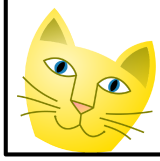
A walk through building & stacking protocols using Kamaelia



Michael Sparks
October 2008

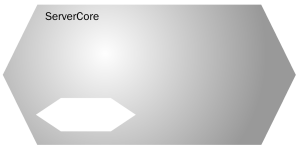
The aim of this document is to guide you through building a system where:

- A user connects to a server and
- Over a secure connection,
- Receives a sequence of JSON encode-able objects

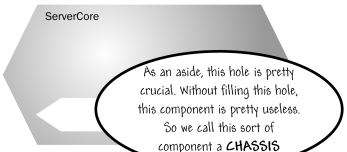
The aim of this document is to guide you through building a system where:

- A user connects to a server and
- Over a secure
- Receives
objects

This doc also uses speech bubbles like this to make "aside" comments regarding particlar panels. You can skip these safely.

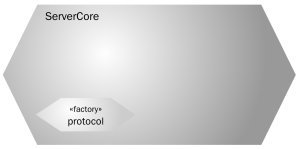


First of all, we start off with the **ServerCore** component. The hole is for fitting a protocol handler factory. eg a class, or function.



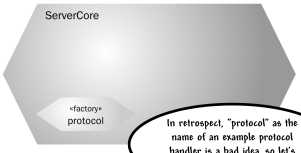
As an aside, this hole is pretty crucial. Without filling this hole, this component is pretty useless. So we call this sort of component a **CHASSIS**

First of all, we start off with the **ServerCore** component. The hole is for fitting a protocol handler factory. eg a class, or function.



So, we provide a protocol factory.

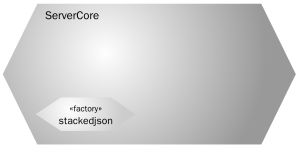
```
ServerCore(protocol=protocol, port=2345)
```



In retrospect, "protocol" as the name of an example protocol handler is a bad idea, so let's change this slightly.

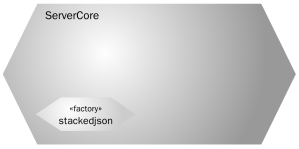
So, we provide a protocol [illegible].

```
ServerCore(protocol=protocol, port=2345)
```



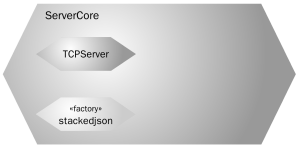
So, we provide a protocol factory.

```
ServerCore(protocol=stackedjson,
           port=2345)
```



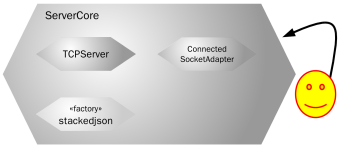
We then activate it, which causes it to start a subcomponent.

```
ServerCore( .... )
```

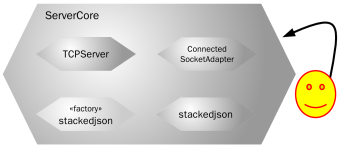


We then activate it. It starts **TCPServer** subcomponent, which listens for connections.

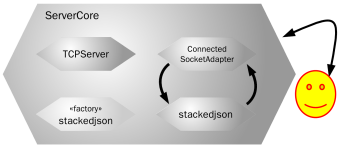
```
ServerCore( .... ).activate()
```



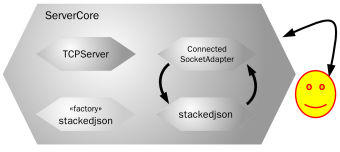
A client then connects to the TCPServer which creates a ConnectedSocketAdapter to handle the mechanics of the connection.



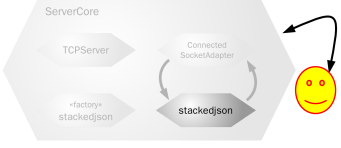
TCPServer tells ServerCore it has done this. ServerCore in response calls the protocol factory to create a handler component...



...which is wired up. The protocol handler just receives bytes and sends bytes, which are sent to the client via the ConnectedSocketAdapter



There, the key thing in creating a server, is to create an appropriate protocol handler component.



There, the key thing in creating a server, is to create an appropriate protocol handler component.

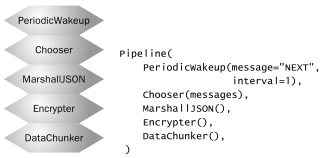
So, let's look inside this component


```
def protocol(*args,**argd):
    return Pipeline(
             PeriodicWakeup(message="NEXT", interval=1),
             Chooser(messages),
             MarshallJSON(),
             Encrypter(),
             DataChunker(),
           )
```

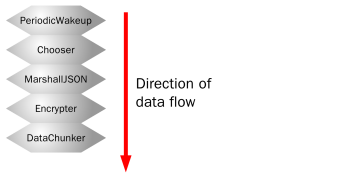
So, let's look inside this component


```
def protocol(*args,**argd):
    return Pipeline(
             PeriodicWakeup(message="NEXT", interval=1),
             Chooser(messages),
             MarshallJSON(),
             Encrypter(),
             DataChunker(),
           )
```

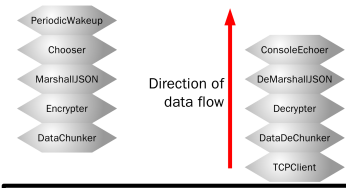
So, let's look inside this component. Clearly this is what actually handles the connection.



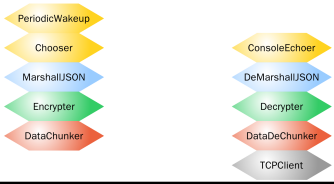
Let's expand that...



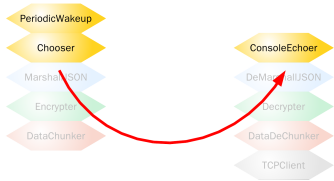
We see this is a uni-directional protocol – as soon as this pipeline starts up. Encrypted JSON data chunks get sent, ignoring client data.



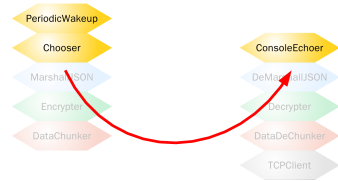
So, given the ServerCore infrastructure means we can more or less ignore the network, what does the client side pipeline look like?



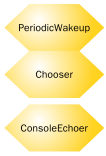
Each component in the server, that is part of the user's protocol, has a matching component in the client.



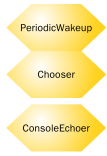
Fundamentally, this protocol allows one side of it to securely throw dictionary objects to clients which can then do something with them



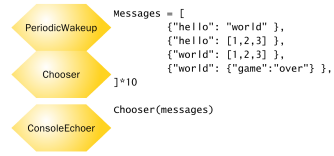
To understand what a client sees, we just look at these three components. The 2 highlighted **server** side components are the core behaviour.



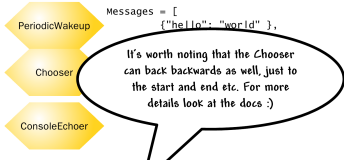
Let's rearrange these, showing the **logical pipeline** that drives what the user sees.



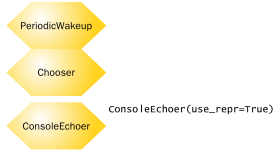
First of all the **PeriodicWakeupComponent** sends out the message "NEXT" once per second.



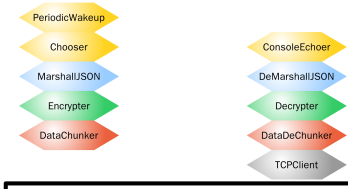
These "NEXT" messages control the **Chooser**. The chooser steps through a list of messages, and when told "NEXT", sends the next message.



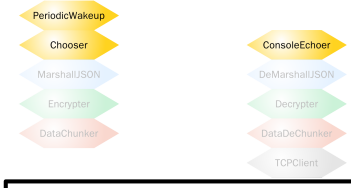
These "NEXT" messages control the **Chooser**. The chooser steps through a list of messages, and when told "NEXT", sends the next message.



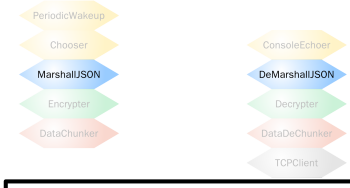
These messages arrive intact at the client – in this case a **ConsoleEchoer**. However it could be something more interesting – eg a game system.



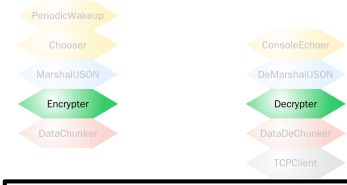
Going back to the overview slide, this diagram shows the various protocols which are stacked on top of each other.



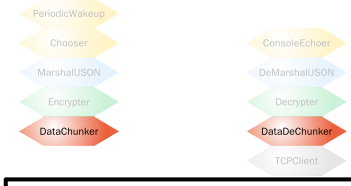
We have our **core application protocol**, stacked on top of a secure serialising communications channel. This truly is at the application level.



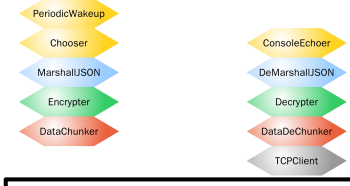
We have our **serialisation protocol**. Clearly this is a message oriented protocol – it requires the lower layer to preserve boundaries.



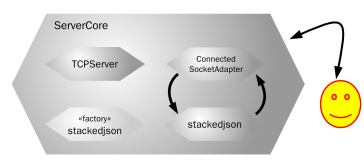
We have a **secure transmission protocol**. This is admittedly simplistic – it assumes predistributed keys. However it gives a flavour



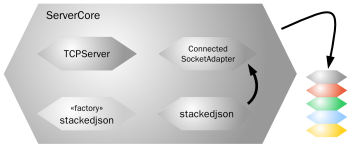
We then have our **chunking protocol**, which preserves message boundaries – necessary because TCP does not guarantee to do so.



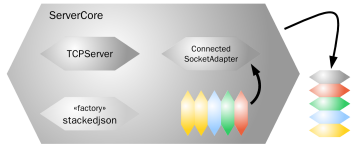
So let's go back and see where these things fit back into the system.



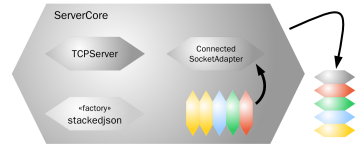
So this is what we had ...



The client really looks like this...
Also, this is a one way protocol



The client really looks like this & this is a one way protocol. The protocol itself looks like this. And that's how those parts fit together.



So, finally, for completeness, we'll include the code.

So, finally, for completeness, we'll include the code.

Though, for code, this is a better location:
http://tinyurl.com/4k3df2

Bunch of imports...

```
import cjson
import Axon

from Kamaelia.Chassis.Pipeline import Pipeline
from Kamaelia.Util.Chooser import Chooser
from Kamaelia.Util.Console import ConsoleEchoer

from Kamaelia.Internet.TCPClient import TCPClient
from Kamaelia.Chassis.ConnectedServer import ServerCore

from Kamaelia.Protocol.Framing import DataChunker, DataDeChunker
from Kamaelia.Apps.Grey.PeriodicWakeup import PeriodicWakeup

from Crypto.Cipher import DES
```

Define the messages being slung to clients

```
messages = [ {"hello": "world" },
             {"hello": [1,2,3] },
             {"world": [1,2,3] },
             {"world": {"game":"over"} },
           ]*10
```

Then the JSON De/Marshalling components

```
class MarshallJSON(Axon.Component.component):
    def main(self):
        while not self.dataReady("control"):
            for j in self.Inbox("inbox"):
                j_encoded = cjson.encode(j)
                self.send(j_encoded, "outbox")
            if not self.anyReady():
                self.pause()
            yield 1

class DeMarshallJSON(Axon.Component.component):
    def main(self):
        while not self.dataReady("control"):
            for j in self.Inbox("inbox"):
                j_decoded = cjson.decode(j)
                self.send(j_decoded, "outbox")
            if not self.anyReady():
                self.pause()
            yield 1
```

Define our encoding layer using normal code...1

```
class Encoder(object):
    """Null encoder/base encoder - returns the same string
       for encode/decode"""
    def __init__(self, key, **argd):
        super(Encoder, self).__init__(**argd)
        self.__dict__.update(argd)
        self.key = key
    def encode(self, some_string):
        return some_string
    def decode(self, some_string):
        return some_string
```

Define our encoding layer using normal code...2

```
class DES_CRYPT(Encoder):
    def __init__(self, key, **argd):
        super(DES_CRYPT, self).__init__(key, **argd)
        self.key = self.pad_eight(key)[:8]
        self.obj = obj=DES.new(self.key, DES.MODE_ECB)

    def encode(self, some_string):
        padded = self.pad_eight(some_string)
        encrypted = self.obj.encrypt(padded)
        return encrypted

    def decode(self, some_string):
        padded = self.obj.decrypt(some_string)
        decoded = self.unpad(padded)
        return decoded

# ... continued
```

Define our encoding layer using normal code...3

```
# class DES_CRYPT(Encoder): # ... continued from before

    def pad_eight(self, some_string):
        X = len(some_string)
        if X % 8 != 0:
            pad_needed = 8-X % 8
        else:
            pad_needed = 8
        pad_needed = 8-(X % 8)
        PAD = pad_needed * chr(pad_needed)
        return some_string+PAD

    def unpad(self, some_string):
        x = ord(some_string[-1])
        return some_string[:-x]
```

Create Encryption components using this

```
class Encrypter(Axon.Component.component):
    key = "ABCD"
    def main(self):
        crypter = DES_CRYPT(self.key)
        while not self.dataReady("control"):
            for j in self.Inbox("inbox"):
                j_encoded = crypter.encode(j)
                self.send(j_encoded, "outbox")
            if not self.anyReady():
                self.pause()
            yield 1
```

Note, this is not a serious protocol, it is an illustration. This form of pre-shared key would be a bad idea, but the pattern of usage will be useful.

Create Decryption components using this

```
class Decrypter(Axon.Component.component):
    key = "ABCD"
    def main(self):
        crypter = DES_CRYPT(self.key)
        while not self.dataReady("control"):
            for j in self.Inbox("inbox"):
                j_decoded = crypter.decode(j)
                self.send(j_decoded, "outbox")
            if not self.anyReady():
                self.pause()
            yield 1
```

Note, this is not a serious protocol, it is an illustration. This form of pre-shared key would be a bad idea, but the pattern of usage will be useful.

Create the protocol handler factory & Client

```
def stackedjson(*args,**argd):
    return Pipeline(
             PeriodicWakeup(message="NEXT", interval=1),
             Chooser(messages),
             MarshallJSON(),
             Encrypter(),       # Encrypt on the way out
             DataChunker(),
        )

def json_client_prefab(ip, port):
    return Pipeline(
             TCPClient(ip, port=port),
             DataDeChunker(),
             Decrypter(),       # Decrypt on the way in
             DeMarshallJSON(),
             ConsoleEchoer(use_repr=True)
        )
```

And finally start the server and point a client at the server.

```
ServerCore(protocol=protocol, port=2345).activate()
json_client_prefab("127.0.0.1", 2345).run()
```