THE QUEEN'S AWARDS FOR ENTERPRISE 2001

BBC

R&D White Paper

WHP 113

June 2005

Kamaelia:
highly concurrent and network systems tamed

M. Sparks

Research & Development
BRITISH BROADCASTING CORPORATION

Kamaelia: Highly Concurrent and Network Systems Tamed

Michael Sparks

Abstract

Kamaelia is a project aimed at building large scale online media delivery systems for the long term. Large scale media systems are naturally concurrent systems since they assume large numbers of people watching programmes simultaneously. In the long term the systems people will be using to access and serve BBC services online will be also be naturally concurrent. A key aim of Kamaelia is to enable even novice programmers to create scalable and safe concurrent systems, quickly and easily.

Kamaelia provides a tool set for dealing with large scale concurrency in a manner very similar to Unix pipelines, and is based on taking a hardware approach to software construction. This leads naturally to ease of system composition. The tool set includes a wide variety of pre-built components for creating network servers and clients along with components for handling media, interactive systems and text processing.

Kamaelia's architecture operates efficiently on existing single CPU architectures. It also encourages the construction of components that will also take advantage of the naturally parallel mainstream systems being developed by all major hardware vendors.

Kamaelia enables the majority of developers to create safer, more stable high performance systems, rather than a select few. Kamaelia encourages fine grained parallelism without the need for complex state-machine based designs or the overheads of large numbers of parallel threads of execution

This tool set furthers the core goal in Kamaelia - allowing the BBC to experiment with systems for long term, large scale, online media delivery. By designing Kamaelia to lower the barrier to contribution we allow the community to join with us in building these systems.

Additional key words: concurrency, networks, parallelism, architecture, protocol, python

White Papers are distributed freely on request.

Authorisation of the Chief Scientist is required for publication.

BBC Research & Development White Paper WHP 113

Kamaelia: Highly Concurrent and Network Systems Tamed

Michael Sparks

1 Kamaelia

Kamaelia*[1]* is a project aimed at building large scale online media delivery systems for the long term. These systems must be:

- Able to serve a large audience, measured in millions[1], content that the BBC has available, which is measured in petabytes[2]. This requires new approaches to building systems, and a low barrier of entry to broaden perspective.
- Stable, reliable and take a long term view. This is affected by hardware trends and the need to simplify long term maintenance. Taking a short term approach will result in a need for constant redevelopment, and change.

Large scale media systems are naturally concurrent systems since they assume large numbers of people watching programmes simultaneously. These systems do not use concurrency for performance reasons - but rather because it is the natural state for the system.

In the long term the systems people will be using to access and serve BBC services online will be also be naturally concurrent. The advent of multiple core CPUs from Intel *[2]* and AMD *[3]*, and the appearance the highly parallel CELL CPU (*[4]*, *[5]*) in consumer systems marks a key change in the way hardware is being made. This also requires a change to the way software is written. *[6]*

Handling software concurrency safely and correctly is currently a problem that even experienced programmers can find difficult, and hence is a rarely used tool. By contrast, even a child can wire up a battery, lights and motor in parallel. In digital systems, the concept of clocked circuits is used widely to simplify building systems. In modern asynchronous systems, composition and local synchronisation are used to limit system complexity. These tools allow hardware systems to contain millions of components operating concurrently.

Due to exploring new delivery mechanisms, a key aim of Kamaelia is to enable even novice programmers to create scalable and safe concurrent systems, quickly and easily (This lowers barrier to entry to the project - novices seem possibilities, not problems). To achieve this, and noting that hardware system are naturally parallel, we have chosen to try and make building software more like building hardware.

"A key aim of Kamaelia is to enable even novice programmers to create scalable and safe concurrent systems, quickly and easily"

Kamaelia encourages the creation of scalable concurrent systems using modular composition of simple, communicating processing elements. This is similar to systems like Occam *[7]*, Cray-C *[8]* or CSP *[9]* based C derivatives. Unlike these systems, Kamaelia is designed to be simple for novices. Kamaelia works on the assumption that novices want to write simple solutions, allows these to be reused and assists composition when creating simple solutions. This is based on the assumption that doing so will encourage the creation of naturally parallel solutions, encouraging good performance on parallel hardware.

Despite these long term goals, the Kamaelia project sits in the present day: the majority of systems in use have each CPU in a server farm handling as many as thousands of concurrent requests. This means whilst the system is naturally concurrent, the tools we use for managing this concurrency in what is essentially a single threaded environment are crucial to Kamaelia.

[1] Mature audience sizes for BBC content is generally up to 20 million viewers/listeners

[2] If stored at archival quality, the BBC archive requires around 15 Petabytes. (~15,000,000 GB)

As a result, at Kamaelia's core is a concurrency toolkit, focussed mainly on experimenting with network protocols. This toolkit has two key portions - Axon and Kamaelia.

Axon is the core component infrastructure aimed at making concurrent systems easy to work with. It is written in the python programming language, though a simplistic C++ version also exists. In python terms it implements components using simple generators, *[10]* and systems are created by setting up communications between components.

The Kamaelia portion is the larger of the two, which is why it is named after the main project. Kamaelia is a collection of components that use Axon. These range from components for building network servers and clients through to music playback, audio codec decode through to components suitable for viewing network topologies.

This paper will discuss the project's motivations, the current status of the project, the background to Kamaelia's core concurrency technology, provide a simple example of building systems and components using Axon, and finally finish with some possible options for longer term work, and an invitation to work with us!

Python Generators

Python generators provide a kind of function that can return an intermediate result to its caller, but maintaining the function's local state so that the function can be resumed again right where it left off. A very simple example:

```
def fib():
   a, b = 0, 1
   while 1:
      yield b
      a, b = b, a+b
```

When fib() is first invoked, it sets a to 0 and b to 1, then yields b back to its caller. The caller sees 1. When fib is resumed, from its point of view the yield statement is really the same as, say, a print statement: fib continues after the yield with all local state intact. a and b then become 1 and 1, and fib loops back to the yield, yielding 1 to its invoker. The next value yielded is 2, followed by 3, 5, 8 and so on.

-- Adapted from Simple Generators [10]

1.1 Motivations

The core motivations for the Kamaelia project stem from three main points:

- What does the BBC do currently with regard to large scale streaming?
- How might demand change in future?
- ***What happens <u>if</u>...*** *(say) the BBC opened the entire archive online and everyone in the UK had broadband?*

Currently, the BBC streams a large amount of content to the UK audience. This amounts to several million streams per day to tens of thousands of concurrent viewers. Given the three points above it is likely that at some point the BBC may be handling millions of concurrent viewers online.

As a thought experiment, we can also imagine ***what <u>might</u> happen if*** (say) 10 years from now, the BBC opened up the entire archive to the entire UK audience online?

- The Creative Archive *[11]* is not that ambitious - after all it is based around content what we have the rights to, and the BBC does not currently have those rights. This may not always be the case, and given copyright does expire eventually, this will mean that significant chunks will become available in coming years.

Such a scenario is not a total unknown - there are online stores with thousands upon thousands of items of content. People's choices tend to follow a well known curve - called a Zipf distribution *[12]* – also known as a long tailed *[13]* distribution. This name is due to the shape of the graph that you get if you plot the number of requests for an object (e.g. 10 million people watching east enders) vs chart position (e.g. number 10, number 100, number 10000 in the charts).

It turns out that with long tail distributions the bulk of traffic and work is done in this immensely long tail - not in the head. Put another way, the most popular programmes are a fraction of the amount of work you have to do, because given the choice, people tend to choose from the entire range of content available. You ***can*** have 20 million homes watching up to 20 million different things, which is very different problem to 20 million people all watching the same thing on TV!

Other issues arise when you scale up to these levels as well. The only open protocol currently in heavy use for streaming is the Real-time Transport Protocol *[14]*. However, RTP was originally conceived for Internet based Audio/Video conferencing/telephony, which means certain aspects don't scale well for large scale unidirectional streaming. (For example, RTP requires all participants in a session to essentially inform all other participants in a session of their reception quality. For a large enough unidirectional multicast group this becomes either useless, or buggy implementations result in more messages about quality, than the content being served.)

As a result BBC R&D needed a platform for designing, implementing and testing new open standards to scale in this way. Scalability and ability to experiment often conflict, which is why we needed to write our own system. Furthermore large scale means highly parallel, and scalable concurrency often has a high barrier to entry. From our perspective this was a real problem since a high barrier to entry can limit the ability to take in new ideas and limits areas of collaboration. After all those most capable of implementing an idea may not be the same people capable of imagining the idea.

1.2 Status

The Kamaelia project has been released as an open source project licensed under the MPL/GPL/LGPL *[15]* tri-license. This is the same licensing scheme as Dirac *[16]* and is intended to allow the use of Kamaelia by as wide an audience as possible.

Axon is deemed feature stable, and currently stands at version 1.0.3. Axon runs on Windows, Mac OS X, Linux and Series 60 mobile phones.

Kamaelia can already be used for a wide variety of tasks from network servers, through to graphical interactive systems, but we feel has a long way to go before the system can be considered reaching a 1.0 release. As a result the current version number reflects this and currently stands at 0.1.2.

That said, like Axon, Kamaelia runs on Windows, Mac OS X, Linux and Series 60 mobile phones. Kamaelia has also been used to test our ease of use hypothesis on a pre-university trainee, with promising results.

2 Axon's Design Background

Axon forms the core concurrency subsystem in Kamaelia. It provides the smallest useful tool set for concurrency needed for Kamaelia components, but no smaller (By comparison it is possible to produce smaller systems, which are useful for exploring various ideas, but tend to be incomplete).

We decided that Axon should:

- First and foremost use a scalable approach, which is also portable between operating systems.
- Naturally encourage re-use.
- Be simple. That is, be sufficiently simple that novice programmers can learn the system and quickly find it easy to use to produce useful systems. This is partly because we want to experiment with new ides and novices see possibilities, not problems.
- Provide a *safe* environment for building concurrent systems. Users should not have to worry unduly (or preferably at all) about race hazards, and ideally the user should not have to worry about locks. i.e. we wanted a non-locking architecture.

Many of these points will seem contradictory to many used to dealing with scalable concurrent systems - after all even experienced professionals can find scalable concurrency difficult.

I believe that Axon achieves these goals.

2.1 Scaling Concurrency

> *"Threads are for people who can't program state machines."*
> *-- Alan Cox (http://tinyurl.com/a66es)*

Let's start at the beginning - methods for scaling concurrent systems in a portable manner. There are essentially three well understood and commonly used methods for handling concurrency: processes, threads, and "build your own".

Processes and threads are very well known approaches to handling concurrency, and indeed are tools built into the operating system allowing it to provide programs with the illusion of sole control of the system. Threads differ from processes in that they reside within a process, and share data within that process.

Unfortunately the scalability of process and thread based concurrency does not tend to scale equally across different platforms (*[17]*,*[18]*). Furthermore code written using processes and to a lesser extent threads suffers from a key problem - the operating system has no way of determining that these processes and threads are working together towards a common goal, rather than competing. Under heavy loads context switching can also severely hamper scalability using processes or threads, depending on the operating system or even version of operating system.

The majority of large scale network software, and hence large scale concurrent systems tends to use a "build your own" approach to handling concurrency. In practice, this normally means using state machines.

An interesting question arises: what about the people who can't program state machines? This is a serious question – Axon is targeting as wide a group of people as possible and aiming to include novices within that group.

2.2 State Machines

Where a state machine is used to control software, it often uses a state variable to index code fragments associated with the next state. These code fragments may update the state variable to move the state machine into a new state when it is given CPU time again. A state machine is often used to model a piece of sequential processing that can release control half way and be restarted retaining state.

What *really* is the problem with state machines? They are hard to get 100% right even when you're pretty certain you are doing the right thing. This is especially true for novices. Also it is useful to allow people to be able to pick up other people's code and reuse it. This increases the likelihood that someone maintaining code will not be the original author, and unfortunately debugging someone else's state machine is at least twice as hard as designing the state machine in the first place.

There are various frameworks that exist that try to make creating systems like this a lot simpler. However despite best of breed frameworks providing lots of high quality assistance, they still tend to have a high barrier to entry that is often too high for the novice.

2.3 Scalability or ease?

For ease of development an experienced developer would normally choose a process based approach for handling concurrency. For scalability the same developer would choose a state machine approach. However does that developer really have to choose between these two, or is there an alternative with the benefits of both?

Consider:

- *A state machine **is often used to model** a piece of sequential processing that can release control half way and be restarted retaining state*
- *A generator **is** a piece of sequential processing that can release control half way and be restarted retaining state*

Generators in this context are logically equivalent to a software state machine but look single threaded. Users can use the same approach for development as they would with process based code, but potentially with similar performance to a state machine approach.

Using generators, novices can write small pieces of code completely single threaded to test their ideas. After this they can then modify their code to work in a parallel manner by simply adding a few yield statements in the right places, and changing their interaction with other systems and users. This transition from single threaded to working cleanly with the concurrency toolkit is something that can be done slowly and incrementally, often without radical code restructuring.

Generators avoids cross platform issues associated with threads. Running essentially single threaded, when on a single CPU, avoids many of the problems associated with context switching between thousands of processes. Finally because the code is written in essentially the same way as process based systems, novices avoid problems that can arise with state machines:

- There are no shared variables between concurrently executing units. (This is a cause of classic problems in state machine and thread based systems)
- There is a clear execution path through the code, which makes it simpler for a maintainer of code to understand the code.
- Since the concurrency system is entirely under user control, the design of the scheduler can understand that our components co-operate rather than compete. This allows for greater efficiency in the system.

2.4 Concurrency is Easy?

Creating code that can run concurrently is one thing. Actually making systems that are naturally concurrent and easy to use is not considered common.

However, there is one field of IT and computing that uses simple, easy to use concurrency day in day out and does so on a regular basis - Unix systems administration. Many Unix systems administrators would find the following small script simple *[19]* to understand and very similar to the sort of shell script they write on a regular basis:

```
find -type f . |
egrep -v '/build/^./MANIFEST' |
while read i; do
   cp ../Source/$i $i;
done
```

This can be argued to have four processing units, with three concurrent units, one of which consists of two serialised processing units. The 'find' statement runs in concurrently to the 'egrep' statement which runs in parallel to the 'while' statement. The 'while' statement itself is controlled by the 'read' statement, and dependent on the value controls execution of the 'cp' statement.

This is a relatively complex parallel computing task that many systems administrators would find natural to write and would naturally take good advantage of a three core/CPU system. However whilst many systems administrators would not consider themselves the world's best parallel systems programmers, it is probably true that they are the world's most common parallel programmers and highly proficient at it.

Axon models itself on hardware systems that follow a very similar model to Unix pipelines. What are the key characteristics of a Unix pipelines then?

- They're essentially concurrent sequential processes - both in practice, and also in the manner of the classic paper on the topic (CSP). One major limitation of Unix pipelines is that they are essentially linear - largely only allowing data flow in one direction inside the pipeline.
- Items have no concept of what might be next in the pipeline or even if they are inside a pipeline.
- Items in the pipeline simply communicate with three local file handles: stdin, stdout, stderr.

These three points are often considered sufficient for any CSP-style system, however there are a couple of further details which are worth noting:

- How data passes between processes. In Unix pipelines processes can send data to the pipeline whenever they want, and will simply block when the hidden intermediate buffers between processes are full. This allows to operating system to handle race hazards between processes. (Consider a program that produces data faster than another that consumes and uses that data)
- The system environment. This allows processes to place information in a standard place for notification of standard resources. The most commonly used standard resource for example is where to find other commands - specifically the PATH variable in the standard environment. At it's most basic, the system environment forms a basic key/value lookup tool.

3 Axon

This section discusses the key features of Axon, what they do and how to use them. The key classes in Axon are as follows:

- **Component**. A component is a self pausing sequential objects - typically a generator - that sends data to local interfaces
- **Linkage**. A linkage is a facility for joining interfaces, allowing system composition
- **Scheduler**. The scheduler is a mechanism for giving components CPU time. The current scheduler is relatively primitive, however schedulers may also schedule other schedulers however allowing more complex priority mechanisms to be built if desired.
- **Postman**. The postman is a facility for tracking linkages, and handling data transferral. It is expected that the postman will decrease in importance with time and end up simply tracking linkages.
- **Co-ordinating Assistant/Tracker** (cat). The co-ordinating assistant tracker provides environmental facilities for the same reasons as the Unix environment. In practical terms this is similar to a Linda tuple space *[20]*, and could evolve towards a general facility.

3.1 Component

A component in Axon is an instance of a class that has a generator method called "main". The fact that this generator is a method means that it has access to a local object state. This local object state is accessible by code that manipulates the object, forming a mechanism for communicating with the generator.

Axon provides a standardised mechanism for communicating with the generator. Thus, components do not need to share data, other than to hand off ownership of data between themselves. (Much like Unix processes do not share data, but can hand off data between processes by sending data to stdout/stderr)

Component classes are augmented by a list of named inboxes and a list of named outboxes. Inboxes and outboxes form queues for sending data to and from a component. A component only communicates with inboxes and outboxes.

The exception to this is components that communicate with the outside world in some manner - such as reading from files, network sockets, or displaying information. Such components are often named adaptors by convention to simplify identifying such components since testing these can be more complex.

The default set of inboxes are named "inbox" and "control".

- **inbox** is expected to receive bulk data - much like stdin for processes.
- **control** is expected to receive signalling information which may control the component. An example message that may be received on a control inbox is "producerFinished". This allows the component to know that the component sending data on the inbox has finished. (e.g. end of file)

The default set of outboxes are named "outbox" and "signal".

- **outbox** is expected to be sent bulk data - much like stdout for processes.
- **signal** is expected to be sent signalling information that may or may not control another component. A file reader for example may read data from a file and send the data to its "outbox", and when it reaches the end of file may send a "producerFinished" message to "signal".

The simplest example component that is useful is probably as follows:

```
class Echo(component):
   def main(self):
      while 1:
         if self.dataReady("inbox"):
            data = self.recv("inbox")
            self.send(data, "outbox")
         yield 1
```

The logic of this component should be clear. When data is ready in the inbox "inbox", the component takes that data from that inbox. Finally, it sends the data to its outbox "outbox".

Whilst this is a simple "pass through" component it forms an extremely useful tool. Since it is a component, I can also use it as a basic network protocol - one that echoes back to the client what they send. This is extremely useful when testing systems.

3.2 Scheduler

The scheduler periodically provides components with CPU time. Its process of operation is as follows:

- It holds a run queue containing activated components
- It calls the generator for each component sequentially, which may yield a variety of possible values for communication to the scheduler

There are three different return values that hold meaning at present.

Continue Running

- If the value yielded by a component's generator is true then the component is deemed to still be active, but to have relinquished control for some reason. (For example it may be waiting for information from other components, a network socket or the user)

Component Activation

- If the value yielded is a “newComponent” object, the components contained within that object are activated (essentially their main() method is called, and the resulting generator stored in the run queue).

Component Deactivation

- If the value yielded is false, the component is removed from the run queue
- If the component allows an exception to emerge from the generator for whatever reason it is also deemed to have exited.

3.2.1 Scheduling policy

The scheduler in Axon is currently a very primitive linear/round robin scheduler. If the scheduler has 10 things to run, currently it runs them all one after another repeatedly with no change in order. This particular policy is not however guaranteed to stay the same but was chosen for simplicity and fairness.

3.3 Linkages

Components run in parallel and need a mechanism for enabling connections between them. Linkages form this mechanism, and unlike components and the scheduler are traditional passive objects.

Linkages normally join outboxes to inboxes between components. However when dealing with connections to sub-/nested/child components outbox-outbox and inbox-inbox connections make sense. In the case where there is a link between a component and a nested component the parent components inbox (or outbox) associated with the name of that box is prefixed by an underscore by convention to indicate an internal linkage.

Linkages can currently only be created inside a component. Therefore in order to create a pipeline of five components, that pipeline must itself be a component.

Linkages are therefore our mechanism for forming composition of components, and this approach or components not knowing who they are linked to tends to encourage reuse of components.

3.3.1 Linkage Example

Take for example the following simple streaming client:

```
class SimpleStreamingClient(component):
   def main(self):
      client = TCPClient("127.0.0.1", 1500)
      decoder = VorbisDecode()
      player = AOAudioPlaybackAdaptor()
      self.link((client, "outbox"), (decoder, "inbox"))
      self.link((decoder, "outbox"), (player, "inbox"))

      self.addChildren(decoder, player, client)
      yield newComponent(decoder, player, client)
      while 1:
         self.pause()
         yield 1
```

Since it is constructed from a number of components, it too must be a component. The component creates three components - one for connecting to the server, one for decoding and one for playback. These are then linked together, and activated by yielding them back to the scheduler. After that, the only purpose of this component is to acts as a shell. As a result, it returns control to the scheduler and hints (self.pause()) that it no longer needs any CPU time.

Similarly a server capable of serving an ad-hoc file to this simple client might look like this:

```
def AdHocFileProtocolHandler(filename):
   class klass(Kamaelia.ReadFileAdaptor.ReadFileAdaptor):
      def __init__(self, *argv, **argd):
         super(klass, self).__init__(filename, readmode="bitrate",
                                     bitrate=400000)
   return klass


class SimpleStreamingServer(component):

   def main(self):
      server = SimpleServer(protocol=AdHocFileProtocolHandler("foo.ogg"),
                     port=clientServerTestPort)
      self.addChildren(server)
      yield _Axon.Ipc.newComponent(server)
      while 1:
         self.pause()
         yield 1
```

What this example does is to use the generic simple server component for the bulk of the network handling. When a client connects the server needs to be able to create components for handling that connection. Specifically these components handle the actual protocol for speaking to the client over the network connection. As a result the protocol argument takes a class reference to allow protocol objects to be created.

In this particular case, the protocol class is created dynamically and will always create a component that reads the file “foo.ogg”, and sends it at a constant bit rate to it's outbox.

What this means is in practice when a client connects to the port the server is listening on, the specified protocol component will be created and receive network data on it's inbox and any data it sends to it's outbox will be sent to the client. This enables a wide range of behaviours to take place and to be tested in complete isolation from the network.

3.3.2 Linkage Example: Reuse

Suppose that rather than connecting to a TCP based server, the requirements change to needing a multicast based service. Rather than using TCP Server, a multicast component is used instead, decoded and played back. The following code snippet shows in bold the required changes to the program:

```
class SimpleStreamingClient(component):
   def main(self):
      client = Multicast_transceiver("0.0.0.0", 1600, "224.168.2.9", 0)
      adapt = detuple(1)
      decoder = VorbisDecode()
      player = AOAudioPlaybackAdaptor()
      self.link((client, "outbox"), (adapt, "inbox"))
      self.link((adapt, "outbox"), (decoder, "inbox"))
      self.link((decoder, "outbox"), (player, "inbox"))

      self.addChildren(decoder, adapt, player, client)
      yield newComponent(decoder, adapt, player, client)
      while 1:
         self.pause()
         yield 1
```

Essentially I made the following changes:

- The system uses a Multicast_transceiver component Instead of a TCPClient,
- The multicast_transceiver component sends tuples of (sender, data) to its output. This is largely because data received over a multicast socket is a UDP socket which isn't connected.
- In order to extract the data from this tuple as part of the decode chain I simply chain the data from the Multicast_transceiver into a detuple component.
- The rest of the decode chain then remains the same

As a result, this allows reuse of the bulk of the streaming server client, and also leaves the decoding and playback components unchanged.

3.4 Co-ordinating Assistant Tracker

The co-ordinating assistant tracker (CAT) tracks two distinct types of values in an Axon system: values and services.

3.4.1 Values

The cat tracks values associated with keys. This provides facilities similar to environment variables (or a Linda tuple space) and is intended to facilitate the collection of statistics in network servers. At present not many components make use of this facility, but experience with similar systems imply that this is needed.

3.4.2 Services

A service is a name given to a (component, inbox) tuple. This allows a component to advertise a service upon which it may expect values to be sent. This facility is deliberately modelled on the concept of named ports in AmigaOS and can be viewed as similar to services in a rendevous/zero conf environment.

Components can contact request access to a service via the CAT by name and then make use of the service. If the component requires communication back to itself from the service, it can send the service a service.

Whilst sounding hideously recursive and theoretical, services were borne out of a very specific real world use case:

- Suppose there are 2 server components in a system.
- Both of these will have a number of active sockets, which are traditionally managed via a select call. In Axon this maps directly to a selector component that simply tracks sockets and sends components managing those sockets "data ready" messages in order activate them.
- Clearly if there are 2 (or more) servers in a system they could all share the same selector component.
- At this point in time the problem arises of how a developer will find this single selector component. Axon could enforce this on users of the selector component using a singleton pattern, but this can prevent components from being generic. The alternative is to allow a selector to register somewhere that it exists and that it is willing to handle arbitrary connections.

Services provide a way of allowing components to share active functionality:

- As a server starts up, it can then check for the existence of a selector service. Since none exists, it creates a selector component. That selector component registers itself as a service with the CAT.
- The next server that starts up can also check for the existence of a selector. This server finds a selector service. It can then send that service a message stating "wake me up on this inbox if you find activity on this socket". It does this, effectively, by sending a reference to the socket, itself and the inbox name to the selector. (This is why Axon 's documentation talks about sending a service to a service)

Note that the only one of these two components needs to know how to create a selector component. It is possible that as time progresses that this aspect of services may expand.

4 Component Creation Overview

This section describes how to create a basic component that deals with one of the most common network formats, and translates this from textual form to a python dictionary.

MIME/RFC2822 type objects are common in network protocols, used in email, web, Usenet systems and several others.

```
From: michaels@rd.bbc.co.uk
Subject: MimeDictComponent

And so we see a sample mime dict...
```

This forms a collection of serialised key/value pairs. In python the closest natural structure that is key/value based is the "dict" type. This leads us to the idea that if I had a MIME Dict component I could place this between network connections handling components and other components.

This "MIME Dict" component should:

- Accept dict like objects, but translates them to MIME-like messages
- Accept MIME-like messages and converts them to dicts.

4.1 MimeDictComponent

The MimeDictComponent is a real component in Kamaelia. This section largely discusses how it was written, rather than laboriously going through code (The code is always available in CVS and releases).

The basic approach was as follows. The core functionality was written first. That is I first created a component that subclassed dict. The __str__ method was replaced with a custom implementation that returned an RFC2822/MIME style message. After that was written, a static method "fromString" was added to the MimeDict class that could accept an RFC2822/MIME style message.

A further key point is that all this code was written entirely test first, with no special considerations.

Only after the basic class was able to perform the desired transformation did I consider the specific interface that the component may need. It's worth noting that should this interface have required changes to the MimeDict class these would have been added as tests to the test suite for MimeDict first and implemented independently of the component system.

When designing the MimeDictComponent I needed to decide what interface I desired from the system. Specifically this relates to the inboxes and outboxes on the component, and what data I expect to send/receive to/from the component. I decided upon the following interface:

- **control** - from which the component may receive a "shutdown" message
- **signal** – to which the component will send "shutdown" messages
- **demarshall** - an inbox to which you send strings for turning into dicts
- **demarshalled** - an outbox which spits out parsed strings as dicts
- **marshall** - an inbox to which you send objects for turning into strings
- **marshalled** - an outbox which spits out translated dicts as strings

An irony here though is that it turned out to be simpler to create a generic marshalling component instead. During creation the user passes over a reference to the MimeDict. Then the generic marshalling code would use the facilities of the MimeDict class to perform the actual transformations detailed above.

For example, the main loop of the component ended up looking like this:

```
while 1:
   self.pause()
   if self.dataReady("control"):
      data = self.recv("control")
      if isinstance(data, Axon.Ipc.producerFinished):
         self.send(Axon.Ipc.producerFinished(), "signal")
         return
   if self.dataReady("marshall"):
      data = self.recv("marshall")
      self.send(str(data),"marshalled")
   if self.dataReady("demarshall"):
      data = self.recv("demarshall")
      self.send(self.klass.fromString(data), "demarshalled")
   yield 1
```

For the specific marshaller that I wanted I could then take a traditional subclassing style of approach:

```
class MimeDictMarshaller(MarshallComponent):
   def __init__(self, *argv, **argd):
      super(MimeDictMarshaller, self).__init__(MimeDict, *argv, **argd)
```

Or I could choose a class decoration approach:

```
def MarshallerFactory(klass):
   class newclass(MarshalComponent):
      def __init__(self, *argv, **argd):
         super(newclass, self).__init__(klass, *argv, **argd)
   return newclass

MimeDictMarshaller = MarshallerFactory(MimeDict)
```

It is interesting to note that this approach naturally encouraged the creation of a generic component that allows for greater use of non-component based code in the component framework.

4.2 Summary

This has been a relatively high level brief overview of how to go about designing and implementing a component. There is a longer tutorial on the Kamaelia website*[1]* revolving around creating a multicast transceiver. Again, that code was designed and written using much the same approach:

- Don't worry about concurrency, write single threaded
- When the code works, then convert to components
- Change control methods into inboxes/outboxes

5 Ease of Use, Efficiency and Integration Issues

5.1 Ease of use Hypothesis

The Kamaelia project hypothesises that using simple components communicating to/from local inboxes/outboxes composed into systems may be simpler to work with when building concurrent systems than traditional approaches.

The Kamaelia Project was recently able to test this hypothesis on a pre-university trainee, who was happy to let me describe him as a novice programmer. For the rest of the document he is referred to as "C" . Prior to joining our group "C" had done A-Level computer studies. This involved a small amount of Visual Basic programming and creating a small Access database.

This trainee had a three month placement within our group which involved the following:

- Started off learning python and axon (2 weeks)
- Created a "learning system" based around parsing a Shakespeare play:
  - Performs filtering, character identification, demultiplexing, etc
  - Used pygame for display, stopped short of using pyTTS... (a text to speech library)
- As well as working on his main project

The project "C" worked on is not normally something you would not normally give to a novice programmer with "C"'s background.

“C” was asked to created a simplistic low bandwidth video streamer, based on a scalable architecture. The server would have an MPEG video, and take a frame as JPEG every n seconds. This is sent to the client over a framing protocol “C” designed and implemented. The client then displays the images as they arrive. The client systems "C" would be required to implement would be both PC based for testing, with the primary target platform being series 60 mobiles.

The idea is this simulates previewing PVR content on a mobile.

The project was successful, "C" achieved the goals, and wrote components satisfying every part of that of the description. For network handling "C" was able to use the "SimpleServer" and simple "TCPClient" components. It is also interesting to note that rather than finding this a frustrating experience (given his background) that he found the experience fun.

It would be interesting to retry this experiment, both with Axon/Kamaelia and other frameworks for clients and servers. (After all this only provides a single data point)

5.2 CSP vs State Machines

From a performance perspective the question arises: is a CSP based approach - such as that taken by Axon - better or worse than a state machine style system? At present I would suggest that neither is better or worse than the other - at least theoretically.

After all, state machine systems often have intermediate buffers (even single variables) for hand off between states and state machines. In many respects this is akin to outboxes and inboxes. If they are collapsed into one, as planned, this is probably as efficient as traditional frameworks.

Some preliminary tests have been performed using a simplified version of the component system and it does tend to imply that collapsing outbox/inbox pairs into one is effectively as efficient as the non-componentised system. The difference of course is that the componentised system is easy to reuse.

5.3 Integration with other systems

Kamaelia does not exist in a void. Having the ability to assimilate code and functionality from other systems easily and quickly is something needed by Kamaelia. Given that python generators do not exist in other languages, a mechanism is needed for interacting with traditional procedural languages.

Providing this mechanism allows components to be written in languages other than python providing an incremental optimisation step where needed.

The mechanism provided is a default main generator which unless overridden by a subclass calls three default callback:

- initialiseComponent()
- mainBody()
- closeDownComponent()

The reason for just these three callbacks is because almost every program that exists essentially matches the following underlying structure:

perform some initialisation

loop until some condition is true:

    perform the bulk of the work of the program (normally)

perform shutdown code

Sometimes different parts of this template program may be empty. For example crash based systems are designed never to shutdown. They are designed only to recover from crashes. Likewise a simple "hello world" program generally has an empty loop and no shutdown. Nevertheless many programs can be brought down to these three phases of a program.

This logic, is managed by the default main generator, which looks like this:

```
def main(self):
   result = self.initialiseComponent()
   if not result: result = 1
   yield result
   while(result):
      result = self.mainBody()
      if result:
         yield result
   yield self.closeDownComponent()
```

Note that any returned value from a callback is yielded back to the scheduler allowing components using the callback form to start new components as required.

The secondary advantage of also having this callback mechanism in addition to the generator approach is that some programmers simply find a callback approach easier to work with. As a result having this call back approach as well as the generator approach is extremely useful.

6 Futures

Currently the version number of Kamaelia, whilst stable and useful stands at 0.1.2. I feel there are a lot of enhancements to come to both Axon and Kamaelia before Kamaelia reaches a 1.0 release (A label indicating feature stablity and a level of completeness). This section details a few of the areas which the Kamaelia Project intends to extend Kamaelia and Axon into.

6.1 Axon for C++

The approach and benefits found in Axon/Kamaelia should not be simply limited to those using the python language. As a result, when adding features the consideration "how could this be implemented in other languages?" has always been high on the list. With regard to python's generators, one can use Duff's device to implement generator like functionality in C++.

As a proof of concept, I produced a simple and *naive* translation of a "mini-axon" system into C++. This is includes a version of C++ generators. The following is the C++ version of the Echo component from section 3.1:

```
struct Echo : public SimpleComponent {

    mystring msg;
    Echo()  {     };
    ~Echo () {     };

    virtual int next() {
    GENERATOR_CODE_START
        while(1) {
           if dataReady() {
              msg = recv("inbox");
              send(&msg);
           }
           YIELD(1);
        };
    GENERATOR_CODE_END
    };
};
```

One interesting target system for a C++ based system using Axon would be IBM's CELL processor, which is most notably known for powering Sony's PS3. It current appears that Axon's data-flow architecture would be a very good match for the data-flow architecture inside the CELL CPU.

6.2 Axon Optimisations, Updates

Axon currently focusses on correctness, safety and ease of use over speed. One example of this is that linkages are currently registered with a Postman who then manages deliveries from outboxes to inboxes. This process involves copying of (references to) data from outboxes to inboxes.

One optimisation that can be made here for example is to collapse inboxes into outboxes. Some initial tests with a model implementation of a subset of Axon suggest that this does

indeed bring performance benefits, and is a worthwhile optimisation. When the issues in collapsing inboxes into outboxes are sufficiently well understood they will be merged into Axon.

At present some parts of Axon's API, and creation of pipelines can be clunky, and so means to make this clearer and easier to read are under investigation.

6.3 Automated Component Distribution over Clusters

Since components in Axon only communicate with other components over local interfaces, they have no way of knowing whether the component they're communicating with is on the same machine or not. This makes the creation and testing of protocols fairly simple. To develop a new protocol implementation you can test the client protocol handler and server protocol handlers by joining their in/out-boxes together. When that works you simply place the client and server protocol handlers in generic client and server code on different machines.

This opens up the possibility of running entire axon systems over clusters of machines transparently. Axon is designed of course to work well with a single CPU system, but could be used to scale over clusters. One tool that would be needed here is opaque component creation. A method that may be of use here is to extend the services model of the co-ordinating assistant tracker.

Automated component distribution over clusters would also assist in naturally taking advantage of systems like the CELL CPU.

6.4 Kamaelia Component Repository

A component system is only as useful as the components available in that system. As a result the main focus of Kamaelia at present is on creating new components useful in specific real world systems that BBC R&D needs. However a standard repository (but perhaps not centralised) for components such that people can find new components quickly and easily to solve their task would magnify the value of Kamaelia/Axon significantly.

At present the Kamaelia project itself is a component repository, but the project team would be extremely interested in proposals that encourage sharing and reuse between groups.

6.5 More Concurrency Tools

The current releases of Kamaelia only include support for components written as either generators or using the callback mechanism. The CVS version has support for components written to use threading instead of generators, and these are focussed around dealing with objects that can only be used in a blocking manner. (The impetus for this was due to the socket module in the Nokia Series 60 implementation of python not supporting non-blocking sockets)

Other concurrency tools that would be interesting to provide based components for include:

- Operating system level processes
- Greenlets*[21]*- essentially an implementation of coroutines for python
- Twisted*[22]* objects. (There is no reason why Axon/Kamaelia should compete with alternative frameworks – collaboration and co-operation is more useful)

6.6 Extensions to Kamaelia: More protocols, experimental servers

Since the purpose is to scale delivery of BBC content, and there appear to be issues with some of the existing open protocols for streaming the best way to demonstrate this is to implement these protocols and show what happens.

Initially implemention is focussing on RTSP/RTP for streaming Ogg Vorbis, before working on other protocols. As well as this support for experimentation with different protocols for peer to peer and collaborative client hub systems is vital.

7 Collaboration

If you're interested in working with us (BBC R&D), please do: join the mailing list, download, the code, play with it. If you want to write docs, code, components, or constructive feedback, please do. If you want to explore some of the more "future" oriented ideas rather than work on the stabilising the code base, you're more than welcome.

If you find the code looks vaguely interesting, please use it and give us feedback. We're very open to exploring changes to the system and willing to give people CVS commit access in order to try their ideas, within some fair and free bounds.

Anyone working with alternative frameworks for single threaded concurrency is very welcome to come and criticise and suggest new ideas. If you would like to integrate our system with yours, please let us know, we're very interested in hearing your thoughts.

Contacts, project blog:

- michaels@rd.bbc.co.uk, kamaelia-list@lists.sourceforge.net
- http://kamaelia.sourceforge.net/cgi-bin/blog/blog.cgi

8 Summary

Kamaelia provides a tool set for dealing with large scale concurrency in a manner very similar to Unix pipelines, and is based on taking a hardware approach to software construction. This leads naturally to ease of composition. Also the tool set includes a wide variety of pre-built components for creating network servers and clients along with components for handling media, interactive systems and text processing.

Kamaelia's approach of writing components single threaded, and then making simple modifications to allow the code to be used in a parallel manner, simplifies integrating existing code into the Kamaelia framework. It simply becomes a matter of replacing core looping control constructs with python generators, and adding some communications. This ability to assimilate existing code into Kamaelia systems with minimal changes, I feel is a great strength.

The tool set has an architecture that operates efficiently on existing architectures, but encourages the construction of components that will also take advantage of the naturally parallel mainstream systems being developed by all major hardware vendors. It is our hope that this will result in safer, more stable high performance systems, and allow these systems to be developed by the majority of developers rather than a select few.

This tool set also furthers the goal of allowing the BBC to experiment with systems allowing the delivery of BBC content to the user where they want it, when they want it, and however they want it. Designing the tool set to lower the barrier to contribution invites the community to join with the BBC in building these systems. The sooner we, as a community, build new systems allowing the delivery of all BBC content online, the sooner we as a community will benefit.

9 References

[1] The Kamaelia Project Homepage: http://kamaelia.sourceforge.net/

[2] *Multi-core: Intel's New Processor Architecture Architecture Explained*, Andrew Binstock, Intel, http://www.intel.com/cd/ids/developer/asmo-na/eng/211198.htm

[3] *Multi-Core Processors - The Next Evolution In Computing*, AMD Inc, AMD Multi-Core Technology White Paper, http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf

[4] *CELL: A New Platform For Digital Entertainment*, Dominic Mallinson, Mark DeLoura, Sony Computer Entertainment US Research and Development, http://www.research.scea.com/research/html/CellGDC05/index.html

[5] Index of IBM Technical Reports on the Cell, IBM, http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell

[6] *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, Herb Sutter, Dr. Dobb's Journal, 30(3), March 2005, http://www.gotw.ca/publications/concurrency-ddj.htm

[7] *Programming in Occam*, Geraint Jones, ISBN: 0137297734

[8] *Cray C and C++ Reference Manual*, Cray Inc, http://www.cray.com/craydoc/manuals/S-2179-50/html-S-2179-50/S-2179-50-toc.html

[9] *Communicating Sequential Processes*, Tony Hoare, 1985, ISBN: 0131532898. Also available online in PDF form: http://www.usingcsp.com/cspbook.pdf

[10] *"Simple Generators"*, Neil Schemenauer, Tim Peters, Magnus Lie Hetland http://www.python.org/peps/pep-0255.html

[11] The home page of the Creative Archive. http://creativearchive.bbc.co.uk/

[12] *Zipf's law (definition)*, National Institute of Standards and Technology, http://www.nist.gov/dads/HTML/zipfslaw.html

[13] *The Long Tail*, Chris Anderson, Wired 12.10, October 2004, http://www.wired.com/wired/archive/12.10/tail.html

[14] *RFC 3550: RTP: A Transport Protocol for Real-Time Applications*, Schulzrinne, Casner, Frederick, Jacobson, http://www.ietf.org/rfc/rfc3550.txt

[15] *Kamaelia's License* file containing the MPL/GPL/LGPL tri-license, http://kamaelia.sourceforge.net/COPYING

[16] The BBC's Dirac project, http://dirac.sourceforge.net/

[17] *High Performance Server Architecture*, Jeff Darcy, http://pl.atyp.us/content/tech/servers

[18] *The C10K problem*, Dan Kegel, http://www.kegel.com/c10k.html

[19] *Teach Yourself Shell Programming in 24 Hours*, Sriranga Veeraraghavan, ISBN: 0672323583

[20] *Generative communication in Linda*, David Gelernter, ACM Transactions on Programming Languages and Systems (TOPLAS), archive Volume 7 , Issue 1 (January 1985), ISSN:0164-0925

[21] py.magic.greenlet: Lightweight concurrent programming, Armin Rigo, http://codespeak.net/py/current/doc/greenlet.html

[22] Twisted, An event-driven networking framework written in Python, Glyph Lefkowitz, Sean Riley, and a team of dozens, http://twistedmatrix.com/developers/credits