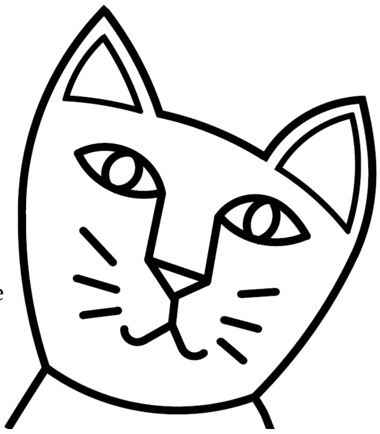
Kamaelia: Concurrency made simpler



We've been told for a long time that concurrency is hard. That concurrency is something the average developer cannot work with reliably.

In the meantime we see hardware going massively parallel. We see 2 cores, 8 cores, 80 core systems becoming reality. We're told that software developers have to get smarter in order to deal with this.

However, we live in a concurrent world and many problems we face day to day are concurrent. When we order coffee in a cafe, when we all read a different book, or read the same article. 100s of drivers in cars on the roads deal with the fact there's something else happening right now. People create mashups between webservices, and they work, even though the webservices are all concurrent.

Our computers' hardware are all highly parallel systems with many thousands of subsystems running concurrently that just work. Furthermore these systems are more often than not these days created using high level languages not dissimilar to a modern programming language. It's all highly concurrent. It all just works. Why?

The reason is because there are some fundamental rules in place that make this happen.

Enter Kamaelia

Kamaelia was originated by BBC Research with these problems and ideas in mind. Kamaelia's core goal is to make the creation of large scale concurrent systems natural, simple and easy to maintain. This has a side effect of making many concurrent systems simpler to write. It also means that a natural solution is often highly concurrent - that is suitable to work naturally on massively multi-core systems.

The reason Kamaelia exists is because the problem domains Kamaelia is designed to work with - networked multimedia delivery of BBC content - are naturally concurrent domains, such as 20 million people all watching the same thing is concurrent. Furthermore, Kamaelia is designed as a toolset to explore solutions to these problems, meaning it is also designed for performance and flexibility.

So, what's the basic approach? Essentially we take the same approach as unix pipeline, but update it for the modern object oriented age.

Kamaelia's ethos (with apologies to Doug McIlroy):
 Write components that do one thing and do it well.
 Write components to work together.
 Write components to handle python objects, because that is a universal interface.

Simplistic Video Serving

So what is a component? Well, it clearly sits in a pipeline or two:

```
# A simple application to stream Dirac
# encoded video from one machine to another
# for playback
Pipeline(
  ReadFileAdaptor( filename = 'video.drc',
                   bitrate = 400000 ),
  SingleServer(  ),
).activate()

Pipeline(
    TCPClient( host = "127.0.0.1",
               port = 1601 ),
    DiracDecoder(  ),
    MessageRateLimit( messages_per_second = 15,
                      buffer = 15 ),
    VideoOverlay(  ),
).run()
```

They sit there, and run concurrently with other components, sending out data. This may initially be just flat data from a file served out of a network connection, but this can be decoded into individual video frames that may need rate limiting before display.

However that doesn't really tell you what a component is.

Making Presentation Tools

I could say, well, rather than just pipelines we can have arbitrary graph shapes, and as a result have Graphlines, which look like this:

```
Graphline(
  NEXT = Button(caption="Next",
             msg="NEXT", position=(72,8)),
  PREV = 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"),
     ("PREV","outbox") : ("CHOOSER","inbox"),
     ("FIRST","outbox") : ("CHOOSER","inbox"),
     ("LAST","outbox") : ("CHOOSER","inbox"),
     ("CHOOSER","outbox") : ("IMAGE","inbox"),
  }
).run()
```

Here you can clearly see that we have an Image component that is controlled by something - a Chooser - that knows about a set of files. You can also see that the Chooser is controlled by some buttons, for first, last, next, previous. I could even say that these are pygame components and look nice, and you can also now clearly see that rather thna stdin/stdout we have inboxes and outboxes.

Whilst this does show you how to build the basics of a presentation tool, this still doesn't tell you what a component looks like inside. What you should be able to see however is that we create interesting systems from composing components into systems.

Look inside a component

Let's look inside a component - let's choose something simple - such as something that reads from the console. If you wanted to naively loop and read from stdin and echo to stdout, you might write something like this:

```
 import sys
 eol = "\n"
 while 1:
   line = raw_input(">>> ")
   line = line + eol
   sys.stdout.write(line)
   sys.stdout.flush()
```

In Kamaelia components we replace stdout with an outbox. We also put this inside the main method of a class that in this case (due to raw_input blocking) is a threadedcomponent:

```
 from Axon.ThreadedComponent import
threadedcomponent
 class ConsoleReader(threadedcomponent):
   def main(self):
      eol = "\n"
      while 1:
         # this blocks so we use a thread
         line = raw_input(">>> ")
         line = line + eol
         self.send(line, "outbox")
```

However, we might also want to output information as well. In which case, we would create a normal non-threaded component for this:

```
class ConsoleEchoer(component):
   def main(self):
      while 1:
          while self.dataReady("inbox"):
              data = self.recv("inbox")
              _sys.stdout.write(str(data))
              _sys.stdout.flush()
          # Note the threaded component doesn't
          # do this 'yield' that's the only
          # difference from a development
          # perspective
          yield 1
```

Now this looks pretty similar - we recieve some data on an inbox, grab it and output it.

We have however also gained a "yield 1" line. This signifies that the main method is a generator. The vast majority of components are of this form. What this means in practice is that this function can be called to gives us something else we can repeatedly call. When we do, it runs until the yield and then returns the value 1 as indicated. We can do this repeatedly.

What this means is that whilst we're writing code that is naturally concurrent it scales well on a single CPU machine because we're effectively running single threaded.

As before we can now plug these two together as follows:

```
 Pipeline( ConsoleReader(),
           ConsoleEchoer(),
         ).run()
```

Why Kamaelia Works

Kamaelia essentially models the real world. The real/physical world is really good at making concurrency work, and it's due to things like it being impossible to be in two places at once.

The metaphor of taking something out of an inbox does a number of things - by definition you conceptually own it and can do what you like with it. When you're happy with your work, you can then send it to an outbox. At that point in time you've literally released the object.

This naturally leads people away from shared state on objects and where there is the act of taking from an inbox means you implicitly have a lock, and when you put something in an outbox you have implicitly released a lock.

This also means that at any point in time any piece of data only has a single reader and single writer. The combination of encouraging safe behaviours, a conceptual model that encourages creating things by composition, and a sprinkling of underlying theory mean that if you still within the rules of no shared data, generally speaking many Kamaelia systems are relatively simple to create.

P2P Video Streaming

Now that we've seen the basics, we can now introduce 2 more concepts - the Backplane and SimpleServer - in the context of creating a basic P2P Video streaming system.

The Backplane was created to make it easier to allow a component publish data to, and to allow components to subscribe to data streams. Anything thing published to a named backplane is recieved by all subscribers to that named backplane. By comparison, SimpleServer is designed to simplify a particular scenario - where everyone connecting to a server gets the same protocol.

The core of any P2P Video streaming system consists of a something that gets the video from somewhere, and sends it to all connected parties, as far as possible. In this case, let's assume that we grab the video from a dedicated video source, and publish it to a local backplane:

```
  from Kamaelia.Util.Backplane import *
  from Kamaelia.Chassis.Pipeline import Pipeline
  from Kamaelia.Internet.TCPClient import \
          TCPClient
  # Create the backplane
  Backplane("VIDEO").activate()
  Pipeline(
     TCPClient("192.168.1.1", 2080),
     PublishTo("VIDEO"),
  ).activate()
```

The flipside of this is that we need to run a server in the same program that allows a client who connects to get a copy of the stream. Now the SimpleServer is designed very much with this usecase in mind. We provide the SimpleServer with a reference to a function that is to be called whenever a new connection starts:

```
  from Kamaelia.Util.Backplane import *
  from Kamaelia.Chassis.ConnectedServer import \
          SimpleServer
  def VideoProtocol():
     return SubscribeTo("VIDEO"),

  SimpleServer(protocol = VideoProtocol,
               port = 1500).run()
```

Let's take a brief second look at that:

```
  Backplane("VIDEO").activate()
  Pipeline(
     TCPClient("192.168.1.1", 2080),
     PublishTo("VIDEO"),
  ).activate()
  def VideoProtocol():
      return SubscribeTo("VIDEO"),
  SimpleServer(protocol = VideoProtocol,
               port=1500).run()
```

That's essentially all that is needed for the bare minimum of a P2P streaming core. Sure you can add on things like buffering, quality of service, and discovery of other hosts, but this is a start.

Now that we have this, we now need a video source. Given the size of video, you're unlikely to be able to serve this over the internet since your upload capacity may not be that high. In which case, lets take a video source from DVB-T. For this we'll have a dedicated server. In the following the tuning parameters suit my local area - you may need to change your tuning!

```
  from Kamaelia.Device.DVB.Core import \
          DVB_Multiplex
  from Kamaelia.Chassis.Pipeline import Pipeline
  from Kamaelia.File.Writing import \
          SimpleFileWriter

  # Multiplex frequency
  freq = 754
  # Programme Stream IDs
  pids = [640,641]

  Backplane("VIDSOURCE").activate()
  Pipeline(
    DVB_Multiplex(freq, pids),
    PublishTo("VIDSOURCE"),
  ).activate()
  def Video(): return SubscribeTo("VIDSOURCE"),
  SimpleServer(protocol = Video,
               port=1500).run()
```

We could then cause the Peer 2 Peer core to point at this, and then the next core can point at that and so on. The issue we have here is how do we play this back?! Well, in this case we don't have to reinvent the wheel. We can making a unix shell out. We can either do this with a standalone player:

```
    Pipeline(
        TCPClient("192.168.1.127", 1500)
        Pipethrough("mplayer -"),
    ).run()
```

Or we can incorporate this into our peer:

```
  Backplane("VIDEO").activate()
  Pipeline(
     TCPClient("192.168.1.1", 1500),
     PublishTo("VIDEO"),
  ).activate()

  Pipeline(
        SubscribeTo("VIDEO"),
        Pipethrough("mplayer -"),
  ).run()

  def VideoProtocol():
      return SubscribeTo("VIDEO")
  SimpleServer(protocol = VideoProtocol,
               port=1500).run()
```

Beyond this, creating a viable service would requiring transcoding, which we won't cover here. (There are examples in the Distribution)

What's been done so far?

Systems we've (BBC Research) built include:

- Kamaelia Compose: A graphical system composition tool (included in the distribution)
- Video Whiteboard: A proof of concept video annotation tool
- A system for producing podcasts of BBC Radio
- Kamaelia Macro: A system for capture and transcoding all BBC TV content
- Kamaelia Whiteboard: A collaborative whiteboarding tool that sends audio in Speex format over the network. This supports ad-hoc mesh creation in a peer to peer fashion since servers are as simple to create as clients. This also supports recording and playback of sessions. (included in the distribution)
- A reframer for mobile video
- A shot change detector for automated frame splitting
- We've also prototyped collaborative community radio
- A simple handwriting recognition tool
- Simple pygame based games
- Presentation tools

During Summer 2006, we also had 4 students from Google's Summer of Code working on the following:

- 1 student in England working on BitTorrent integration. In practice he also implemented a basic scalable webserver and tools for creating web clients, along with simple audio and IRC tools. The bit torrent tools have gone on to see use in the Kamaelia community.

- 1 student in Austria working on OpenGL integration. This resulted in a wide set of components allowing us to take our existing 2D pygame based components and place them as textures onto 3D surfaces – eg video playback on a 3D surface.

- 2 students - one based in India, one in the US working on proof of concept tools for trusted communications. The aim here is for the user of the system to be able to trust the system for things like communication of sensitive private details.

Obtaining Kamaelia

The best way to get hold of Kamaelia is via subversion. You can check out the current working system as follows:

```
   svn co \
   https://svn.sourceforge.net/svnroot/kamaelia/t
runk kamaelia-trunk
```

For the rest of this section, I'll refer to that directory as $RT. Key highlights:

- `$RT/Code/Python/Axon` – Root of all Axon distribution.
- `$RT/Code/Python/Kamaelia` – Root of Kamaelia dist.
- `$RT/Code/Python/Bindings` – Root of bindings for things like Dirac, Vorbis, Linux DVB..

Installing Axon: (as root)

```
  cd $RT/Code/Python/Axon
  python setup.py install
```

Installing Kamaelia (as root)

```
  cd $RT/Code/Python/Kamaelia
  python setup.py install
```

If you want to install, say, the python DVB bindings, then assuming you have Pyrex already installed, you do this: (as root)

```
   cd $RT/Code/Python/Bindings/python-dvb3
   python setup.py install
```

If you don't have pyrex installed, it's well worth checking out and you can get it from here:

- http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

What's Pyrex? Just a really great tool for creating python bindings with!

Summary

Hopefully this tutorial has been useful in introducing Kamaelia, and demystified what Kamaelia is. Throughout the examples we've really focused on composition as a means to structure programs clearly, but as a side effect these systems are also naturally concurrent.

One of the best examples of this is a basic P2P collaborative whiteboarding tool, including audio mixing and retransmission included in the Kamaelia distribution. It is entirely structured in a compositional form, and looks natural for doing so. If you example the number of components however, there are well over a 100 components running efficiently. If rather than using generators we used OS threads or processes, this opens up the door to naturally taking advantage of massively multi-core systems.

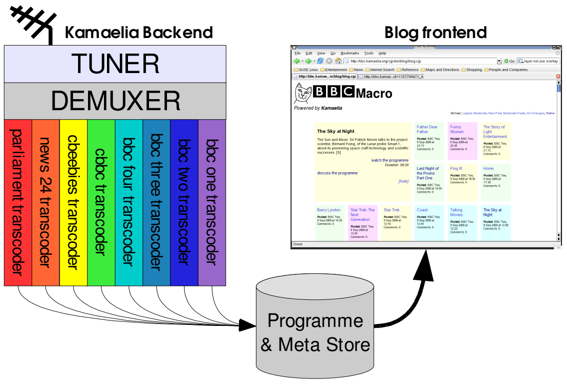
Beyond this we've only scratched the surface of the components available. There's components for scalable TCP & UDP based clients and servers, DVB-T capture & handling, integration with Pymedia, PyGame, OpenGL, BitTorrent, a webserver with component sessions, tools for multicast, Vorbis, Dirac, graph visualisation, system introspection, as well as experimental XML support, among many other things... In /Sketches you'll find a nascent video whiteboard, simple handwriting recognition tools and more.

Kamaelia is as flexible as you need it to be, and has the aim to enable you in the long term to take advantage of multicore systems naturally. The ideas it uses however are carefully designed to be implementable in other languages such as ruby, C, & C++. (Indeed ShedSkin – the python to C++ compiler can already today convert very simplistic Kamaelia systems directly to C++)

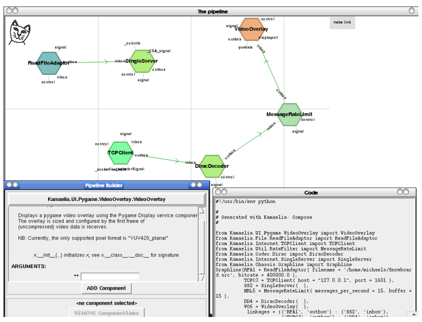
Enjoy!

Pictures that can be included in no specific position:

Kamaelia Macro: A Kamaelia system that captures an transcodes everything broadcast by the BBC as a testbed potential PVR. The implementation naturally follows the diagram below:



Kamaelia Compose: A nascent graphical tool for prototyping Kamaelia systems rapidly.



Displays a pygame video overlay using the Pygame Display service componer
The overlay is sized and configured by the first frame of
(uncompressed) video data is receives.

NB: Currently, the only supported pixel format is "YUV420_planar"

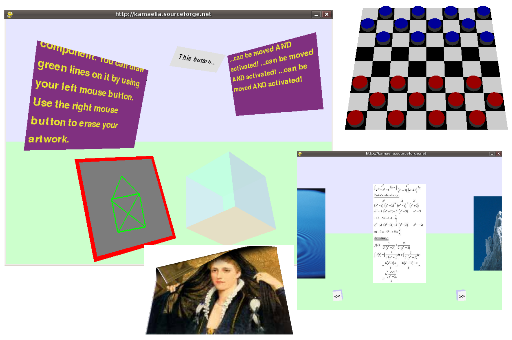
x.__init__(...) initializes x; see x.__class__.__doc__ for signature

```
#!/usr/bin/env python

#
# Generated with Kamaelia: Compose
#

from Kamaelia.UI.Pygame.VideoOverlay import VideoOverlay
from Kamaelia.File.ReadFileAdaptor import ReadFileAdaptor
from Kamaelia.Internet.TCPClient import TCPClient
from Kamaelia.Util.RateFilter import MessageRateLimit
from Kamaelia.Codec.Dirac import DiracDecoder
from Kamaelia.Internet.SingleServer import SingleServer
from Kamaelia.Chassis.Graphline import Graphline
Graphline(RFA1 = ReadFileAdaptor( filename = '/home/michaels/Snowboar
d.src', bitrate = 400000.0 ),
          TCPC3 = TCPClient( host = "127.0.0.1", port = 1601 ),
          SS2 = SingleServer(  ),
          MRL5 = MessageRateLimit( messages_per_second = 15, buffer =
15 ),
          DD4 = DiracDecoder(  ),
          VO6 = VideoOverlay(  ),
           linkages = (('RFA1', 'outbox') : ('SS2', 'inbox'),
```

Google Summer of Code 2006: Tools created included components for working with 3D by integrating OpenGL.



http://kamaelia.sourceforge.net
component. You can draw green lines on it by using your left mouse button. Use the right mouse button to erase your artwork.
This button...
...can be moved AND activated! ...can be moved AND activated! ...can be moved AND activated!

Kamaelia Logo:
It'd be nice if this could be included somewhere: