does it work?

Once activated, this component introspects the current local topology of a Kamaelia system.

Local? This component examines its scheduler to find components and postmen. It then examines them to determine their inboxes and outboxes and the linkages between them. In effect, it determines the current topology of the system.

If this component is not active, then it will see no scheduler and will report nothing.

What is output is how the topology changes. Immediately after activation, the topology is assumed to be empty, so the first set of changes describes adding nodes and linkages to the topology to build up the current state of it.

Subsequent output just describes the changes - adding or deleting linkages and nodes as appropriate.

Nodes in the topology represent components and postboxes. A linkage between a component node and a postbox node expresses the fact that that postbox belongs to that component. A linkage between two postboxes represents a linkage in the Axon system, from one component to another.

This topology change data is output as string containing one or more lines. It is output through the "outbox" outbox. Each line may be one of the following:

- "DEL ALL"
  - the first thing sent immediately after activation - to ensure that the receiver of this data understand that we are starting from nothing
- "ADD NODE <id> <name> randompos component"
- "ADD NODE <id> <name> randompos inbox"
- "ADD NODE <id> <name> randompos outbox"
  - an instruction to add a node to the topology, representing a component, inbox or outbox. <id> is a unique identifier. <name> is a 'friendly' textual label for the node.
- "DEL NODE <id>"
  - an instruction to delete a node, specified by its unique id
- "ADD LINK <id1> <id2>"
  - an instruction to add a link between the two identified nodes. The link is deemed to be directional, from <id1> to <id2>
- "DEL LINK <id1> <id2>"
  - an instruction to delete any link between the two identified nodes. Again, the directionality is from <id1> to <id2>.

the <id> and <name> fields may be encapsulated in double quote marks ("). This will definitely be so if they contain space characters.

If there are no topology changes then nothing is output.

This component ignores anything arriving at its "inbox" inbox.

If a shutdownMicroprocess message is received on the "control" inbox, it is sent on to the "signal" outbox and the component will terminate.

Components

Introspector

NO DOCS

Kamaelia.Util.LineSplit

Components

LineSplit

Split each message into its separate lines and send them on as separate messages

Kamaelia.Util.LossyConnector

Lossy connections between components

A component that passes on any data it receives, but will throw it away if the next component's inbox is unable to accept new items.

Example Usage

Using a lossy connector to drop excess data:: src = fastProducer().activate() lsy = LossyConnector().activate() dst = slowConsumer().activate()

src.link((src,"outbox"), (lsy,"inbox")) src.link((lsy,"outbox"), (dst,"inbox"), pipewidth=1)

The outbox of the lossy connector is joined to a linkage that can buffer a maximum of one item. Once full, the lossy connector causes items to be dropped.

How does it work?

This component receives data on its "inbox" inbox and immediately sends it on out of its "oubox" outbox.

If the act of sending the data causes a noSpaceInBox exception, then it is caught, and the data that it was trying to send is simply discarded.

I a producerFinished or shutdownMicroprocess message is received on the component's "control" inbox, then the message is forwarded on out of its "signal" outbox and the component then immediately terminates.

Components

LossyConnector

LossyConnector() -> new LossyConnector component

Component that forwards data from inbox to outbox, but discards data if destination is full.

Kamaelia.Util.MarshallComponent

Legacy stub for BasicMarshallComponent

The functionality of this component has been superceeded by the Marshaller and DeMarshaller components in Kamaelia.Util.Marshalling. Please use these in preference.

This component contains both marshalling and demarshalling facilities. It is a thin wrapper combining a Marshalling and DeMarshalling component.

Example Usage

None at present.

How does it work?

Behaviour is consistent with that of Kamaelia.Util.Marshalling, except that the "inbox" inbox and "outbox" outbox are not used.

Marshall data by sending it to the "marshall" inbox. The marshalled data is sent to the "marshalled" outbox.

Demarshall data by sending it to the "demarshall" inbox. The marshalled data is sent to the "demarshalled" outbox.

Other

BasicMarshallComponent

NO DOCS

Kamaelia.Util.Marshalling

Simple Marshalling/demarshalling framework

A pair of components for marshalling and demarshalling data respectively. You supply a class containing methods to marshall and demarshall the data in the way you want.

The idea is that you would place this between your logic and a network socket to transform the data to and from a form suitable for transport.

Example usage

Marshalling and demarshalling a stream of integers:

```
class SerialiseInt:

    def marshall(int):
        return str(int)
    marshall = staticmethod(marshall)

    def demarshall(string):
        return int(string)
    demarshall = staticmethod(demarshall)

Pipeline( producer(...),
          Marshaller(SerialiseInt),
          sender(...)
        ).activate()

Pipeline( receiver(...),
          DeMarshaller(SerialiseInt),
          consumer(...)
        ).activate()
```

How does it work?

When instantiating the Marshaller or DeMarshaller components, you provide an object (eg. class) containing these static methods:

- marshall(item) - returns the item serialised for transmission
- demarshall(item) - returns the original item, deserialised

Marshaller requires only the marshall(...) static method, and DeMarshaller requires only demarshall(...).

Why static methods? Because marshalling/demarshalling is a stateless activity. This distinguishes marshalling activity from other protocols and other processes that can be implemented with a similar style of framework.

For simplicity this component expects to be given an entire object to marshall or demarshall. This requires the user to deal with the framing and deframing of objects separately.

Any data sent to the Marshaller or DeMarshaller components' "inbox" inbox is passed to the marshall(...) or demarshall(...) method respectively of the class you supplied. The result is immediately sent on out of the components' "outbox" outbox.

If a producerFinished or shutdownMicroprocess message is received on the components' "control" inbox, it is sent on out of the "signal" outbox. The

component will then immediately terminate.

Post script

The initial data format this is designed to work with is the MimeDict object.

It is expected that there will be a more complex marshaller that supports receiving that is capable of receiving objects segmented over multiple messages.

Components

Marshaller

Marshaller(klass) -> new Marshaller component.

A component for marshalling data (serialising it to a string).

Keyword arguments:

- klass -- a class with static method: marshall(data) that returns the data, marshalled.

DeMarshaller

DeMarshaller(klass) -> new DeMarshaller component.

A component for demarshalling data (deserialising it from a string).

Keyword arguments: - klass -- a class with static method: demarshall(data) that returns the data, demarshalled.

Kamaelia.Util.Max

Find the maximum of a set of values

Send a list of values to Max and it will send out the maximum value in the list.

Example Usage

Supplying three lists of items, and the greatest is selected from each.

»> Pipeline(DataSource([(1,4,2,3), ('d','a','b','c'), ('xx','xxx')]), Max(), ConsoleEchoer(),).run()

When run this will output:

```
4dxxx
```

Behaviour

Send a list of comparable items to the "inbox" inbox and the greatest of those items (the maximum of the list) will be sent out of the "outbox" outbox.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

Max

Max() -> new Max component.

Send a list of values to its "inbox" inbox, and the maximum value from that list is sent out the "outbox" outbox.

Kamaelia.Util.NullSink

Null sink component. To ignore a component's outbox connect it to this component and the box will be emptied but not used in any way. This will be necessary with syncronized linkages.

Components

nullSinkComponent

NO DOCS

Kamaelia.Util.NullSinkComponent

Null sink component. To ignore a component's outbox connect it to this component and the box will be emptied but not used in any way. This will be necessary with syncronized linkages.

Components

nullSinkComponent

NO DOCS

Kamaelia.Util.OneShot

One-shot sending data

OneShot and TriggeredOneShot send a single specified item to their "outbox" outbox and immediately terminate.

TriggeredOneShot waits first for anything to arrive at its "inbox" inbox, whereas OneShot acts as soon as it is activated.

Example Usage

A way to create a component that writes data to a given filename, based on (filename,data) messages sent to its "next" inbox:

```
Carousel( lambda filename, data :
            Pipeline( OneShot(data),
                      SimpleFileWriter(filename),
                    ),
        )
```

A graphline that opens a TCP connection to myserver.com port 1500, and sends an a one off message:

```
Pipeline( OneShot("data to send to server"),
          TCPClient("myserver.com", 1500),
        ).run()
```

Shutting down a connection to myserver.com port 1500 as soon as a reply is received from the server:

```
Graphline( NET   = TCPClient("myserver.com", 1500),
           SPLIT = TwoWaySplitter(),
           STOP  = TriggeredOneShot(producerFinished()),
           linkages = {
               ("", "inbox" )      : ("NET", "inbox"),
               ("NET", "outbox")   : ("SPLIT", "inbox"),
               ("SPLIT", "outbox") : ("", "outbox"),

               ("SPLIT", "outbox2") : ("STOP", "inbox"),
               ("STOP", "outbox")   : ("NET", "control"),
               ("", "control")      : ("NET", "control"),
               ("NET", "signal")    : ("SPLIT", "control"),
               ("SPLIT", "signal")  : ("", "signal"),
               ("SPLIT", "signal2"),: ("STOP", "control"),
           },
         )
```

OneShot Behaviour

At initialisation, specify the message to be sent by OneShot.

As soon as OneShot is activated, the specified message is sent out of the "outbox" outbox. A producerFinished message is also sent out of the "signal" outbox. The component then immediately terminates.

TriggeredOneShot Behaviour

At initialisation, specify the message to be sent by TriggeredOneShot.

Send anything to the "inbox" inbox and TriggeredOneShot will immediately send the specified message out of the "outbox" outbox. A producerFinished message is also sent out of the "signal" outbox. The component then immediately terminates.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

OneShot

OneShot(msg) -> new OneShot component.

Immediately sends the specified message and terminates.

Keyword arguments:

```
- msg  -- the message to send out
```

TriggeredOneShot

OneShot(msg) -> new OneShot component.

Waits for anything to arrive at its "inbox" inbox, then immediately sends the specified message and terminates.

Keyword arguments:

```
- msg  -- the message to send out
```

Kamaelia.Util.PassThrough

Passthrough of data

The PassThrough component simply passes through data from its "inbox" inbox to its "outbox" outbox.

This can be used, for example, as a dummy 'protocol' component - slotting it into a system where ordinarily a component would go that somehow changes or processes the data passing through it.

Example Usage

Creating a simple tcp socket server on port 1850 that echoes back to clients whatever they send to it:

```
def echoProtocol:
    return PassThrough()

SimpleServer( protocol=echoProtocol, port=1850 ).run()
```

More Detail

Send any item to PassThrough component's "inbox" inbox and it will immediately be sent on out of the "outbox" outbox.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox then this component will immediately terminate. It will send the message on out of its "signal" outbox. Any pending data waiting in the "inbox" inbox may be lost.

Components

PassThrough

Other

passThrough

NO DOCS

Kamaelia.Util.Pipeline

This is a deprecation stub, due for later removal.

Other

Pipeline

NO DOCS

pipeline

NO DOCS

Kamaelia.Util.PipelineComponent

This is a deprecation stub, due for later removal.

Other

pipeline

NO DOCS

Kamaelia.Util.PromptedTurnstile

Buffering of data items until requested one at a time

PromptedTurnstile buffers items received, then sends them out one at a time in response to requests, first-in first-out style.

This is useful for controlling or limiting the rate of flow of data.

Example Usage

Displaying a script from a file, one line at a time, when a pygame button is clicked:

```
Graphline(
    SOURCE = RateControlledFileReader("script.txt",readmode="lines", ...),
    GATE   = PromptedTurnstile(),
    SINK   = ConsoleEchoer(),
    NEXT   = Button(label="Click for next line of script"),
    linkages = {
        ("SOURCE", "outbox") : ("GATE", "inbox"),
        ("GATE",   "outbox") : ("SINK", "inbox"),
        ("NEXT",   "outbox") : ("GATE", "next"),

        ("SOURCE", "signal") : ("GATE", "control"),
        ("GATE",   "signal") : ("SINK", "control"),
        ("SINK",   "signal") : ("NEXT", "control"),
        }
    )
```

Behaviour

Send items to the "inbox" inbox where they will queue up. Then each time you send anything to the "next" inbox; one item will be taken from the queue and forwarded out of the "outbox" outbox.

Think of it like a turnstile gate with people queuing up for it. Each message sent to the "next" inbox is a signal to let one person through the turnstile.

This component supports sending data out of its outbox to a size limited inbox. If the size limited inbox is full, this component will pause until it is able to send out the data. Data will not be consumed from the inbox if this component is waiting to send to the outbox.

If there is a backlog of "next" requests (because there is nothing left in the buffer) those items will be sent out as soon as they arrive. There is no need to send another "next" request.

Send a producerFinished message to the "control" inbox to tell PromptedTurnstile that there will be no more data. When prompted turnstile then receives a "next" request and has nothing left queued, it will send a producerFinised() message to its "signal" outbox and immediately terminate.

If a shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

PromptedTurnstile

> PromptedTurnstile() -> new PromptedTurnstile component.
>
> Buffers all items sent to its "inbox" inbox, and only sends them out, one at a time when requested.

Kamaelia.Util.PureTransformer

Pure Transformer component

This component applies a function specified at its creation to messages received (a filter). If the function returns None, no message is sent, otherwise the result of the function is sent to "outbox".

Example Usage

To read in lines of text, convert to upper case and then write to the console:

```
Pipeline(
    ConsoleReader(),
    PureTransformer(lambda x : x.upper()),
    ConsoleEchoer()
).run()
```

Components

PureTransformer

NO DOCS

Kamaelia.Util.RangeFilter

Filter items out that are not in range

RangeFilter passes through items received on its "inbox" inbox where item[0] lies within one or more of a specfied set of ranges of value. Items that don't match this are discarded.

Example Usage

Reading all video frames from a YUV4MPEG format video file, but only passing on video frames 25-49 and 100-199 inclusive further along the pipeline:

```
Pipeline( RateControlledFileReader("myvideo.yuv4mpeg",readmode="bytes"),
          YUV4MPEGToFrame(),
          TagWithSequenceNumber(),
          RangeFilter(ranges=[ (25,49), (100,199) ]),
          ...
        ).run()
```

Behaviour

At initialisation, specify a list of value ranges that RangeFilter should allow. The list should be of the form:

```
[ (low,high), (low,high), (low, high), ... ]
```

The ranges specified are inclusive.

Send an item to the "inbox" inbox of the form (value, ...). If the value matches one or more of the ranges specified, then the whole item (including the value) will immediately be sent on out of the "outbox" outbox.

RangeFilter can therefore be used to select slices through sequence numbered or timestamped data.

If the size limited inbox is full, this component will pause until it is able to send out the data,.

If a producerFinished message is received on the "control" inbox, this component will complete parsing any data pending in its inbox, and finish sending any resulting data to its outbox. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately

terminate. It will not complete processing, or sending on any pending data.

Components

RangeFilter

RangeFilter(ranges) -> new RangeFilter component.

Filters out items of the form (value, ...) not within at least one of a specified value set of range. Items within range are passed through.

Keyword arguments:

```
- ranges  -- list of (low,high) pairs representing ranges of value. Ranges are inclusiv
```

Kamaelia.Util.RateChunker

Breaks data into chunks matching a required chunk rate

Send data, such as binary strings to this component and it will break it down to roughly constant sized chunks, to match a required 'rate' of chunk emission.

This is not about 'real time' chunking of a live data source, but is instead about precisely chunking data that you know has been generated, or will be consumed, at a particular rate.

You specify the 'rate' of the incoming data and the rate you want chunks sent out at. RateChunker will determine what size the chunks need to be, applying dynamic rounding to precisely match the rate without drift over time.

Example Usage

Chunking a stream of 48KHz 16bit stereo audio into 25 chunks per second of audio data (one chunk for each frame of a corresponding piece of 25fps video):

```
bps = bytesPerSample = 2*2

Pipeline( AudioSource(),
          RateChunker(datarate=48000*bps, quantasize=bps, chunkrate=25),
          ...
        )
```

The quanta size ensures that the chunks RateChunker sends out always contain a whole number of samples (4 bytes per sample).

Behaviour

At initialisation, specify:

- the rate of the incoming data (eg. bytes/second)

- the required rate of outgoing chunks of data
- the minimum quanta size (see below)

Send slicable data items, such as strings containing binary data to the "inbox" inbox. The same data is sent out of the "outbox" outbox, rechunked to meet the required chunk rate.

The outgoing chunk sizes are dynamically varied to match the required chunk rate as accurately as possible. The quantasize parameter dictates the minimum unit by which the chunksize will be varied.

For example, for 16bit stereo audio data, there are 4 bytes per sample, so a quantasize of 4 should be specified, to make sure samples remain whole.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

RateChunker

RateChunker(datarate,quantasize,chunkrate) -> new Chunk component.

Alters the chunksize of incoming data to match a desired chunkrate.

Keyword arguments:

- datarate -- rate of the incoming data
- quantasize -- minimum granularity with which the data can be split
- chunkrate -- desired chunk rate

Kamaelia.Util.RateFilter

These components limit the rate of data flow, either by buffering or by taking charge and requesting data at a given rate.

Message Rate limiting

This component buffers incoming messages and limits the rate at which they are sent on.

Example Usage

Regulating video to a constant framerate, buffering 2 seconds of data before starting to emit frames:

```
Pipeline( RateControlledFileReader(...),
          DiracDecoder(),
          MessageRateLimit(messages_per_second=framerate, buffer=2*framerate),
          VideoOverlay(),
        ).activate()
```

How does it work?

Data items sent to this component's "inbox" inbox are buffered. Once the buffer is full, the component starts to emit items at the specified rate to its "outbox" outbox.

If there is a shortage of data in the buffer, then the specified rate of output will, obviously, not be sustained. Items will be output when they are available.

The specified rate serves as a ceiling limit - items will never be emitted faster than that rate, though they may be emitted slower.

Make sure you choose a sufficient buffer size to handle any expected jitter/temporary shortages of data.

If a producerFinished or shutdownMicroprocess message is received on the components' "control" inbox, it is sent on out of the "signal" outbox. The component will then immediately terminate.

Rate Control

These components control the rate of a system by requesting data at a given rate. The 'variable' version allows this rate to the changed whilst running.

Example Usage

Reading from a file at a fixed rate:

```
Graphline( ctrl   = ByteRate_RequestControl(rate=1000, chunksize=32),
           reader = PromptedFileReader(filename="myfile", readmode="bytes"),
           linkages = {
                ("ctrl", "outbox") : ("reader","inbox"),
                ("reader", "outbox") : ("self", "outbox"),

                ("self", "control") : ("reader", "control"),
                ("reader", "signal") : ("ctrl", "control"),
                ("ctrl, "signal") : ("self", "signal"),
              }
```

Note that the "signal"-"control" path goes in the opposite direction so that when the file is finished reading, the ByteRate_RequestControl component receives a shutdown message.

Reading from a file at a varying rate (send new rates to the "inbox" inbox):

```
Graphline( ctrl   = VariableByteRate_RequestControl(rate=1000, chunksize=32),
           reader = PromptedFileReader(filename="myfile", readmode="bytes"),
           linkages = {
                  ("self", "inbox") : ("ctrl", "inbox"),
                  ("ctrl", "outbox") : ("reader","inbox"),
                  ("reader", "outbox") : ("self", "outbox"),

                  ("self", "control") : ("reader", "control"),
                  ("reader", "signal") : ("ctrl", "control"),
                  ("ctrl, "signal") : ("self", "signal"),
              }
         ).activate()
```

Note that the "signal"-"control" path goes in the opposite direction so that when the file is finished reading, the VariableByteRate_RequestControl component receives a shutdown message.

How does it work?

These components emit from their "outbox" outboxes, requests for data at the specified rate. Each request is an integer specifying the number of items.

Rates are in no particular units (eg. bitrate, framerate) - you can use it for whatever purpose you wish. Just ensure your values fit the units you are working in.

At initialisation, you specify not only the rate, but also the chunk size or chunk rate. For example, a rate of 12 and chunksize of 4 will result in 3 requests per second, each for 4 items. Conversely, specifying a rate of 12 and a chunkrate of 2 will result in 2 requests per second, each for 6 items.

The rate and chunk size or chunk rate you specify does not have to be integer or divide into integers. For example, you can specify a rate of 10 and a chunksize of 3. Requests will then be emitted every 0.3 seconds, each for 3 items.

When requests are emitted, they will always be for an integer number of items. Rounding errors are averaged out over time, and should not accumulate. Rounding will occur if chunksize, either specified, or calculated from chunkrate, is non-integer.

At initialisation, you can also specify that chunk 'aggregation' is permitted. If permitted, then the component can choose to exceed the specified chunksize. For example if, for some reason, the component gets behind, it might aggregate two requests together - the next request will be for twice as many items.

Another example would be if you, for example, specify a rate of 100 and chunkrate of 3. The 3 requests emitted every second will then be for 33, 33 and 34 items.

The VariableByteRate_RequestControl component allows the rate to be changed on-the-fly. Send a new rate to the component's "inbox" inbox and it will be

adopted immediately. You cannot change the chunkrate or chunksize.

The new rate is adopted at the instant it is received. There will be no glitches in the apparent rate of requests due to your changing the rate.

If a producerFinished or shutdownMicroprocess message is received on the components' "control" inbox, it is sent on out of the "signal" outbox. The component will then immediately terminate.

Flow limiting by request

This component buffers incoming data and emits it one item at a time, whenever a "NEXT" request is received.

Example Usage

An app that reads data items from a file, then does something with then one at a time when the user clicks a visual button in pygame:

```
Graphline( source   = RateControlledFileReader(..., readmode="lines"),
           limiter  = OnDemandLimit(),
           trigger  = Button(caption="Click for next",msg="NEXT"),
           dest     = consumer(...),
           linkages = {
                   ("source", "outbox") : ("limiter", "inbox"),
                   ("limiter", "outbox") : ("dest", "inbox"),
                   ("trigger", "outbox") : ("limiter", "slidecontrol")
               }
         ).activate()
```

How does it work?

Data items sent to the component's "inbox" inbox are buffered in a queue. Whenever a "NEXT" message is received on the component's "slidecontrol" inbox, an item is taken out of the queue and sent out of the "outbox" outbox.

Items come out in the same order they go in.

If a "NEXT" message is received but there are no items waiting in the queue, the "NEXT" message is discarded and nothing is emitted.

If a producerFinished message is received on the components' "control" inbox, it is sent on out of the "signal" outbox. The component will then immediately terminate.

Components

MessageRateLimit

MessageRateLimit(messages_per_second[, buffer]) -> new MessageRateLimit component.

Buffers messages and outputs them at a rate limited by the specified rate once the buffer is full.

Keyword arguments:

- messages_per_second -- maximum output rate
- buffer -- size of buffer (0 or greater) (default=60)

ByteRate_RequestControl

ByteRate_RequestControl([rate][,chunksize][,chunkrate][,allowchunkaggregation]) -> new ByteRate_RequestControl component.

Controls rate of a data source by, at a controlled rate, emitting integers saying how much data to emit.

Keyword arguments:

- rate -- qty of data items per second (default=100000)
- chunksize -- None or qty of items per 'chunk' (default=None)
- chunkrate -- None or number of chunks per second (default=10)
- allowchunkaggregation -- if True, chunksize will be enlarged if 'catching up' is necessary (default=False)

Specify either chunksize or chunkrate, but not both.

VariableByteRate_RequestControl

ByteRate_RequestControl([rate][,chunksize][,chunkrate][,allowchunkaggregation]) -> new ByteRate_RequestControl component.

Controls rate of a data source by, at a controlled rate, emitting integers saying how much data to emit. Rate can be changed at runtime.

Keyword arguments: - rate -- qty of data items per second (default=100000) - chunksize -- None or qty of items per 'chunk' (default=None) - chunkrate -- None or number of chunks per second (default=10) - allowchunkaggregation -- if True, chunksize will be enlarged if 'catching up' is necessary (default=False)

Specify either chunksize or chunkrate, but not both.

OnDemandLimit

OnDemandLimit() -> new OnDemandLimit component.

A component that receives data items, but only emits them on demand, one at a time, when "NEXT" messages are received on the "slidecontrol" inbox.

Kamaelia.Util.SequentialTransformer

Sequential Transformer component

This component applies all the functions supplied to incoming messages. If the output from the final function is None, no message is sent.

Example Usage

To read in lines of text, convert to upper case, prepend "foo", and append "bar!" and then write to the console:

```
Pipeline(
    ConsoleReader(eol=""),
    SequentialTransformer( str,
                           str.upper,
                           lambda x : "foo" + x,
                           lambda x : x + "bar!",
                         ),
    ConsoleEchoer(),
).run()
```

Components

SequentialTransformer

NO DOCS

Kamaelia.Util.Splitter

Simple Fanout of messages

A component that splits a data source, fanning it out to multiple destinations.

Example Usage

Component connecting to a Splitter:

```
class Consumer(Axon.Component.component):
    Outboxes = [ "outbox", "signal", "splitter_config" ]

    def main(self):
         self.send( addsink( self, "inbox", "control" ), "splitter_config")
```

```
        yield 1
        ... do stuff when data is received on "inbox" inbox

mysplitter = Splitter()
Pipeline( producer(), mysplitter ).activate()

myconsumer = Consumer().activate()
myconsumer.link( (myconsumer, "splitter_config"), ("mysplitter", "configuration") )
```

How does it work?

Any data sent to the component's "inbox" inbox is sent out to multiple destinations (but not to the "outbox" outbox).

Add a destination by sending an addsink(...) message to the "configuration" inbox of the component. Splitter will then wire up to the 'sinkbox' inbox specified in the message, and send it any data sent to its "inbox" inbox.

NOTE: Splitter only does this for the 'sinkbox' inbox, not for the 'sinkcontrol' inbox. If one is specified, it is ignored.

There is no limit on the number of 'sinks' that can be connected to the splitter. The same component can add itself as a sink multiple times, provided different named inboxes are used each time.

NOTE: The data is not duplicated - the same item is sent to all destinations. Care must therefore be taken if the data item is mutable.

If one or more destinations cause a noSpaceInBox exception, the data item will be queued, and Splitter will attempt to resend it to the destinations in question until successful. It will stop forwarding any new incoming data until it has succeeded, thereby ensuring the order of data is not altered.

Stop data being sent to a destination by sending a removesink(...) message to the "configuration" inbox of the Splitter component. Splitter will then cease sending messages to the 'sinkbox' inbox specified in the message and will unwire from it.

Any messages sent to the "control" inbox are ignored. The "outbox" and "signal" outboxes are not used.

This component does not terminate.

Pluggable Fanout of messages

The PlugSplitter component splits a data source, fanning it out to multiple destinations. The Plug component allows you to easily 'plug' a destination into the splitter.

Example Usage

Two consumers receiving the same data from a single consumer. Producer and consumers are encapsulated by PlugSplitter and Plug components respectively:

```
mysplitter = PlugSplitter( producer() ).activate()

Plug(mysplitter, consumer() ).activate()
Plug(mysplitter, consumer() ).activate()
```

The same, but the producer and consumers are not encapsulated:

```
mysplitter = PlugSplitter()
Pipeline( producer, mysplitter ).activate()

Pipeline( Plug(mysplitter), consumer() ).activate()
Pipeline( Plug(mysplitter), consumer() ).activate()
```

How does it work?

Any data sent to the component's "inbox" and "control" inboxes is sent out to multiple destinations. It is also sent onto the components "outbox" and "signal" outboxes, respectively.

Alternatively, initialisation you can specify a 'source' component. If you do, then data to be sent out to multiple destinations is instead received from that component's "outbox" and "signal" outboxes, respectively. Any data sent to the "inbox" and "control" inboxes of the PlugSplitter component will be forwarded to the "inbox" and "control" inboxes of the 'source' component, respectively.

This source component is encapsulated as a child within the PlugSplitter component, and so must not be separately activated. Activating PlugSplitter will also activate this child component.

Add a destination by making a Plug component, specifying the PlugSplitter component to 'plug into'. See documentation for the Plug component for more information.

Alternatively, you can add and remove destinations manually:

- Add a destination by sending an addsink(...) message to the "configuration" inbox of the component.

  If a 'sinkbox' inbox is specified in the message, then PlugSplitter will wire up to it and forward to it any 'inbox'/'outbox' data. If a 'sinkcontrol' inbox is specified, then Plugsplitter will wire up to it and forward to it any 'control'/'signal' data.

- Stop data being sent to a destination by sending a removesink(...) message to the "configuration" inbox of the Splitter component.

Splitter will then cease sending messages to the 'sinkbox' inbox specified in the message and will unwire from it.

There is no limit on the number of 'sinks' that can be connected to the splitter. The same component can add itself as a sink multiple times, provided different named inboxes are used each time.

NOTE: The data is not duplicated - the same item is sent to all destinations. Care must therefore be taken if the data item is mutable.

If a shutdownMicroprocess or producerFinished message is received on the "control" inbox and there is NO 'source' child component, then the message is forwarded onto all 'control' destinations and the 'signal' outbox. The component then immediately terminates, unwiring from all destinations.

If there is a child component then PlugSplitter will terminate when the child component terminates, unwiring from all destinations.

Plug for PlugSplitter

The Plug component 'plugs into' a PlugSplitter as a destination to which the source data is split.

Example Usage

See PlugSplitter documentation.

How does it work?

Initialise the Plug component by specifying a PlugSplitter component to connect to and the component that wants to receive the data from the Plugsplitter.

The destination/sink component is encapsulated as a child component, and is therefore activated by the Plug component when it is activated. Do not activate it yourself.

The Plug component connects to the PlugSplitter component by wiring its "splitter_config" outbox to the "configuration" inbox of the PlugSplitter component and sending it an addsink(...) message. This causes PlugSplitter to wire up to the Plug's "inbox" and "control" inboxes.

The Plug's "inbox" and "control" inboxes are forwarded to the "inbox" and "control" inboxes of the child component respectively. The "outbox" and "signal" outboxes of the child component are forwarded to the "outbox" and "signal" outboxes of the Plug component respectively.

When the child component terminates, the Plug component sends a removesink(...) message to the PlugSplitter, causing PlugSplitter to unwire from it. It then terminates.

Thoughts

PlugSplitter is probably more reliable than Splitter however it *feels* too complex. However the actual "Splitter" class in this file is not the preferable option.

Components

Splitter

Splitter() -> new Splitter component.

Splits incoming data out to multiple destinations. Send addsink(...) and removesink(...) messages to the 'configuration' inbox to add and remove destinations.

PlugSplitter

PlugSplitter([sourceComponent]) -> new PlugSplitter component.

Splits incoming data out to multiple destinations. Send addsink(...) and removesink(...) messages to the 'configuration' inbox to add and remove destinations.

Keyword arguments:

- sourceComponent -- None, or component to act as data source

Plug

Plug(splitter,component) -> new Plug component.

A component that 'plugs' the specified component into the specified splitter as a destination for data.

Keyword arguments:

- splitter -- splitter component to plug into (any component that accepts addsink(...) and removesink(...) messages on a 'configuration' inbox
- component -- component to receive data from the splitter

Other

Axon

Axon - the core concurrency system for Kamaelia

Axon is a component concurrency framework. With it you can create software "components" that can run concurrently with each other. Components have "inboxes" and "outboxes" through with they communicate with other components.

A component may send a message to one of its outboxes. If a linkage has been created from that outbox to another component's inbox; then that message will arrive in the inbox of the other component. In this way, components can send and receive data - allowing you to create systems by linking many components together.

Each component is a microprocess - rather like a thread of execution. A scheduler takes care of making sure all microprocesses (and therefore all components) get regularly executed. It also looks after putting microprocesses to sleep (when they ask to be) and waking them up (for example, when something arrives in one of their inboxes).

Base classes for building your own components

- **Axon.Component**
  - defines the basic component. Subclass it to write your own components.
- **Axon.AdaptiveCommsComponent**
  - like a basic component but with facilties to let you add and remove inboxes and outboxes during runtime.
- **Axon.ThreadedComponent**
  - like ordinary components, but which truly run in a separate thread - meaning they can perform blocking tasks (since they don't have to yield control to the scheduler for other components to continue executing)

Underlying concurrency system

- **Axon.Microprocess**
  - Turns a python generator into a schedulable microprocess - something that can be started, paused, reawoken and stopped. Subclass it to make your own.
- **Axon.Scheduler**
  - Runs the microprocesses. Manages the starting, stopping, pausing and waking of them. Is also a microprocess itself!

Services, statistics, Instrospection

- **Axon.CoordinatingAssistantTracker**
  - provides mechanisms for components to advertising and discover services they can provide for each other.
  - acts as a repository for collecting statistics from components in the system
- **Axon.Introspector**
  - outputs live topology data describing what components there are in a running axon system and how they are linked together.

Exceptions, Messages and Misc

- **Axon.Base**
  - base metaclass for key Axon classes
- **Axon.AxonExceptions**
  - classes defining various exceptions in Axon.
- **Axon.Ipc**
  - classes defining various IPC messages in Axon used for signalling shutdown, errors, notifications, etc...
- **Axon.idGen**
  - unique id value generation
- **Axon.util**
  - various miscellaneous support utility methods

Integration with other systems

- **Axon.background**
  - use Axon components within other python programs by wrapping them in a scheduler running in a separate thread
- **Axon.Handle**
  - a Handle for getting data into and out of the standard inboxes and outboxes of a component from a non Axon based piece of code. Useful in combination with Axon.background

Internals for implementing inboxes, outboxes and linkages

- **Axon.Box**
  - The base implementation of inboxes and outboxes.
- **Axon.Postoffice**
  - All components have one of these for creating, destroying and tracking linkages.
- **Axon.Linkage**
  - handles used to describe linkages from one postbox to another

What, no Postman? Optimisations made to Axon have dropped the Postman. Inboxes and outboxes handle the delivery of messages themselves now.

Debugging support

- **Axon.debug**
  - defines a debugging output object.
- **Axon.debugConfigFile**
  - defines a method for loading a debugging configuration file that determines what debugging output gets displayed and what gets filtered out.
- **Axon.debugConfigDefaults**
  - defines a method that supplies a default debugging configuration.

addsink

addsink(sink[,sinkbox][,sinkcontrol][passthrough]) -> new addsink message.

Message specifying a target component and inbox(es), requesting they be wired to receive a data source (to be the data sink).

Keywork arguments:

- sink -- target component
- sinkbox -- target component's inbox name (default="inbox")
- sinkcontrol -- None, or target component's 'control' inbox name (default=None)
- passthrough -- 0, or 1 if linkage is inbox-inbox, or 2 if outbox-outbox (default=0)

removesink

removesink(sink[,sinkbox][,sinkcontrol]) -> new removesink message.

Message specifying a target component and inbox(es), requesting they be unwired from a data source (no longer act as a data sink)

Kamaelia.Util.Stringify

Convert Data to Strings

A simple component that takes data items and converts them to strings.

Example Usage

A simple pipeline:

```
Pipeline( sourceOfNonStrings(),
          Stringify(),
          consumerThatWantsStrings(),
        ).activate()
```

How does it work?

Send data items to this component's "inbox" inbox. They are converted to strings using the str(...) function, and sent on out of the "outbox" outbox.

Anything sent to this component's "control" inbox is ignored.

This component does not terminate.

Components

Stringify

Stringify() -> new Stringify.

A component that converts data items received on its "inbox" inbox to strings and sends them on out of its "outbox" outbox.

Kamaelia.Util.Sync

Wait for 'n' items before sending one of them on

For every 'n' items received, one is sent out (the first one received in the latest batch).

Example Usage

Wait for two tasks to finish, before propagating the shutdown message:

```
Graphline( A    = TaskA(),
           B    = TaskB(),
           SYNC = Sync(2),
           linkages = {
               ("A", "signal") : ("SYNC", "inbox"),
               ("B", "signal") : ("SYNC", "inbox"),

               ("SYNC", "outbox") : ("SYNC", "control"),
               ("SYNC", "signal") : ("", "signal"),
           }
```

The slightly strange wiring is to make sure the Sync component is also shut down. The shutdown message is used to shutdown Sync itself. The shutdown message it emits is then the one that propogates out of the graphline.

Behaviour

At initialisation, specify the number of items Sync should wait for.

Once that number of items have arrived at Sync's "inbox" inbox; the first that arrived is sent on out of its "outbox" outbox. This process is repeated until Sync is shut down.

If more han the specified number of items arrive in one go; the excess items roll over to the next cycle. They are not ignored or lost.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

Sync

Sync([n]) -> new Sync component.

After ever 'n' items received, the first in each batch received is sent on.

Keyword arguments:

```
- n  -- The number of items to expect (default=2)
```

Kamaelia.Util.TagWithSequenceNumber

Tags items with an incrementing sequence number

TagWithSequenceNumber tags items with a sequence numbers, for example: 0, 1, 2, 3, ... etc. The default initial value of the sequence is a 0.

It takes in items on its "inbox" inbox and outputs (seqnum, item) tuples on its "outbox" outbox.

Example Usage

Tagging frames from a Dirac video file with a frame number, starting with 1:

```
Pipeline( RateControlledFileReader("videofile.dirac", readmode="bytes", rate=... ),
          DiracDecoder(),
          TagWithSequenceNumber(initial=1),
          ...
        )
```

Behaviour

At initialisation, specify the initial sequence number to use.

Send an item to TagWithSequenceNumber's "inbox" inbox, and it will send (seqnum, item) to its "outbox" outbox.

The sequence numbers begin 0, 1, 2, 3, ... etc ad infinitum.

If a producerFinished or shutdownMicroprocess message is received on the "control" inbox. It is immediately sent on out of the "signal" outbox and the component then immediately terminates.

Components

TagWithSequenceNumber

TagWithSequenceNumber() -> new TagWithSequenceNumber component.

Send 'item' to the "inbox" inbox and it will be tagged with a sequence number, and sent out as (seqnum, 'item') to the "outbox" outbox.

Sequence numbering goes 0, 1, 2, 3, ... etc.

Kamaelia.Util.TestResult

Basic Result Tester

A simple component for testing that a stream of data tests true. This is NOT intended for live systems, but for testing and development purposes only.

Example Usage

:: Pipeline(source(), TestResult()).activate()

Raises an assertion error if source() generates a value that doesn't test true.

How does it work?

If the component receives a value on its "inbox" inbox that does not test true, then an AssertionError is raised.

If the component receives a StopSystem message on its "control" inbox then a StopSystemException message is raised as an exception.

This component does not terminate (unless it throws an exception).

It does not pass on the data it receives.

Components

TestResult

TestResult() -> new TestResult.

Component that raises an AssertionError if it receives data on its "inbox" inbox that does not test true. Or raises a StopSystemException if a StopSystem message is received on its "control" inbox.

Other

StopSystem

This IPC message is the command to the component to throw a StopSystemException and bring the Axon system to a halt.

StopSystemException

This exception is used to stop the whole Axon system.

Kamaelia.Util.TestResultComponent

This is a deprecation stub, due for later removal.

Other

testResultComponent

NO DOCS

Kamaelia.Util.ToStringComponent

This is a deprecation stub, due for later removal.

Other

ToStringComponent

NO DOCS

Kamaelia.Util.Tokenisation.Simple

Other

EscapedListMarshalling

NO DOCS

lines_to_tokenlists

NO DOCS

substitutions

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

tokenlists_to_lines

NO DOCS

Kamaelia.Util.TwoWaySplitter

Send stuff to two places

Splits a data source sending it to two destinations. Forwards both things sent to its "inbox" inbox and "control" inboxes, so shutdown messages propogate through this splitter. Fully supports delivery to size limited inboxes.

Example Usage

Send from a data source to two destinations. Do this for both the inbox->outbox path and the signal->control paths, so both destinations receive shutdown messages when the data source finishes:

```
Graphline( SOURCE = MyDataSource(),
           SPLIT  = TwoWaySplitter(),
           DEST1  = MyDataSink1(),
           DEST2  = MyDataSink2(),
           linkages = {
               ("SOURCE", "outbox") : ("SPLIT", "inbox"),
               ("SOURCE", "signal") : ("SPLIT", "control")

               ("SPLIT", "outbox") : ("DEST1", "inbox"),
               ("SPLIT", "signal") : ("DEST1", "control"),

               ("SPLIT", "outbox2") : ("DEST1", "inbox"),
               ("SPLIT", "signal2") : ("DEST1", "control"),
           }
         ).run()
```

Behaviour

Send a message to the "inbox" inbox of this component and it will be sent on out of the "outbox" and "outbox2" outboxes.

This component supports sending to a size limited inbox. If the size limited inbox is full, this component will pause until it is able to send out the data.

Send a message to the "control" inbox of this component and it will be sent on out of the "signal" and "signal2" outboxes. If the message is a shutdownMicroprocess message then this component will also terminate immediately. If it is a producerFinished message then this component will finish sending any messages still waiting at its "inbox" inbox, then immediately terminate.

Components

TwoWaySplitter

TwoWaySplitter() -> new TwoWaySplitter component

Anything sent to the "inbox" or "control" inboxes is sent on out of the "outbox" and "outbox2" or "signal" and "signal2" outboxes respectively.

Kamaelia.Util.UnseenOnly

UnseenOnly component

This component forwards on any messages it receives that it has not seen before.

Example Usage

Lines entered into this setup will only be duplicated on screen the first time they are entered:

```
pipeline(
    ConsoleReader(),
    UnseenOnly(),
    ConsoleEchoer()
).run()
```

Components

UnseenOnly

UnseenOnly() -> new UnseenOnly component.

Send items to the "inbox" inbox. Any items not "seen" already will be forwarded out of the "outbox" outbox. Send the same thing two or more times and it will only be sent on the first time.

Kamaelia.Video.CropAndScale

Video frame cropping and scaling

This component applies a crop and/or scaling operation to frames of RGB video.

Requires PIL - the python imaging library.

Example Usage

Crop and scale a YUV4MPEG format uncompressed video file so that the output is the region (100,100) ->(400,300), scaled up to 720x576:

```
from Kamaelia.File.Reading import RateControlledFileReader

Pipeline( RateControlledFileReader("input.yuv4mpeg", ... ),
          YUV4MPEGToFrame(),
          ToRGB_interleaved(),
          CropAndScale(newsize=(720,576), cropbounds=(100,100,400,300)),
          ToYUV420_planar(),
          FrameToYUV4MPEG(),
          SimpleFileWriter("cropped_and_scaled.yuv4mpeg"),
        ).run()
```

More details

Initialise this component specifying the region of the incoming video frames to crop to, and the size of the desired output (the cropped region will be scaled up/down to match this).

Send frame data structures to the "inbox" inbox of this component. The frames will be cropped and scaled and output from the "outbox" outbox. Only frames with one of the following pixel formats are currently supported:

"RGB_interleaved" "RGBA_interleaved" "Y_planar"

See below for a description of the uncompressed frame data structure format. Send uncompressed video frames, in the format described below,

This component supports sending data out of its outbox to a size limited inbox. If the size limited inbox is full, this component will pause until it is able to send out the data. Data will not be consumed from the inbox if this component is waiting to send to the outbox.

If a producerFinished message is received on the "control" inbox, this component will complete parsing any data pending in its inbox, and finish sending any resulting data to its outbox. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete processing, or sending on any pending data.

UNCOMPRESSED FRAME FORMAT

A frame is a dictionary data structure, containing at minimum one of these combinations:

```
{
  "yuv" : luminance_data
  "pixformat" :  pixelformat        # format of raw video data
  "size" : (width, height)          # in pixels
```

```
}

{
  "rgb" : rgb_interleaved_data
  "pixformat" :  pixelformat        # format of raw video data
  "size" : (width, height)          # in pixels
}
```

CropAndScale only guarantees to fill in the fields above. Any other fields will be transparently passed through, unmodified.

Components

CropAndScale

CropAndScale(newsize, cropbounds) -> new CropAndScale component.

Crops and scales frames of video in RGB format.

Keyword arguments:

- newsize -- (width, height) of the resulting output video frames (in pixels)
- cropbounds -- (x0,y0,x1,y1) region to crop out from the incoming video frames (in pixels)

Kamaelia.Video.DetectShotChanges

Detecting cuts/shot changes in video

DetectShotChanges takes in (framenumber, videoframe) tuples on its "inbox" inbox and attempts to detect where shot changes have probably occurred in the sequence. When it thinks one has ocurred, a (framenumber, confidencevalue) tuple is sent out of the "outbox" outbox.

Example Usage

Reading in a video in uncompressed YUV4MPEG file format, and outputting the frame numbers (and confidence values) where cuts probably occur:

```
Pipeline( RateControlledFileReader(..)
          YUV4MPEGToFrame(),
          TagWithSequenceNumber(),      # pair up frames with a frame number
          DetectShotChanges(threshold=0.85),
          ConsoleEchoer(),
        ).run()
```

Expect output like this:

```
(17, 0.885)(18, 0.912)(56, 0.91922)(212, 0.818)(213, 0.825)(214, 0.904) ...
```

More detail

Send (frame-number, video-frame) tuples to this component's "inbox" inbox and (frame-number, confidence-value) tuples will be sent out of the "outbox" outbox whenever it thinks a cut has occurred.

Frames must be in a YUV format. See below for details. Frame numbers need not necessarily be sequential; but they must be unique! If they are not, then it is your own fault when you can't match up detected shot changes to actual video frames!

Internally, the cut detector calculates a 'confidence' value representing how likely that a shot change has occurred. At initialisation you set a threshold value - if the confidence value reaches or exceeds this threshold, then a cut is deemed to have taken place, and output will be generated.

How do you choose a threshold? It is a rather inexact science (as is the subjective decision of whether something consitutes a shot change!) - you really need to get a feel for it experimentally. As a rough guide, values between 0.8 and 0.9 are usually reasonable, depending on the type of video material.

Because of the necessary signal processing, this component has a delay of several frames of data through it before you will get output. It therefore will not necessarily detect cuts in the first 15 frames or so of a sequence sent to it. Neither will it generate any output for the last 15 frames or so - they will never make it through the internal signal processing.

Send a producerFinished() or shutdownMicroprocess() message to this component's "control" inbox and it will immediately terminate. It will also forward on the message out of its "signal" outbox.

Implementation details

The algorithm used is based on a simple "mean absolute difference" between pixels of one frame and the next; with some signal processing on the resulting stream of frame-to-frame difference values, to detect a spike possibly indicating a shot change.

The algorithm is courtesy of Jim Easterbrook of BBC Research. It is also available in its own right as an independent open source library here.

As signal processing is done on the confidence values internally to emphasise spikes - which are likely to indicate a sudden increase in the level of change from one frame to the next - a conseuqence is that this component internally buffers inter-frame difference values for several frames, resulting in a delay of about 15 frames through this component. This is the reason why it is necessary to pair up video frames with a frame number, otherwise you cannot guarantee being

able to match up the resulting detected cuts with the actual frame where they took place!

The change detection algorithm only looks at the Y (luminance) data in the video frame.

UNCOMPRESSED FRAME FORMAT

A frame is a dictionary data structure. It must, for this component, at minimum contain a key "yuv" that returns a tuple containing (y_data, u_data, v_data).

Any other entries are ignored.

Components

DetectShotChanges

DetectShotChanges([threshold]) -> new DetectShotChanges component.

Send (framenumber, videoframe) tuples to the "inbox" inbox. Sends out (framenumber, confidence) to its "outbox" outbox when a cut has probably occurred in the video sequence.

Keyword arguments:

- threshold -- threshold for the confidence value, above which a cut is detected (default=0.9)

Kamaelia.Video.PixFormatConversion

Converting the pixel format of video frames

These components convert the pixel format of video frames, for example, from interleaved RGB to planar YUV 420.

Example Usage

Decoding a Dirac encoded video file, then converting it to RGB for display on a pygame display surface:

```
Pipeline( RateControlledFileReader("video.drc",readmode="bytes", rate=100000),
          DiracDecoder(),
          ToRGB_interleaved(),
          VideoSurface(),
        ).run()
```

Which component for which conversion?

The components here are currently capable of the following pixel format conversions:

From	To	Which component?
"RGB_interleaved"	"RGB_interleaved"	ToRGB_interleaved
"YUV420_planar"	"RGB_interleaved"	ToRGB_interleaved
"YUV422_planar"	"RGB_interleaved"	ToRGB_interleaved
"RGB_interleaved"	"YUV420_planar"	ToYUV420_planar
"YUV420_planar"	"YUV420_planar"	ToYUV420_planar

More details

Send video frames to the "inbox" inbox of these components. They will be converted to the destination pixel format and sent out of the "outbox" outbox. Video frames are dictionaries, they must have the following keys:

- "rgb" or "yuv" -- containing the pixel data
- "pixformat" -- the pixel format
- "size" -- (width,height) in pixels

Any other fields will be transparently passed through, unmodified.

These components support sending data out of its outbox to a size limited inbox. If the size limited inbox is full, these components will pause until able to send out the data. Data will not be consumed from the inbox if these components are waiting to send to the outbox.

If a producerFinished message is received on the "control" inbox, these components will complete converting and frames pending in its inbox, and finish sending any resulting data to its outbox. They will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, these components will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete processing, or sending on any pending data.

Components

ToRGB_interleaved

" ToRGB_interleaved() -> new ToRGB_interleaved component.

Converts video frames sent to its "inbox" inbox, to "RGB_interleaved" pixel format and sends them out of its "outbox"

Supports conversion from:

- YUV420_planar
- YUV422_planar
- RGB_interleaved (passthrough)

ToYUV420_planar

" ToYUV420_planar() -> new ToYUV420_planar component.

Converts video frames sent to its "inbox" inbox, to "ToYUV420_planar" pixel format and sends them out of its "outbox"

Supports conversion from:

- RGB_interleaved
- YUV420_planar (passthrough)

Kamaelia.Visualisation.Axon.AxonLaws

Particle Laws for Axon Visualisation

This class is a specialisation of Kamaelia.Physics.Simple.MultipleLaws that specifies the laws for particle interactions when visualising Axon/Kamaelia systems.

Example Usage

Instantiate a topology viewer using rendering code for each particle type, and the appropriate laws to govern their interactions:: visualiser = TopologyViewer(particleTypes={ "component" : PComponent, "inbox" : PPostbox.Inbox, "outbox" : PPostbox.Outbox }, laws = AxonLaws(),).activate()

How does it work?

AxonLaws is a subclass of MultipleLaws. It sets the bond lengths and strengths of forces for two types of particles that represent components and inboxes/outboxes respectively.

These laws are mapped to work for particles identified as being of type "component" and "postbox".

At initialisation you can specify the length of the postbox to postbox bond (which represents Axon linkages). The ranges over which forces act (but not their magnitude) are scaled appropriately.

See MultipleLaws for information on the role of this class for physics simulation and topology visualisation.

Other

AxonLaws

AxonLaws([postboxBondLength]) -> new AxonLaws object.

Encapsulates laws for interactions between particles of types "Component" and "Postbox" in a physics simulation. Subclass of MultipleLaws.

Keyword arguments:

- postboxBondLength -- length of bond that represents Axon linkages (default=100)

Kamaelia

This is a doc string, will it be of use?

Kamaelia.Visualisation.Axon.AxonVisualiserServer

Axon/Kamaelia Visualisation Server

A specialisation of TopologyViewerServer for visualising Axon/Kamaelia systems.

Example Usage

Visualiser that listens on its default port for a TCP connection through which it receives Introspection topology data to render:: AxonVisualiserServer().run()

How does it work?

AxonVisualiserServer is a subclass of TopologyViewerServer, where the following are already specified: - types of particles - their laws of interaction - the number of simulation cycles per redraw - extra window furniture

The remaining keyword arguments of the TopologyviewerServer component can all be specified when initialising AxonVisualiserServer.

The particles used are: - Kamaelia.Visualisation.Axon.PComponent - Kamaelia.Visualisation.Axon.PPostbox

The laws used are Kamaelia.Visualisation.Axon.AxonLaws.

The extra window furniture is supplied by Kamaelia.Visualisation.Axon.ExtraWindowFurniture.

Other

AxonVisualiser

AxonVisualiserServer(...) -> new AxonVisualiserServer component.

A specialisation of the TopologyViewerServer component for viewing Axon/Kamaelia systems.

Keyword arguments are those for TopologyViewerServer, excluding:

- particleTypes
- laws
- simCyclesPerRedraw
- extraWindowFurniture

AxonVisualiserServer

AxonVisualiserServer(...) -> new AxonVisualiserServer component.

A specialisation of the TopologyViewerServer component for viewing Axon/Kamaelia systems.

Keyword arguments are those for TopologyViewerServer, excluding:

- particleTypes
- laws
- simCyclesPerRedraw
- extraWindowFurniture

Kamaelia

This is a doc string, will it be of use?

text_to_token_lists

NO DOCS

Kamaelia.Visualisation.Axon.ExtraWindowFurniture

Kamaelia Cat logo renderer

Renderer for the topology viewer framework that renders a Kamaelia cat logo to the top left corner on rendering pass 10.

Example Usage

Create a topology viewer component that also renders 'ExtraWindowFurniture' to the display surface:

```
TopologyViewer( extraDrawing = ExtraWindowFurniture(),
                ...
              ).activate()
```

How does it work?

Instances of this class provide a render() generator that renders a Kamaelia cat logo at coordinates (8,8) to the specified pygame surface. The cat logo is scaled so its longest dimension (width or height) is 64 pixels.

Rendering is performed by the generator, returned when the render() method is called. Its behaviour is that needed for the framework for multi-pass rendering that is used by TopologyViewer.

The generator yields the number of the rendering pass it wishes to be next on next. Each time it is subsequently called, it performs the rendering required for that pass. It then yields the number of the next required pass or completes if there is no more rendering required.

An setOffset() method is also implemented to allow the particles coordinates to be offset. This therefore makes it possible to scroll the particles around the display surface.

See TopologyViewer for more details.

Other

ExtraWindowFurniture

> ExtraWindowFurniture() -> new ExtraWindowFurniture object.
>
> Renders a Kamaelia cat logo to the top left corner of a pygame surface when the render() generator is called.

Kamaelia.Visualisation.Axon.PComponent

"Component" particle for Axon/Kamaelia visualisation

This is an implementation of a rendering particle for "Component" particles in topology visualisation of Axon/Kamaelia systems.

Example Usage

See Kamaelia.Visualisation.Axon.AxonLaws or Kamaelia.Visualisation.Axon.AxonVisualiserServer

How does it work?

This object subclasses Kamaelia.Physics.Simple.Particle and adds methods to support rendering. Specifically, rendering to represent a component in an Axon/Kamaelia system.

At initialisation, provide a unique ID, a starting (x,y) position tuple, and a name. The name is displayed on the particle. If it is a dot-delimited string then only the final term is displayed on the actual particle.

If the particle becomes selected, then it will render its full name at the top of the display surface.

It also renders bonds *from* this particle *to* another. They are rendered as simple grey lines.

Rendering is performed by a generator, returned when the render() method is called. Its behaviour is that needed for the framework for multi-pass rendering that is used by TopologyViewer.

The generator yields the number of the rendering pass it wishes to be next on next. Each time it is subsequently called, it performs the rendering required for that pass. It then yields the number of the next required pass or completes if there is no more rendering required.

An setOffset() method is also implemented to allow the particles coordinates to be offset. This therefore makes it possible to scroll the particles around the display surface.

See TopologyViewer for more details.

Other

PComponent

PComponent(ID,position,name) -> new PComponent object.

Particle representing an Axon/Kamaelia Component for topology visualisation.

Keyword arguments:

- ID -- a unique ID for this particle
- position -- (x,y) tuple of particle coordinates
- name -- The full dot-delimited pathname of the component being represented

abbreviate

Abbreviates dot-delimited string to the final (RHS) term

acronym

Abbreviates strings to capitals, word starts and numerics and underscores

colours

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Kamaelia.Visualisation.Axon.PPostbox

"Postbox" particle for Axon/Kamaelia visualisation

This is an implementation of a rendering particle for "Postbox" particles in topology visualisation of Axon/Kamaelia systems, representing inboxes/outboxes.

Example Usage

See Kamaelia.Visualisation.Axon.AxonLaws or Kamaelia.Visualisation.Axon.AxonVisualiserServer

How does it work?

This object subclasses Kamaelia.Physics.Simple.Particle and adds methods to support rendering. Specifically, rendering to represent an inbox or outbox in an Axon/Kamaelia system.

At initialisation, provide a unique ID, a starting (x,y) position tuple, a name and whether it is an inbox or outbox. The name is abbreviated and displayed as the particle.

If the particle becomes selected, then it will render its full name at the top of the display surface.

At initialisation the label is rendered at several different 45 degree angles. When rendering, the appropriate one is chosen depending on the directions of bonds (linkages) this particle is involved in.

It also renders bonds *from* this particle *to* another. Their colour depends on whether they represent ordinary or passthrough linkages. This is determined by looking at whether both postbox particles involved are of the same type.

It is assumed that any bonds going *from* this particle *to* another go to another postbox particle (not a component particle). If this is not the case then behaviour is undefined.

Rendering is performed by a generator, returned when the render() method is called. Its behaviour is that needed for the framework for multi-pass rendering that is used by TopologyViewer.

The generator yields the number of the rendering pass it wishes to be next on next. Each time it is subsequently called, it performs the rendering required for that pass. It then yields the number of the next required pass or completes if there is no more rendering required.

An setOffset() method is also implemented to allow the particles coordinates to be offset. This therefore makes it possible to scroll the particles around the display surface.

See TopologyViewer for more details.

Other

PPostbox

PPostbox -> new PPostbox object.

Particle representing an Axon/Kamaelia inbox/outbox for topology visualisation.

Keyword arguments:

- position -- (x,y) tuple of particle coordinates
- name -- Name for the inbox/outbox being represented
- boxtype -- "inbox" or "outbox"

abbreviate

Abbreviates strings to capitals, word starts and numerics and underscores

angleMappings

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

nearest45DegreeStep

Returns (in degrees) the nearest 45 degree angle match to the supplied vector.

Returned values are one of 0, 45, 90, 135, 180, 225, 270, 315. If the supplied vector is (0,0), the returned angle is 0.

Kamaelia.Visualisation.ER.ERLaws

Other

AxonLaws

NO DOCS

Kamaelia

This is a doc string, will it be of use?

Kamaelia.Visualisation.ER.ERVisualiserServer

Other

ERVisualiser

- particleTypes
- laws
- simCyclesPerRedraw
- extraWindowFurniture

ERVisualiserServer

- particleTypes
- laws
- simCyclesPerRedraw
- extraWindowFurniture

Kamaelia

This is a doc string, will it be of use?

text_to_token_lists

NO DOCS

Kamaelia.Visualisation.ER.ExtraWindowFurniture

Kamaelia Cat logo renderer

Renderer for the topology viewer framework that renders a Kamaelia cat logo to the top left corner on rendering pass 10.

Example Usage

Create a topology viewer component that also renders 'ExtraWindowFurniture' to the display surface:

```
TopologyViewer( extraDrawing = ExtraWindowFurniture(),
                ...
              ).activate()
```

How does it work?

Instances of this class provide a render() generator that renders a Kamaelia cat logo at coordinates (8,8) to the specified pygame surface. The cat logo is scaled so its longest dimension (width or height) is 64 pixels.

Rendering is performed by the generator, returned when the render() method is called. Its behaviour is that needed for the framework for multi-pass rendering that is used by TopologyViewer.

The generator yields the number of the rendering pass it wishes to be next on next. Each time it is subsequently called, it performs the rendering required for that pass. It then yields the number of the next required pass or completes if there is no more rendering required.

An setOffset() method is also implemented to allow the particles coordinates to be offset. This therefore makes it possible to scroll the particles around the display surface.

See TopologyViewer for more details.

Other

ExtraWindowFurniture

> ExtraWindowFurniture() -> new ExtraWindowFurniture object.
>
> Renders a Kamaelia cat logo to the top left corner of a pygame surface when the render() generator is called.

Kamaelia.Visualisation.ER.PAttribute

Other

PAttribute

NO DOCS

abbreviate

Abbreviates strings to capitals, word starts and numerics and underscores

angleMappings

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

nearest45DegreeStep

Returns (in degrees) the nearest 45 degree angle match to the supplied vector.

> Returned values are one of 0, 45, 90, 135, 180, 225, 270, 315. If the supplied vector is (0,0), the returned angle is 0.

Kamaelia.Visualisation.ER.PEntity

Other

PEntity

NO DOCS

abbreviate

Abbreviates dot-delimited string to the final (RHS) term

acronym

Abbreviates strings to capitals, word starts and numerics and underscores

colours

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Kamaelia.Visualisation.ER.PISA

Other

PISA

NO DOCS

abbreviate

Abbreviates strings to capitals, word starts and numerics and underscores

angleMappings

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

nearest45DegreeStep

Returns (in degrees) the nearest 45 degree angle match to the supplied vector.

> Returned values are one of 0, 45, 90, 135, 180, 225, 270, 315. If the supplied vector is (0,0), the returned angle is 0.

Kamaelia.Visualisation.ER.PRelation

Other

PRelation

NO DOCS

abbreviate

Abbreviates strings to capitals, word starts and numerics and underscores

angleMappings

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d[k] = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

nearest45DegreeStep

Returns (in degrees) the nearest 45 degree angle match to the supplied vector.

> Returned values are one of 0, 45, 90, 135, 180, 225, 270, 315. If the supplied vector is (0,0), the returned angle is 0.

Kamaelia.Visualisation.PhysicsGraph.GridRenderer

Grid Renderer

Renderer for the topology viewer framework that renders horizontal and vertical gridlines on pass -1.

Example Usage

Already used by Kamaelia.Visualisation.PhysicsGraph.TopologyViewer.

Rendering a grid in light grey with grid cell size of 100x100:

```
grid = GridRenderer(size=100, colour=(200,200,200))
renderer = grid.render( <pygame surface> )
for rendering_pass in renderer:
  print "Rendering for pass ", rendering_pass
```

How does it work?

Instances of this class provide a render() generator that renders horizontal and vertical grid lines to conver the specified pygame surface. The colour and spacing of the grid lines are specified at initialisation.

Rendering is performed by the generator, returned when the render() method is called. Its behaviour is that needed for the framework for multi-pass rendering that is used by TopologyViewer.

The generator yields the number of the rendering pass it wishes to be next on next. Each time it is subsequently called, it performs the rendering required for that pass. It then yields the number of the next required pass or completes if there is no more rendering required.

A setOffset() method is also implemented to allow the rendering position to be offset. This therefore makes it possible to scroll the grid around the display surface.

See TopologyViewer for more details.

Other

GridRenderer

> GridRenderer(size,colour) -> new GridRenderer object.
>
> Renders a grid of horizontal and vertical lines with the specified grid square size and colour to cover a pygame surface when the render() generator is called.

Kamaelia.Visualisation.PhysicsGraph.ParticleDragger

Drag handler for Topology Viewer

A subclass of Kamaelia.UI.MH.DragHandler that implements "click and hold" dragging of particles for the TopologyViewer.

Example Usage

See source for TopologyViewer.

How does it work?

This is an implementation of Kamaelia.UI.MH.DragHandler. See that for more details.

The detect() method uses the withinRadius method of the physics attribute of the 'app' to determine which (if any) particle the mouse is hovering over when the drag is started. If there is no particle, then the drag does not begin.

At the start of the drag, the particle is 'frozen' to prevent motion due to the physics model of the topology viewer. This is achieved by calling the freeze() and unfreeze() methods of the particle. The particle is also 'selected'.

During the drag the particle's coordinates are updated and the physics model is notified of the change.

Other

ParticleDragger

ParticleDragger(event,app) -> new ParticleDragger object.

Implements mouse dragging of particles in a topology viewer. Bind the handle(...) class method to the MOUSEBUTTONDOWN pygame event to use it (via a lambda function or equivalent)

Keyword Arguments:

- event -- pygame event object cuasing this
- app -- PyGameApp component this is happening in

Kamaelia.Visualisation.PhysicsGraph.RenderingParticle

Simple generic particle for Topology visualisation

This is an implementation of a simple rendering particle for topology visualisation.

Example Usage

A topology viewer where particles of type "-" are rendered by RenderingParticle instances:

```
TopologyViewer( particleTypes = {"-":RenderingParticle},
                laws = Kamaelia.Support.Particles.SimpleLaws(),
              ).run()
```

SimpleLaws are used that apply the same simple physics laws for all particle types.

How does it work?

This object subclasses Kamaelia.Support.Particles.Particle and adds methods to support rendering.

At initialisation, provide a unique ID, a starting (x,y) position tuple, and a name. The name is displayed as a label ontop of the particle.

If the particle becomes selected it changes its visual appearance to reflect this.

It also renders bonds *from* this particle *to* another. They are rendered as simple lines.

Rendering is performed by a generator, returned when the render() method is called. Its behaviour is that needed for the framework for multi-pass rendering that is used by TopologyViewer.

The generator yields the number of the rendering pass it wishes to be next on next. Each time it is subsequently called, it performs the rendering required for that pass. It then yields the number of the next required pass or completes if there is no more rendering required.

An setOffset() method is also implemented to allow the particles coordinates to be offset. This therefore makes it possible to scroll the particles around the display surface.

See TopologyViewer for more details.

Other

RenderingParticle

RenderingParticle(ID,position,name) -> new RenderingParticle object.

Simple generic particle for topology visualisation.

Keyword arguments:

- ID -- a unique ID for this particle
- position -- (x,y) tuple of particle coordinates
- name -- A name this particle will be labelled with

Kamaelia.Visualisation.PhysicsGraph.TopologyViewer

Generic Topology Viewer

Pygame based display of graph topologies. A simple physics model assists with visual layout. Rendering and physics laws can be customised for specific applications.

Example Usage

A simple console driven topology viewer:

```
Pipeline( ConsoleReader(),
          lines_to_tokenlists(),
          TopologyViewer(),
        ).run()
```

Then at runtime try typing these commands to change the topology in real time:

```
>>> DEL ALL
>>> ADD NODE 1 "1st node" randompos -
>>> ADD NODE 2 "2nd node" randompos -
>>> ADD NODE 3 "3rd node" randompos -
>>> ADD LINK 1 2
>>> ADD LINK 3 2
>>> DEL LINK 1 2
>>> DEL NODE 1
```

User Interface

TopologyViewer manifests as a pygame display surface. As it is sent topology information nodes and links between them will appear.

You can click a node with the mouse to select it. Depending on the application, this may display additional data or, if integrated into another app, have some other effect.

Click and drag with the left mouse button to move nodes around. Note that a simple physics model or repulsion and attraction forces is always active. This causes nodes to move around to help make it visually clearer, however you may still need to drag nodes about to tidy it up.

The surface on which the nodes appear is notionally infinite. Scroll around using the arrow keys.

Press the 'f' key to toggle between windowed and fullscreen modes.

How does it work?

TopologyViewer is a specialisation of the Kamaeila.UI.MH.PyGameApp component. See documentation for that component to understand how it obtains and handles events for a pygame display surface.

A topology (graph) of nodes and links between them is rendered to the surface.

You can specify an initial topology by providing a list of instantiated particles and another list of pairs of those particles to show how they are linked.

TopologyViewer reponds to commands arriving at its "inbox" inbox instructing it on how to change the topology. A command is a list/tuple.

Commands recognised are:

["ADD", "NODE", <id>, <name>, <posSpec>, <particle type>]
Add a node, using:

- id -- a unique ID used to refer to the particle in other topology commands. Cannot be None.
- name -- string name label for the particle

- posSpec -- string describing initial x,y (see _generateXY)
- particleType -- particle type (default provided is "-", unless custom types are provided - see below)

[**"DEL", "NODE", <id>**] Remove a node (also removes all links to and from it)

[**"ADD", "LINK", <id from>, <id to>**] Add a link, directional from fromID to toID

[**"DEL", "LINK", <id from>, <id to>**] Remove a link, directional from fromID to toID

[**"DEL", "ALL"**] Clears all nodes and links

[**"GET", "ALL"**] Outputs the current topology as a list of commands, just like those used to build it. The list begins with a 'DEL ALL'.

[**"UPDATE_NAME", "NODE", <id>, <new name>**] If the node does not already exist, this does NOT cause it to be created.

[**"GET_NAME", "NODE", <id>**] Returns UPDATE_NAME NODE message for the specified node

[**"FREEZE", "ALL"**] Freezes all particles in the system, essentially halting the simulation

[**"UNFREEZE", "ALL"**] Unfreezes all particles in the system, essentially restarting the simulation

Commands are processed immediately, in the order in which they arrive. You therefore cannot refer to a node or linkage that has not yet been created, or that has already been destroyed.

If a stream of commands arrives in quick succession, rendering and physics will be temporarily stopped, so commands can be processed more quickly. This is necessary because when there is a large number of particles, physics and rendering starts to take a long time, and will therefore bottleneck the handling of commands.

However, there is a 1 second timeout, so at least one update of the visual output is guaranteed per second.

TopologyViewer sends any output to its "outbox" outbox in the same list/tuple format as used for commands sent to its "inbox" inbox. The following may be output:

[**"SELECT", "NODE", <id>**] Notification that a given node has been selected.

[**"SELECT", "NODE", None**] Notificaion that *no node* is now selected.

[**"ERROR", <error string>**] Notification of errors - eg. unrecognised commands arriving at the "inbox" inbox

[**"TOPOLOGY", <topology command list>**] List of commands needed to build the topology, as it currently stands. The list will start with a ("DEL","ALL") command. This is sent in response to receiving a ("GET","ALL") command.

Termination

If a shutdownMicroprocess message is received on this component's "control" inbox this it will pass it on out of its "signal" outbox and immediately terminate.

Historical note for short term: this has changed as of May 2008. In the past, this component would also shutdown when it recieved a producerFinished message. This has transpired to be a mistake for a number of different systems, hence the change to only shutting down when it recieves a shutdownMicroprocess message.

NOTE: Termination is currently rather cludgy - it raises an exception which will cause the rest of a kamaelia system to halt. Do not rely on this behaviour as it will be changed to provide cleaner termination at some point.

Customising the topology viewer

You can customise:

- the 'types' of particles (nodes)
- visual appearance of particles (nodes) and the links between them;
- the physics laws used to assist with layout
- extra visual furniture to be rendered

For example, see Kamaelia.Visualisation.Axon.AxonVisualiserServer. This component uses two types of particle - to represent components and inboxes/outboxes. Each has a different visual appearance, and the laws acting between them differ depending on which particle types are involved in the interaction.

Use the particleTypes argument of the initialiser to specify classes that should be instantiated to render each type of particle (nodes). particleTypes should be a dictionary mapping names for particle types to the respective classes, for example:

```
{ "major" : BigParticle,  "minor"  : SmallParticle  }
```

See below for information on how to write your own particle classes.

Layout of the nodes on the surface is assisted by a physics model, provided by an instance of the Kamaelia.Support.Particles.ParticleSystem class.

Customise the laws used for each particle type by providing a Kamaelia.Phyics.Simple.MultipleLaws object at initialisation.

Writing your own particle class

should inherit from Kamaelia.Support.Particles.Particle and implement the following methods (for rendering purposes):

setOffset((left,top)) Notification that the surface has been scrolled by the user. Particles should adjust the coordinates at which they render. For example, a particle at (x, y) should be rendered at (x-left, y-top). You can assume, until setOffset(...) is called, that (left,top) is (0,0).

select() Called to inform the particle that it is selected (has been clicked on)

deselect() Called to inform the particle that is has been deselected.

render(surface) -> generator Called to get a generator for multi-pass rendering of the particle (see below)

The coordinates of the particle are updated automatically both due to mouse dragging and due to the physics model. See Kamaelia.Support.Particles.Particle for more information.

The render(...) method should return a generator that will render the particle itself and its links/bonds to other particles.

Rendering by the TopologyViewer is multi-pass. This is done so that irrespective of the order in which particles are chosen to be rendered, things that need to be rendered before (underneath) other things can be done consistently.

The generator should yield the number of the rendering pass it wishes to be next called on. Each time it is subsequently called, it should perform the rendering required for that pass. It then yields the number of the next required pass or completes if there is no more rendering required. Passes go in ascending numerical order.

For example, Kamaelia.Visualisation.PhysicsGraph.RenderingParticle renders in two passes:

```
def render(self, surface):
    yield 1
    # render lines for bonds *from* this particle *to* others
    yield 2
    # render a blob and the name label for the particle
```

...in this case it ensures that the blobs for the particles always appear on top of the lines representing the bonds between them.

Note that rendering passes must be coded in ascending order, but the numbering can otherwise be arbitrary: The first pass can be any value you like; subsequent

passes can also be any value, provided it is higher.

When writing rendering code for particle(s), make sure they all agree on who should render what. It is inefficient if all bonds are being rendered twice. For example, RenderingParticle only renders links *from* that particle *to* another, but not in another direction.

Components

TopologyViewer

TopologyViewer(...) -> new TopologyViewer component.

A component that takes incoming topology (change) data and displays it live using pygame. A simple physics model assists with visual layout. Particle types, appearance and physics interactions can be customised.

Keyword arguments (in order):

- screensize -- (width,height) of the display area (default = (800,600))
- fullscreen -- True to start up in fullscreen mode (default = False)
- caption -- Caption for the pygame window (default = "Topology Viewer")
- particleTypes -- dict("type" -> klass) mapping types of particle to classes used to render them (default = {"-":RenderingParticle})
- initialTopology -- (nodes,bonds) where bonds=list((src,dst)) starting state for the topology (default=([],[]))
- laws -- Physics laws to apply between particles (default = SimpleLaws(bondlength=100))
- simCyclesPerRedraw -- number of physics sim cycles to run between each redraw (default=1)
- border -- Minimum distance from edge of display area that new particles appear (default=100)
- extraDrawing -- Optional extra object to be rendered (default=None)
- showGrid -- False, or True to show gridlines (default=True)
- transparency -- None, or (r,g,b) colour to make transparent
- position -- None, or (left,top) position for surface within pygame window

Other

Kamaelia

This is a doc string, will it be of use?

Kamaelia.Visualisation.PhysicsGraph.TopologyViewerComponent

This is a deprecation stub, due for later removal.

Other

TopologyViewerComponent

NO DOCS

Kamaelia.Visualisation.PhysicsGraph.TopologyViewerServer

Generic Topology Viewer Server

A generic topology viewer that one client can connect to at a time over a TCP socket and send topology change data for visualisation.

Example Usage

Visualiser that listens on port 1500 for a TCP connection through which it receives topology change data to render:

```
TopologyViewerServer( serverPort = 1500 ).run()
```

A simple client to drive the visualiser:

```
Pipeline( ConsoleReader(),
          TCPClient( server=<address>, port=1500 ),
        ).run()
```

Run the server, then run the client:

```
>>> DEL ALL
>>> ADD NODE 1 "1st node" randompos -
>>> ADD NODE 2 "2nd node" randompos -
>>> ADD NODE 3 "3rd node" randompos -
>>> ADD LINK 1 2
>>> ADD LINK 3 2
>>> DEL LINK 1 2
>>> DEL NODE 1
```

See also Kamaelia.Visualisation.Axon.AxonVisualiserServer - which is a specialisation of this component.

How does it work?

TopologyViewerServer is a Pipeline of the following components:

- Kamaelia.Internet.SingleServer
- chunks_to_lines

- lines_to_tokenlists
- TopologyViewer
- ConsoleEchoer

This Pipeline serves to listen on the specified port (defaults to 1500) for clients. One client is allowed to connect at a time.

That client can then send topology change commands formatted as lines of text. The lines are parsed and tokenised for the TopologyViewer.

Any output from the TopologyViewer is sent to the console.

If the noServer option is used at initialisation, then the Pipeline is built without the SingleServer component. It then becomes a TopologyViewer capable of processing non-tokenised input and with diagnostic console output.

See TopologyViewer for more detail on topology change data and its behaviour.

Other

TextControlledTopologyViewer

NO DOCS

TopologyViewerServer

> TopologyViewerServer([noServer][,serverPort],**args) -> new TopologyViewerServer component.
>
> Multiple-clients-at-a-time TCP socket Topology viewer server. Connect on the specified port and send topology change data for display by a TopologyViewer.
>
> Keyword arguments:
>
> - serverPort -- None, or port number to listen on (default=1500)
> - args -- all remaining keyword arguments passed onto TopologyViewer

Users

NO DOCS

Kamaelia.Visualisation.PhysicsGraph.chunks_to_lines

Text line splitter

This component takes chunks of text and splits them at line breaks into individual lines.

Example usage

A system that connects to a server and receives fragmented text data, but then displays it a whole line at a time:

```
Pipeline( TCPClient(host=..., port=...),
  chunks_to_lines(),
  ConsoleEcho()
).run()
```

How does it work?

chunks_to_lines buffers all text it receives on its "inbox" inbox. If there is a line break ("n") in the text it has buffered, then it extracts that line of text, including the line break character and sends it on out of its "outbox" outbox.

It also removes any "r" characters in the text.

If a producerFinished() or shutdownMicroprocess() message is received on this component's "control" inbox, then it will send it on out of its "signal" outbox and immediately terminate. It will not flush any whole lines of text that may still be buffered.

Components

chunks_to_lines

chunks_to_lines() -> new chunks_to_lines component.

Takes in chunked textual data and splits it at line breaks into individual lines.

Kamaelia.Visualisation.PhysicsGraph.lines_to_tokenlists

Simple line-of-text tokeniser

This component takes a line of text and splits it into space character separated tokens. Tokens can be encapsulated with single or double quote marks, allowing spaces to appear within a token.

Example Usage

A simple pieline that takes each line you type and splits it into a list of tokens, showing you the result:

```
Pipeline( ConsoleReader(),
          lines_to_tokenlists(),
          ConsoleEchoer()
        ).run()
```

At runtime:: »> Hello world "how are you" 'john said "hi"' "i replied "hi"" "c:windows" end ['Hello', 'world', 'how are you', 'john said "hi"', 'i replied "hi"', 'c:windows', 'end']

How does it work?

lines_to_tokenlists receives individual lines of text on its "inbox" inbox. A line is converted to a list of tokens, which is sent out of its "outbox" outbox.

Space characters are treated as the token separator, however a token can be encapsulated in single or double quotes allowing space characters to appear within it.

If you need to use a quote mark or backslash within a token encapsulated by quote marks, it must be escaped by prefixing it with a backslash. Only do this if the token is encapsulated.

encapsulating quote marks are removed when the line is tokenised. Escaped backslashes and quote marks are converted to plain backslashes and quote marks.

If a producerFinished() or shutdownMicroprocess() message is received on this component's "control" inbox, then it will send it on out of its "signal" outbox and immediately terminate. It will not flush any whole lines of text that may still be buffered.

Components

lines_to_tokenlists

lines_to_tokenlists() -> new lines_to_tokenlists component.

Takes individual lines of text and separates them into white space separated tokens. Tokens can be enclosed with single or double quote marks.

Kamaelia.Visualisation.PhysicsGraph3D.Particles3D

Particle3D: Simple generic/ supertype particle for 3D Topology visualisation

This is an implementation of a simple supertype particle for 3D topology visualisation.

Example Usage

Subclass it and extend it by adding draw() method to render any shape you want the particle to be.

How does it work?

This object subclasses Kamaelia.Support.Particles.Particle and adds 3D elements.

At initialisation, provide a unique ID, a starting (x,y,z) position tuple, and a name. The name is displayed as a label on top of the particle. For other parameters, such as bgcolour and fgcolour, see its doc string below.

If the particle becomes selected it changes its visual appearance to reflect this.

There are two kinds of textures, i.e. text label and image textures. When the 'image' argument is provided, it uses image textures; otherwise, it uses text label textures in which particle name is used as the caption of the label. Note, the value of the 'image' argument is the uri of the image; it could be a path in local drive, a network address or an internet address.

It only serves as a superclass of 3D particle and has no rendering (draw) method, so it leaves the shape rendering to subclasses.

CuboidParticle3D: cuboid rendering particle for 3D Topology visualisation

This is an implementation of a simple cuboid particle for 3D topology visualisation.

Example Usage

A 3D topology viewer where particles of type "-" are rendered by CuboidParticle3D instances:

```
TopologyViewer3D( particleTypes = {"-":CuboidParticle3D},
                  laws = Kamaelia.Support.Particles.SimpleLaws(bondLength=2),
                ).run()
```

SimpleLaws are used that apply the same simple physics laws for all particle types.

How does it work?

This object subclasses Kamaelia.Visualisation.PhysicsGraph3D.Particles3D.Particle3D and adds methods to support rendering (draw).

SphereParticle3D: sphere rendering particle for 3D Topology visualisation

This is an implementation of a simple sphere particle for 3D topology visualisation.

Note: it would be much slower than CuboidParticle3D because it uses GLU library.

Example Usage

A 3D topology viewer where particles of type "sphere" are rendered by SphereParticle3D instances:

```
TopologyViewer3D( particleTypes = {"sphere":SphereParticle3D},
                  laws = Kamaelia.Support.Particles.SimpleLaws(bondLength=2),
                ).run()
```

SimpleLaws are used that apply the same simple physics laws for all particle types.

How does it work?

This object subclasses Kamaelia.Visualisation.PhysicsGraph3D.Particles3D.Particle3D and adds methods to support rendering (draw).

TeapotParticle3D: teapot rendering particle for 3D Topology visualisation

This is an implementation of a simple teapot particle for 3D topology visualisation.

Note: it would be much slower than CuboidParticle3D and SphereParticle3D because it uses GLUT library.

Example Usage

A 3D topology viewer where particles of type "teapot" are rendered by CuboidParticle3D instances:

```
TopologyViewer3D( particleTypes = {"teapot":TeapotParticle3D},
                  laws = Kamaelia.Support.Particles.SimpleLaws(bondLength=2),
                ).run()
```

SimpleLaws are used that apply the same simple physics laws for all particle types.

How does it work?

This object subclasses Kamaelia.Visualisation.PhysicsGraph3D.Particles3D.Particle3D and adds methods to support rendering (draw).

References: 1. Kamaelia.UI.OpenGL.Button 2. Kamaelia.UI.OpenGL.OpenGLComponent

Other

CuboidParticle3D

Cuboid rendering particle

Particle3D

A super class for 3D particles super(RenderingParticle3D, self).___init___(**argd)

Simple 3D generic superclass particle for topology visualisation.

Keyword arguments:

- ID -- a unique ID for this particle
- position -- (x,y,z) tuple of particle coordinates
- name -- A name this particle will be labelled with
- bgcolour -- Colour of surfaces behind text label (default=(230,230,230)), only apply to label texture
- fgcolour -- Colour of the text label (default=(0,0,0), only apply to label texture
- sidecolour -- Colour of side planes (default=(200,200,244)), only apply to CuboidParticle3D
- bgcolourselected -- Background colour when the particle is selected (default=(0,0,0)
- bgcolourselected -- Frontground colour when the particle is selected (default=(244,244,244))
- sidecolourselected -- Side colour when the particle is selected (default=(0,0,100))
- size -- Size of particle (length, width, depth), it depends on texture size if unspecified
- margin -- Margin size in pixels (default=8)
- fontsize -- Font size for label text (default=50)
- pixelscaling -- Factor to convert pixels to units in 3d, ignored if size is specified (default=100)
- thickness -- Thickness of button widget, ignored if size is specified (default=0.3)
- image -- The uri of image, image texture instead of label texture is used if specified

SphereParticle3D

Sphere rendering particle

TeapotParticle3D

Teapot rendering particle

Kamaelia.Visualisation.PhysicsGraph3D.TopologyViewer3D

Generic 3D Topology Viewer

A 3D version of TopologyViewer plus hierarchy topology support, pygame based display of graph topologies. Rendering and physics laws can be customised for specific applications.

Example Usage

A simple console driven topology viewer:

```
Pipeline( ConsoleReader(),
          lines_to_tokenlists(),
          TopologyViewer3D(),
        ).run()
```

Then at runtime try typing these commands to change the topology in real time:

```
>>> DEL ALL
>>> ADD NODE 1 "1st node" (0,0,-10) teapot
>>> ADD NODE 2 "2nd node" randompos sphere
>>> ADD NODE 3 "3rd node" randompos -
>>> ADD NODE 1:1 "1st child node of the 1st node" " ( 0 , 0 , -10 ) " -
>>> ADD NODE 1:2 "2nd child node of the 1st node" randompos -
>>> ADD LINK 1 2
>>> ADD LINK 3 2
>>> DEL LINK 1 2
>>> ADD LINK 1:1 1:2
>>> DEL NODE 1
```

User Interface

TopologyViewer3D manifests as a pygame OpenGL display surface. As it is sent topology information, nodes and links between them will appear.

You can click a node with the mouse to select it. Depending on the application, this may display additional data or, if integrated into another app, have some other effect.

Click and drag with the left mouse button to move nodes around. Note that a simple physics model or repulsion and attraction forces is always active. This causes nodes to move around to help make it visually clearer, however you may still need to drag nodes about to tidy it up.

For hierarchy topology, double-click a particle (or select one then press return key) to show its child topology; right-click (or press backspace key) to show last level's topology.

Operations supported:

- esc --- quit
- a --- viewer position moves left
- d --- viewer position moves right
- w --- viewer position moves up
- s --- viewer position moves down
- pgup --- viewer position moves forward (zoom in)
- pgdn --- viewer position moves backward (zoom out)
- left --- rotate selected particles to left around y axis (all particles if none of them is selected)
- right --- rotate selected particles to right around y axis (all particles if none of them is selected)
- up --- rotate selected particles to up around x axis (all particles if none of them is selected)
- down --- rotate selected particles to down around x axis (all particles if none of them is selected)
- < --- rotate selected particles anticlock-wise around z axis (all particles if none of them is selected)
- > --- rotate selected particles clock-wise around z axis (all particles if none of them is selected)
- return --- show next level's topology of the selected particle when only one particle is selected
- backspace --- show last level's topology
- Mouse click --- click particle to select one, click empty area to deselect all
- Mouse drag --- move particles
- Mouse double-click --- show next level's topology of the particle clicked
- Mouse right-click --- show last level's topology
- shift --- multi Select Mode; shift+click for multiple selection/ deselection
- **ctrl --- rotation Mode; when ctrl is pressed, mouse motion will rotate the selected par** (all particles if none of them is selected)

How does it work?

TopologyViewer3D is a Kamaeila component which renders Topology on a pygame OpenGL display surface.

A 3D topology (graph) of nodes and links between them is rendered to the surface.

You can specify an initial topology by providing a list of instantiated particles and another list of pairs of those particles to show how they are linked.

TopologyViewer3D responds to commands arriving at its "inbox" inbox instructing it on how to change the topology. A command is a list/tuple.

Commands recognised are:

[**"ADD", "NODE", <id>, <name>, <posSpec>, <particle type>**]
Add a node, using:

- **id -- a unique ID used to refer to the particle in other topology commands. Canno**
  For hierarchy topology, the id is joined by its parent id with ":" to represent the hierarchy structure. E.g., suppose the topology has 3 levels. The id of a particle in the 1st level is 1Node; it has a child particle whose id is 2Node; 2Node also has a child particle whose id is 3Node; then their ids are represented as 1Node 1Node:2Node 1Node:2Node:3Node
- name -- string name label for the particle
- **posSpec -- string describing initial (x,y,z) (see _generateXY); spaces are allowed**
  within the tuple, but quotation is needed in this case. E.g., " (0 , 0 , -10) "
- **particleType -- particle type (default provided is "-", unless custom types are prov**
  currently supported: "-" same as cuboid, cuboid, sphere and teapot Note: it would be much slower than cuboid if either sphere or teapot is used.

[**"DEL", "NODE", <id>**] Remove a node (also removes all links to and from it)

[**"ADD", "LINK", <id from>, <id to>**] Add a link, directional from fromID to toID

[**"DEL", "LINK", <id from>, <id to>**] Remove a link, directional from fromID to toID

[**"DEL", "ALL"**] Clears all nodes and links

[**"GET", "ALL"**] Outputs the current topology as a list of commands, just like those used to build it. The list begins with a 'DEL ALL'.

[**"UPDATE_NAME", "NODE", <id>, <new name>**] If the node does not already exist, this does NOT cause it to be created.

[**"GET_NAME", "NODE", <id>**] Returns UPDATE_NAME NODE message for the specified node

Commands are processed immediately, in the order in which they arrive. You therefore cannot refer to a node or linkage that has not yet been created, or that has already been destroyed.

If a stream of commands arrives in quick succession, rendering and physics will be temporarily stopped, so commands can be processed more quickly. This is necessary because when there is a large number of particles, physics and rendering starts to take a long time, and will therefore bottleneck the handling of commands.

However, there is a 1 second timeout, so at least one update of the visual output is guaranteed per second.

TopologyViewer sends any output to its "outbox" outbox in the same list/tuple format as used for commands sent to its "inbox" inbox. The following may be output:

[**"SELECT", "NODE", <id>**] Notification that a given node has been selected.

[**"SELECT", "NODE", None**] Notificaion that *no node* is now selected.

[**"TOPOLOGY", <topology command list>**] List of commands needed to build the topology, as it currently stands. The list will start with a ("DEL","ALL") command. This is sent in response to receiving a ("GET","ALL") command.

Error and tip information is printed out directly when applied.

For hierarchy topology, the id of particles should be joined by its parent id with ":" to represent the hierarchy structure. See "ADD NODE" command above for more information.

Termination

If a shutdownMicroprocess message is received on this component's "control" inbox, it will pass it on out of its "signal" outbox and immediately terminate.

NOTE: Termination is currently rather cludgy - it raises an exception which will cause the rest of a kamaelia system to halt. Do not rely on this behaviour as it will be changed to provide cleaner termination at some point.

Customising the 3D topology viewer

You can customise:

- the 'types' of particles (nodes)
- visual appearance of particles (nodes) and the links between them;
- the physics laws used to assist with layout

Use the particleTypes argument of the initialiser to specify classes that should be instantiated to render each type of particle (nodes). particleTypes should be a dictionary mapping names for particle types to the respective classes, for example:

```
{ "major" : BigParticle,  "minor"  : SmallParticle  }
```

See below for information on how to write your own particle classes.

Layout of the nodes on the surface is assisted by a physics model, provided by an instance of the Kamaelia.Support.Particles.ParticleSystem class. Freeze them if you want to make some particles not subject to the law (particle.freeze()).

Customise the laws used for each particle type by providing a Kamaelia.Phyics.Simple.MultipleLaws object at initialisation.

Writing your own particle class

should inherit from Kamaelia.PhysicsGraph3D.Particles3D.Particle3D and implement the following method (for rendering purposes):

draw() draw OpenGL particles and links in this method.

TODO: Reduce CPU usage, improve responsive speed

References: 1. Kamaelia.Visualisation.PhysicsGraph.TopologyViewer 2. Kamaelia.UI.OpenGL.OpenGLComponent 3. Kamaelia.UI.OpenGL.MatchedTranslationInteractor

Components

TopologyViewer3D

TopologyViewer3D(...) -> new TopologyViewer3D component.

A component that takes incoming topology (change) data and displays it live using pygame OpenGL. A simple physics model assists with visual layout. Particle types, appearance and physics interactions can be customised.

Keyword arguments (in order):

- screensize -- (width,height) of the display area (default = (800,600))
- fullscreen -- True to start up in fullscreen mode (default = False)
- caption -- Caption for the pygame window (default = "3D Topology Viewer")

- particleTypes -- dict("type" -> klass) mapping types of particle to classes used to render them (default = {"-":CuboidParticle3D})
- initialTopology -- (nodes,bonds) where bonds=list((src,dst)) starting state for the topology (default=([],[]))
- laws -- Physics laws to apply between particles (default = SimpleLaws(bondlength=2))
- simCyclesPerRedraw -- number of physics sim cycles to run between each redraw (default=1)
- border -- Minimum distance from edge of display area that new particles appear (default=0)

Other

Kamaelia

This is a doc string, will it be of use?

Kamaelia.Visualisation.PhysicsGraph3D.TopologyViewer3DWithPara

Generic 3D Topology Viewer With more Parameters supports

Extend TopologyViewer3D by supporting additional parameters of "ADD" and "UPDATE" commands.

Example Usage

A simple console driven topology viewer:

```
Pipeline( ConsoleReader(),
          lines_to_tokenlists(),
          TopologyViewer3DWithParams(),
        ).run()
```

Then at runtime try typing these commands to change the topology in real time:

```
>>> DEL ALL
>>> ADD NODE 1 "1st node" (0,0,-10) teapot
>>> ADD NODE 2 "2nd node" randompos sphere image=../../../Docs/cat.gif
>>> ADD NODE 3 "3rd node" randompos - bgcolour=(255,255,0);bgcolour=(0,255,255)
>>> UPDATE NODE 1 name=1st;bgcolour=(0,255,0)
>>> UPDATE NODE 3 name=3rd;bgcolour=(255,0,0);fgcolour=(0,0,255);fontsize=100
>>> ADD NODE 1:1 "1st child node of the 1st node" " ( 0 , 0 , -10 ) " -
>>> ADD NODE 1:2 "2nd child node of the 1st node" randompos - "fontsize = 20"
>>> ADD LINK 1 2
>>> ADD LINK 3 2
```

```
>>> DEL LINK 1 2
>>> ADD LINK 1:1 1:2
>>> DEL NODE 1
```

How does it work?

Extend TopologyViewer3D by supporting additional parameters of "ADD" and "UPDATE" commands.

The format of "ADD" commands: ["ADD", "NODE", <id>, <name>, <positionSpec>, <particle type>, <parameters>]

The format of "UPDATE" commands: ["UPDATE", "NODE", <id>, <parameters>]

The format of parameters: pa=pa_value;pb=pb_value

Add quotation if there are spaces within parameters.

Available parameters:

- bgcolour -- Colour of surfaces behind text label (default=(230,230,230)), only apply to label texture
- fgcolour -- Colour of the text label (default=(0,0,0), only apply to label texture
- sidecolour -- Colour of side planes (default=(200,200,244)), only apply to CuboidParticle3D
- bgcolourselected -- Background colour when the particle is selected (default=(0,0,0)
- bgcolourselected -- Frontground colour when the particle is selected (default=(244,244,244))
- sidecolourselected -- Side colour when the particle is selected (default=(0,0,100))
- margin -- Margin size in pixels (default=8)
- fontsize -- Font size for label text (default=50)
- pixelscaling -- Factor to convert pixels to units in 3d, ignored if size is specified (default=100)
- thickness -- Thickness of button widget, ignored if size is specified (default=0.3)
- image -- The uri of image, image texture instead of label texture is used if specified

See Kamaelia.PhysicsGraph3D.TopologyViewer3D.TopologyViewer3D for more information.

Components

TopologyViewer3DWithParams

TopologyViewer3DWithParams(...) -> new TopologyViewer3DWithParams component.

A component that takes incoming topology (change) data and displays it live using pygame OpenGL. A simple physics model assists with visual layout. Particle types, appearance and physics interactions can be customised.

It extends TopologyViewer3D by supporting additional parameters of "ADD" commands.

Keyword arguments (in order):

- screensize -- (width,height) of the display area (default = (800,600))
- fullscreen -- True to start up in fullscreen mode (default = False)
- caption -- Caption for the pygame window (default = "Topology Viewer")
- particleTypes -- dict("type" -> klass) mapping types of particle to classes used to render them (default = {"-":RenderingParticle})
- initialTopology -- (nodes,bonds) where bonds=list((src,dst)) starting state for the topology (default=([],[]))
- laws -- Physics laws to apply between particles (default = SimpleLaws(bondlength=100))
- simCyclesPerRedraw -- number of physics sim cycles to run between each redraw (default=1)
- border -- Minimum distance from edge of display area that new particles appear (default=100)

Other

paramStr2paramDict

Transform a parameters string to a parameters dictionary.

Kamaelia.XML.SimpleXMLParser

Simple/Basic parsing of XML using SAX

XMLParser parses XML data sent to its "inbox" inbox using SAX, and sends out "document", "element" and "character" events out of its "outbox" outbox.

Example Usage

The following code:

```
Pipeline( RateControlledFileReader("Myfile.xml"),
          SimpleXMLParser(),
          ConsoleEchoer(),
        ).run()
```

If given the following file as input:

```
<EDL xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="Mo
    <FileID>File identifier</FileID>

    <Edit>
        <Start frame="0"  />
        <End   frame="24" />
        <Crop  x1="0" y1="0" x2="400" y2="100" />
    </Edit>
    <Edit>
        <Start frame="25" />
        <End   frame="49" />
        <Crop  x1="80" y1="40" x2="480" y2="140" />
    </Edit>
</EDL>
```

Will output the following (albeit without the newlines, added here for clarity):

```
('document',)
('element', u'EDL', <xml.sax.xmlreader.AttributesImpl instance at 0x2aaaaca86e60>)
('chars', u'>')
('chars', u'\n')
('chars', u'    ')
('element', u'FileID', <xml.sax.xmlreader.AttributesImpl instance at 0x2aaaac028758>)
('chars', u'File identifier')
('/element', u'FileID')
('chars', u'\n')
('chars', u'    ')
('chars', u'\n')
('chars', u'    ')
('element', u'Edit', <xml.sax.xmlreader.AttributesImpl instance at 0x2aaaac028908>)
('chars', u'\n')
('chars', u'        ')
('element', u'Start', <xml.sax.xmlreader.AttributesImpl instance at 0x2aaaac028908>)
('/element', u'Start')
('chars', u'\n')
('chars', u'        ')
('element', u'End', <xml.sax.xmlreader.AttributesImpl instance at 0x2aaaaca86e60>)
('/element', u'End')
('chars', u'\n')
('chars', u'        ')
('element', u'Crop', <xml.sax.xmlreader.AttributesImpl instance at 0x2aaaac028908>)
```

```
('/element', u'Crop')
('chars', u'\n')
('chars', u'    ')
('/element', u'Edit')
('chars', u'\n')
('chars', u'    ')
('/element', u'EDL')
```

What does it output?

What is output is effectively a simple, but useful, parsing of the XML. It is a set of messages representing a subset of the python SAX ContentHandler. All are in the form of a simple tuple where the first term is always the type of "thing" identified - 'document' start or finish, 'element' start or finish, or raw text characters:

"("document",)"

- Start of the XML document

"("/document",)"

- End of the XML document (not sent if shutdown with a shutdownMicroprocess() message)

"("element", name, attributes)"

- Start tag for an element. `name` is the name of the element. `attributes` is a SAX attributes object - which behaves just like a dictionary - mapping attribute names to strings of their values. For example:

  ```
  ("element", "img", {"src":"/images/mypic.jpg"})
  ```

"("/element", name)"

- End tag for an element. `name` is the name of the element.

"("chars", textfragment)"

- Fragment of text from within an element. Note that the text contained in a single element be comprised of multiple fragments. They will include whitespace and newline characters.

Behaviour

Send chunks of text making up an XML document to this component's "inbox" inbox. The XML Parser used will output identified items from its "outbox" outbox.

This component supports sending its output to a size limited inbox. If the size limited inbox is full, this component will pause until it is able to send out the data.

If a producerFinished message is received on the "control" inbox, this component will complete parsing any data pending in its inbox, and finish sending any resulting data to its outbox. It will then send the producerFinished message on out of its "signal" outbox and terminate.

If a shutdownMicroprocess message is received on the "control" inbox, this component will immediately send it on out of its "signal" outbox and immediately terminate. It will not complete processing, or sending on any pending data.

Components

SimpleXMLParser

SimpleXMLParser() -> new SimpleXMLParser component.

Send XML data to the "inbox" inbox, and events describing documents, elements and blocks of characters (as parsed by SAX) will be sent out of the "outbox" outbox.