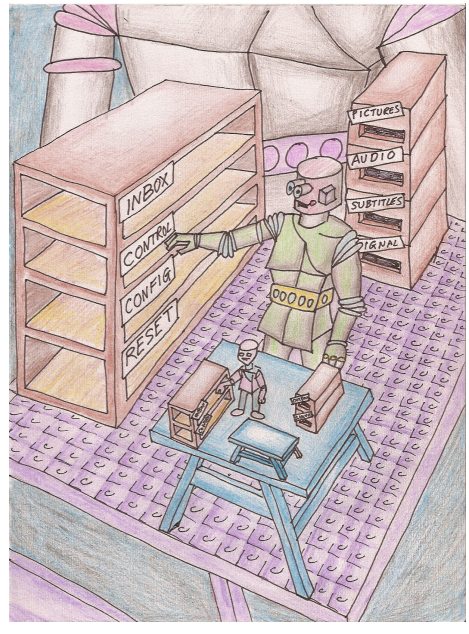
Kamaelia:

Pragmatic Concurrency

Michael Sparks



Tutorial Notes

europython 2009

Contents

Preface

Why use concurrency? Since concurrency is viewed as an advanced topic by many developers, this question is often overlooked. However, many real world systems, including transportation, companies, electronics and Unix systems are highly concurrent and accessible by the majority of people. So, one motivation can be “many hands make light work”.

However, with software this maxim often appears to be false – in no small part due to the tools we use to create concurrent systems. Despite this, the need for concurrency often creeps into many systems – even something as basic as “attaching a debugger”.

Kamaelia is a toolset and mindset aimed at assisting in structuring your code such that you can focus on the problem you want to solve, but in a way that results in naturally reusable code that happens to be concurrent.

This tutorial aims to introduce you to the follow 4 core aspects of Kamaelia systems:

Axon

All Kamaelia systems are dependent on this core library – it provides you with tools for making systems which are naturally concurrent. Its primary metaphor is components with inboxes and outboxes, which get linked by parent components.

Kamaelia Components

The bulk of Kamaelia is actually a large collection of components. By themselves each component is useful, but their real power comes from being linked to each other, like programs in /usr/bin get linked together, forming pipelines.

Applications

This is the point of Kamaelia – to build useful systems. Kamaelia was originally designed for naturally concurrent problems, so there was a desire to make this simple(r) to work with. Using Axon based components means applications generate more reusable components, and also have a naturally concurrent structure. This can simplify many applications, allowing their reuse in unexpected ways.

Testing & Debugging

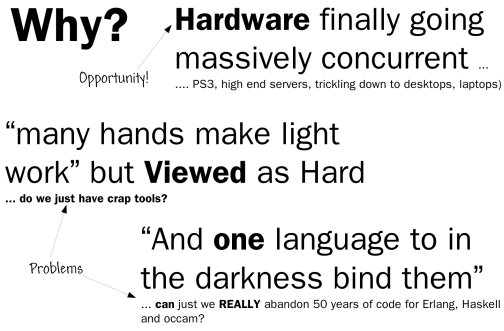
Testing & debugging concurrent systems is considered hard, we'll cover some approaches we can use in Kamaelia for debugging systems. Some of these are surprisingly familiar. Indeed, some can be used in non-Kamaelia based systems.

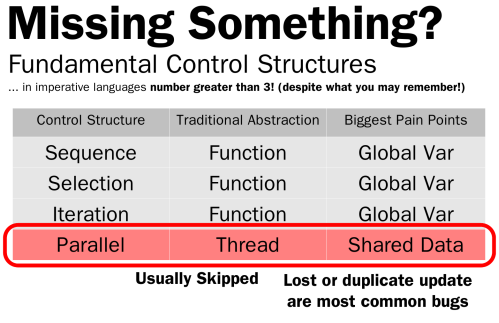
Kamaelia was designed originally to make maintenance of highly concurrent network systems simpler, but has general application in a wider variety of problem domains, including desktop applications, web backend systems (eg video transcode & SMS services), through to tools for teaching a child to read and write.

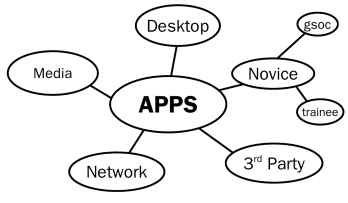
Kamaelia: Pragmatic Concurrency

The 30,000 foot view

We start at the beginning with an overview of the various aspects of a Kamaelia.







In order to understand Kamaelia it's useful to review **why** it exists. The original use case was in the case of network systems and a recognition that hardware is going massively multicore.

Whilst there are concurrency languages – Erlang, Occam, Bash(!) – it is likely that no one language will rule. So, in order to work with either naturally concurrent problems and/or hardware, we need better tools – conceptual, libraries, etc.

Kamaelia's core goal is to harness concurrency.

Against this backdrop, few developers get properly trained with dealing with concurrency, because crucially parallel control flow is often skipped. The downside of this is that it means that many developers do not get taught the downsides of its traditional abstraction.

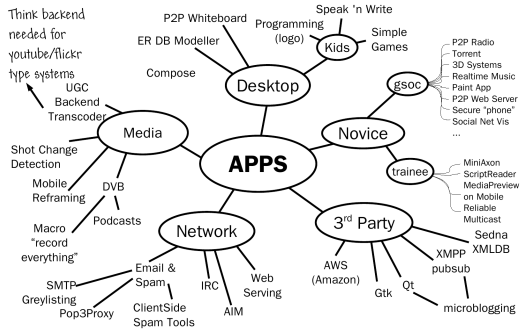
For example, it's well known that global data is generally a bad idea in traditional code, but less so that shared data in a parallel system is a similar cause of pain, in similar ways.

Beyond this, Kamaelia is not intended to be a theoretical system.

Its original use case was in the realm of experimental network servers, and since then it's grown to encompass desktop applications, media rich systems, server systems, access to third party libraries, etc.

Crucially, we've also worked on trying to make Kamaelia's approach & metaphors accessible – to both novices and experienced developers alike. We've tested this in through participation in GSOC mentoring programmes, and also a variety of other systems.

That leads to the question “what sort of systems?” The diagram on the right gives a general overview of the scope of the sort of applications, and more detail on these can be found on the website.

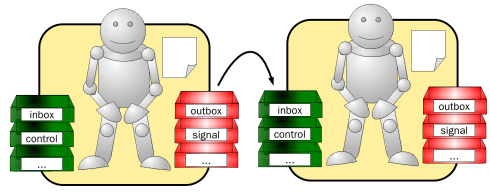


Every one of these systems is built, like shell scripts, using small, communicating chunks of code, which can be used in other systems.

This mixability allows entire sub-systems to be easily integrated with many others, meaning Kamaelia becomes more flexible with the more systems that you build.

Core Approach:

Concurrent things with comms points
Generally send messages
Keep data private, don't share

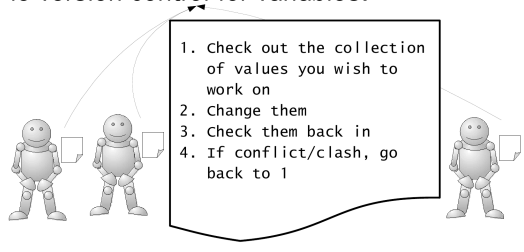


The core approach boils down to a few common points:

- All data has a single owner at any point in time - no unconstrained sharing of data.
- Components don't know about each other, they only know about inboxes and outboxes
- Data flows from outboxes to inboxes
- Code **is** data – you cannot call functions owned by other components, you send a message.

But I must share data?

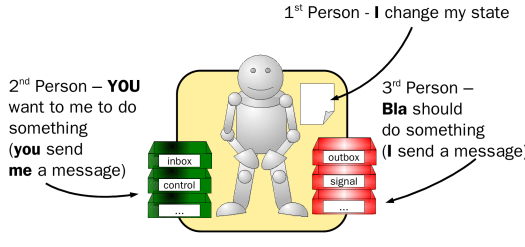
Use Software Transactional Memory
ie version control for variables.



This naturally then leads to the question of “how do I share data, since I **have** to sometimes”. The approach taken is to use “Software Transactional Memory”, which can be very closely approximated as “version control for variables”.

It doesn't actually store the history, but it allows you to find out if something can go wrong, and prevents you from storing bad data accidentally.

Perspectives in APIs! (1/2)

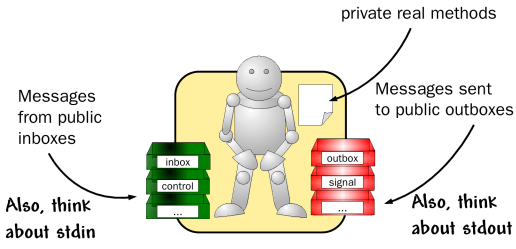


Slightly more philosophically, when talking about concurrency metaphors, like little robots doing work, or inboxes & outboxes are useful, since they start allowing us to think about what's happening more clearly. Specifically 1st,2nd,3rd person views.

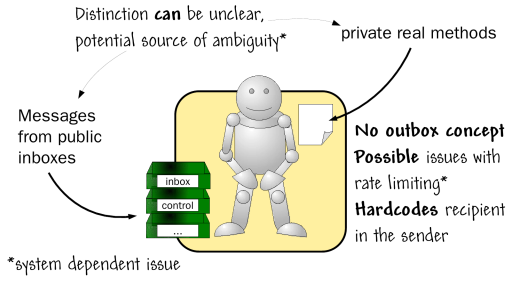
Eg 1st person: “**I** change my state” using a method or action. 2nd Person: **YOU** may want 'me' to do something, so **you** send me a message on an inbox.

Whereas 3rd person maps to “I may want '**someone else**' to act on a piece of work I've done” so I send a message to an outbox.

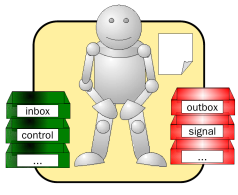
Perspectives in APIs! (2/2)



Actor Systems



Advantages of outboxes



No hardcoding of recipient allows:
- Late Binding
- Dynamic rewiring

Concurrency Patterns as Reusable Code
... a concurrency DSL

A Core Concurrency DSL

```
Pipeline(A,B,C)
Graphline(A=A,B=B, C=C, linkages = {})
Tpipe(cond, C)
Seq(A,B,C), PAR(), ALT()
Backplane("name"), PublishTo("name"), SubscribeTo("name")
Carousel(...)
PureTransformer(...)
StatefulTransformer(...)
PureServer(...)
MessageDemuxer(...)
Source(*messages)
NullSink
```

Some of these are *work in progress* – they've been identified as useful, but not implemented as chassis, yet

It also becomes natural to think of "who changes what?". Inboxes are public, write only outside, read only inside. Outboxes are also public, and write only from inside and read only outside.

When you take data from an inbox you own it, when you put it in an outbox, you don't.

All other data & methods are then considered private to the component, since their operation requires a first person view of the system.

As a comparison with Actor systems, these are generally defined as only having inboxes. This means that in order to send a message, you first need to know the recipient. Another aspect is that objects have both "normal" methods and methods which are actually message passing.

This leads to the fact that you can be expected to call what looks like a method on another object. This can be a source of ambiguity about what you can/should call, as well as receiver hardcoding.

By using explicit outboxes you gain some relatively obvious benefits:

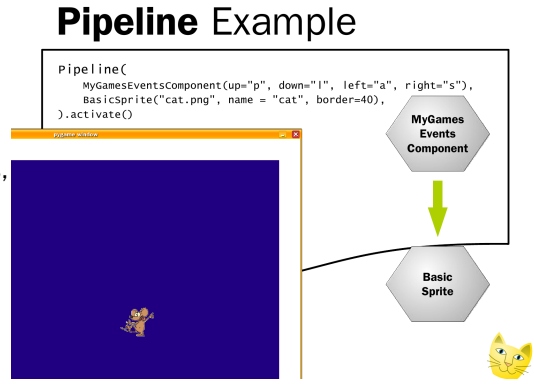
- Late binding, allowing dynamic rewiring
- The wrapper/trace components insertable into systems without changing code
- System introspection
- Clarity about what methods you can call on an object. (__init__(), .activate() & .run())

This also means that you naturally start creating higher level components that join things together, forming a little language of joining things.

Some of the earliest were things like Pipeline & Graphline, but it becomes natural to build others, as the diagram to the left shows. This DSL which is still be codified, then effectively allow you to start building new systems in a declarative manner, and also makes the application structure significantly clearer, as we'll see. (esp. with systems you've not seen before!)

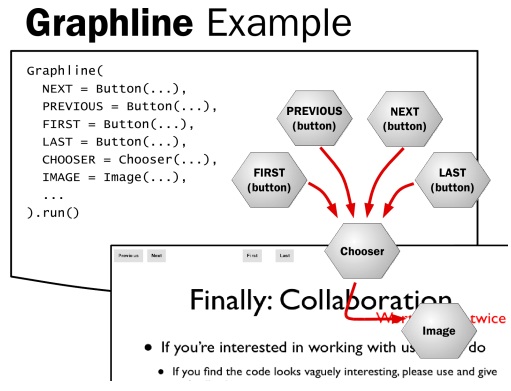
Some of these are **Chassis** components – they need to have other components plugged into it to be useful. Let's look at a few.

Pipeline performs a well known pattern. In this example, a component converts keyboard presses, and into messages regarding direction. These are piped to the second component acts upon them moving the sprite on screen. Pipeline naturally pipes messages from the outboxes of one component to the inboxes of the next.



Graphlines form another pattern. This example has a "Chooser", which knows about image names. When sent a message from one of the buttons, it emits an filename to the Image component which displays it. This creates a simple presentation tool.

In this case, the Graphline is passed another parameter – linkages – explicitly linking the outputs from components to each other. (This example in is the Examples/ directory).



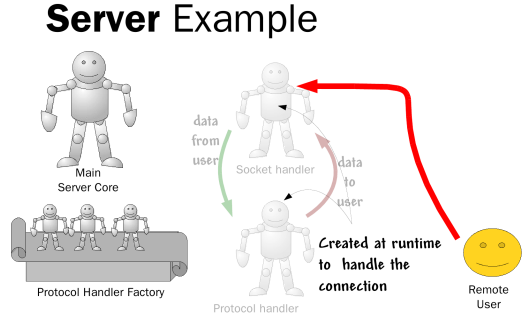
ServerCore is a slightly more complex chassis. Rather than plug one component into it, you provide it with a factory method which will create components to talk to the user.

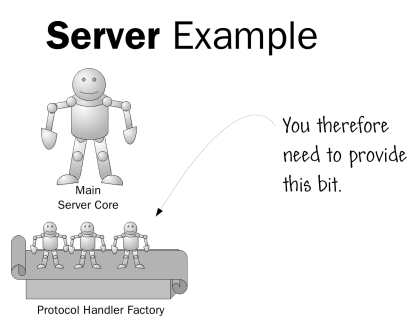
Fundamentally, it handles all the boring parts of a server, leaving you to handle the "chatting to the user" part of the problem.

As a result a protocol handler component receives bytes from the user on its main inbox, and sends bytes to the user on its main outbox. (Much like getting & sending data from/to stdin & stdout)

This allows you to build servers of a variety of different kinds, from echo servers, through email, web & even embedded python consoles.

We'll see a variety of different servers as we go through this tutorial.





Server Example

```
from Kamaelia.Chassis.ConnectedServer import ServerCore
from Kamaelia.Util.PureTransformer import PureTransformer

def greeter(*argv, **argd):
    return PureTransformer(lambda x: "hello" +x)

class GreeterServer(ServerCore):
    protocol=greeter
    port=1601

GreeterServer().run()
```

This example server waits for connections on port 1601. When a user connects, GreeterServer calls “greeter” to create a component to handle the connection.

The component created here is a **PureTransformer** component that takes the message, transforms it some way, and sends the transformed version back to the user. In this case the transformation is to prepend “hello” (PureTransformer makes no sense without the supplied transform)

Backplane Example

```
# Streaming Server for raw DVB of Radio 1
Backplane("Radio").activate()

Pipeline(
   DVB_Multiplex(850.16, [6210], feparams), # RADIO ONE
   PublishTo("RADIO"),
).activate()

def radio(*argv,**argd):
     return SubscribeTo("RADIO")

ServerCore(protocol=radio, port=1600).run()
```

This example is substantially more interesting, and uses both **ServerCore** and another form of component – **Backplanes**. Backplane provides a service – named “Radio” here – which can be used by other components.

Specifically, **PublishTo** is a component that takes messages from its main inbox, and publishes them to the named backplane.

SubscribeTo plugs into named backplane's output, sending messages it receives to its main outbox. These three component – Backplane, PublishTo, SubscribeTo provide a Kamaelia system an internal broadcast system.

As a result, this example does this...

- Creates a **Backplane** service called Radio
- Creates a component **DVB_Multiplex** that tunes into Radio 1, pipes that into a **PublishTo** component that publishes that to the Radio backplane.
- Runs a **ServerCore** that waits for users to connect on port 1600

Then, when a user connects to port 1600...

- **ServerCore** calls the function 'radio' – the supplied protocol factory. This returns a **SubscribeTo** component, which takes a copy of all radio data sent to the backplane Radio, and sends it out its main outbox.
- This results in the user being sent back the DVB-T stream for Radio 1.

Summary

Kamaelia's core goal is to make it easier to build concurrent systems. It uses a component model, that encourages the user to link together basic building blocks into more complex systems explicitly. A side effect of this is that systems can be much clearer & simpler as a result, with the fact we're using concurrency being – by the by – it can simplify the code – allowing us to focus on the higher level problems.

2. Building your own Kamaelia Core

One of the more common issues that I hear when experienced developers look at a Kamaelia system – for example regarding the Radio 1 re-streamer – is that “it can't be that simple”. However, this is often due to questions effectively relating to “how does that work under the hood”.

As a result, in order to explain this, we have a tutorial designed to teach the underlying principles in Kamaelia, based on the old saying: *I hear, and I forget. I see, and I remember. I do, and I understand*. By building your own core, and seeing what actually goes into it – even simplified slightly – demystifies many of the core aspects of Kamaelia.

This part of the tutorial covers this, and is gently paced – since it was originally written for a novice developer. However the core of the idea is that you will hopefully understand the system better if you understand what's going on under the hood.

Python pre-requisites:

- classes, methods, functions, if/elif/else, while, try..except, for..in.., generators (yield), lists, dictionaries, tuples.

Generators have a brief recap at the end of this booklet.

2.1 Microprocesses - A Generator with Context

Axon is built on top of generators with some added context. Whilst the most common version of this a user actually uses is called a component, this is a specialisation of the general concept - a generator with context.

Exercise: Given this outline class, fill in the described functionality.

```
class microprocess(object):
    def __init__(self):
```
Takes no arguments. (aside from self)

into this put any initialisation you might need (hint: call the `super`-class's __init__)
```
    def main(self):
```
This should be a generator that simply yields 1 value - specifically a 1

Discussion:

Clearly we can create 5 of these now:

```
a = microprocess()
b = microprocess()
c = microprocess()
```

Calling their main method results in us being given a generator:

```
>>> a.main()
<generator object at 0x40396d2c>
>>> b.main()
<generator object at 0x40396ccc>
```

```
>>> c.main()
<generator object at 0x40396d2c>
```

We can then run these generators in the usual way:

```
>>> for i in a.main():
...     print "Value", I
...
Value 1
>>> for i in b.main():
...     print "Value", I
...
Value 1
>>> for i in c.main():
...     print "Value", I
...
Value 1
```

Since these generators have access to an object – `self` - we have a mechanism for adding context to generators, and we've called that a microprocess. The next step is to enable us to set lots of these running.

2.2 Scheduler - A means of running lots of microprocesses

The purpose of this exercise is to enable you to be able to allow any number of microprocesses you've just created to be run at once. This exercise is essentially intended to demystify the scheduler. Its internals boil down to “run this, then this, then this, then this”.

Exercise: Again, given this outline class, fill in the described functionality.

`class scheduler(microprocess):`

`def __init__(self):`

Initialise the scheduler by setting up the runqueue (`active`) and `newqueue` lists after calling the `super`-class's `__init__` method.

Remember to call your `super`-class's `__init__` method

Then add your attributes to the object

`active` – should be a list. (initially empty)

`newqueue` - should also be a list. (initially empty)

`def main(self):`

This loops through all the microprocesses and runs each of them, 100 times.

The main body of this consists of a loop – make it stop after 100 iterations.

Inside this loop...

Using `current` as a loop value, have another loop, that loops through the contents of `self.active`, and ...

yield control here immediately a value that is not -1

try calling current.next()

If a `StopIteration` is raised, silence it, and move onto the next iteration of `current`. (ie allow a microprocess to stop if it just “stops”)

Otherwise, as long as the value returned was not -1, append `current` onto the `self.newqueue` list. (ie allow a microprocess to stop if it "stops" cleanly)

After looping through `self.active`, replace it with the contents of `self.newqueue`, and replace `self.newqueue` with a new, empty list.

`def activateMicroprocess(self, someprocess):`

This adds new microprocesses to the runqueue for the scheduler.

Assume that someprocess is a microprocess. Call its `main()` method returning a generator, and append this generator to the end of self.newqueue

Discussion:

This class provides us with a rudimentary way of activating generators embedded inside a class, adding them to a runqueue and then letting something run them. So let's try it. The default microprocess is relatively boring, so let's create some microprocesses that are little more than an age old program that repeatedly displays a messae. To do that we declare a class subclassing microprocess and provide a generator called main.

We'll also capture a provided argument:

```
class printer(microprocess):
    def __init__(self, tag):
        super(printer, self).__init__()
        self.tag = tag
    def main(self):
        while 1:
            yield 1 # Must be a generator
            print self.tag
```

Note that this generator doesn't ever exit. We can then create a couple of these printers:

```
X = printer("Hello World")
Y = printer("Game Over") # Another well known 2 word phrase :-)
```

Next we can create a scheduler:

```
myscheduler = scheduler()
```

We can then ask this scheduler to activate the two microprocesses - X & Y :

```
myscheduler.activateMicroprocess(X)
myscheduler.activateMicroprocess(Y)
```

We can then run our scheduler by iterating through its main method:

```
for _ in myscheduler.main():
    pass
```

If we run this we get the following output (middle of output snipped):

```
>>> for _ in myscheduler.main():
...     pass
...
Hello World
Game Over
Hello World
Game Over
...
```

```
Hello World
Game Over
>>>
```

As you can see, the scheduler hits the 100th iteration and then halts.

2.3 Interlude

So far we've created a mechanism for giving a generator some implicit context by embedding it inside a microprocess class. We've also created a simple microprocess that repeatedly displays the same message over and over again. We've also created a simple mechanism for setting lots of microprocesses running and watching them just go.

This is all well and good and core aspects of Axon. However another core aspect is enabling these generators to talk to each other. Doing this means we can divide responsibility for a task between file reading, and display. The metaphor we choose to use in Axon is a very old one - that of a worker at a desk with a number of inboxes and a number of outboxes. The worker receives messages on his/her inboxes. He/She does some work, and send results on his/her outboxes. We can then have something that takes messages from an outbox (called saying "finance") and delivers them to the inbox of somewhere else (say the inbox "in" on the finance desk/component).

An alternate analogy we don't take here is one of computer chips with pins and wires. Signals would get sent to pins transmitted along the wires (links) to other pins on other chips. A more software oriented alternative is unix pipelines and standard file descriptors. A unix command line program always* has access to stdin, which it reads but has no idea of the source; stdout it can write to, but has no idea of destination (and stderr). Obviously however unix command line programs don't know if they're in a pipeline, or standalone.

The key point we have is active objects talking only to local interfaces, and not knowing how those local interfaces are used.

So the next step is to first create this standard interface for external communications, and then a mechanism for allowing communication between these interface.

2.4 Simple Component - Microprocesses with standard external interfaces

Exercise: Again, given this outline class, fill in the described functionality.

```
class component(microprocess):
    def __init__(self):
```

As well as calling the `super`-class's `__init__` method, this method sets up a collection of inboxes and outboxes – these are named, and are represented using lists.

Call the `super`-class `__init__` method
Create a dictionary – `self.boxes` – with 2 keys - “inbox” and “outbox”.
The values for both should be empty lists. You could create more inboxes and outboxes this way, but lets keep things simple initially.

`def send(self,value, outboxname):`

This is for putting values into outboxes – for them to be later picked up and passed on elsewhere.

Find the list called `outboxname` in `self.boxes` and append `value` to it.

`def recv(self,inboxname):`

This is the logical opposite of send. We take values from inboxes and return it.

Find the list called `inboxname` in `self.boxes`. Pop the first value from it, and return that to the user. This could throw an exception. If it does, let the user deal with it.

`def dataReady(self, inboxname):`

This returns a true value if there is any data in the given inbox, and if so, how many values are waiting.

Having found the list `inboxname` in `self.boxes`, simply return its length.

Discussion:

Ok that's a fairly long description, but a fairly simple implementation. So what's this done? It's enabled us to send data to a running generator and recieve data back. We're not worried what the generator is doing at any point in time, and so the communications between us and the generator (or between generators) is asynchronous.

An extension to the suggested __init__ is to do the following:

```
class component(microprocess):
    Boxes = {
        "inbox" : "This is where we expect to receive messages",
        "outbox" : "This is where we send results/messages"
    }
    def __init__(self):
        super(simplecomponent, self).__init__()
        self.boxes = {}
        for box in self.Boxes:
            self.boxes[box] = list()
```

This small extension means that classes subclassing component can have a different set of inboxes and outboxes. For example:

```
class spinnyThing(component):
    Boxes = {
        "inbox" : "As per default",
        "outbox" : "As per defaults",
        "rotation" : "Expect an integer between 0 and 359 (degrees)",
    }
```

That said, components by themselves are relatively boring. Unless we have some way of moving the data between generators we haven't gained anything (really) beyond the printer example above. So we need someone/something that can move data/messages from outboxes and deliver to inboxes...

2.5 Postman - A Microprocess that performs deliveries!

Given we have outboxes and inboxes, it makes sense to have something that can handle deliveries between the two. For the purpose of this exercise, we'll create a microprocess that can look at a single outbox for a single component, take any messages deposited there and pass them the an inbox of another component. In terms of the component implementation so far we can use dataReady to check for availability of messages, recv to collect the message from the outbox, and send to deliver the message to the recipient inbox.

Exercise: Again, given this outline class, fill in the described functionality.

```
class postman(microprocess):
    def __init__(self, source, sourcebox, sink, sinkbox):
```

Store copies of all the arguments as object attributes, after remembering to call the `super`-class __init__ method.

```
    def main(self):
```

Loop for ever, and...

When `data` (is) `Ready` at the `source` component's `sourcebox`, `recv` the latest value waiting, and `send` it onto the `sink` components `sinkbox`.

Discussion:

Given this, we can now start building interesting systems. We have mechanisms for enabling concurrency in a single process (microprocess & scheduler), a mechanism for adding communications (postboxes) to a microprocess (component) and a mechanism for enabling deliveries between components. Whilst we (the Kamaelia team) can see from an optimised version that the postman can actually be optimised out of the system, this simple mini-axon shows the core elements of Kamaelia quite nearly in a microcosm.

One full version of this mini-axon can be found here: Mini Axon Full, which should now be clear what it's doing how and why.

A simple example we can now create is a trivial system with one component creating some data and sending it to another one for display.

```
class Producer(component):
    def __init__(self, message):
        self.message = message
    def main(self):
        while 1:
            yield 1
            self.send(self.message, "outbox")

class Consumer(component):
    def main(self):
        count = 0
        while 1:
            yield 1
            count += 1 # This is to show our data is changing :-)
```

```
            if self.dataReady("inbox"):
                data = self.recv("inbox")
                print data, count

p = Producer("Hello World")
c = Consumer()
postie = postman(p, "outbox", c, "inbox")

myscheduler = scheduler()
myscheduler.activateMicroprocess(p)
myscheduler.activateMicroprocess(c)
myscheduler.activateMicroprocess(postie)

for _ in myscheduler.main():
    pass
```

Running the above system then results in the following output:

```
Hello World 2
Hello World 3
...
Hello World 97
Hello World 98
```

2.6 Interlude 2

If you've come this far, you may be wondering the worth of what you've acheived. Essentially you've managed to implement the core of a working Axon system, specifically on the most used aspects of the system. Sure, there is some syntactic sugar relating to creation and managing of links, but that's what it is - sugar.

One of the longer examples on the Kamaelia website, specifically in the blog area, is how to build new components. That's probably the next logical place to start looking. However, taking one of the components on that page, we find that the core implementation of them matches the same core API as the component system you've implemented.

For example, let's take a look at the multicast sender.

```
class Multicast_sender(component):
```

This grabs some initial values, and calls the super class's initialiser:

```
   def __init__(self, local_addr, local_port, remote_addr, remote_port):
       super(Multicast_sender, self).__init__()
       self.local_addr = local_addr
       self.local_port = local_port
       self.remote_addr = remote_addr
       self.remote_port = remote_port
```

The main function/generator then just- sets up the socket, waits for data and sends it out:

```
def main(self):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM,
                         socket.IPPROTO_UDP)
    sock.bind((self.local_addr,self.local_port))
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 10)
```

```
            while 1:
               if self.dataReady("inbox"):
                  data = self.recv()
                  l = sock.sendto(data, (self.remote_addr,self.remote_port) );
               yield 1
```

From this, it should be clear that this will work inside the mini-axon system you've created.

Similarly, we can create a simple file reading component thus:

```
class FileReader(component):
    def __init__(self, filename):
        super(ReadFileAdapter, self).__init__()
        self.file = open(filename, "rb",0)
    def main(self):
        yield 1
        for line in self.file.xreadlines():
            self.send(line, "outbox")
            yield 1
```

This can then also be used using the component system you've just created to build a simplistic system for sending data to a multicast group:

```
reader = FileReader("fortune.txt")
sender = Multicast_sender("0.0.0.0", 0, "224.168.2.9", 1600)
postie = Postman(reader, "outbox", sender, "inbox")
```

That can then be activated and run in the usual way:

```
myscheduler = scheduler()
myscheduler.activateMicroprocess(reader)
myscheduler.activateMicroprocess(sender)
myscheduler.activateMicroprocess(postie)
for _ in myscheduler.main():
 pass
```

2.7 Summary

This page has hopefully helped you build a core component system based on Kamaelia's Axon. It should be clear as well from this that the core of Kamaelia is actually quite small. We've found a number of aspects which we can optimise, add in syntactic sugar, and we're discovering that certain facilities are needed, and can be useful. However the raw core is simple - it's about generators communicating with inboxes and outboxes, and then building interesting systems on top of that.

The next step we'd normally recommend at this point is to build some interesting systems. Some exercises which will hopefully be helpful will appear as time progresses.

The next step we'd normally recommend at this point is to build some interesting systems. Some exercises which will hopefully be helpful will appear as time progresses.

2.8 Mini-Axon - Full Source

```
class microprocess(object):
    def __init__(self):
        super(microprocess, self).__init__()
    def main(self):
        yield 1

class scheduler(microprocess):
    def __init__(self):
        super(scheduler, self).__init__()
        self.active = []
        self.newqueue = []
    def main(self):
        for i in xrange(100):
            for current in self.active:
                yield 1
                try:
                    result = current.next()
                    if result is not -1:
                        self.newqueue.append(current)
                except StopIteration:
                    pass
            self.active = self.newqueue
            self.newqueue = []
    def activateMicroprocess(self, someprocess):
        microthread = someprocess.main()
        self.newqueue.append(microthread)

class component(microprocess):
    Boxes = {
        "inbox" : "This is where we expect to receive messages for work",
        "outbox" : "This is where we expect to send results/messages to after doing work"
    }
    def __init__(self):
        super(component, self).__init__()
        self.boxes = {}
        for box in self.Boxes:
            self.boxes[box] = list()
    def send(self, value, outboxname):
        self.boxes[outboxname].append(value)
    def recv(self, inboxname):
        result = self.boxes[inboxname][0]
        del self.boxes[inboxname][0]
        return result
    def dataReady(self, inboxname):
        return len(self.boxes[inboxname])
```

```
class postman(microprocess):
    def __init__(self, source, sourcebox, sink, sinkbox):
        self.source = source
        self.sourcebox = sourcebox
        self.sink = sink
        self.sinkbox = sinkbox
    def main(self):
        while 1:
            yield 1
            if self.source.dataReady(self.sourcebox):
                d = self.source.recv(self.sourcebox)
                self.sink.send(d, self.sinkbox)
#
# Some sample code using the mini component
# system defined above
#
class Producer(component):
    def __init__(self, message):
        super(Producer, self).__init__()
        self.message = message
    def main(self):
        while 1:
            yield 1
            self.send(self.message, "outbox")

class Consumer(component):
    def main(self):
        count = 0
        while 1:
            yield 1
            count += 1 # This is to show our data is changing :-)
            if self.dataReady("inbox"):
                data = self.recv("inbox")
                print data, count

p = Producer("Hello World")
c = Consumer()
postie = postman(p, "outbox", c, "inbox")

myscheduler = scheduler()
myscheduler.activateMicroprocess(p)
myscheduler.activateMicroprocess(c)
myscheduler.activateMicroprocess(postie)

for _ in myscheduler.main():
    pass
```

3. Axon & Kamaelia – Deep Dive

Having looked at how you build your own mini-Axon, which gives an idea of what's going on underneath the hood, let's review Axon proper. As you might expect there are a number of extra facilities and structural differences in practice in order to handle things like threads, optimisations, efficient message passing, not hogging CPU, advertising services inside an application, and so on, but fundamentally the basics remain the same.

As with any system, the internals of Axon are likely to continue to evolve and improve with time, with a focus on simplicity, flexibility, and performance. However, components created with Axon today will continue to work with Axon tomorrow.

This section will not seek to replace Axon's documentation which can be found here:

- http://www.kamaelia.org/Docs/Axon/Axon

But is intended to serve as a guide to the more important parts of Axon from the perspective of a developer using Kamaelia. We'll cover components, and container components (which form systems), and work out from there.

Nor is it intended to resplace Kamaelia's component documentation, which can be found here:

- http://www.kamaelia.org/Components

But this section intended to give you signposts to some of the components which are commonly used in many Kamaelia systems and applications. After doing this, it should be noted that one of the best ways to learn how to write Kamaelia systems and components is to examine existing components to see how they work. As a result, this section also does deep dives through a handful of components as a guide.

Also note, the components in the main repository, except those in the Kamaelia.Apps namespace, tend to contain large amounts of documentation about how they work, why they work a certain way, and so forth.

The Kamaelia Cookbook is a growing collection of examples, of small systems and growing tutorials as to “what next”. That can be found here:

- http://www.kamaelia.org/Cookbook

3.1 Components

As has been mentioned, components are the core of any Kamaelia system.

A rather extreme form of component definition can look something like this:

```
class TaggedTracer(Axon.Component.component):
    Inboxes = {
        "inbox" : "Data to display",
        "control" : "Any message sent here causes this component to shutdown",
    }
    Outboxes = {
        "outbox" : "Data is passed here unchanged",
        "signal" : "Messages passed to our control inbox are forwarded here",
    }
    def __init__(self, tag="> ", **argd):
        super(TaggedTracer, self).__init__(**argd)
        self.tag = tag

    def main(self):
        while not self.dataReady("control"):
            while self.dataReady("inbox"):
                msg = self.recv("inbox")
                print self.tag, str(msg)
                self.send(msg, "outbox")
            if not self.anyReady():
                self.pause()
            yield 1
        self.send(self.recv("control"), "signal")
```

This is rather over the top for a simple component, and there is a simpler form, which we'll come back to.

However, this explicit form is useful for explaining what's happening here. There are clearly some things here which are:

- Definition
- Configuration
- Code - the main loop, with:
    - Handling of control messages
    - Handling of data messages

Let's look at these, and then delve inside, and see how this maps to Mini-Axon. Then we'll take a look at some syntactic sugar that simplifies this.

After that we'll look at a container component, and how linkages are made enabling data to slow between components, and why container components are useful.

3.1.1 Component Definition

These parts define the inboxes and outboxes the component expects to exist:

```
class TaggedTracer(Axon.Component.component):
    Inboxes = {
        "inbox" : "Data to display",
        "control" : "Any message sent here causes this component to shutdown",
    }
    Outboxes = {
        "outbox" : "Data is passed here unchanged",
        "signal" : "Messages passed to our control inbox are forwarded here",
    }
```

The actual inboxes and outboxes are created by `Axon.Component.component.__init__` . Since this `TaggedTracer` component has an `__init__` method, the way it calls `component`'s `__init__` method is as follows:

```
super(TaggedTracer, self).__init__(**argd)
```

Thus, simply defining our interface declaratively ensures our component has the right interface for the problem in hand.

3.1.2 Simple Component Configuration

Like many python classes, this component takes some arguments to its `__init__` initialiser and uses those directly to configure the component.

```
class TaggedTracer(Axon.Component.component):
    ...
    def __init__(self, tag="> ", **argd):
        super(TaggedTracer, self).__init__(**argd)
        self.tag = tag
```

In this case it sets an attribute `self.tag`. This is pretty normal stuff, with a call like `TaggerTracer(tag="Checking this")` setting `self.tag` to `"Checking this"`.

With Kamaelia components, in the initialiser `__init__` it is extremely unusual to do anything other than copy these values into attributes in the object. As a result there is a more compact form for handling this common pattern, which we will come back to when we look at some syntactic sugar. However, like all syntactic sugar, its use is optional and depends on what you want to do.

3.1.3 Component Runtime

In order to run the component, the component has 2 special methods - `.activate()` and `.run()`. They perform the following two tasks.

- `.activate()` activates the component and sets it running, if the scheduler is running.

  Internally, with generator components it calls the main() method, and stores a copy of that generator, allowing the scheduler to give it CPU time. The component is then added to the schedulers run queue. If the scheduler is running, then the component will start running, otherwise It will wait for the scheduler to start up.

- .run() also activates the component, but also runs the scheduler.

As a result, from a user perspective, `.activate()` is a call that returns immediately, whereas `.run()` only exits when the scheduler finishes. It is therefore common to create a collection of components and call `.activate()` on them and finish by calling `.run()` on the last one.

You can see this behaviour in the Radio 1 example:

```
Backplane("Radio").activate()

Pipeline(
    DVB_Multipex(850.16, [6210], feparams), # RADIO ONE
    PublishTo("Radio"),
).activate()

def radio(*argv, **argd):
    return SubscribeTo("Radio")

ServerCore(protocol=radio, port=1600).run()
```

From then on, the logic of what's happening with the inboxes, outboxes, and so in is very explicit:

```
def main(self):

        while not self.dataReady("control"):

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

            if not self.anyReady():
                self.pause()
            yield 1

        self.send(self.recv("control"), "signal")
```

Specifically, this loops until any single message is received on the component's `control` inbox that message is forwarded to the component's `signal` outbox before exiting.

Inside the loop, 3 things happen:

- The system performs some actions based on messages arriving on the main inbox `inbox`
- Then if there's no messages waiting on any inbox, it sets a pause flag, by calling `self.pause()`
- It then releases control back to the scheduler using `yield 1`

The actions on the main inbox are:

- While there are messages to be processed, grab them one at a time
- print them, preceded by the tag the user gave us
- pass the message on to the next component.

This is all made very explicit in this notation. Before we come back to the slightly more compact way of representing this, let's look inside this component and see what's going on.

3.1.4 Delving Inside Components

Recalling this definition, what actually happens?

```
class TaggedTracer(Axon.Component.component):
    Inboxes = {
        "inbox" : "Data to display",
        "control" : "Any message sent here causes this component to shutdown",
    }
    Outboxes = {
        "outbox" : "Data is passed here unchanged",
        "signal" : "Messages passed to our control inbox are forwarded here",
    }
```

During initialisation, we ensure that Axon.Component.component.__init__ is called. Inside there, it uses these definitions of Inboxes and Outboxes to instantiate the actual inboxes and outboxes which are used for communications. How this happens actually depends on the component type.

Inside normal, generator components, this happens:

```
from Box import makeInbox,makeOutbox
...
class component(microprocess):
...
   def __init__(self, *args, **argd):
      super(component, self).__init__()
      self.__dict__.update(argd)
       self.inboxes = dict()
      self.outboxes = dict()

      for boxname in self.Inboxes:
          self.inboxes[boxname] = makeInbox(notify=self.unpause)
      for boxname in self.Outboxes:
          self.outboxes[boxname] = makeOutbox(notify=self.unpause)
```

Looking at this, it clearly creates the actual inboxes & outboxes to based on the user's definition.

By using a dict we gain the opportunity to document the usage on the inboxes & outboxes in a useful location. In fact, it actually just iterates through the Inboxes and Outboxes definition, so you could just use a list if you like. We'll come back to "`Box`", but essentially that's implemented as a list - in the same way as mini-axon.

Digging into `Axon.Component`, you'll find that `self.recv` & `self.send` are implemented as follows:

```
def send(self,message, boxname="outbox"):
      self.outboxes[boxname].append(message)

   def recv(self,boxname="inbox"):
      return self.inboxes[boxname].pop(0)
```

Again, exactly as with Mini-Axon. Note that `main()` is a generator.

If this had been a threaded component, what's essentially different? Well, if we look at how the initialiser for threaded component:

```
import Queue
...
class threadedcomponent(Component.component):
...
   def __init__(self,queuelengths=DefaultQueueSize, **argd):
      super(threadedcomponent,self).__init__(**argd)
      ...
      self.queuelengths = queuelengths
      self.inqueues = dict()
      self.outqueues = dict()
      for box in self.inboxes.iterkeys():
         self.inqueues[box] = Queue.Queue(self.queuelengths)
      for box in self.outboxes.iterkeys():
         self.outqueues[box] = Queue.Queue(self.queuelengths)
```

We can see that `Axon.Component.component.__init__` is still called. This means that `threadedcomponents` still have the same inboxes and outboxes as generator components. This enables them to talk to generator components cleanly. However the set of inboxes & outboxes is also used to create a collection of `inqueues` and `outqueues`.

The reason for this is because `threadedcomponent` is essentially a wrapper around a thread, in the same way `component` is a wrapper around generator.

This means `self.recv` & `self.send` – need to work differently, due to threading issues:

```
def recv(self,boxname="inbox"):
    Component.component.unpause(self)
    return self.inqueues[boxname].get()

def send(self,message, boxname="outbox"):
    try:
        self.outqueues[boxname].put_nowait(message)
        Component.component.unpause(self)
    except Queue.Full:
        raise noSpaceInBox(self.outqueues[boxname].qsize(), self.queuelengths)
```

That said, ignoring the optimisation aspects (pausing), essentially this boils down to the same logic as before, except using threadsafe `Queues` rather than lists (the movement of data between the in/out-boxes and in/out-queues is handled by the generator part of the component, to maintain safety).

3.2 Syntactic Sugar For Components

Revisiting configuration – Inheritable Defaults

Buried above is this line:

```
class component(microprocess):
...
   def __init__(self, *args, **argd):
      super(component, self).__init__()
      self.__dict__.update(argd)
```

Let's examine what this does, on the console, with a slightly simplified version:

```
>>> class component(object):
...     def __init__(self, **argd):
...         self.__dict__.update(argd)
...
>>> X=component(greeting="hello world", location="screen")
>>> X.greeting
'hello world'
>>> X.location
'screen'
```

Clearly this enables automation of this sort of code:

```
class component(object):
    def __init__(self, greeting=None, location=None):
        self.greeting = greeting
        self.location = location
```

Similarly, when python looks up the values for `self.greeting` & `self.location`, it also looks in the class, as well as the object. So we could also do this:

```
>>> class mycomponent(object):
...     greeting = "Game Over"
...     location = "Games console"
...
>>> T=mycomponent()
>>> T.greeting
'Game Over'
>>> T.location
'Games console'
```

By combining these two facts, we gain something rather special:

```
>>> class mycomponent(component):
...     greeting = "Game Over"
...     location = "Games console"
...
>>> X=mycomponent()
>>> X.greeting, X.location
('Game Over', 'Games console')
>>> X=mycomponent(greeting="Hello World", location="display")
>>> X.greeting, X.location
('Hello World', 'display')
>>> class new_mycomponent(mycomponent):
...     location = "debug"
...
>>> X=new_mycomponent()
>>> X.greeting, X.location
('Game Over', 'debug')
```

Specifically, we gain a way to inherit the default values of the parent class, as well as a way of populating attributes in the object.

As a result, since this construct...

```
class SomeComponent(component):
    def __init__(self,foo=None, bar=None, baz=None, Bla=None):
        super(SomeComponent, self).__init__()
        self.foo = foo
        self.bar = bar
        self.baz = baz
        self.bla = bla
```

... is so common, we tend to use this "inheritable default values" approach more and more in real world systems.

In the introduction for example, we saw this:

```
def greeter(*argv, **argd):
    return PureTransformer(lambda x: "hello" + x )

class GreeterServer(ServerCore):
    protocol = greeter
    port = 1601

GreeterServer().run()
```

This is explicitly making use of this approach. This could be also written like this:

```
class GreeterServer(ServerCore):
    protocol = greeter
    port = 1601
    def protocol(*argd, **argd):
        return PureTransformer(lambda x: "hello" + x )

GreeterServer().run()
```

Alternatively, this can be written as follows:

```
ServerCore(protocol = greeter, port=1601).run()
```

Default Inboxes & Outboxes

If a component class is defined without an explicit Inboxes and Outboxes definition, then the component is given the default set. Specifically inboxes: "inbox", "control" and outboxes "outbox", "signal"

Iterating Inboxes

Again, this code structure is relatively common:

```
while self.dataReady("someinbox"):
    msg = self.recv("someinbox")
    <do something>
```

As a result, this iteration pattern has the following syntactic sugar:

```
for msg in self.Inbox("someinbox"):
    <do something>
```

Bringing it all together

Given all this, we can rewrite the component under discussion as follows:

```
class TaggedTracer(Axon.Component.component):
    tag = "> "
    def main(self):
        while not self.dataReady("control"):
            for msg in self.Inbox("inbox"):
                print self.tag, str(msg)
                self.send(msg, "outbox")
            self.pause()
            yield 1

self.send(self.recv("control"), "signal")
```

At present this is the best practice form for many components.

This has the same behaviour as before - you can still type `TaggedTracer(tag="my tag")`, and `self.tag` inside is customised in that way. Unlike before however, if you use a particular form of tracer, you can do this instead:

```
class DebugTracer(TaggedTracer):
    tag = "debug> "

class WarnTracer(TaggedTracer):
    tag = "warn> "

class ErrorTracer(TaggedTracer):
    tag = "error> "
```

You could also use a factory function instead, but this form enables the user of your component to override it and reuse it in ways you may not expect.

Syntactic Sugar for Control Handling?

At present it's becoming rapidly apparent that this structure in itself is a common form:

```
class SomeComponentName(Axon.Component.component):
    def main(self):
        while not self.dataReady("control"):
             * generator body *
        self.send(self.recv("control"), "signal")
```

And it's tempting to consider writing a decorator that would be used like this...

```
@GeneratorComponent(Inboxes = ["inbox", "control"],
                    Outboxes = ["outbox", "signal"])
def Tracer(self):
    while 1:
        for msg in self.Inbox("inbox"):
            print self.tag, str(msg)
            self.send(msg, "outbox")
        self.pause()
        yield 1
```

... and to use PEP 342 facilities to shutdown this sort of generator component. Whether this makes sense is currently a matter under discussion, you're welcome to join in!

3.3 Building Components

This fact that generator components and threaded components have the same API provdes us with a way of making components relatively simply. On the Kamaelia website there's an example of how a `MulticastTransceiver` can be written. For this case, we'll look at how to build a component for working with the new `pygame.camera` API (due in 1.9 release).

In order to do this, let's review how to use `pygame.camera` API.

First we pull in the necessary imports, and initialisation.

```
import pygame
import pygame.camera

pygame.init()
pygame.camera.init()
```

We define some values we'll use

```
Displaysize = (800, 600)
capturesize = ( 640, 480 )
imagesize = (352,288)
imageorigin = (0,0)
device = "/dev/video0"
```

Initialise the display, allocate a camera and activate it.

```
display = pygame.display.set_mode( displaysize )
camera = X=pygame.camera.Camera(device, capturesize)
camera.start()
```

Then loop round capturing images, resizing them as necessary, blit to the display and flip.

```
while 1:
    snapshot = camera.get_image()
    snapshot = pygame.transform.scale(snapshot, imagesize)
    display.blit(snapshot, imageorigin)
    pygame.display.flip()
```

And when you run it, you get to see whatever the camera is pointed at:

So far so good. Before we componentise this, let's transform this into a normal python class which makes it easier to work with.

The key benefit this process gives you of course is reuse & customisation.

As before, necessary imports and initialisation

```
import pygame
import pygame.camera

pygame.init()
pygame.camera.init()
```

The class...

We can store our configuration in class attributes. These form defaults.

In the constructor, allow the user to override the defaults.

As before, initialise the display, allocate a camera and activate it.

```
class VideoCapturePlayer(object):

    displaysize = (800, 600)
    capturesize = ( 640, 480 )
    imagesize = (352,288)
    imageorigin = (0,0)
    device = "/dev/video0"

    def __init__(self, **argd):
        self.__dict__.update(**argd)
        super(VideoCapturePlayer, self).__init__(**argd)


        self.display = pygame.display.set_mode( self.displaysize )
        self.camera = X=pygame.camera.Camera(self.device,
                                             self.capturesize)
        self.camera.start()
```

Wrap up the body of the loop in a method, largely unchanged.

Then provide a method for the user to call

Finally, instantiate this and run it.

```
    def get_and_flip(self):
        snapshot = self.camera.get_image()
        snapshot = pygame.transform.scale(snapshot,self.imagesize)
        self.display.blit(snapshot, self.imageorigin)
        pygame.display.flip()

    def main(self):
        while 1:
            self.get_and_flip()

VideoCapturePlayer().main()
```

To use this in a Kamaelia system we could go through the following steps:

- Convert it to a threaded component
- We'd then want to seperate out capture from display

- This would then allow the WebCam code to be used with other pygame components, and allow us to use it as a source for other purposes – for example recording images (eg stop motion) or across the network.

Conversion to a threaded component is the first obvious step, so let's do that.

As before, necessary imports and initialisation. We've added in one import.

```
import pygame
import pygame.camera

pygame.init()
pygame.camera.init()
From Axon.ThreadedComponent import threadedcomponent
```

Change the baseclass

```
class VideoCapturePlayer(threadedcomponent):
```

Same config options

```
    displaysize = (800, 600)
    capturesize = ( 640, 480 )
    imagesize = (352,288)
    imageorigin = (0,0)
    device = "/dev/video0"
```

As before, still the user can still override defaults.

```
    def __init__(self, **argd):
        super(VideoCapturePlayer, self).__init__(**argd)
```

As before initialise the display, allocate camera and activate it.

```
        self.display = pygame.display.set_mode( self.displaysize )
        self.camera = pygame.camera.Camera("/dev/video0",
                                           self.capturesize)
        self.camera.start()
```

We've added this wrapper function.

```
    def pygame_display_flip(self):
        pygame.display.flip()
```

As before same logic for the body of the loop. Only change is to call our wrapper around `.flip()`

```
    def get_and_flip(self):
        snapshot = self.camera.get_image()
        snapshot = pygame.transform.scale(snapshot,self.imagesize)
        self.display.blit(snapshot, self.imageorigin)
        self.pygame_display_flip()
```

The main method is unchanged.

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

And finally run as before.

```
VideoCapturePlayer().main()
```

From this, it should be clear that making threaded components are relatively painless. Indeed, this sort of change is relatively common – at least initially. Specifically, get a version of the code you want to use running, and then componentise it and evolve it towards where you want it to go.

The next step is to seperate out capture form display.

First the resulting webcam capture component.

```
class VideoCaptureSource(Axon.ThreadedComponent.threadedcomponent):
    capturesize = ( 640, 480 )
    delay = 0
    fps = -1
    def __init__(self, **argd):
        super(VideoCaptureSource, self).__init__(**argd)
        self.camera = pygame.camera.Camera("/dev/video0", self.capturesize)
        self.camera.start()
        if self.fps != -1:
            self.delay = 1.0/self.fps # fps overrides delay

    def capture_one(self):
        snapshot = self.camera.get_image()
        return snapshot

    def main(self):
        while 1:
            self.send(self.capture_one(), "outbox")
            time.sleep(self.delay)
```

There's a handful of changes here:

- It doesn't touch the display, simply sending images out our main outbox.
- A delay between taking images has been added, since the rate of image capture is no longer limited by render speed.

It's worth noting that this component is now more useful, even though it's shorter.

On the flip side, separating out the display elements looks like this:

```
class SurfaceDisplayer(Axon.Component.component):
    displaysize = (800, 600)
    def __init__(self, **argd):
        super(SurfaceDisplayer, self).__init__(**argd)
        self.display = pygame.display.set_mode( self.displaysize )

    def pygame_display_flip(self):
        pygame.display.flip()

    def main(self):
        while 1:
            while self.dataReady("inbox"):
                snapshot = self.recv("inbox")
                self.display.blit(snapshot, (80,60))
                self.pygame_display_flip()
            while not self.anyReady():
                self.pause()
                yield 1
            yield 1
```

Usage of these two then looks like this:

```
from Kamaelia.Chassis.Pipeline import Pipeline
Pipeline(
    VideoCaptureSource(),
    SurfaceDisplayer()
).run()
```

Similarly this video capture component can also be used to save images to disk.

For example, if instead of the image displayer, we create a component for saving the images to disk? Such a component could look like this:

```
class LossyPicSaver(Axon.Component.component):
    base = "vid/"
    def main(self):
        base = self.base
        while 1:
            snapshot = None
            for snapshot in self.Inbox("inbox"):
                pass
            if snapshot:
                filename = base + str(self.scheduler.time)+".jpg"
                pygame.image.save(snapshot, filename)
            while not self.anyReady():
                self.pause()
                yield 1
            yield 1
```

Again this follows the same structure, expecting images on the main “inbox”, but has a couple of oddities:

- Picking the smaller first, it does this: `str(self.scheduler.time)`
  The reason for this is because the scheduler already has the time representing this timeslice, so we may as well just use it. (This is the reason this attribute exists)
- It also has the following construct:

  ```
  snapshot = None
  for snapshot in self.Inbox("inbox"):
      pass
  if snapshot:
  ```

  The reason for this is because saving images to disk obviously takes time. If we want to keep pace with the speed at which we're likely to receive them, we want to throw away images we're not going to have time to save (without incurring a memory leak). It's due to this, that I called this a lossy picture saver.

Usage of this is as follows:

```
Pipeline(
    VideoCapturePlayer(),
    LossyPicSaver(base="vid/"),
).run()
```

3.3.1 Katas

Rather than specific exercises here, 2 Kata's which are appropriate are this:

- Find example stubs for your favourite library, and wrapping them to form components, eg audio capture, Lego NXT components, etc
- Taking these components and connecting them together using pipeline or Graphline (For example an audio recorder, stop motion animation capture).

3.4 Components Core to Systems

Having covered components in some detail, it should be apparent that components can be relatively complex – such as a complete webcam viewing application. However by making simpler components we are able to recombine them in interesting and more flexible ways.

We've already seen that you can use Pipeline to combine components, but let's examine systems in more detail – both in higher level terms and all the way down to the bottom

The following higher level mechanisms are most useful today for building systems:

Kamaelia.Chassis:

- **Pipeline** – causes data & signals to flow from one component to the next in the pipeline.
- **Graphline** – causes data & signals to flow in a user customised form (more verbose, more flexible)
- **Seq** – Looks like a pipeline, but slightly different. It activates each subcomponent one after another, replumbing the external in/out-boxes to the newly active subcomponent as they go. We'll come back to how this is useful.
- **PAR** – this component runs all the subcomponents in parallel, combining their output into one destination.
- **ServerCore** – this takes a factory method for new

Kamaelia.Util:

- **PureTransformer** – whilst this component doesn't contain a component, it is generally only useful when supplied with a transformation function. As a result, it can be viewed as a function chassis, as opposed to a component chassis.
- **Backplane, SubscribeTo, PublishTo –** these components used together provide a mechanism for creating named services that can be easily plumbed into the rest of the system.

3.5 Pipelines

Pipelines are amongst the most widely used component that we've seen so far, so let's examine them in slightly more detail.

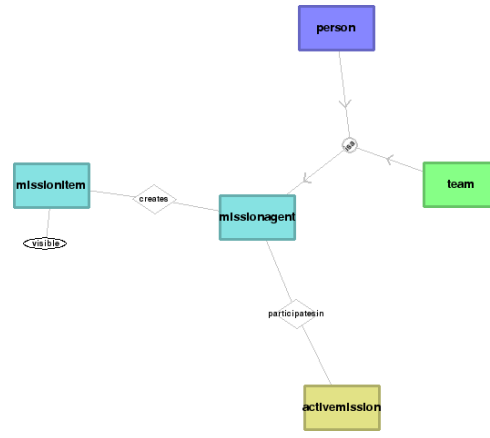
The Kamaelia ER Modeller is a useful example here. It was a tool written in an afternoon, including the components, created for the purpose making it easy to play with a textual database definition and to automatically lay out what it looks like. The purpose behind this was to enable “playing” with the a particular data model at a high level, and allowing Kamaelia to handle the layout.

This was quick and simple to do because Kamaelia provides a topology visualisation tool, which creates a dynamic layout based on a simple physics model.

Specifically, it allows you to type this:

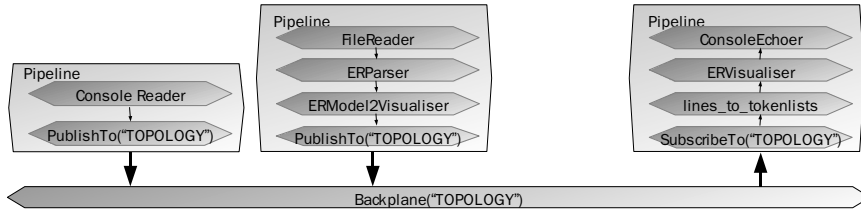
```
entity missionagent
entity person(missionagent)
entity team(missionagent)

entity missionitem:
     simpleattributes visible

entity activemission

relation
participatesin(activemission,missionagent)
relation creates(missionagent,missionitem)
```

And Kamaelia renders this:



The **full** code for the main file – Modeller.py & diagram – is this

```
import sys
from Kamaelia.Util.Backplane import *
from Kamaelia.Util.Console import *
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Visualisation.PhysicsGraph.TopologyViewer import TopologyViewer
from Kamaelia.Visualisation.PhysicsGraph.lines_to_tokenlists import lines_to_tokenlists
from Kamaelia.File.ReadFileAdaptor import ReadFileAdaptor
from Kamaelia.Visualisation.ER.ERVisualiserServer import ERVisualiser
from Kamaelia.Experimental.ERParsing import ERParser,ERModel2Visualiser

Backplane("TOPOLOGY").activate()

Pipeline(
    ConsoleReader(">>> "),
    PublishTo("TOPOLOGY"),
).activate()

if len(sys.argv)> 1:
    Pipeline(
        ReadFileAdaptor(sys.argv[1]),
        ERParser(),
        ERModel2Visualiser(),
        PublishTo("TOPOLOGY"),
    ).activate()

Pipeline(
    SubscribeTo("TOPOLOGY"),
    lines_to_tokenlists(),
    ERVisualiser(screensize = (1024,768), fullscreen = True),
    ConsoleEchoer(),
).run()
```



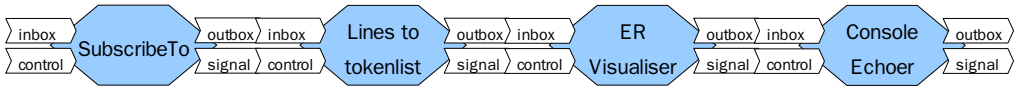
3.5.1 Deep dive inside pipelines, linkages, sub components

I'm not going to go through the ER modeller code in detail, but rather focus on one part of it, specifically this pipeline:

```
Pipeline(
    SubscribeTo("TOPOLOGY"),
    lines_to_tokenlists(),
    ERVisualiser(screensize = (1024,768), fullscreen = True),
    ConsoleEchoer(),
).run()
```

Internal Links

Now in order for this to work, the Pipeline component links the outputs of these components to each other:



As you can see Pipeline creates linkages between components inside it – these collapse the outboxes into inboxes, ensuring direct delivery of messages from on component to the next.

The code that does this looks like this:

```
   def __init__(self, *components, **argv):
....
      self.components = list(components)
...
   def main(self):
...
      Pipeline = self.components[:]
      source = Pipeline[0]
      del Pipeline[0]
      while len(Pipeline)>0:
         dest = Pipeline[0]
         del Pipeline[0]
         self.link((source,"outbox"), (dest,"inbox"))
         self.link((source,"signal"), (dest,"control"))
         source = dest
```

That's a little obfuscated to make this more general, so lets give the components some real names, and trace out the `.link()` calls made:

```
SUB     =  SubscribeTo("TOPOLOGY"),
LINES   = lines_to_tokenlists(),
VIS     = ERVisualiser(screensize = (1024,768), fullscreen = True),
CONSOLE = ConsoleEchoer(),
```

The first set of .link calls made, therefore look like this:

```
self.link((SUB,"outbox"), (LINES,"inbox"))
self.link((SUB,"signal"), (LINES,"control"))

self.link((LINES,"outbox"), (VIS,"inbox"))
self.link((LINES,"signal"), (VIS,"control"))

self.link((VIS,"outbox"), (CONSOLE,"inbox"))
self.link((VIS,"signal"), (CONSOLE,"control"))
```

Clearly `self.link` makes the linkages described, but what does that mean here? Well, it means that the method component.link gets called:

```
def link(self, source,sink,*optionalargs, **kwoptionalargs):
    return self.postoffice.link(source, sink, *optionalargs, **kwoptionalargs)
```

This is clearly a proxy method for the component's local `postoffice`.

What's a `postoffice`? Well, unlike mini-axon which has an actual `microprocess` handling deliveries, we collapse outboxes into inboxes, meaning delivery happens immediately. The `postoffice`'s purpose revolves around tracking what links are created and destroyed.

So in order to find out what's going on we dive inside `postoffice.link`, and find:

```
def link(self, source, sink, *optionalargs, **kwoptionalargs):
    (sourcecomp, sourcebox) = source
    (sinkcomp, sinkbox) = sink
    thelink = linkage(sourcecomp,sinkcomp,sourcebox,sinkbox,*optionalargs,**kwoptionalargs)
    try:
        thelink.getSinkbox().addsource( thelink.getSourcebox() )
    except BoxAlreadyLinkedToDestination, e:
        raise e
    self.linkages.append(thelink)
    return thelink
```

So this extracts the specific source component/boxname & sink component/boxname. It then calls linkage – creating a linkage object – using this information, along with any `optionalargs` and any `kwoptionalargs`.

The next step is to collapse the collapse the two box's storage into one. This is done by the `addsource` method call.

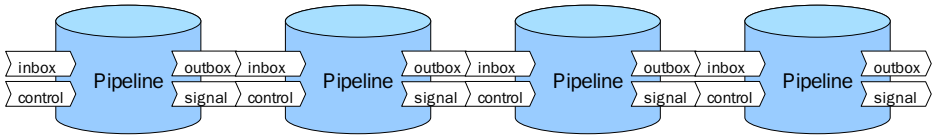
Assuming that works, the `postoffice` stores a copy of the `linkage` locally, and returns a copy back to the user. This enables a user to say `.unlink()` at a later point in time, and to break the linkage – perhaps to point an outbox or inbox in a different direction.

Note: until an outbox is linked to an inbox, it doesn't actually have any storage space, and once it is, an outbox is effectively a proxy to the storage for an inbox.

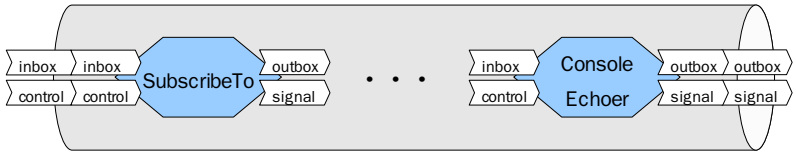
A key result of this, indeed the point of this, is sending a message to an outbox means that it is instantly delivered to the recipient without copying. This is safe for generator components because only one generator is active at any point in time.

External To Internal Interface

Now that we've dealt with the links inside the Pipeline component, we clearly want the following circumstance to be efficient as well.



For this to work as you'd expect, we need to examine what happens inside a Pipeline component at, at the edges. If you do, you'll find that the links look like this:



The sorts of linkages – inbox to inbox and outbox to outbox – are termed passthrough linkages. If you make a passthrough linkage, you are explicitly saying “I'm not going to touch these – I won't get in the way of the conversation between the outside component talking to my subcomponent”.

In Pipeline the way this link is (generally) made is as follows:

```
self.link((self,"inbox"), (self.components[0],"inbox"), passthrough=1)
self.link((self,"control"), (self.components[0],"control"), passthrough=1)

self.link((self.components[-1],"outbox"), (self,"outbox"), passthrough=2)
self.link((self.components[-1],"signal"), (self,"signal"), passthrough=2)
```

This ends up being passed all the way down to the `linkage` class which uses the passthrough number to determine that it's OK to link an inbox to an inbox and an outbox to an outbox. This is for practical reasons – since we do have the logical distinction between inboxes and outboxes, and for safety reasons. Doing thi reduces the chances of accidentally linking an inbox to an inbox and outbox to outbox.

The reason the comment above says generally, is because whilst the majority of pipelines are simple linear “stuff comes in one end, stuff comes out the other” affairs, it is occasionally useful to feed the output of the pipeline back into the front.

If you need to do this, you would make a call like this:

```
Pipeline( ComponentOne(), ComponentTwo(), circular=True)
```

Then the output of ComponentTwo also becomes the source for ComponentOne.

Managing shutdown

A Pipeline component is a Chassis component – a component that requires subcomponents to be active for it to make sense. Given that it uses passthrough linkages for all inboxes and outboxes, how does it know when the children have exitted ?

Well, this is where the concept of **child components** comes in. A child component "lives inside" another component, and that parent component gets notification when a sub component shutsdown. Ie if the component was `.paused()` – sleeping until awoken by an event from the scheduler.

The part of Pipeline that manages this is this:

```
   def __init__(self, *components, **argv):
...
      self.components = list(components)
...
   def main(self):
      self.addChildren(*self.components)
...
      for child in self.children:
          child.activate()
      while not self.childrenDone():
          self.pause()
          yield 1

   def childrenDone(self):
       for child in self.childComponents():
           if child._isStopped():
               self.removeChild(child)   # deregisters linkages for us
       return 0==len(self.childComponents())
```

As a result, what we see is exactly what you expect:

- Pipeline adds the provided components as children
- Pipeline then activates them, after having created all the links
- Then it enters a loop where it knows it will be awoken when a child component exits.
- Its condition for exiting that loop is when all the child components have exited. It does this by calling the `._isStopped()` method to determine this. Exited children are then removed using `removeChild`. This de-registers linkages involving this component and breaks those linkages (via `.unlink()`) as well.

As you might expect, the following components operate in effectively a similar way to Pipeline internally with their own unique foibles: Graphline, Seq & PAR.

Clearly any component can call `.link()` and therefore all components can be container components of varying specialisation. When you do so, rather than using Pipeline, Graphline, Seq, or PAR, then this section should be a useful guide.

Syntactic Sugar

From all this, it should be clear that Pipeline, Graphline, Seq, PAR, and indeed Carousel, are themselves syntactic sugar. As a result, rather than supply syntactic sugar inside Axon, these components themselves provide that functionality.

As a result, let's examine some examples of using them.

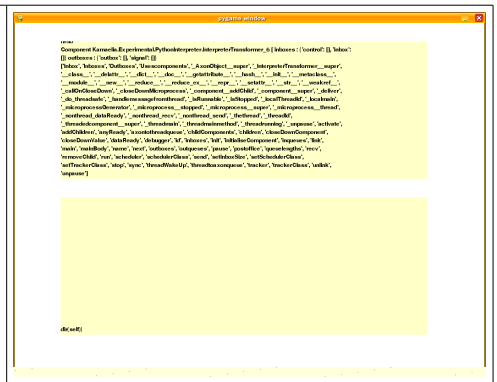
3.5.2 Using Pipelines

As we've seen, pipelines can be used to link a collection of components together to form a system.

Example Pipelines

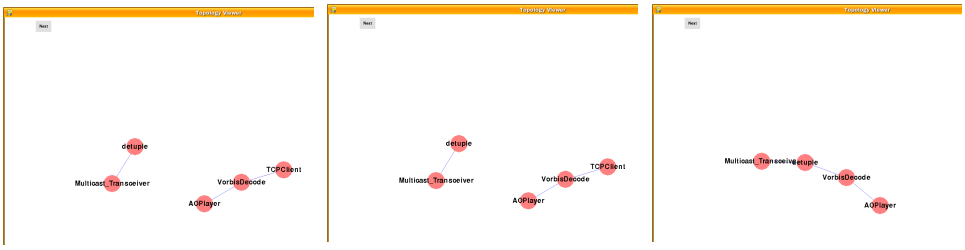
A pygame based graphical python interpreter:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.UI.Pygame.Text import Textbox from
Kamaelia.UI.Pygame.Text import TextDisplayer

from Kamaelia.Experimental.PythonInterpreter
import InterpreterTransformer

Pipeline(
    Textbox(size = (800, 300),
            position = (100,380)),
    InterpreterTransformer(),
    TextDisplayer(size = (800, 300),
                  position = (100,40)),
).run()
```



Clearly this links together 3 components, one handling reading from the user, another displaying to the user and another to interpret any commands given.

Building something more complex, we can create a tool that steps through some rules that describe a topology, which updates when we click a button. A topology slideshow – example:



For this we need a set of instructions, a button to hit, something that when it gets a message that emits one of these instructions & the topology viewer. Since the topology viewer expects pre-parsed instructions, we need something to tokenise the instructions (the topology viewer code includes such a tokeniser).

So we start with out imports:

```
from Kamaelia.UI.Pygame.Button import Button
from Kamaelia.Util.Chooser import Chooser
from Kamaelia.Visualisation.PhysicsGraph.lines_to_tokenlists import lines_to_tokenlists
from Kamaelia.Visualisation.PhysicsGraph.TopologyViewer import TopologyViewer
from Kamaelia.Chassis.Pipeline import Pipeline
```

We then create a list of instructions to step through:

```
graph = """\

ADD NODE TCPClient TCPClient auto -
ADD NODE VorbisDecode VorbisDecode auto -
ADD NODE AOPlayer AOPlayer auto -
ADD LINK TCPClient VorbisDecode
ADD LINK VorbisDecode AOPlayer
ADD NODE Multicast_Transceiver Multicast_Transceiver auto -
ADD NODE detuple detuple auto -
ADD LINK Multicast_Transceiver detuple
DEL NODE TCPClient
ADD LINK detuple VorbisDecode
DEL ALL
""".split("\n")
```

Finally we build the pipeline:

```
Pipeline(
     Button(caption="Next", msg="NEXT", position=(72,8)),
     Chooser(items = graph),
     lines_to_tokenlists(),
     TopologyViewer(transparency = (255,255,255), showGrid = False),
).run()
```

A recent example system which I used at work for a project in the past year was a system that accepted SMS messages from a user. These ended up being placed into a directory for processing – since the SMS provider forwarded the SMS messages over http.

These would be read, processed, and depending on user input, the user would get a response. The core of this part of the system was this:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Apps.Facilitate.SMSFileParser import SMSFileParser
from Kamaelia.Apps.Facilitate.SMSProcesser import SMSProcesser
from Kamaelia.Apps.Facilitate.SMSSender import SMSSender
from Kamaelia.Util.Console import ConsoleEchoer

Pipeline(
    DirectoryWatcher(watch = "/srv/www/sites/www.bickermanor.org/cgi/app/incomingsms"),
    SMSFileParser(),
    SMSProcesser(),
    SMSSender(),
    ConsoleEchoer(),
).activate()
```

Incidentally, this code is likely to be available by Europython. The reason for the ConsoleEchoer on the ends was to trace the system was sending messages.

Similarly, another part of that same system needed to accept user images uploaded, and convert them to a variety of sizes, and transcode any videos uploaded to SWF format for playback on the web. They then needed to be placed into a moderation queue before being made visible. Like the SMS system, the web front end would place content into an incoming directory, and the first step of the system was to sort that content.

As a result, the core of this system – Kamaelia-FileProcessor in the release directory – is actually 4 pipelines:

- One to sort images from a main incoming directory
- One to sort videos from a main incoming directory
- One to pass filenames to an image converter component, which the converted the images.
- One to pass filenames to an video transcoder component, which the converted the images.

The core of that looks relatively simple:

```
Pipeline(
    DirectoryWatcher(watch = conf["main_incoming_queue"]),
    ImageMover(destdir = conf["image_queue"]),
).activate()

Pipeline(
    DirectoryWatcher(watch = conf["image_queue"]),
    ImageTranscoder(destdir = conf["image_moderation_queue"]),
).activate()

Pipeline(
    DirectoryWatcher(watch = conf["main_incoming_queue"]),
    VideoMover(destdir = conf["video_queue"]),
).activate()

Pipeline(
    DirectoryWatcher(watch = conf["video_queue"]),
    VideoTranscoder(destdir = conf["video_moderation_queue"]),
).run()
```

Again, the reason that looks relatively simple is because it's comprised of components which are focused on doing a single task very well, and can then be combined in interesting ways. We also see here that we can `.activate()` several pipelines one after another before choosing to `.run()` the last one.

Clearly this approach would allow us to spawn as many `VideoTranscoders` as we liked to keep our CPU's busy. Furthermore, by using a queueing system like this, rather than on-demand, it means that even if there's

As a result, as well as being clearer, using concurrency in this case also allows for better resource planning.

3.6 Graphlines

Graphlines perform a similar role to Pipelines in Kamaelia, in that they are also container or **Chassis** components. However, Graphlines are not limited to any particular set of links or topologies. You can form whatever structures you like.

An early example of a graphline was a presentation tool, so let's examine that. In this case it took a collection of images, had a previous/next/first/last buttons. It then had something that could step backwards and forwards between those (a “Chooser” component) and then something to display the chosen images.

Putting that together, we grab the imports:

```
import os
from Kamaelia.UI.Pygame.Button import Button
from Kamaelia.UI.Pygame.Image import Image
from Kamaelia.Util.Chooser import Chooser
from Kamaelia.Chassis.Graphline import Graphline
```

Then grab all the file-names from a subdirectory, which contains all the images:

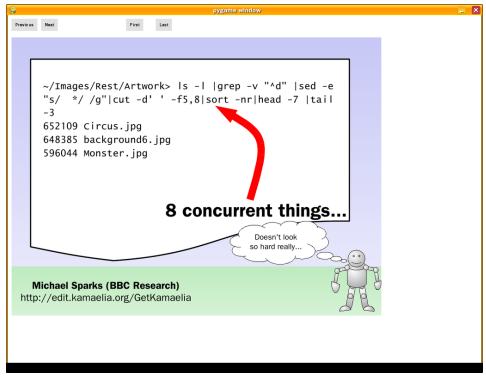
```
path = "Slides"
extn = ".gif"
allfiles = os.listdir(path)
files = list()
for fname in allfiles:
    if fname[-len(extn):]==extn:
        files.append(os.path.join(path,fname))

files.sort()
```

Finally we build our graphline out of these components, define the links, and run them:

```
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()
```

That then looks like this:

You should be able to see the 4 buttons along the top, and that the main panel is the image display.



More generally, Graphlines tend to be used extensively in Kamaelia applications. Going back to that SMS system I mentioned earlier, as well as responding to SMS messages the user sent us, it was also necessary for the web system, and other parts to send SMS messages.

Again, in order to balance resources effectively, and in order to have a responsive web system, sending SMS messages to the user was handed off to another Kamaelia system – this one for sending.

This system needed to do the following:

- Accept sms's placed in an outgoing SMS directory
- Then the SMS would need to be read, and passed onto the sender
- Additionally it was necessary to send the message

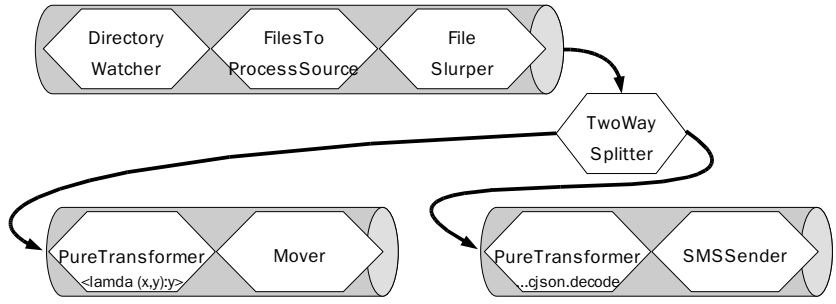
Again, most of the components for doing this were pre-existing. (For example the SMSSender was reused, the file mover code was reused, etc)

```
Graphline(
    SMS_SOURCE = Pipeline(
                    DirectoryWatcher(watch = "outgoingsms"),
                    FilesToProcessSource(),
                    FileSlurper(tuplemode=True),
                ),
    CLEANER = Pipeline(
                  PureTransformer(lambda (x,y): y ),
                  Mover(),
              ),
    SENDER = Pipeline(
                  PureTransformer(lambda (x,y): cjson.decode(x) ),
                  SMSSender(),
             ),
    SPLIT = TwoWaySplitter(),
    linkages = {
         ("SMS_SOURCE","outbox") : ("SPLIT","inbox"),
         ("SPLIT", "outbox") : ("CLEANER", "inbox"),
         ("SPLIT", "outbox2") : ("SENDER", "inbox"),
    }
).run()
```

This graphline doesn't just consist of basic components- but also includes pipelines as

sources and sinks. This shows rather nicely that many real world systems that get built tend to use Graphlines to for top level linkages & Pipelines for linking subsystems together.

Additionally there is sufficient information here for us to draw an diagram about what's going on:



Furthermore, you can place graphlines inside pipelines, graphlines in graphlines, and so on.

3.7 Incrementally Growing Systems: Nesting Pipelines & Graphlines

This allows you to develop gradually more complex systems, testing them along the way, piece by piece. Put another way, this allows you to safely create and extend Kamaelia based systems in a naturally safe manner.

For example, one application developed in this way:

- A simple scribbler application was written
- Buttons that emitted a colour spec were written and hooked into that.
- Then a simple protocol was written for recording the strokes drew, to drive the scribbler
- Then something was written that serialised these and able to be sent over the network. The other end was another scribbler, which could deserialise these, and display them.
- Then this was mirrored by the other end – resulting in a whiteboard application which could be used by 2 users in 2 locations.
- This was then generalised to any number of users.
- Then someone wrote a “speex” audio codec component, and started using the same technique for sharing audio – meaning the people in both locations could talk to each other as well.
- Then multiple pages were added, defaulting to saving when changing pages, resulting in a shared, multipage notebook.

This then resulted in the creation, incrementally of whiteboard capable of being both a client and server, mixing audio in each location. Again the core of that system is also a graphline,

which is in Kamaelia/Tools/Whiteboard/Whiteboard.py . We'll rummage around in this in the tutorial on the day, but to give an idea of what it looks like, we'll look at it here briefly.

As noted, we had a Canvas and something capable of Painting on it.

```
Graphline( CANVAS  = Canvas( position=(left,top+32),size=(width,height-32) ),
           PAINTER = Painter(),
```

Then we want some controls regarding colour, erasing & clearing:

```
           PALETTE = buildPalette( cols=colours, order=colours_order,
                                   topleft=(left+64,top), size=32 ),
           ERASER  = Eraser(left,top),
           CLEAR = ClearPage(left+(64*5)+32*len(colours),top),
```

Then there's controls over pages:

```
           PAGINGCONTROLS = PagingControls(left+64+32*len(colours),top),
           LOCALPAGINGCONTROLS = LocalPagingControls(left+(64*6)+32*len(colours),top),
           LOCALPAGEEVENTS = LocalPageEventsFilter(),
```

Then there's a component regarding converting these page controls into commands which manage the page loading/etc.

```
HISTORY = CheckpointSequencer(lambda X: [["LOAD", SLIDESPEC % (X,)]],
                              lambda X: [["SAVE", SLIDESPEC % (X,)]],
                              lambda X: [["CLEAR"]],
                              initial = 1,
                              highest = num_pages,
          ),
```

Then we need to duplicate events to the canvas and the network – like earlier we can use a two way splitter to do this:

```
                      PAINT_SPLITTER = TwoWaySplitter(),
                      LOCALEVENT_SPLITTER = TwoWaySplitter(),
                      DEBUG   = ConsoleEchoer(),
```

And finally make all the actual links:

```
linkages = {
    ("CANVAS",  "eventsOut") : ("PAINTER", "inbox"),
    ("PALETTE", "outbox")    : ("PAINTER", "colour"),
    ("ERASER", "outbox")     : ("PAINTER", "erase"),
    ("PAINTER", "outbox")    : ("PAINT_SPLITTER", "inbox"),
    ("CLEAR","outbox")       : ("PAINT_SPLITTER", "inbox"),
    ("PAINT_SPLITTER", "outbox")  : ("CANVAS", "inbox"),
    ("PAINT_SPLITTER", "outbox2") : ("", "outbox"), # send to network
    ("LOCALPAGINGCONTROLS","outbox")  : ("LOCALEVENT_SPLITTER", "inbox"),
    ("LOCALEVENT_SPLITTER", "outbox2"): ("", "outbox"), # send to network
    ("LOCALEVENT_SPLITTER", "outbox") : ("LOCALPAGEEVENTS", "inbox"),
    ("", "inbox")        : ("LOCALPAGEEVENTS", "inbox"),
    ("LOCALPAGEEVENTS", "false")  : ("CANVAS", "inbox"),
    ("LOCALPAGEEVENTS", "true")  : ("HISTORY", "inbox"),
    ("PAGINGCONTROLS","outbox") : ("HISTORY", "inbox"),
    ("HISTORY","outbox")     : ("CANVAS", "inbox"),
    ("CANVAS", "outbox")     : ("", "outbox"),
    ("CANVAS","surfacechanged") : ("HISTORY", "inbox"),
}, )
```

Clearly, this didn't just “jump” into being in this form, and as noted this code evolved in this way, based on a gradual accretion of functionality.

As a result, doing a deep dive through this code wouldn't make much sense, since it would not explain why the code is this structure. Indeed in doing so it would probably point to why it would be better to do it a different way, so we will come back to how you build up complex systems shortly, after touching on two other Chassis components.

Furthermore by showing how systems grow and evolve, it should help should you need to do this.

3.8 PAR Components

As we've seen, Pipeline is a component that takes a bunch of components, creates links between then all in a predefined way and runs them all concurrently. Graphline is a component that takes a bunch of components, and links them up based on a user definition.

Similarly, PAR is also a component that takes a bunch of components, but unlike the others, they are **not** wired up/connected together.

Specifically PAR:

- Is passed a collection of components
- It links their output to its output
- It activates them all
- And exits when they do.

This is a relatively new component, and there are plans to make it possible to define a policy function regarding what to do with messages coming in on the standard inboxes, but that is something that will come later.

For the moment however it enables us to rewrite some graphlines in a more direct style. For example, these two pipeline/par & graphlines are equivalent, but one is more direct and explicit than the other:

```
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)),
```

```
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()
```

```
    Linkages = {
        ("NEXT","outbox") : ("CHOOSER","inbox"),
    ("PREVIOUS","outbox") : ("CHOOSER","inbox"),
       ("FIRST","outbox") : ("CHOOSER","inbox"),
        ("LAST","outbox") : ("CHOOSER","inbox"),
     ("CHOOSER","outbox") : ("IMAGE","inbox"),
         }
).run()
```

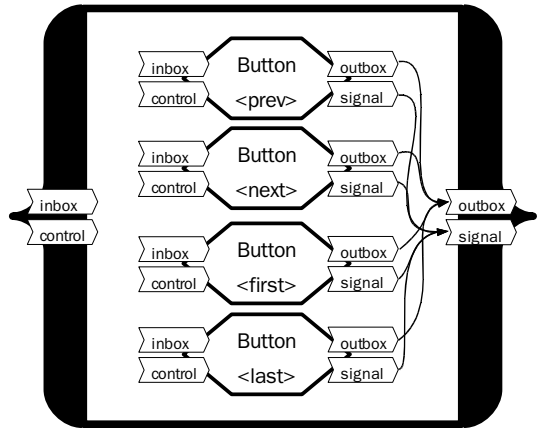
Clearly the version using “PAR” is more compact. Which is clearer from a maintainers perspective probably depends on whether they know what PAR does or not, but the potential for clarifying code through compactness like this is useful.

Specifically it is removing the apparent complexity of the linkages, and being more explicit about what should happen with the data flow.

As noted, PAR is a relatively new component, but it is likely to be useful in simplifying many systems.

Diagrammatically, these two are equivalent:

```
PAR(
    Button(msg="NEXT" ...
    Button(msg="PREV" ...
    Button(msg="FIRST" ...
    Button(msg="LAST" ...
)
```



An extension to come in the next few weeks will be control over the inbound wiring, through the use of a policy function.

3.9 Seq Components

Clearly **PAR** allows you take a collection of components and easily set them running in **PAR**allel – with the output from all subcomponents all piped to one location.

Seq however, allows you to take a collection of components and easily set them running **seq**uentially – one after another. Ie they allow you to do this:

- Seq(RunThis(), ThenThis(), ThenThat(), AndFinallyThis())

This is useful because the inboxes and outboxes of these are rewired after each, allowing you to pipe data through one component & then another.

This allows you to do a variety of things, but examples include :

- Sending an initial starting message before running a component which generates output.
- The MobileReframer (in Kamaelia/Tools) uses it to divide processing of video files up into stages. First it decodes and seperates frames, then processes some edits, and then performs some cleanup. Each of the stages is a seperate Graphline.
- Performing authentication on a network connection before allowing a user to speak with the main connection. We'll see how this can work later.

Seq also provides you with a means of forcing a particular sequence of events – with components running one after another – for example where you need to ensure that some state (of the system, computer etc) is updated in a particular sequence. Examples here include:

- Setting the Pygame Display to a particular configuration before running other pygame components. This can be useful to set settings in sub processes when working with multiple processes.

  Example: (from a GSOC multi process paint app)

```
ProcessGraphline(
    COLOURS = Seq(
         DisplayConfig(width=270, height=600),
         ToolBox(size=(270, 600)),
         ),
    WINDOW1 = Seq(
              DisplayConfig(width=555, height=520),
              Paint(bgcolour=(100,100,172),position=(10,10), size = (500,500),
                    transparent = False),
               ),
```

- Another can be to sequence data senders and data receivers. This can be useful in an “expect” type system, for example:

```
def authentication_sequence(state):
        return Seq(
            Sender(“password: ”),
            LineReceive(state, “username”, stripEOL=true),
            Sender(“password: “)
            LineReceive(state, “password”, stripEOL=true),
            ConfirmIdentity(state),
        )
```

  NB. Neither Expect nor Sender exist as yet!

Exercise

Write Sender & Expect. Use them to replace the Authentication component in the bulletin board example, which we'll see later. Bonus points for contributing this back to the Kamaelia project!

3.10 Backplanes – Broadcast for Components

A backplane is essentially a named broadcast mechanism for Kamaelia systems, and consists of 3 components which are used together:

Backplane

- This creates and advertises a "Backplane" **service** to other components. Specifically, components can connect to this service via a **SubscribeTo** or **PublishTo** component. Any data published to the backplane by **PublishTo** components is duplicated and sent to any connected **SubscribeTo** components.

PublishTo

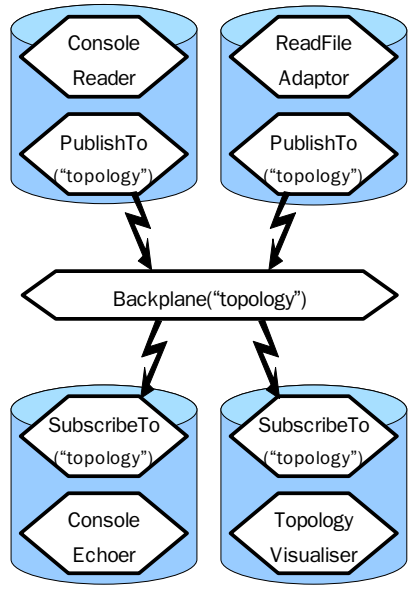
- This looks for a previously advertised named service. When it finds it, it connects its main outbox to the service, and any data it receives on its main inbox is sent to the backplane.

SubscribeTo

- This looks for a previously advertised named service. When it finds it, it sends it a message saying "send me data to my main inbox". When the backplane does this, any data SubscribeTo receives is forwarded to its main outbox.

As a result, given these you can build a vast array of interesting and flexibile topologies using this. For example, to have an application take data from the console & a file, and to enable the user to watch this on the console, and for the same data to drive, say, a topology visualiser, using a backplane makes sense sense:

```
from Backplane import *
from Kamaelia.Chassis.PAR import PAR
from Kamaelia.Chassis.Pipeline import Pipeline

from Kamaelia.Util.Console import *
from Kamaelia.Visualisation.PhysicsGraph.TopologyViewer \
       import LineControlledTopologyViewer

PAR(

  Backplane("topology"),

  Pipeline( ConsoleReader(),PublishTo("topology") ),
  Pipeline( ReadFileAdaptor(filename),
            PublishTo("topology") ),

  Pipeline(SubscribeTo("topology"), ConsoleEchoer() ),
  Pipeline(
    SubscribeTo("topology")
    LineControlledTopologyViewer(filename))

).run()
```



Note: We've used PAR here to activate all these components in this order, by to run concurrently.
Using PAR for this is much cleaner, than .activate()

3.10.1 Services ?

Briefly, you can advertise any inbox to the rest of the system for others to find and connect to.

Service

- A service is defined as a tuple that has this form (component, “someinboxname”).
- **NB.** A service is something that will sit as the second argument of a self.link call.

Public Service

- A public service is a service that has been named and registered with the coordinating assistant tracker.

Providing a service

You register services like this:

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

    # Once your component is ready, advertise your service as follows:
    def advertiseService(servicename=”myservice”):
        # First find the co-ordinating assistant tracker
        theCAT = coordinatingassistanttracker.getcat()
        # Then a component can register a service with it:
        theCAT.registerService(servicename, self, "inbox")
```

When your component is shutting down, cease advertising your service as follows:

-
```
    def deregisterService(servicename=”myservice”):
        theCAT.deRegisterService(servicename)
```

It is your responsibility to ensure that clients of this service are aware that this is happening. (eg messages you send them telling them this is going on)

Using a service

To use a service, you must look for the service...

```
theCAT = coordinatingassistanttracker.getcat()
the_service = theCAT.retrieveService("some service")
```

...and then connect an outbox of yours to it. Keeping a reference to the linkage is a good idea at this point.

```
self.link_to_service = self.link( (self, “outbox”), the_service)
```

You can then send it messages. For example, you may wish to receive data back from the service, you may want to send it the name of an outbox to talk to. For it to do that you send it a reference to a private service for it to talk to:

```
talk_to_me_here = ( self, “inbox )
self.send(  (“send me data”, talk_to_me_here) , “outbox”)
```

When you're done using it, you need to unlink the service. You do this as follows:

```
self.unlink( self.link_to_service )
```

The service you're using may require you to send a message to it before hand to let it know you're doing this. You'll probably need to include some information so it knows who's doing this, unless it's just interested in receiving data.

Eg:

```
self.send(  ("unsubscribing", talk_to_me_here) , "outbox")
self.unlink( self.link_to_service )
```

When are services used?

When there's a resource that multiple components may wish to use.

Examples:

- Selector – used by lots of active connections, servers, clients and some file readers (including UnixProcess) in order to know when they can read or write to from connections & files.
- PygameDisplay – uses this to advertise a display service to pygame components. This allows one component to manage all user input, and farm off surfaces and handling to components.
- Backplane – for providing broadcast facilities inside an application

Tools for uses services include:

- Backplane – Backplane, SubscribeTo, PublishTo
- Kamaelia.Experimental.Services – RegisterService, Subscribe, ToService

There is likely to be more components over time for using services.

Generally speaking though, if you need to provide a service, consider using Backplane, rather than rolling your own code. (It's likely to simplify things!)

3.10.2 Backplane Internals

You don't need to understand what's going on inside these components to use them. For those interested however:

Backplane

- This creates and advertises a "Backplane" **service** to other components. Specifically, components can connect to this service via a **SubscribeTo** or **PublishTo** component. Any data published to the backplane by **PublishTo** components is duplicated and sent to any connected **SubscribeTo** components.

  Under the hood this is a `Kamaelia.Util.Splitter.PlugSplitter`, and it registers 2 services with Axon's "Co-ordinating assistant tracker" (Cat :).

These services are:

- `Backplane_I_+<name of backplane>` - **PublishTo** talks to this.
- `Backplane_O_+<name of backplane>` - **SubscribeTo** talks to this.

More information on the Co-ordinating assistant tracker can be found here.

- http://www.kamaelia.org/Docs/Axon/Axon.CoordinatingAssistantTracker

You can simplify the creation of your own services using facilities provided inside `Kamaelia.Experimental.Services`, specifically `RegisterService`, `Subscribe`, `ToService`.

PublishTo

- Simply looks up the backplane's inbound service, `Backplane_I_+<name of backplane>` and connects its outbox to it, copying all input to its output.

SubscribeTo

- Creates a Kamaelia.Util.Splitter.Plug component
- Looks for the backplane's outbound service - `Backplane_O_+<name of backplane>`
- Provides this to the plug, which then hooks itself into the PlugSplitter
- It takes the output from that and passes it to its own output.

If you do decide to write your own services, this is a model worth looking at.

Generally speaking, though, many services are often best implemented using a Backplane, since it enables you to hook up a pipeline to it to see what data is actually flowing to and from your service.

3.11 Server Core

Kamaelia.Chassis.ConnectedServer contains the class ServerCore, this is designed to handle all the nitty gritty details of handling lots of people connected to a TCP based network server. As a result:

- It listens on a particular port
- You can provide some socketOptions to this if you like
- You can even override the TCPServer component it uses to listen for inbound connections, if you need something custom (incredibly rare, but Kamaelia Grey does this).
- Then when it receives a connection, it accepts the connection, sets up a ConnectedSocketAdapter (CSA) to handle reading and writing to/from the connection and needs something to accept the inbound data, and to send data back, for the CSA to send back to the user.

This is where you come in.

As a result, the signature for ServerCore looks like this:

```
ServerCore( protocol[, port=1601] [, socketOptions=None] [, TCPS=TCPServer] )
```

`protocol` is a something which ServerCore will call when a connection is accepted.

This can be an actual component, or it can be something that returns a component. For example, these can both be used:

Class based case

```
Class Echo(Axon.Component.component):

  def main(self):

      while not self.dataReady("control"):
          for msg in self.Inbox("inbox"):
              self.send(self.peer+msg, "outbox")
          self.pause()
          yield 1

      self.send( self.recv("control"), "signal")

ServerCore(protocol=Echo, port=1601).run()
```

Factory based case

```
Class Echo(Axon.Component.component):

    def main(self):

        while not self.dataReady("control"):
            for msg in self.Inbox("inbox"):
              self.send(self.peer+msg, "outbox")
            self.pause()
            yield 1

        self.send( self.recv("control"), "signal")

def myProtocol(**argv):
  return Pipeline(
              ConsoleEchoer(forwarder=True),
              Echo(**argv)
  )

ServerCore(protocol=Echo, port=1601).run()
```

When a client connects, ServerCore will call whatever was provided by the user with some information about the connection, effectively doing this:

```
the_callable = self.protocol
protocolHandler =  the_callable(peer = sock_info["peer"],
                                peerport = sock_info["peerport"],
                                localip = sock_info["localip"],
                                localport = sock_info["localport"])
```

In the class case, this just calls the class, and gets a component object back, and wires it in, and in the factory case, it does the same.

Note: both the class case and factory case here use the same actual protocol. The factory case dumping all inbound data to the console however is useful for debugging!

3.12 ServerCore & Backplanes

Something rather special happens when you combine these two components – let's revist the radio 1 example.

We create a backplane.

We create a DVB tuner for radio 1, and publish radio 1 to that backplane.

We start a ServerCore instance, which when called will create a SubscribeTo which will forward a copy or radio1 data to whomever connects to the server.

```
Backplane("Radio").activate()

Pipeline(
    DVB_Multipex(850.16, [6210], feparams), # RADIO ONE
    PublishTo("Radio"),
).activate()

def radio(*argv, **argd):
    return SubscribeTo("Radio")

ServerCore(protocol=radio, port=1600).run()
```

This ability to create splitting servers incredibly trivially a really useful, and if you're ever unclear as to what's going on inside a system of yours you can just intercept the data flow, publish it to a backplane, connect a server to it and have a rummage.

For example, if you wanted to debug SMS sending application described earlier:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Apps.Facilitate.SMSFileParser import SMSFileParser
from Kamaelia.Apps.Facilitate.SMSProcesser import SMSProcesser
from Kamaelia.Apps.Facilitate.SMSSender import SMSSender
from Kamaelia.Util.Console import ConsoleEchoer

Pipeline(
    DirectoryWatcher(watch = "/srv/www/sites/www.bickermanor.org/cgi/app/incomingsms"),
    SMSFileParser(),
    SMSProcesser(),
    SMSSender(),
    ConsoleEchoer(),
).activate()
```

And wanted to look at what was happening between the SMSProcessor and SMSSender, you could change the above application as follows:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Apps.Facilitate.SMSFileParser import SMSFileParser
from Kamaelia.Apps.Facilitate.SMSProcesser import SMSProcesser
from Kamaelia.Apps.Facilitate.SMSSender import SMSSender
from Kamaelia.Util.Console import ConsoleEchoer
from Kamaelia.Chassis.ConnectedServer import ServerCore
from Kamaelia.Util .Backplane import Backplane
from Kamaelia.Util.PureTransformer import PureTransformer

Backplane("DEBUGSERVER").activate()

def debugSMSSystem (**argd):
    return Pipeline(
              PureTransformer(lambda x: repr(x)),
              SubscribeTo("DEBUGSERVER")
          )

ServerCore(protocol=debugSMSSystem, port=1601).activate()

Pipeline(
    DirectoryWatcher(watch = "/srv/www/sites/www.bickermanor.org/cgi/app/incomingsms"),
    SMSFileParser(),
    SMSProcesser(),
    PublishTo("DEBUGSERVER", forwarder=True),
    SMSSender(),
    ConsoleEchoer(),
).run()
```

The system then continues to work as before, however you can now telnet to port 1601 on that machine, and watch what's actually being sent to the SMSSender for sending.

The ability to attach extra components to existing systems for introspection, debugging or in some cases, rummaging around inside them when they're running, is something which is particularly useful in Kamaelia systems.

3.13 Shell Equivalence

For those used to the shell, and building ad-hoc commandlines which you often think ought to be actual scripts, it's useful to note that a number of Kamaelia forms have direct equivalence to the sort of thing you use on the shell.

This is useful to know because lots of people use concurrency using the shell without even thinking about it, and without that even really being the goal. Kamaelia tries to aim for the same eventual simplicity. There's a little way to go yet :)

Kamaelia Form	Shell Form
`A.activate()`	`A&`
`Pipeline(A, B, C, D)`	`A | B | C | D`
`PAR(A,B,C,D)`	`( A & B & C & D )`
`Pipeline(PAR(A,B,C), D)`	`(A & B & C) | D`
`Seq(A, B, C, D)`	`A; B; C; D`
`Pipeline(Seq(A, B, C), D)`	`(A; B; C) D`
`Graphline( ... )`	?
`Backplane/PublishTo/SubscribeTo`	?

However, unlike the Unix shell you:

- Have Graphlines, which have no direct equivalent.
- Pass fully formed python objects around, not bytes over a byte stream
- Have access to global publishing tools – services like Backplanes – which again, have no equivalent in the shell world.

4 Kamaelia, in non-Kamaelia Systems

Kamaelia does not require you to rewrite your entire system to work with Kamaelia.

Or put another way, if you have a traditional piece of code – for example a GTK based application, or a pygame application, or QT based, etc, and you wish to use some parts of Kamaelia to deal with aspect of your problem you can do this, without rewriting your application.

If you want to embed a debugging server as discussed in section 3.10, then you can do that as well. Similarly, if you want to embed an interactive python console inside your application that you can telnet to in order to rummage around to figure out a particularly tricky bug, you can do that as well, without rewriting your application.

The reason for this is simple: Kamaelia is as happy running in a background thread as it is in the foreground.

4.1 Axon.Handle

Clearly you need a means of then talking to Kamaelia components, without your head exploding, and the way to do that is using Axon.Handle.

4.1.1 File Handles vs Axon Handles

The idea behind Axon.Handle is inspired by plain and simple file handles. It isn't a file handle, so we don't call it that, but it is very similar, hence the name. But let's take a look at file handles, so we have a common starting point, using this simple example:

```
source = open("somefile.txt")
sink = open("someotherfile.txt","w")
for line in source:
    sink.write(l)

source.close()
sink.close()
```

Now, what's actually happening here, in this seemingly innocuous piece of code?

- We have a main thread of control, which is the code that is written.
- We've asked the operating system to allocate some resource for reading from a file, and to allocated some resource for writing to the file.
- The system starts reading the file in a buffered manner. Now whether this is via a state machine, handled on the python side, or on the OS side, something has been allocated that knows it is reading from a file. It's interacting with a file system subsystem for accessing bytes from whatever device "somefile.txt" resides upon, which may actually be over NFS (incurring network reads & writes), which may be reserving something from an sshfs FUSE based file system accessed over an ssh

connection.

- Something similar may be happening with the file writer.
- Then our thread of control is waiting for data, when there's some there taking it, and placing it into some write buffer that will eventually get flushed and written to whereever the file is actually located.

Essentially, despite not worrying about it, file handles are in fact masking significant amounts of concurrent behaviour from a user.

Axon.Handles

Axon.Handle does the same thing, but for Axon/Kamaelia based components.

You ask Axon.Handle to activate a component for you and provide you with a handle to that component. You can then put messages into inboxes, and get messages from inboxes.

The one difference between Axon.Handles and file handles is that Axon.Handles default to being non-blocking. If you try and read from an empty outbox, you'll get a Queue.Empty message as a response (much like reading from a non-blocking file handle gives you a similar message if it's not ready).

4.1.2 Using Axon.Handle

Suppose we have a server we know is running on www.example.com port 1600, we can write tcp client that uses the Kamaelia TCP client in normal linear code.

Necessary imports for handles to work:

```
from Axon.background import background
from Axon.Handle import Handle
```

Necessary for handle when data isn't ready:

```
import Queue
import time
```

The component we want to use:

```
from Kamaelia.Internet.TCPClient import TCPClient
```

We then start the scheduler in the background:

```
background(slowmo=0.01).start()
```

Then we can run use Axon.Handle to read/write to/from the network connection.

```
our_client = Handle(TCPClient(host = "www.example.com", port = 1600)).activate()
while True:
    user_data = raw_input(">>> ")
   our_client.put(user_data,"inbox")
    while 1:
        try:
            print our_client.get("outbox")
            break
        except Queue.Empty:
            time.sleep(0.01)
```

4.2 Augmenting Existing Systems

Now, let's suppose you've writte a GTK application, and you want to do some user testing.

You want to accurately track why they're consistently getting something wrong, and you're not getting good data from user reports, and you suspect they're hitting a button they shouldn't. In that case it'd be useful to augment it such that such events got sent to a central server for logging. Let's look at how you could do that with Kamaelia, and Axon handles.

So this is our basic GTK application:

```
import gtk

class HelloWorld:
    def hello(self, widget, data=None):
        print "Hello World"

    def delete_event(self, widget, event, data=None):
        print "delete event occurred"
        return False

    def destroy(self, widget, data=None):
        print "me"
        gtk.main_quit()

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.connect("delete_event", self.delete_event)
        self.window.connect("destroy", self.destroy)
        self.window.set_border_width(10)
        self.button = gtk.Button("Hello World")
        self.button.connect("clicked", self.hello, None)
        self.window.add(self.button)
        self.button.show()
        self.window.show()


def main():
    gtk.main()

if __name__ == "__main__":
    hello = HelloWorld()
    main()
```

Let's say we have our event logging server running this on a seperate server running on 192.168.2.5 , port 1500. Our imports for that might look like this:

```
import time
from Kamaelia.Util.Backplane import *
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Chassis.ServerCore import ServerCore
from Kamaelia.File.Append import Append
from Kamaelia.Util.PureTransformer import PureTransformer
```

We then want to have our server log to the file. Since we may want to have lots of people sending us data, we'll tag data on the way in, and send it to a backplane, and then let a

single logger handle logging for us. It'll also timestamp events.

The backplane set up:

```
Backplane("UI_EVENTS").activate()
```

The server side part that handles timestamping & logging.

```
Pipeline(
    SubscribeTo("UI_EVENTS"),
    PureTransformer(lambda x: str(time.time()+"|"+x),
    Append(filename = "userlog.txt"),
).activate()
```

The protocol handler we have for the server would be this – note that it tags inbound data with the client IP:

```
def log_events (**argd):
    peer = str(argd.get(peer, ""))
    return Pipeline(
                PureTransformer(lambda x: peer + "|" + x),
                PublishTo("UI_EVENTS")
           )
```

Finally, we run the logging server:

```
ServerCore(protocol=log_events, port=1500).run()
```

How do we get the GTK application to talk to this? In the same way as we did with the hello world application – let's take a look:

```
import gtk

from Axon.background import background
from Axon.Handle import Handle
from Kamaelia.Internet.TCPClient import TCPClient

background(slowmo=0.01).start()
our_client= Handle(TCPClient(host = "192.168.2.5", port = 1500)).activate()

class HelloWorld:
    def hello(self, widget, data=None):
        print "Hello World"
        our_client.put("Hello World Clicked")


    def delete_event(self, widget, event, data=None):
        print "delete event occurred"
        return False

    def destroy(self, widget, data=None):
        print "me"
        gtk.main_quit()

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.window.connect("delete_event", self.delete_event)
        self.window.connect("destroy", self.destroy)
        self.window.set_border_width(10)
```

```
        self.button = gtk.Button("Hello World")
        self.button.connect("clicked", self.hello, None)
        self.window.add(self.button)
        self.button.show()
        self.window.show()


def main():
    gtk.main()

if __name__ == "__main__":
    hello = HelloWorld()
    main()
```

As you can see the GTK application is largely unchanged.

4.3 Using Axon.Handles with Kamaelia Systems

Since a Kamaelia system is itself a component, with a bit of effort you can control any part of a Kamaelia system this way. For example, inserting a SubscribeTo component somewhere means you can put a handle round a PublishTo component, and post data anywhere inside the Kamaelia system. Or vice versa.

So if you want to use the topology visualiser, you can use that with your twisted based network system if you really want to.

4.4 Embedding a Python Interpreter

Kamaelia has a python interpreter component, which we touched upon earlier. This can be be used inside (say) a pygame window like this:

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.UI.Pygame.Text import Textbox
from Kamaelia.UI.Pygame.Text import TextDisplayer
from Kamaelia.Experimental.PythonInterpreter import InterpreterTransformer

Pipeline(
    Textbox(size = (800, 300), position = (100,380)),
    InterpreterTransformer(),
    TextDisplayer(size = (800, 300), position = (100,40)),
).run()
```

We can obviously attach this to any python system – whether Kamaelia based or not. For example, to embed it in the logging server we discussed, our logging server changes to this:

```
import time
from Kamaelia.Util.Backplane import *
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Chassis.ServerCore import ServerCore
from Kamaelia.File.Append import Append
from Kamaelia.Util.PureTransformer import PureTransformer
```

```
from Kamaelia.UI.Pygame.Text import Textbox
from Kamaelia.UI.Pygame.Text import TextDisplayer
from Kamaelia.Experimental.PythonInterpreter import InterpreterTransformer

Backplane("UI_EVENTS").activate()

Pipeline(
    SubscribeTo("UI_EVENTS"),
    PureTransformer(lambda x: str(time.time()+"|"+x),
    Append(filename = "userlog.txt"),
).activate()

def log_events (**argd):
    peer = str(argd.get(peer, ""))
    return Pipeline(
                PureTransformer(lambda x: peer + "|" + x),
                PublishTo("UI_EVENTS")
           )

Pipeline(
    Textbox(size = (800, 300), position = (100,380)),
    InterpreterTransformer(),
    TextDisplayer(size = (800, 300), position = (100,40)),
).activate()

ServerCore(protocol=log_events, port=1500).run()
```

Whilst this may seem odd to be able to have a running interpreter inside a server, being able to do this is a natural consequence of Kamaelia. Once you can run whatever components you want in parallel with all the others, you can run anything with anything.

4.4.1 Embedding a Network accessible Kamaelia Python Console in non-Kamaelia systems

Suppose however that this wasn't enough for us to figure out what was going wrong with the user application we modified earlier. Suppose we wanted to login to a python console embedded to our GTK application, over the network. Can we do that? Sure we'd have to do serious amounts of rummaging around to find what we needed, but can we do it?

Yes, we can :-)

This turns out to be simpler than you might think – it looks like this:

```
import gtk

class HelloWorld:
    def hello(self, widget, data=None):
        print "Hello World"

    def delete_event(self, widget, event, data=None):
        print "delete event occurred"
        return False

    def destroy(self, widget, data=None):
        print "me"
        gtk.main_quit()

    def __init__(self):
        self.window = gtk.Window(gtk.WINDOW_TOPLEVEL)
```

```
            self.window.connect("delete_event", self.delete_event)
            self.window.connect("destroy", self.destroy)
            self.window.set_border_width(10)
            self.button = gtk.Button("Hello World")
            self.button.connect("clicked", self.hello, None)
            self.window.add(self.button)
            self.button.show()
            self.window.show()


    def main():
        gtk.main()

    if __name__ == "__main__":
        from Axon.background import background

        from Kamaelia.Chassis.Pipeline import Pipeline
        from Kamaelia.Chassis.ConnectedServer import ServerCore
        from Kamaelia.Util.PureTransformer import PureTransformer
        from Kamaelia.Experimental.PythonInterpreter import InterpreterTransformer

        background(slowmo=0.01).start()

        def NetInterpreter(**argv):
            return Pipeline(
                   PureTransformer(lambda x: str(x).rstrip()),
                   PureTransformer(lambda x: str(x).replace("\r","")),
                   InterpreterTransformer(),
                   PureTransformer(lambda x: str(x)+"\r\n>>> "),
               )

        ServerCore(protocol=NetInterpreter, port=9765).activate()

        hello = HelloWorld()
        main()
```

You can then telnet to this application – eg 127.0.0.1 port 9765, and you're presented with a python console, and can rummage around inside to your hearts content.

4.4.2 Health Warning

It's a VERY bad idea to connect a network accessible python interpreter to the bare internet. Very, Very Very bad things can happen if you do this.
Don't do it, only use it in a completely controlled environment.

That said, in a completely controlled environment, this is an incredibly useful tool to have available!

5 Building A Bulletin Board

As our final example, we'll build an old-school style bulletin board. It'll have the following behaviour:

- You'll telnet to port 1600 on the server to login
- You'll be prompted for a username & password. For simplicity we won't hide the password from the user when they're typing it. It'll keep prompting for a username/password until they're logged in or disconnect.
- The system will then retrieve the users state. (Actually we'll leave this as a stub)
- The user will then be able to interact with the bulletin board.
- When they choose the logout option, their state will be saved. (again we'll leave that as a stub)
- The final version should also be capable of handling strings sent to the client which have returns in odd places or strings which don't contain carriage returns. (Telnet under linux happens to default to line oriented, but not all telnet applications work that way)

Once logged in, the user will be presented with the option of doing three things:

- Entering a message reading/sending mode.
- Displaying help – h, followed by return.
- Quitting – q followed by return.

When in the message reading mode, they will be presented with the next unread message, followed and then they again have some options:

- Return to view the next message
- r – to enter a reply to another message. (we won't implement this, but you could)
- d – to delete the message (again, we won't implement this, but you could)
- h – for help
- x – to exit message reading back to the menu

For brevity's sake we won't implement message read marking nor adding messages. However the system will stored all the messages in a directory called “messages”. They will be stored as serialised JSON objects, which will be structures of the following form:

```
{
     'from': 'michael',
     'to': 'michael',
     'message': '4',
     'date': 'TBD',
     'subject': 'test',
     'reply-to': ['3', '2', '1'],
    '__body__': 'Testing\nTesting\n123'
}
```

The strings in reply-to are references to other messages. Clearly creating these message objects is relatively simple once, you have the rest of the system in place.

5.1 Building up the initial protocol

The question with this system, is where do you start?

Well we want a network server, so let's start there, and we're responding to the user's, so let's be a little unimaginative and create a server with a protocol handle by a RequestResponseComponent.

The skeleton of that looks like this:

```
import Axon
from Kamaelia.Chassis.ConnectedServer import ServerCore

class RequestResponseComponent(Axon.Component.component):
    def main(self):
        while not self.dataReady("control"):
            for msg in self.Inbox("inbox"):
                self.send(msg, "outbox")
            self.pause()
            yield 1
        self.send(self.recv("control"), "signal")

ServerCore(protocol=RequestResponseComponent,
           port=1600).run()
```

- **File:** BB1.py

When you run this, if kill the server without killing all the clients by hitting control-C, this can leave the server unable to accept connections for 60 seconds. This is due to the operating system & TCP/IP stack rather than Kamaelia.

Since it's the operating system default, it's also Kamaelia's default. It can however be annoying when you're writing a server, or server needing a fast restart. Let's make it so that we can restart the server quicker:

```
import socket
import Axon
from Kamaelia.Chassis.ConnectedServer import ServerCore

class RequestResponseComponent(Axon.Component.component):
    def main(self):
        while not self.dataReady("control"):
            for msg in self.Inbox("inbox"):
                self.send(msg, "outbox")
            self.pause()
            yield 1
        self.send(self.recv("control"), "signal")

ServerCore(protocol=RequestResponseComponent,
           socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1),
           port=1600).run()
```

- **File:** BB2.py

Now, what we'd like to write really is something like this:

```
self.send("username: ")
username = self.getResponse()
self.send("username: ")
username = self.getResponse()



if loggedin:
    self.netPrint("login successful")
else:
    self.netPrint("login failed")
```

Now, in practice, we can't do this – generator components simply don't work this way. However we can do something close (and maybe closer in future). Rather than saying:

```
yield 1
```

To release control back to the scheduler, we can do this:

```
yield WaitComplete( <some generator> )
```

and allow the scheduler to run that generator before coming back to use. In essence this gives us a function call.

Since we don't need netPrint yet, we'll ignore that for now, and aim for handling request & response cycles. Initially, let's aim for this:

```
yield WaitComplete( self.waitMsg() )
msg = self.getMsg()
```

The changes we need to do this then becomes this:

```
import socket
import Axon
from Axon.Ipc import WaitComplete
from Kamaelia.Chassis.ConnectedServer import ServerCore

class RequestResponseComponent(Axon.Component.component):

    def waitMsg(self):
        while not self.dataReady("inbox"):
            self.pause()
            yield 1

    def getMsg(self):
        return self.recv("inbox")

    def main(self):
        while not self.dataReady("control"):
            for msg in self.Inbox("inbox"):
                self.send(msg, "outbox")
            self.pause()
            yield 1
        self.send(self.recv("control"), "signal")

ServerCore(protocol=RequestResponseComponent,
           socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1),
           port=1600).run()
```

- **File:** BB3.py

So far so good, we can now use this to do a little request response behaviour – we change the main body of the loop from the stub that's copying user input to standard output to something that loops asking for a login & password.

Those changes look like this:

```
import socket
import Axon
from Axon.Ipc import WaitComplete
from Kamaelia.Chassis.ConnectedServer import ServerCore

class RequestResponseComponent(Axon.Component.component):
    def waitMsg(self):
        while not self.dataReady("inbox"):
            self.pause()
            yield 1

    def getMsg(self):
        return self.recv("inbox")

    def main(self):
        while not self.dataReady("control"):
            self.send("login: ", "outbox")
            yield WaitComplete( self.waitMsg() )
            username = self.getMsg()

            self.send("password: ", "outbox")
            yield WaitComplete( self.waitMsg() )
            password= self.getMsg()

            print
            print repr(username), repr(password)

            self.pause()
            yield 1
        self.send(self.recv("control"), "signal")

ServerCore(protocol=RequestResponseComponent,
           socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1),
           port=1600).run()
```

- **File:** BB4.py

When we run this, we see this on the server side:

```
~/Incoming/Europython/Demo> ./BB4.py

'michael\r\n' 'password\r\n'
```

And this on the client side:

```
~/Incoming/Europython> telnet 127.0.0.1 1600
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
login: michael
password: password
```

Now this is getting very close to what we want to write, generally speaking, for this kind of protocol.

But there's an issue. The standard approach for checking for control messages doesn't really work now, because in waitMsg we're not listening for them and they could get missed.

So what we'll do is we'll redefine some behaviour:

- We'll say that waitMsg can return when there's a message on “inbox” or “control”
- We'll change getMsg to check for a message on control, and if it finds one, it'll raise an exception called GotShutdownMessage.
- Then when we do waitMsg/getMsg, we'll break out of the loop as needed.

Then we need to change our main method such that it catches this exception, and if caught sends on the control message, or otherwise sends on a `producerFinished` message. (the message we should do when we're done)

Also, we note that the username & password are raw data, so we need to remember that with anything that's using these values from the network.

All that results in the following changes: (**File:** BB5.py)

```
import socket
import Axon
from Axon.Ipc import WaitComplete
from Kamaelia.Chassis.ConnectedServer import ServerCore

class GotShutdownMessage(Exception):
    pass

class RequestResponseComponent(Axon.Component.component):
    def waitMsg(self):
        while (not self.dataReady("inbox")) and (not self.dataReady("control")):
            self.pause()
            yield 1

    def getMsg(self):
        if self.dataReady("control"):
            raise GotShutdownMessage()
        return self.recv("inbox")

    def main(self):
        try:
            while 1:
                self.send("login: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                username = self.getMsg()

                self.send("password: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                password= self.getMsg()

                print
                print repr(username), repr(password)
                self.pause()
                yield 1
        except GotShutdownMessage:
            self.send(self.recv("control"), "signal")
        self.send( Axon.Ipc.producerFinished(), "signal")

ServerCore(protocol=RequestResponseComponent,
           socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1),
```

```
                port=1600).run()
```

Now before we go any further, let's consider.

- This is essentially the authentication part of the system, so it ought to be named something better. Authenticator is a good name.
- Most of the main bulletin board code is likely to work in a similar fashion, so reusing some of what we have here for that would be useful.
- As a result, factoring out the protocol specific bits from the reusable bits looks like a good idea. We could do this now or later, but for this exercise we'll do it now.

That refactoring then looks like this, note the change to the ServerCore line: (**File:** BB6.py)

```
import socket
import Axon
from Axon.Ipc import WaitComplete
from Kamaelia.Chassis.ConnectedServer import ServerCore

class GotShutdownMessage(Exception):
    pass

class RequestResponseComponent(Axon.Component.component):
    def waitMsg(self):
        while (not self.dataReady("inbox")) and (not self.dataReady("control")):
            self.pause()
            yield 1

    def getMsg(self):
        if self.dataReady("control"):
            raise GotShutdownMessage()
        return self.recv("inbox")

    def main(self):
        self.send("no protocol attached\r\n\r\n")
        self.send( Axon.Ipc.producerFinished(), "signal")
        yield 1

class Authenticator(RequestResponseComponent):
    def main(self):
        try:
            while 1:
                self.send("login: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                username = self.getMsg()

                self.send("password: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                password= self.getMsg()

                print
                print repr(username), repr(password)
                self.pause()
                yield 1
        except GotShutdownMessage:
            self.send(self.recv("control"), "signal")
        self.send( Axon.Ipc.producerFinished(), "signal")

ServerCore(protocol=Authenticator,
           socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1),
```

```
                port=1600).run()
```

We'll make one final change to RequestResponseComponent – specifically to wrap the "checking control and raise an exception" idea, since it's likely that other components may want to check that in places.

We'll also add in the netPrint function we wanted earlier.

This results in that class looking like this:

```
class RequestResponseComponent(Axon.Component.component):
    def waitMsg(self):
        while (not self.dataReady("inbox")) and (not self.dataReady("control")):
            self.pause()
            yield 1

    def checkControl(self):
        if self.dataReady("control"):
            raise GotShutdownMessage()

    def getMsg(self):
        if self.dataReady("control"):
            raise GotShutdownMessage()
        return self.recv("inbox")

    def netPrint(self, arg):
        self.send(arg + "\r\n", "outbox")

    def main(self):
        self.send("no protocol attached\r\n\r\n")
        self.send( Axon.Ipc.producerFinished(), "signal")
        yield 1
```

So let's think – we have two options. We can either continue to make the authenticator more complex, or we can create components for each of these stages:

- Authentication
- User info retrieval
- Main bulletin board logic/ UI
- User state storage

And run them in sequence. If we do that we'll probably want to pass along user state between the components.

Whilst that may sound tricky, since we know that these will be run one after another, we can just pass them all a reference to the same dictionary, and know that it'll be updated by the time it gets to the next component. (This is a bit like WSGI, for those that know WSGI)

In Kamaelia terms, that means rather than creating just an Authenticator to handle the connection, it'd be useful if the ServerCore something like this to happen, and used the resulting component:

```
ConnectionInfo = {}
Seq(
    Authenticator(State = ConnectionInfo),
    UserRetriever(State = ConnectionInfo),
    MessageBoardUI(State = ConnectionInfo),
    StateSaverLogout(State = ConnectionInfo),
)
```

Since this seems like a nice thing to do, rather than make the Authenticator more complex, let's make the changes needed to allow this to happen, even though we only have one of these components so far.

Specifically we're changing the ServerCore line and the initialisation of the Authenticator. In the authenticator, since we can rely on inheritable defaults to add an attribute in the right place, all we need to do is to provide a default value in the class definition. In the ServerCore line, we're changing that to use a factory function instead.

So the changed parts of the code are:

```
from Kamaelia.Chassis.Seq import Seq

class Authenticator(RequestResponseComponent):
    State = {}
    def main(self):
        try:
            while 1:
                self.send("login: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                username = self.getMsg()

                self.send("password: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                password= self.getMsg()

                print
                print repr(username), repr(password)
                self.pause()
                yield 1
        except GotShutdownMessage:
            self.send(self.recv("control"), "signal")
        self.send( Axon.Ipc.producerFinished(), "signal")

def CompositeBulletinBoardProtocol(**argd):
    ConnectionInfo = {}
    ConnectionInfo.update(argd)
    return Seq(
                Authenticator(State = ConnectionInfo),
            )


ServerCore(protocol=CompositeBulletinBoardProtocol,
           socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1),
           port=1600).run()
```

- **File:** BB8.py

You'll also note that rather just passing in an empty dictionary, we've decided to pass in the 4 attributes that ServerCore passes us – peer, peerport, localip, localport – in case the protocols we're writing (and stacking) find that information useful.

Now that we've done that the authenticator has a clear role. After reading in a username/password file it should wait for the user to give them a username & password, and when they succeed, add the username to the State dictionary it's been passed and then exit. This isn't a strong authentication scheme of course, but then what we're

protecting is a simple bulletin board – it doesn't need to be bullet proof.

However, if you made any stronger authentication component, you could swap it in as a replacement for this one, as long as it updated the State's username appropriately. (in much the same way you can have different authentication schemes in WSGI)

The changes necessary to make that happen are:

- Reading the custom password file
- Change the loop to depend on whether the user has logged in or not
- Check user provided data against the password data, remembering that network strings are terminated with "\r\n", so stripping that data.
- We'll make it slightly more pleasant from the user's perspective by adding some blank lines in places as well, using netPrint.
- We no longer want it to terminate the connection, so we stop sending a producerFinished message.

The resulting changed code looks like this:

```
import cjson
...
class Authenticator(RequestResponseComponent):
    State = {}
    def main(self):
        loggedin = False
        try:
            self.netPrint("")
            while not loggedin:
                self.send("login: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                username = self.getMsg()[:-2] # strip \r\n

                self.send("password: ", "outbox")
                yield WaitComplete( self.waitMsg() )
                password= self.getMsg()[:-2] # strip \r\n

                self.netPrint("")
                if users.get(username.lower(), None) == password:
                    self.netPrint("Login Successful")
                    loggedin = True
                else:
                    self.netPrint("Login Failed!")

        except GotShutdownMessage:
            self.send(self.recv("control"), "signal")

        if loggedin:
            self.State["remoteuser"] = username

def CompositeBulletinBoardProtocol(**argd):
    ConnectionInfo = {}
    ConnectionInfo.update(argd)
    return Seq(
                Authenticator(State = ConnectionInfo),
            )
```

```
def readUsers():
    f = open("users.passwd")
    users = f.read()
    f.close()
    users = cjson.decode(users)
    return users

users = readUsers()

ServerCore(protocol=CompositeBulletinBoardProtocol,
           socketOptions=(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1),
           port=1600).run()
```

- **File:** BB9.py

If we run this, the interested user will now see this:

```
~/Incoming/Europython> telnet 127.0.0.1 1600
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

login: michael
password: sparks

Login Failed!
login: michael
password: password

Login Successful
Connection closed by foreign host.
```

We appear to be getting somewhere! We can cross one part of the system off the list:

```
Seq(
    Authenticator(State = ConnectionInfo),
    UserRetriever(State = ConnectionInfo),
    MessageBoardUI(State = ConnectionInfo),
    StateSaverLogout(State = ConnectionInfo),
)
```

Let's now write stubs for the UserRetriever and StateSaverLogout protocol handers. These can be trivial, since they're really for another time if you decided to implement user state (which is very likely).

Those changes look like this:

```
class UserRetriever(RequestResponseComponent):
    State = {}
    def main(self):
        self.netPrint("")
        self.netPrint("Retrieving user data...")
        self.netPrint("")
        yield 1

class StateSaverLogout(RequestResponseComponent):
    State = {}
    def main(self):
        self.netPrint("")
        self.netPrint("Saving user data...")
```

```
            self.netPrint("")
            self.netPrint("Goodbye!")
            self.netPrint("")
            yield 1

    def CompositeBulletinBoardProtocol(**argd):
        ConnectionInfo = {}
        ConnectionInfo.update(argd)
        return Seq(
                    Authenticator(State = ConnectionInfo),
                    UserRetriever(State = ConnectionInfo),
                    StateSaverLogout(State = ConnectionInfo),
                )
```

- **File:** BB10.py

Finally, we need to write the Bulletin Board logic itself. Before we do that however, let's make one final change to RequestResponseComponent so that rather than typing:

- `yield WaitComplete( self.waitMsg() )`

We can write:

- `yield self.waitMsg()`

Before

```
def waitMsg(self):
    while (not self.dataReady("inbox")) and
          (not self.dataReady("control")):
        self.pause()
        yield 1
```

After

```
def waitMsg(self):
  def _waitMsg(self):
    while (not self.dataReady("inbox")) and
          (not self.dataReady("control")):
        self.pause()
        yield 1
  return WaitComplete( _waitMsg(self) )
```

This change makes it it simpler to work with this approach, changing the relevant parts of the Authenticator like this:

```
self.send("login: ", "outbox")
yield self.waitMsg()
username = self.getMsg()[:-2] # strip \r\n

self.send("password: ", "outbox")
yield self.waitMsg()
password= self.getMsg()[:-2] # strip \r\n
```

Which is more or less what we wanted.

5.2 Writing the Bulletin Board UI

We now have a base class that we can use for this, so first of all, let's implement the top level menu system, and use a stub function for the reading messages option. Specifically we want to do this:

- Wait for a user input
- If it's empty, show messages
- If it's an “h” show some help

- If it's a “q”, quit.
- We also want to insert it into the protocol sequence.

Since we have a means of asking the user questions, and prompts, we can write this pretty directly. The changes we make to the program are this:

```
class MessageBoardUI(RequestResponseComponent): # (all new)
    State = {}
    def doMainHelp(self):
        self.netPrint("<return> - browse messages")
        self.netPrint("h – help")
        self.netPrint("q - quit")

    def doMessagesMenu(self, user):
        pass

    def main(self):
        user = self.State.get("remoteuser", "anonymous")
        try:
            self.netPrint("")
            self.netPrint("Hello, "+user)
            while 1:
                self.send("main> ", "outbox")
                yield self.waitMsg()
                command = self.getMsg()[:-2]
                if command == "h":
                    self.doMainHelp()
                if command == "q":
                    break
                if command == "":
                    self.doMessagesMenu(user)
        except GotShutdownMessage:
            self.send(self.recv("control"), "signal")
        yield 1

def CompositeBulletinBoardProtocol(**argd):
    ConnectionInfo = {}
    ConnectionInfo.update(argd)
    return Seq(
                Authenticator(State = ConnectionInfo),
                UserRetriever(State = ConnectionInfo),
                MessageBoardUI(State = ConnectionInfo),
                StateSaverLogout(State = ConnectionInfo),
            )
```

- **File:** BB12.py

The next step is to implement **doMessagesMenu** such that it actually shows messages which can act in a very similar way to this main method.

Since it'll be a generator, we'll need to call it using the `yield WaitComplete(...)` approach, and when that returns, we'll want to check control, in case that was the reason for exiting that method. If it was the case, by doing so we'll re-throw our exception and be well behaved.

The logic for the **doMessagesMenu** method should be this:

- Read in all the unread messages for the user
- While there are any unread messages
    - Tell the user how many unread messages they have & wait for some user input. The prompt should change to show that they're in “messages> “ mode.
    - If they just press return, show the next unread message
    - If they type “h”, show some help, this can be a stub for now
    - If they type “x”, quit

For the moment, we'll implement the unread messages as a stub, and display of messages as a stub.

Let's make those changes and see the difference:

```
class MessageBoardUI(RequestResponseComponent):
    State = {}
    def doMainHelp(self):
        self.netPrint("<return> - browse messages")
        self.netPrint("h – help")
        self.netPrint("q - quit")

    def getUnreadMessages(self, user):
        return ["to be implemented"]

    def displayMessage(self, message):
        self.netPrint(message)

     def doMessagesHelp(self)
          self.netPrint(“to be implemented)

    def doMessagesMenu(self, user):
        try:
            messages = self.getUnreadMessages(user)
            while len(messages) > 0:
                self.netPrint("")
                self.netPrint("You have "+str(len(messages))+" message(s)"\
                                                                 “ waiting")

                self.send("messages> ", "outbox")
                yield self.waitMsg()
                command = self.getMsg()[:-2]

                if command == "":
                    message = messages.pop(0)
                    self.displayMessage(message)

                if command == "x":
                    break

                if command == "h":
                       self.doMessagesHelp()

        except GotShutdownMessage:
            pass # Expect the "caller" to check for control as well
```

```
    def main(self):
        user = self.State.get("remoteuser", "anonymous")
        try:
            self.netPrint("")
            self.netPrint("Hello, "+user)
            while 1:
                self.send("main> ", "outbox")
                yield self.waitMsg()
                command = self.getMsg()[:-2]
                if command == "h":
                    self.doMainHelp()
                if command == "q":
                    break
                if command == "":
                    yield WaitComplete(self.doMessagesMenu(user))
                    self.checkControl()
        except GotShutdownMessage:
            self.send(self.recv("control"), "signal")
        yield 1
```

- **File:** BB13.py

Now we need a source of actual messages. For this we'll wrap up a collection of messages using the concept of a folder. A folder contains files which are numbered, and those files contain serialised JSON objects. We need tools for getting all the messages, and for convenience one at a time as well.

Since this isn't specific to Kamaelia, this is what we're working with:

```
class Folder(object):
    def __init__(self, folder="messages"):
        super(Folder, self).__init__()
        self.folder = folder
        try:
            f = open(self.folder + "/.meta")
            raw_meta = f.read()
            f.close()
            meta = cjson.decode(raw_meta)
        except IOError:
            meta = {"maxid": 0}
        self.meta = meta
    def getMessage(self, messageid):
        try:
            f = open(self.folder + "/" + str(messageid))
            message = f.read()
            f.close()
            message = cjson.decode(message)
            return message
        except IOError:
            return None
    def getMessages(self):
        messages = []
        for i in os.listdir(self.folder):
            if i[:1] == ".":
                continue
            messages.append(self.getMessage(i))
        return messages
```

Given this we can now finish off the Bulletin Board UI, by replacing these three methods:

```
def getUnreadMessages(self, user):
    return ["to be implemented"]

def displayMessage(self, message):
    self.netPrint(message)

def doMessagesHelp(self)
    self.netPrint("to be implemented)
```

... with actual implementations:

```
def getUnreadMessages(self, user):
    X = Folder()
    return X.getMessages()

def displayMessage(self, message):
    self.netPrint("")
    for key in ["message", "date", "from", "to", "subject",]:
        self.netPrint("%s: %s" % (key, message[key] ) )
    if len(message["reply-to"]) > 0:
        self.netPrint("In-Reply-To: "+(", ".join(message["reply-to"]) ) )
    self.netPrint("")
    self.netPrint(message["__body__"])

def doMessagesHelp(self):
    self.netPrint("<return> - next message (exit if on last message)")
    self.netPrint("r - Reply (to be implemented)")
    self.netPrint("d - Delete message (to be implemented)")
    self.netPrint("h — Help")
    self.netPrint("x - eXit to main menu")
```

- **File:** BB14.py

And now when we run the server, and a user logs in, they get the following experience:

```
~/Incoming/Europython> telnet 127.0.0.1 1600
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

login: michael
password: password

Login Successful

Retrieving user data...

Hello, michael
main>

You have 4 message(s) waiting
messages>

message: 1
date: TBD
from: michael
to:
subject: test
```

```
Testing, Testing, 123

You have 3 message(s) waiting
messages>

message: 2
date: TBD
from: michael
to: michael
subject: test
In-Reply-To: 1

Testing, Testing, 123

You have 2 message(s) waiting
messages> h
<return> - next message (exit if on last message)
r - Reply (to be implemented)
d - Delete message (to be implemented)
h — Help
x - eXit to main menu

You have 2 message(s) waiting
messages> x
main> h
<return> - browse messages
h — help
q — quit
main> q

Saving user data...

Goodbye!

Connection closed by foreign host.
```

And there we have it, the essentials of a bulletin board system written using Kamaelia. Now that we have built up these core components, reusing these to build more bulletin board type systems would be trivial.

For nicety, the complete source of the bulletin board is presented next, and includes 1 further change. Another component — `LineOrientedInputBuffer` — has been added, which is used here:

```
def CompositeBulletinBoardProtocol(**argd):
    ConnectionInfo = {}
    ConnectionInfo.update(argd)
    return Pipeline(
              LineOrientedInputBuffer(),
              Seq(
                  Authenticator(State = ConnectionInfo),
                  UserRetriever(State = ConnectionInfo),
                  MessageBoardUI(State = ConnectionInfo),
                  StateSaverLogout(State = ConnectionInfo),
              )
           )
```

This has been added to join partial line fragments together from the network connection, and to only forward complete lines. It has the extra detail in that it only sends a line when it knows the receiver buffer is ready. For this to work we've put this into a Pipeline, but otherwise the protocol stack is unchanged.

And that as they say is that.

5.3 Summary

This section has seen us build a relatively large Kamaelia based system, with incremental growth, with both growing components, and growing systems. Based on this there's a number of possible extensions that could be made.

For example, this blog post details how to stack Kamaelia protocols on top of each other:

- http://yeoldeclue.com/cgi-bin/blog/blog.cgi?rm=viewpost&nodeid=1223342651

In that it has a simple encryption component. To extend this Bulletin Board to use bidirectional encryption, you could do this:

```
def CompositeBulletinBoardProtocol(**argd):
    ConnectionInfo = {}
    ConnectionInfo.update(argd)
    return Pipeline(
              DataDeChunker(),
              Decrypter(),       # Decrypt on the way in
              LineOrientedInputBuffer(),
              Seq(
                  Authenticator(State = ConnectionInfo),
                  UserRetriever(State = ConnectionInfo),
                  MessageBoardUI(State = ConnectionInfo),
                  StateSaverLogout(State = ConnectionInfo),

              )
              Encrypter(),  # Encrypt on the way out
              DataChunker(),
            )
```

You'd then need a corresponding client, which could be something like this:

```
Pipeline(ConsoleReader(""),
              Encrypter(),  # Encrypt on the way out
              DataChunker(),
              TCPClient(ip, port=port),
              DataDeChunker(),
              Decrypter(),       # Decrypt on the way in
              circular = True).run()
```

Other obvious extensions include implementing replies & deletion, additional folders.

Change to using a telnet protocol, to allow a richer UI. Provide API access to folders for a web interface, etc. A pygame based client – using the TextBox components seen earlier, and so on.

If you do extend it, please send feedback to the project!

Bulletin Board, Full source

```
import os
import cjson
import socket
import Axon
from Axon.Ipc import WaitComplete
from Axon.Ipc import producerFinished
from Axon.Component import component
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Chassis.ConnectedServer import \
                                        ServerCore
from Kamaelia.Chassis.Seq import Seq

def readUsers():
    f = open("users.passwd")
    users = f.read()
    f.close()
    users = cjson.decode(users)
    return users

class GotShutdownMessage(Exception):
    pass

class Folder(object):
    def __init__(self, folder="messages"):
        super(Folder, self).__init__()
        self.folder = folder
        try:
            f = open(self.folder + "/.meta")
            raw_meta = f.read()
            f.close()
            meta = cjson.decode(raw_meta)
        except IOError:
            meta = {"maxid": 0}
        self.meta = meta

    def getMessage(self, messageid):
        try:
            f = open(self.folder + "/" + \
                                    str(messageid))
            message = f.read()
            f.close()
            message = cjson.decode(message)
            return message
        except IOError:
            return None

    def getMessages(self):
        messages = []
        for i in os.listdir(self.folder):
            if i[:1] == ".":
                continue
            messages.append(self.getMessage(i))
        return messages
```

```
class LineOrientedInputBuffer(component):
  def main(self):
    linebuffer = []
    gotline = False
    line = ""
    try:
      while 1:
          # Get a line
          while (not gotline):
              if self.dataReady("control"):
                  raise GotShutdownMessage()

              if self.dataReady("inbox"):
                  msg = self.recv("inbox")
                  if "\r\n" in msg:
                     eol_i =  msg.find("\r\n")+2
                     linebuffer.append( msg[:eol_i])

                     line = "".join(linebuffer)
                     gotline = True
                     linebuffer = [ eol_i:] ]
                  else:
                     linebuffer.append( msg )
              yield 1

          if self.dataReady("control"):
              raise GotShutdownMessage()

          # Wait for receiver to be ready
          while len(self.outboxes["outbox"]) > 0:
              self.pause()
              yield 1
              if self.dataReady("control"):
                  raise GotShutdownMessage()

          # Send them the line, then repeat
          self.send(line, "outbox")
          yield 1
          gotline = False
          line = ""

    except GotShutdownMessage:
      self.send(self.recv("control"), "signal")
      return

    self.send(producerFinished(), "signal")


class RequestResponseComponent(component):
    def waitMsg(self):
        def _waitMsg(self):
            while (not self.dataReady("inbox")) and
                  (not self.dataReady("control")):
                self.pause()
                yield 1
        return WaitComplete( _waitMsg(self) )

    def checkControl(self):
        if self.dataReady("control"):
            raise GotShutdownMessage()
```

```
    def getMsg(self):
        if self.dataReady("control"):
            raise GotShutdownMessage()
        return self.recv("inbox")

    def netPrint(self, arg):
        self.send(arg + "\r\n", "outbox")

    def main(self):
        self.send("no protocol attached\r\n\r\n")
        self.send(producerFinished(), "signal")
        yield 1

class Authenticator(RequestResponseComponent):
  State = {}
  def main(self):
    loggedin = False
    try:
      self.netPrint("")
      while not loggedin:
          self.send("login: ", "outbox")
          yield self.waitMsg()
          username = self.getMsg()[:-2] # strip \r\n
          self.send("password: ", "outbox")
          yield self.waitMsg()
          password= self.getMsg()[:-2] # strip \r\n

          self.netPrint("")
          p = users.get(username.lower(), None)
          if p == password:
              self.netPrint("Login Successful")
              loggedin = True
          else:
              self.netPrint("Login Failed!")

    except GotShutdownMessage:
      self.send(self.recv("control"), "signal")

    if loggedin:
      self.State["remoteuser"] = username

class UserRetriever(RequestResponseComponent):
    State = {}
    def main(self):
        self.netPrint("")
        self.netPrint("Retrieving user data...")
        self.netPrint("")
        yield 1

class StateSaverLogout(RequestResponseComponent):
    State = {}
    def main(self):
        self.netPrint("")
        self.netPrint("Saving user data...")
        self.netPrint("")
        self.netPrint("Goodbye!")
        self.netPrint("")
        yield 1
```

```
class MessageBoardUI(RequestResponseComponent):
    State = {}
    def doMainHelp(self):
        self.netPrint("<return> - browse messages")
        self.netPrint("h – help")
        self.netPrint("q - quit")

    def getUnreadMessages(self, user):
        X = Folder()
        return X.getMessages()

    def displayMessage(self, message):
        self.netPrint("")
        for key in ["message", "date", "from",
                    "to", "subject",]:
            self.netPrint("%s: %s" % (key,
                                      message[key]))

        if len(message["reply-to"]) > 0:
            replies = ", ".join(message["reply-to"])
            self.netPrint("In-Reply-To: "+ replies)
        self.netPrint("")
        self.netPrint(message["__body__"])

    def doMessagesHelp(self):
        self.netPrint("<return> - next message")
        self.netPrint("r - Reply (TBD)")
        self.netPrint("d - Delete message (TBD)")
        self.netPrint("h – Help")
        self.netPrint("x - eXit to main menu")

    def doMessagesMenu(self, user):
        try:
            messages = self.getUnreadMessages(user)
            while len(messages) > 0:
                self.netPrint("")
                n = str(len(messages))
                self.netPrint("You have "+ n +
                              " message(s) waiting")

                self.send("messages> ", "outbox")
                yield self.waitMsg()
                command = self.getMsg()[:-2]

                if command == "":
                    message = messages.pop(0)
                    self.displayMessage(message)

                if command == "x":
                    break

                if command == "h":
                      self.doMessagesHelp()

        except GotShutdownMessage:
            pass # Expect the caller to check for
                 #  control as well
```

```
    def main(self):
      user = self.State.get("remoteuser",
                            "anonymous")
      try:
          self.netPrint("")
          self.netPrint("Hello, "+user)
          while 1:
            self.send("main> ", "outbox")
            yield self.waitMsg()
            command = self.getMsg()[:-2]
            if command == "h":
              self.doMainHelp()
            if command == "q":
              break
            if command == "":
              yield \
             WaitComplete(self.doMessagesMenu(user))
              self.checkControl()
      except GotShutdownMessage:
          self.send(self.recv("control"), "signal")
      yield 1

def CompositeBulletinBoardProtocol(**argd):
  ConnectionInfo = {}
  ConnectionInfo.update(argd)
  return Pipeline(
          LineOrientedInputBuffer(),
          Seq(
              Authenticator(State = ConnectionInfo),
              UserRetriever(State = ConnectionInfo),
             MessageBoardUI(State = ConnectionInfo),
           StateSaverLogout(State = ConnectionInfo),
          )
      )

users = readUsers()

ServerCore(protocol=CompositeBulletinBoardProtocol,
           socketOptions=(socket.SOL_SOCKET,
                          socket.SO_REUSEADDR, 1),
           port=1600).run()
```

6 Where next?

Well, this is the end of this tutorial, but hopefully it's shown a handful of scenarios where using Kamaelia can simplify creating solutions to a problem, in ways that are naturally concurrent.

Beyond this, most of Kamaelia has been driven by a variety of real world problems, or needs. As a result, over time there have been a number of documents and presentations done describing these, each of which may be useful in taking your understanding of Kamaelia further.

However none of these are any substitute for writing your own systems. You may find that early systems just use Axon.Handle, or even just embed a networked python interpreter to assist with debugging non-Kamaelia systems.

Getting Help

The Kamelia IRC channel on freenode is generally inhabited by friendly people, welcoming people new to both python and Kamaelia. Details:

- Channel: #kamaelia
- Network: irc.freenode.net
- Logs: http://www.kamaelia.org/logs/ (the use of logs can lead to asynchronous conversations – meaning a reply seconds or hours later)

Kamaelia's email is divided between sourceforge & google groups.

- General Kamaelia conversations : http://groups.google.com/group/kamaelia/
- Version control commits - kamaelia-commits@lists.sourceforge.net https://lists.sourceforge.net/lists/listinfo/kamaelia-commits

Information sources:

- Slideshare presentations: http://www.slideshare.net/tag/kamaelia
- Kamaelia website: http://www.kamaelia.org/
  - Cookbook - http://www.kamaelia.org/Cookbook
  - Component reference - http://www.kamaelia.org/Components
  - Axon reference - http://www.kamaelia.org/Docs/Axon/Axon
  - http://www.kamaelia.org/Developers/
- Michael's Kamaelia (mainly) blog: http://yeoldeclue.com/blog
- BBC Kamaelia Whitepaper: http://www.bbc.co.uk/rd/pubs/whp/whp113.shtml

Code Repository

- http://code.google.com/p/kamaelia/

Code Layout:

- /trunk/Sketches – shared workspace for “work in progress” - contains a lot of code to take inspiration form.

- /trunk/Code/Python – Where most Kamaelia releases come from
  - /trunk/Code/Python/Axon – Where Axon lives
  - /trunk/Code/Python/Kamaelia – Where the bulk of Kamaelia lives
  - /trunk/Code/Python/Apps – Kamaelia based applications
  - /trunk/Code/Python/Bindings – Bindings for various libraries written during the project lifetime.

Getting Kamaelia

See also: http://www.kamaelia.org/GetKamaelia .

Step 1 - Get Kamaelia & Install it

Get the latest version from here:

http://www.kamaelia.org/release/

Install:

```
~/ > tar zxf Kamaelia-0.6.0.tar.gz
~/ > cd Kamaelia-0.6.0/
~/Kamaelia-0.6.0 > sudo python setup.py install
```

Step 2 - Run & Tweak the Examples

Many of these require pygame or similar libraries. You'll find many examples in the examples directory. You will also find a large number of examples in the Kamaelia Cookbook

Step 3 - Start writing your own components

Hopefully this tutorial is a good starting point!

Step 4 - Wire up your component to a new system

The best way to get started here is to look at the examples in the already Kamaelia Cookbook mentioned. You'll be looking at using a number of the components from the reference area.

Step 5 - Build something new

All sorts of possible ideas exist here - as inspiration please look at the GetKamaelia page for a jumping off point.

Acknowledgements

Kamaelia was started by Michael Sparks at BBC Research, and would not be the state it's in without the support of numerous people, including Peter Shelswell, Ian Childs, Brandon Butterworth, Tim Borer, Joseph Lord, Matt Hammond, Tom Loosemore, Matt Biddulph, Gareth Smith, Sylvain Hellegouarch, the many GSOC students who have worked with the project, and the many who have posed hard questions, improving Kamaelia. Matt H & Joseph merit special thanks for the sheer amount of time and effort they have contributed over the years. Kamaelia would not be what it is without them.

Finally, many thanks to my wife Polina for her unwavering patience with me working on this project.