Walkthrough: Introspecting Running Kamaelia Systems

The Axon Visualiser, is a useful tool for understanding what's happening inside a Kamaelia system. This allows you to look inside running systems since Kamaelia components only communicate via inboxes and outboxes, and also make that information (deliberately) easily accessible for introspection, The next step is to take that introspection information and visualise it. Therefore the Axon Visualiser has 2 halves – an introspector and a visualiser. These are two discrete sets of components.

In this tutorial, let's assume we want to build a simple chat system, and along the way want to see what's happening inside. Since the focus of this tutorial is to talk about the introspector, we'll build this up slowly, one stub at a time.

Table of Contents

Getting Started – Visualising a Console Echoing App....2
Introspecting and Visualising an Echo Server....4
Watching Client Connection, Disconnection and Cleanup....7
- Server Startup....7
- Client Connections....7
- Client Disconnections....8
- What did we see from client connection/disconnection for the Echo Server?....9

Visualising a Minimal Chat Server....10
- Minimal Chat Server Code....10
- Understanding the Minimal Chat Server....11
- Adding Introspection to Visualise the Minimal Chat Server....12
- Watching Client Connections to the Minimal Chat Server....14
- Watching Client Disconnections from the Minimal Chat Server....15

Walkthrough: Introspecting Running Kamaelia Systems

The Axon Visualiser, is a useful tool for understanding what's happening inside a Kamaelia system. This allows you to look inside running systems since Kamaelia components only communicate via inboxes and outboxes, and also make that information (deliberately) easily accessible for introspection, The next step is to take that introspection information and visualise it. Therefore the Axon Visualiser has 2 halves – an introspector and a visualiser. These are two discrete sets of components.

In this tutorial, let's assume we want to build a simple chat system, and along the way want to see what's happening inside. Since the focus of this tutorial is to talk about the introspector, we'll build this up slowly, one stub at a time.

Our initial stub will take what you type on the console and echo it back to you. We'll then introspect that to show the basics of introspection & visualisation, and then turn that into a trial/simplistic chat system.

NOTE: The Axon Visualiser has limitations, isn't perfect, but it is useful.

Getting Started – Visualising a Console Echoing App

First of all, here's our basic command line tool that echoes back anything we type.

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Console import *

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

This isn't particularly interesting, since when we run this and type random stuff to it we get this:

```
$ ./CommandLineEchoExample.py
>>> hello
>>> hello
world
>>> world
game
>>> game
over
>>> over
```

Now, to Introspect and Visualise this, we need two things 1) to run an AxonVisualiser server in the background 2) To add an Introspector component to our CommandLineEchoExample that connects to this server and sends it the introspection information.

Let's modify the program we're writing first. We need to import the Introspector, and the TCP Client, and then create a pipeline from one to the other which connects to our AxonVisualiser on a given port - in this case port 1600.

```
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Console import *

# ------- START OF ADDITION
from Kamaelia.Util.Introspector import Introspector
from Kamaelia.Internet.TCPClient import TCPClient
from Kamaelia.Util.PureTransformer import PureTransformer

Pipeline(
    Introspector(),
    PureTransformer(lambda x: x.encode("utf8")),
    TCPClient("127.0.0.1", 1600),
).activate()
# ------- END OF ADDITION

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

Note: the PureTransformer is required to transform the plain text from the Introspector to bytes as required by network sockets.
As you can see above those additions are made pretty literally.

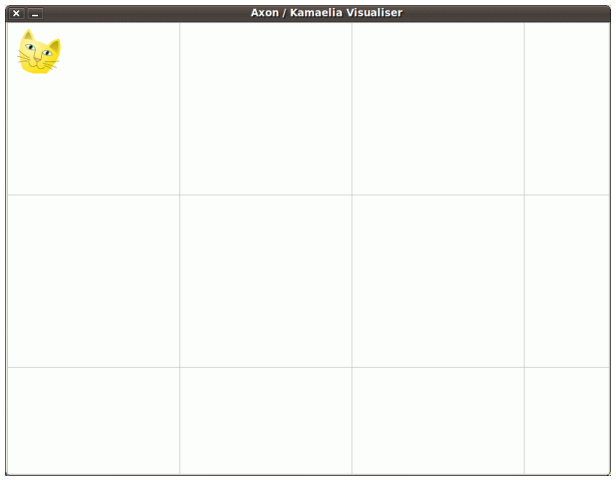
Now, before we run this, we need to run the AxonVisualiser in the background. This sits inside the Kamaelia/Tools directory in the distribution:

```
~/code.google/kamaelia/trunk/Code/Python/Kamaelia$ ls

CHANGELOG  Docs/  Examples/  Kamaelia/  README  Test/  Tools/
MANIFEST.in  setup.py

~/code.google/kamaelia/trunk/Code/Python/Kamaelia$ cd Tools
~/code.google/kamaelia/trunk/Code/Python/Kamaelia/Tools$ ls
DocGen/          VideoShotChangeDetector/  axonshell.py       Compose.py
VideoPlayer.py  VideoReframer/            kamaelia_logo.png  AxonVisualiser.py
Show.py     WhatIsShow.show/
```

We want to run it, listening on port 1600:

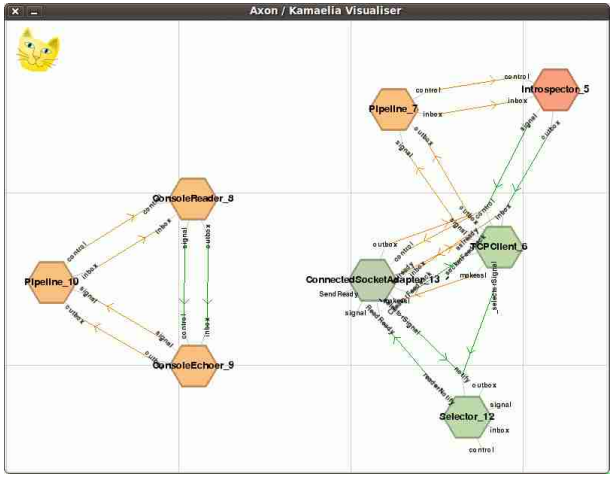
```
$ ./AxonVisualiser.py --port=1600
```

This causes the AxonVisualiser to open and be ready to accept connections describing a running system:



We can then run our Introspecting Command Line Echo Example in another shell, and see the pretty result:

```
$ ./IntrospectingCommandLineEchoExample.py
```



In the above picture, you'll see two distinct groups of components. On the left, you'll see 3 components:

```
Pipeline_10
ConsoleReader_8
ConsoleEchoer_9
```

These 3 relate to the code:

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

On the right hand side you'll see a collection of 6 components (diagram not updated yet):

```
Pipeline_7
Introspector_5
TCPClient_6
ConnectedSocketAdapter_13
Selector_12
PureTransformer_XXX
```

These relate to the code:

```
Pipeline(
    Introspector(),
    PureTransformer(lambda x: x.encode("utf8")),
    TCPClient("127.0.0.1", 1600),
).activate()
```

The reason why the ConnectedSocketAdapter (CSA) and Selector components exist is because TCPClient forced them into existance. The connected socket adapter is the actual component that wraps the actual TCP socket. The selector is a component that wraps a standard "select" call and wakes the connected socket adapter when there's work that needs doing. The TCPClient component therefore just acts as a happy go between that handles connection set up and tear down.

Now, it would be nice if instead of showing a *flat* diagram, that the visualiser showed things nested in the same way as the code - that is (for example) double clicking on a pipeline showed you what was inside - but it doesn't.

That's why this code...

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

... looks like a triangle when visualised rather than like a pipeline. In fact, if you look at the arrows coming in/going out of the pipeline you'll see that the links made are as follows:

```
Pipeline_10 inbox  --> ConsoleReader_8 inbox (orange due to being pass through)
Pipeline_10 control --> ConsoleReader_8 control (orange due to being pass through)
ConsoleReader_8 outbox --> ConsoleEchoer_9 inbox
ConsoleReader_8 signal --> ConsoleEchoer_9 control
```

```
ConsoleEchoer_9 outbox --> Pipeline_10 outbox (orange due to being pass through)
ConsoleEchoer_9 signal --> Pipeline_10 signal (orange due to being pass through)
```

This is exactly what you'd expect the links to be.

So, whilst the visualisation may not be exactly what you expected, it has the benefit of showing you everything that's going on. The downsides are that this may not be expected and that for large scale systems it's less practical.

Introspecting and Visualising an Echo Server

OK, moving on. Let's turn this into a server that simply echoes back everything we type at it. This is simple enough - we replace this code:

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

With this code:

```
from Kamaelia.Protocol.EchoProtocol import EchoProtocol
from Kamaelia.Chassis.ConnectedServer import FastRestartServer

FastRestartServer(protocol=EchoProtocol, port=1500).run()
```

If we run this by itself first of all:

```
$ ./EchoServer.py
```

We can indeed connect to it and see it works:

```
$ telnet 127.0.0.1 1500
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
sdfsdf
sdfsdf
sdsdfsdf
sdsdfsdf
^]
telnet> close
Connection closed.
```

As before we can embed the introspection code into this server as follows:

```
from Kamaelia.Protocol.EchoProtocol import EchoProtocol
from Kamaelia.Chassis.ConnectedServer import FastRestartServer

# ------- START OF ADDITION
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Introspector import Introspector
from Kamaelia.Internet.TCPClient import TCPClient
from Kamaelia.Util.PureTransformer import PureTransformer

Pipeline(
    Introspector(),
    PureTransformer(lambda x: x.encode("utf8")),
    TCPClient("127.0.0.1", 1600),
).activate()
# ------- END OF ADDITION

FastRestartServer(protocol=EchoProtocol, port=1500).run()
```

Now, once again we start the AxonVisualiser first:

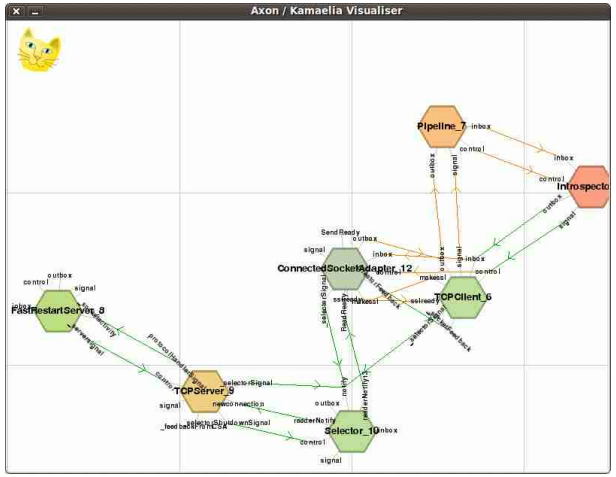
```
$ ./AxonVisualiser.py --port=1600
```

An empty visualiser then appears waiting for a connection.

We then start our IntrospectingEchoServer in another shell:

```
$ ./IntrospectingEchoServer.py
```

... and the visualiser updates as follows:



You'll note from running this that you can drag components and inboxes around, so I've done this here to show that the internal structure of the visualiser has remained the same. In particular, on the right hand side, you'll see the following components as before:

```
Pipeline_7
Introspector_5
TCPClient_6
ConnectedSocketAdapter_12
Selector_10
```

(The numbers are added to ensure uniqueness when introspecting, and will often remain the same within a program, but aren't guaranteed to do so. They're much like a process id)

The left handside again shows the code we're introspecting. In this case it shows two components:

```
FastRestartServer_8
TCPServer_8
```

As with TCPClient, FastRestartServer creates the TCPServer component behind the scenes. TCPServer fundamentally wraps the following behaviour:

- Create a TCP socket to listen on.
- Wait for connections
- When a connection happens
  - Accept the connection
  - Create a ConnectedSocketAdapter to handle the connection
  - Send the ConnectedSocketAdapter to whomever is managing the TCPServer ie out the TCPServer's protocolHandlerSignal outbox (which you can see on the diagram)
- Go back to waiting for connections.

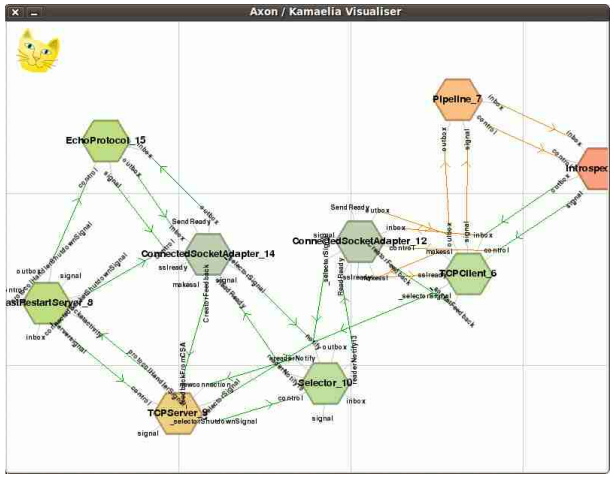
Since a Selector component can handle waking up the TCPServer component, the TCPServer component looks for a Selector service, and wires itself into it. As a result, this is why the TCPServer component has links to the Selector component on the diagram. (This avoids the need for threads, or multiple selectors)

So we now have a server waiting for a connection, and we're looking inside it as it runs.

Let's see what happens when we now connect to the server:
(before we send it any data)

```
$ telnet 127.0.0.1 1500
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
```

Now when we do this, the introspector obviously picks this up, and sends the information to the visualiser, which itself updates accordingly:



You'll note two new additions to the diagram:

```
EchoProtocol_15
ConnectedSocketAdapter_14
```

As you can see, as noted, when the TCPServer accepted the connection, it created a ConnectedSocketAdapter. It then sent (effectively) this to the FastRestartServer for it to figure out what to do. In the meantime, the ConnectedSocketAdapter wired itself into the Selector so that it can read and write to the network connection when appropriate.

When the FastRestartServer receives the ConnectedSocketAdapter it needs to create a protocol handler to talk to the connected socket adapter. Remember that our server code looks like this:

```
FastRestartServer(protocol=EchoProtocol, port=1500).run()
```

As a result it calls the callback provided - in this case EchoProtocol() which results in an EchoProtocol component being created - specifically EchoProtocol_15 in this case, which it wires into ConnectedSocketAdapter_14 .

In particular the FastRestartServer creates the following links between them that you can see on the diagram:

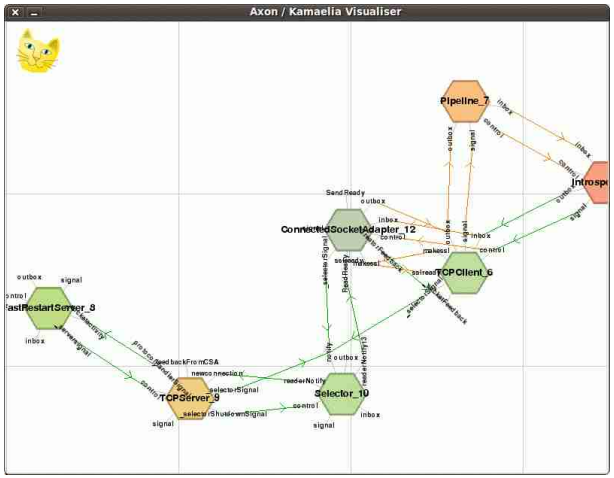
```
ConnectedSocketAdapter_14 outbox -> EchoProtocol_15 inbox
EchoProtocol_15 outbox --> ConnectedSocketAdapter_14 inbox
EchoProtocol_15 signal --> ConnectedSocketAdapter_14 control
```

ie stuff that the ConnectedSocketAdapter receives from the socket ends up on the EchoProtocol's "inbox", and every thing the EchoProtocol sends out its

"outbox" gets sent to the network connection. (ConnectedSocketAdapter handles sending and receiving data to/from TCP sockets after all)

If the client disconnects, then the ConnectedSocketAdapter sends out shutdown message its signal outbox, which the FastRestartServer receives, and passes on to the EchoProtocol (as well as releasing various links etc) and both the ConnectedSocketAdapter and EchoProtocol components shutdown.

Not only that, the Introspector notices, and passes on appropriate messages to the server, and you see them disappear from the visualisation:



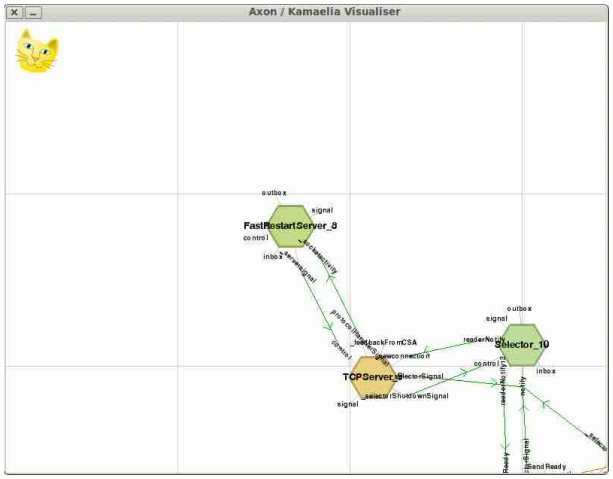
(Incidentally, whilst writing this tutorial, I noted that the EchoProtocol *wasn't* disappearing as expected, which suggested there was a latent bug in EchoProtocol. The rewritten version on /trunk now does not have this bug. If your EchoProtocol *doesn't* disappear, you may want to update your version of Kamaelia)

Watching Client Connection, Disconnection and Cleanup

At this stage it's useful to show what happens when several clients connect and stay connected, one after another. In the snapshots below, I've pushed the visualisation of the Introspector connecting to the visualiser off the side of the screen.

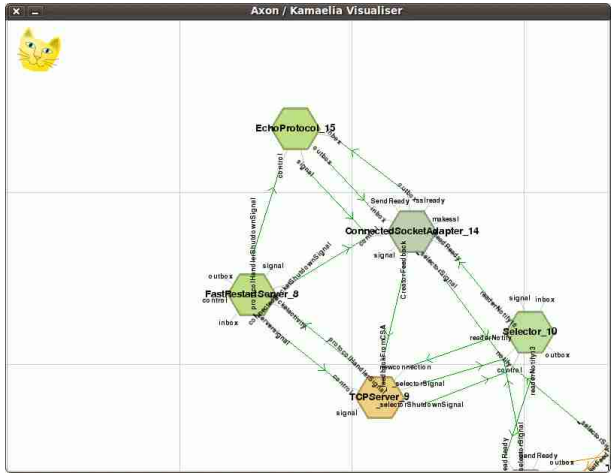
Server Startup

Waiting for connections

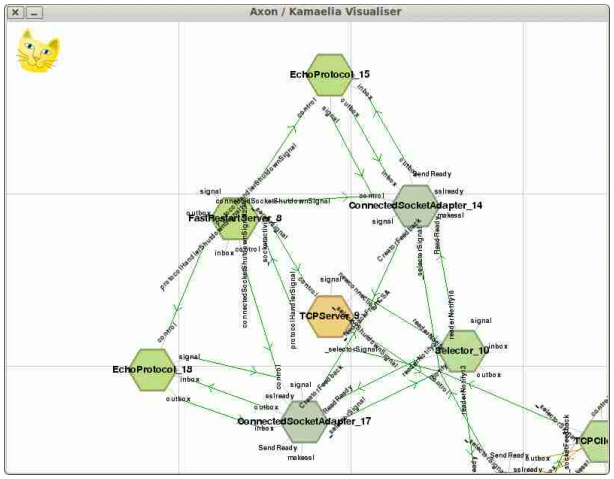


Client Connections

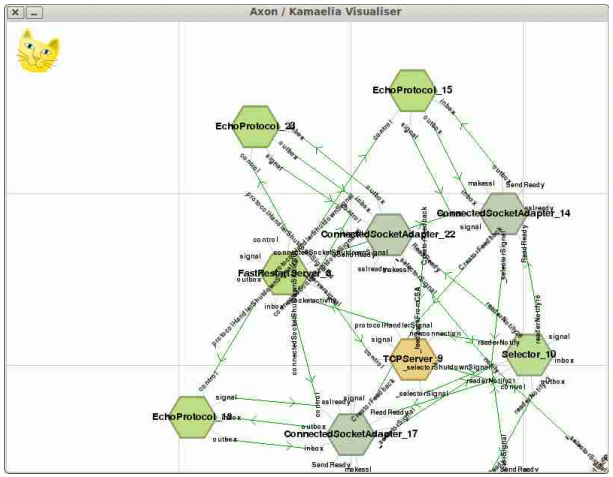
First one client connects:



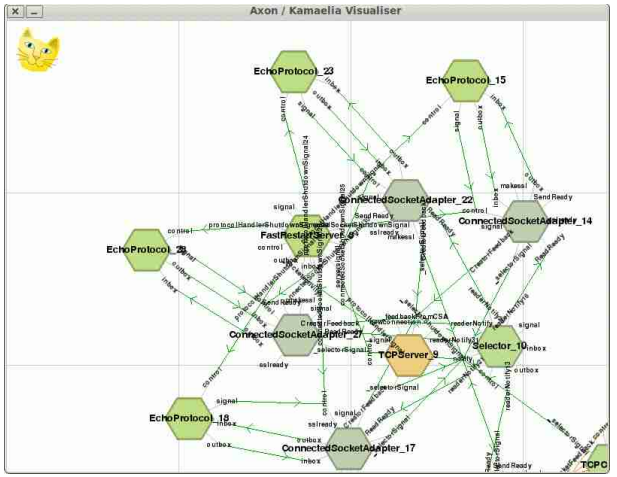
Then another:



And another:

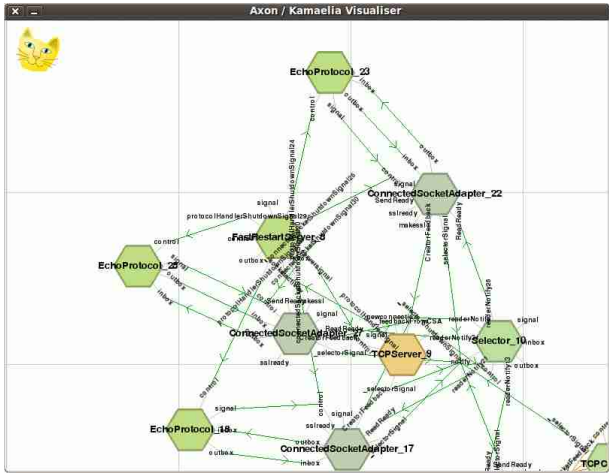


And another:

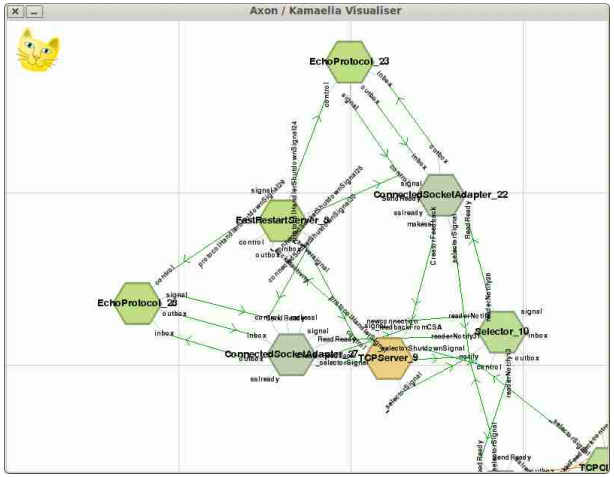


Client Disconnections

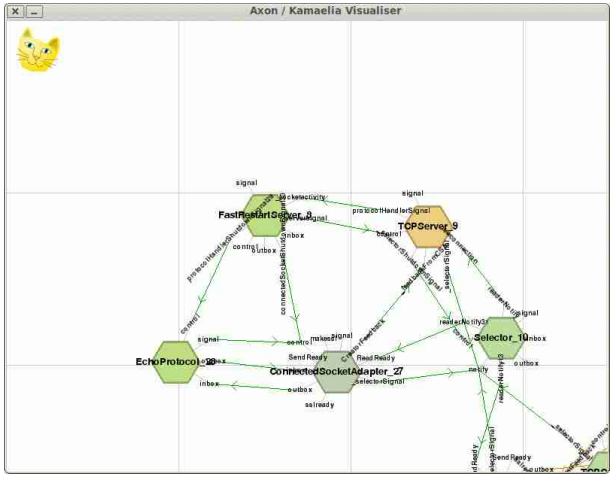
And watching clients disconnect - first client 1:



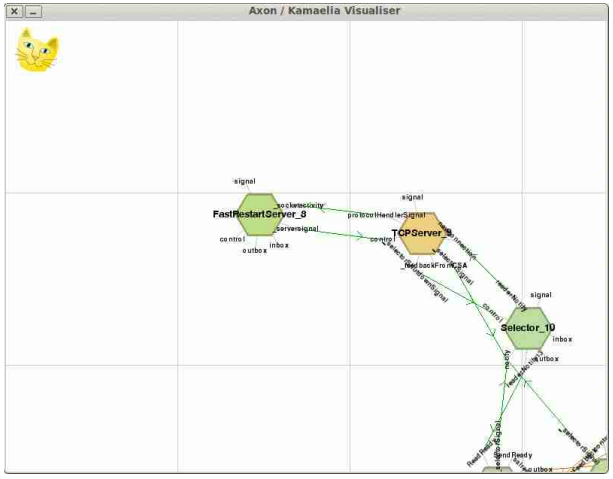
Then client 2:



Then client 3:



Then client 4:



What did we see from client connection/disconnection for the Echo Server?

It may be clear from these what's going on inside the server, but to be clear:

- First the FastRestartServer component is started.
- FastRestartServer starts a TCPServer component.
- The TCPServer component starts the Selector, and waits for activity

Then for each connection:

- A client connects, and the Selector tells the TCPServer there's a connection waiting
- The TCP Server accepts the connection, creates a ConnectedSocketAdapter to handle it, and tells the FastRestartServer that there's a new connection, passing it the ConnectedSocketAdapter .
- Meanwhile the ConnectedSocketAdapter makes links with the Selector to be woken up when there's something to do.
- The FastRestartServer takes the ConnectedSocketAdapter, creates an EchoProtocol component to handle it, wires things up and goes back to sleep.

For each client connection:

- The opposite happens, ConnectedSocketAdapters and EchoProtocols are unwired and exit.

Not only that in Kamaelia's case this isn't metaphorical nor architectural diagrams that bear little relation to the code, this is precisely the way the

components are wired together in practice.

Visualising a Minimal Chat Server

Let's now create a minimal chat server and look inside that. The easiest way to do this is to use a backplane, to make an “EchoEveryone” protocol for our server. That is everything you type is sent to everyone connected to the server.

Minimal Chat Server Code

The server code looks like this:

```
import Axon
from Kamaelia.Chassis.ConnectedServer import FastRestartServer
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Backplane import Backplane, PublishTo, SubscribeTo
from Kamaelia.Util.PureTransformer import PureTransformer

Backplane("CHAT_ONE").activate()

def EchoEveryone(**kwargs):
        peer = str(kwargs.get("peer", "<>"))
        peerport = str(kwargs.get("peerport", "<>"))
        return Pipeline(
                PureTransformer(lambda x: x.decode("utf8")),
                PureTransformer(lambda x: "%s:%s says %s" % (peer,peerport,x)),
                PublishTo("CHAT_ONE"),
                # ------------
                SubscribeTo("CHAT_ONE"),
                PureTransformer(lambda x: x.encode("utf8")),
            )
FastRestartServer(protocol=EchoEveryone, port=1500).run()
```

If you run this, you will indeed see that it's a very simplistic chat server.

First start the server:

```
$ ./ChatServer.py
```

Then start two clients. Here's the connection transcript from client 1:

```
$ telnet 127.0.0.1 1500
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Hello from connection 1
127.0.0.1:38084 says Hello from connection 1
127.0.0.1:38085 says Hello from Connection 2
Woo
127.0.0.1:38084 says Woo
127.0.0.1:38085 says Woo Hoo
```

And from client 2:

```
$ telnet 127.0.0.1 1500
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
127.0.0.1:38084 says Hello from connection 1
Hello from Connection 2
127.0.0.1:38085 says Hello from Connection 2
127.0.0.1:38084 says Woo
Woo Hoo
127.0.0.1:38085 says Woo Hoo
```

As you can see, messages from client 1 / connection 1 are tagged with "127.0.0.1:38084", whereas client 2 / connection 2 are tagged with "127.0.0.1:38085".

Understanding the Minimal Chat Server

So let's walk through this. First a bunch of imports:

```
import Axon
from Kamaelia.Chassis.ConnectedServer import FastRestartServer
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Backplane import Backplane, PublishTo, SubscribeTo
from Kamaelia.Util.PureTransformer import PureTransformer
```

Then we start our Backplane. In this case we call it CHAT_ONE. This is a symbolic name so that the PublishTo and SubscribeTo components can find it later. Under the hood Backplane is in fact a wrapper around another component - a PlugSplitter - which we'll see later (in essence Backplane is syntactic sugar).

```
Backplane("CHAT_ONE").activate()
```

We then define our protocol. In this case, we grab the peer ip and peer port number from the arguments passed to EchoEveryone when the server starts up. The actual protocol is handled by the Pipeline that's returned from the function.

```
def EchoEveryone(**kwargs):
        peer = str(kwargs.get("peer", "<>"))
        peerport = str(kwargs.get("peerport", "<>"))
        return Pipeline(
                PureTransformer(lambda x: x.decode("utf8")),
                PureTransformer(lambda x: "%s:%s says %s" % (peer,peerport,x)),
                PublishTo("CHAT_ONE"),
                # ------------
                SubscribeTo("CHAT_ONE"),
                PureTransformer(lambda x: x.encode("utf8")),
            )
```

(NOTE: Again the PureTransformer of decode/encode are to transform text +to/from bytes type)

ie this:

```
            Pipeline(
```

```
        PureTransformer(lambda x: x.decode("utf8")),
        PureTransformer(lambda x: "%s:%s says %s" % (peer,peerport,x)),
        PublishTo("CHAT_ONE"),
        # ------------
        SubscribeTo("CHAT_ONE"),
        PureTransformer(lambda x: x.encode("utf8")),
    )
```

This looks a little odd, so let's break it down. Anything that comes into the pipeline comes from the network connection, and anything that comes out of the Pipeline gets sent to the connection. ie

```
Pipeline(
       < data sent to the connection needs to be processed on the way in>
       #---------------------
       < data that sent by the final component goes to the connection>
)
```

In this particular case, it's also useful to know that PublishTo does NOT pass data on through the connection. That means we can view this pipeline as two halves. The inbound half handles data from connection:

```
Pipeline(
    PureTransformer(lambda x: x.decode("utf8")),
    PureTransformer(lambda x: "%s:%s says %s" % (peer,peerport,x)),
    PublishTo("CHAT_ONE"),
    # ------------
    ....
)
```

The outbound half handles data sent out the last component's outbox gets sent to the connection:

```
Pipeline(
    ....
    ....
    # ------------
    SubscribeTo("CHAT_ONE"),
    PureTransformer(lambda x: x.encode("utf8")),
)
```

Or overall:

```
Pipeline(
    PureTransformer(lambda x: x.decode("utf8")),
    PureTransformer(lambda x: "%s:%s says %s" % (peer,peerport,x)),
    PublishTo("CHAT_ONE"),
    # ------------
    SubscribeTo("CHAT_ONE"),
    PureTransformer(lambda x: x.encode("utf8")),
)
```

The inbound half takes any messages sent to the network and tags the data with peer & peerport, and then publishes that to CHAT_ONE.

The outbound half subscribes to any messages sent to the CHAT_ONE backplane and sends them to the network connection.

Finally we create and run the server:

```
FastRestartServer(protocol=EchoEveryone, port=1500).run()
```

OK, so that's the logic of a basic chat server - clients connect, any data they send is tagged with their IP and port number, and then pubished to a backplane. Any data sent to the backplane is also subcribed to by all clients and sent as a result to all connected clients.

Adding Introspection to Visualise the Minimal Chat Server

What does this actually look like when running? Well, with the caveat that this is where we start really hitting limitations of the visualiser...

Well to know that we need to modify the server to be an introspecting one:

```
import Axon
from Kamaelia.Chassis.ConnectedServer import FastRestartServer
from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Backplane import Backplane, PublishTo, SubscribeTo
from Kamaelia.Util.PureTransformer import PureTransformer

# --- START OF CODE FRAGMENT NEEDED TO CONNECT TO INTROSPECTOR ---
from Kamaelia.Util.Introspector import Introspector
from Kamaelia.Internet.TCPClient import TCPClient

Pipeline(
    Introspector(),
    TCPClient("127.0.0.1", 1600),
).activate()
# --- END OF CODE FRAGMENT NEEDED TO CONNECT TO INTROSPECTOR ---

Backplane("CHAT_ONE").activate()
def EchoEveryone(**kwargs):
        peer = str(kwargs.get("peer", "<>"))
        peerport = str(kwargs.get("peerport", "<>"))
        return Pipeline(
                PureTransformer(lambda x: x.decode("utf8")),
                PureTransformer(lambda x: "%s:%s says %s" % (peer,peerport,x)),
                PublishTo("CHAT_ONE"),
                # ------------
                SubscribeTo("CHAT_ONE"),
                PureTransformer(lambda x: x.encode("utf8")),
            )

FastRestartServer(protocol=EchoEveryone, port=1500).run()
```

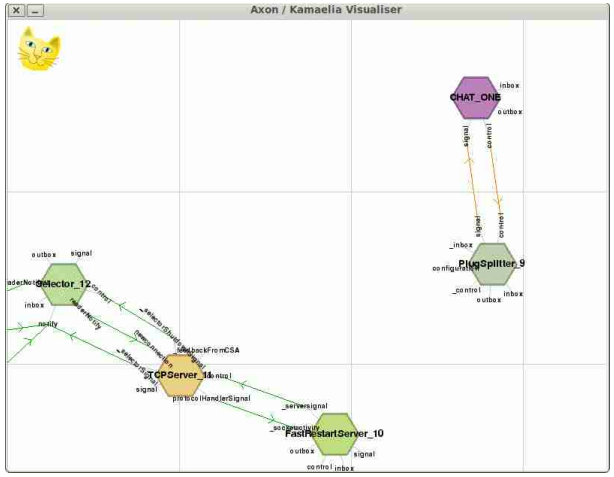
Then we can fire up the AxonVisualiser as before:

```
$ ./AxonVisualiser.py --port=1600
```

Next up, we start our introspecting chat server:

```
$ ./IntrospectingChatServer.py
```

The chat server starts up, and the introspector sends information about the structure to the visualiser, so we see this picture:



As before, I've pushed the structure of the Introspection code itself off screen. The thing to note about this picture is as before, we have the following 3 components handling the server logic:

- FastRestartServer - Connects application logic to connected socket adapters.
- TCPServer - Waits for connections, accepts them and creates connected socket adapters.
- Selector - wakes up things when there's interesting activity on sockets

On the right handside there's two components:

- CHAT_ONE - this is the Backplane component that has given itself a custom name. ... like this:

```
class Backplane(Axon.Component.component):
    ... snip ...
    def __init__(self, name):
        ... snip ...
        self.name = name
```

- PlugSplitter.

What's going on here then? Well, the PlugSplitter works like this:

> The PlugSplitter component splits a data source, fanning it out to multiple destinations. The Plug component allows you to easily 'plug' a destination into the splitter. ... You can add and remove destinations manually [by] sending an addsink(...) message to the "configuration" inbox of [PlugSplitter] [or] sending a removesink(...) message to the "configuration" inbox of the [PlugSplitter]
> http://www.kamaelia.org/Components/pydoc/Kamaelia.Util.Splitter.html

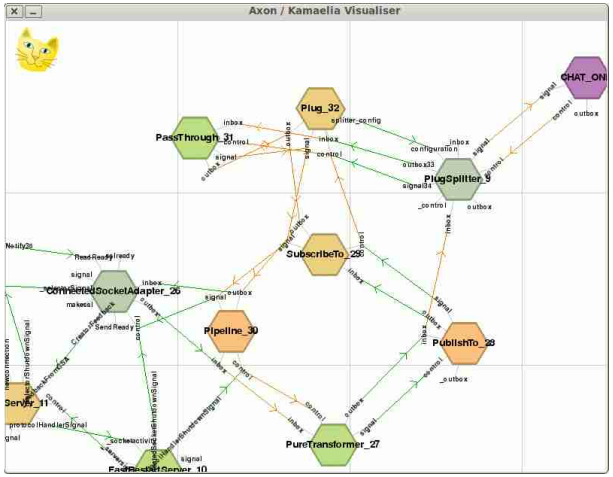
In practice, there's also a "Plug" component that plugs into the plug splitter.

So, the Backplane component essentially advertises the existence of the PlugSplitter using the name CHAT_ONE. In practice this is actually 2 services - one for handling inbound data, and one to enable Plugs to wire themselves in to gain a copy of data.

Watching Client Connections to the Minimal Chat Server

What happens when a client connects ?

Well, as we can see from the AxonVisualiser, this is how things change:



Before we dive into the diagram, remember that this is what the FastRestartServer created to handle the connection when the client connected:

```
Pipeline(
    PureTransformer(lambda x: x.decode("utf8")),
    PureTransformer(lambda x: "%s:%s says %s" % (peer,peerport,x)),
    PublishTo("CHAT_ONE"),
    # ------------
    SubscribeTo("CHAT_ONE"),
    PureTransformer(lambda x: x.encode("utf8")),
)
```

As noted before the visualiser isn't perfect - rather than pipelines being visualised as shapes containing things (ie nested), it shows everything as a flat list of links between in/out boxes.

So in the above diagram, we can see the following linkages:

```
1 ConnectedSocketAdapter_26 outbox -> Pipeline_30 inbox
2 Pipeline_30 inbox - passthrough link to -> PureTransformer_27 inbox
3 PureTransformer_27 outbox -> PublishTo_28 inbox
```

You can also see that there is a pass through linkage from

```
PublishTo_28 inbox to PlugSplitter_9 inbox
```

Since the PlugSplitter is the core of the backplane, this is how PublishTo gets its data to the backplane.

The SubscribeTo half is a little more complex. The Subscribe To component creates a Passthrough component and a Plug component. The passthrough component is there really to work around a requirement of the Plug component's API. (This could be improved, but it's not been a priority)

The upshot though is that data flow out from the PlugSplitter is as follows:

```
PlugSplitter_9 outbox33 -> Plug_32 inbox
Plug_32 inbox - passthrough link to -> PassThrough_31 inbox
PassThrough_31 outbox - passthrough link to -> Plug_32 outbox
Plug_32 outbox - passthrough link to -> SubscribeTo_29 outbox
SubscribeTo_29 outbox -> passthrough link to -> Pipeline_30 outbox
Pipeline_30 outbox -> ConnectedSocketAdapter_26 inbox
```

What's worth noting here is that passthrough linkages are zero copy.

ie the data goes like this on the way in:

```
CSA -> Pipeline -> PureTransformer -> PublishTo -> PlugSplitter
```

On the way out:

```
PlugSplitter -> Plug -> PassThrough -> SubscribeTo -> Pipeline -> CSA
```

Given not all components see or process the data due to direct delivery enabled by passthrough linkages (which are zero copy), in terms of components that actually see and process data in some fashion though the data flow is:

On the way in ...

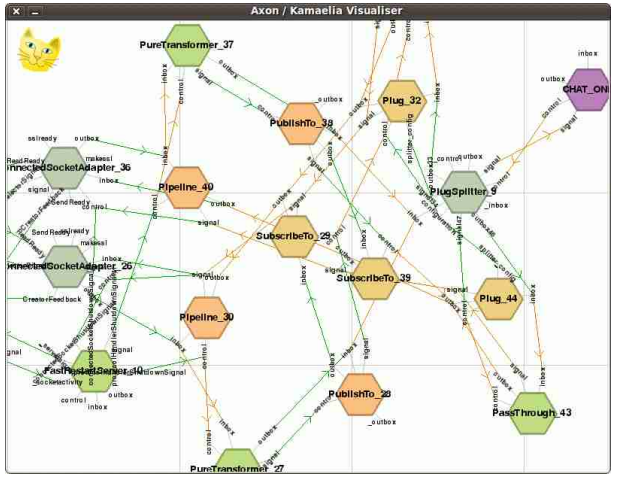
```
CSA -> PureTransformer -> PlugSplitter
```

On the way out...

```
PlugSplitter -> PassThrough -> CSA
```

ie **exactly** what we wanted - data from the connection gets sent to a splitter via a symbolic name "CHAT ONE", and data from the splitter with the symbolic name "CHAT ONE".

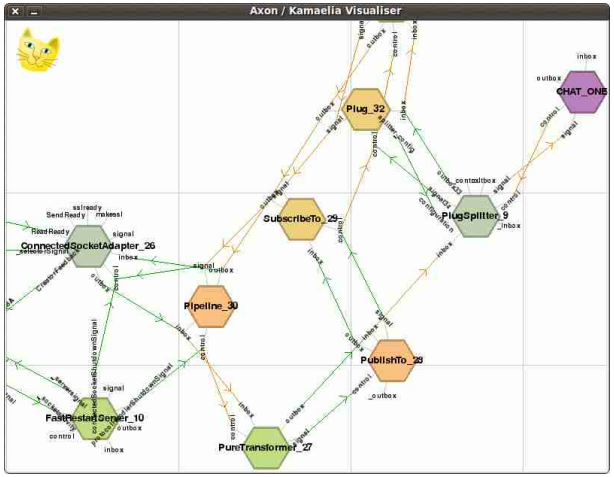
OK, so clearly a bunch of components get created upon connection. For completeness, this is what the diagram looks like when a second client connects:



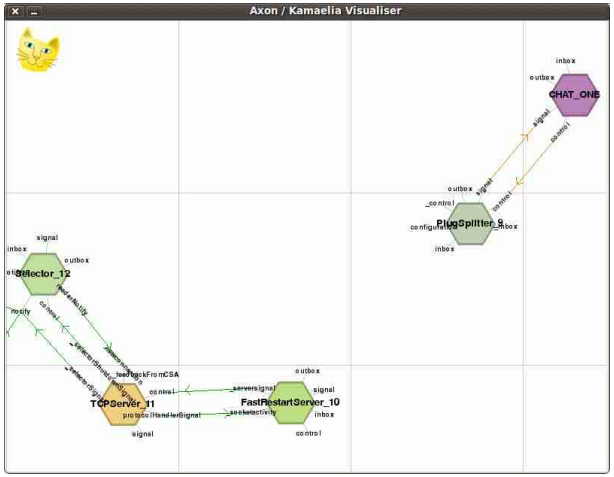
Clearly at this point we've reached the limits of the visualiser. Viewing the network of components as a flat layout is useful from the perspective of seeing “what's going wrong, if anything” for small systems, but for large ones, wrapping up the structure would be nicer. This is something that would be great for someone to work one, but we've not had the time for this.

Watching Client Disconnections from the Minimal Chat Server

Once again, we'll now allow one client to disconnect – leaving one set of components in place:



Finally the last client disconnects:



… and we see that we're back to our original state.

Now, clearly we've reached the limits of the visualiser, but let's go one step further – to show that a server can start and shutdown arbitrary other components based on data from connections, that we can see stay running during connections.