Architecture

Eventually to become a real document, but used for people to share and discuss ideas...

Thoughts on Design

Acceptor-Connector

At the front-end, I would like to try an Acceptor as single-threaded class bound to a particular port. Accepting the incoming connection, of particular type, it will register this connection with the despatch mechanism. The pattern itself can be seen here.

> **elecharny**
>
> It might be a good idea to have more than one thread accepting incoming request, if needed. For instance, if you are accepting requests on more than one port/IP address, at some point, you might want to separate the load. Not necessarily important, but still to consider.

> **jmeehan**
>
> I think we should follow advice below on using framework. With both Netty and MINA we can create Acceptors and a pool of workers easily. I think I will put something together with MINA and send it out for discussion. Seems that the encoders and decoders could be developed quickly enough to use our current type, and I like the way this would start to break things out.
>
> So, create a *ProtocolCodecFactory* implementation that references the *ProtocolEncoder* and *ProtocolDecoder* and put that in the chain, and then add an *ExecutorFilter* to handle pooling of the workers. The *IoHandler* passed to the acceptor would act as a despatcher, examining the request and selecting the appropriate handler from a collection of those available in the same way it does now.

> **slemesle**
>
> First about using Netty, I don't see any benefits on developping a Web Framework on top of an existing one which is already integrated in many higher level WebFramework like Jersey, Play, Restlet. The Netty server already have it's own Http protocol well implemented so here I don't see any added value for AWF. Netty overlaps largely what we are building with AWF. Please notice that I do like Netty and already used it. But the reason I came to Deft was that the server had better performance in response time under huge load. Deft was better at this but, just a bit too instable (Poor HTTP support, few bugs, No multi-thread support...)
>
> Then comes the point of Mina, here again, Mina is really impressive effort and can deliver good performance. I'm sure we can pick-up from Mina some improvement like the Java 6 Selector bug, the memory buffers, the ssl support, ...
>
> Excuse me to come up again with this point, but I really like the IOLoop spirit, where we limit the count of living threads and always execute in the same loop. Callbacks gives a good way develop asynchronous process or get execution time on the targetted IOLoop thread. I think, this is one of the things that allows to get better performance on AWF. What I really like in this paradigm, is that it has no locks, every Thread can run isolated without synchonization point.
>
> So when we come to speak about building AWF on top of a framework, I would say well, picking up some code or feature idea is good point, but we should keep the loop spirit. If we want to be able to build Acceptor loop (I already did this in my fork on github), we just need an event switch allowing differents IOLoop thread to talk together without locks. And here I think we should have a look on Disruptor.

> **jmeehan**
>
> Well, I hadn't viewed either Netty or MINA as web frameworks despite any offered codecs as there would still be much more work to do (see vert.x, for example) but I take the point although I'm still leaning toward MINA. The design interested me as I think it's somewhat more flexible - we could surely implement such things as chaining of processors rather than a single handler invocation as we currently have (although I like the way we do that), but we also need to work on making things more extensible to manage differing protocols as simply as possible. I read through the documentation on Disruptor during lunch yesterday, and I think you've picked-up on something very worth looking at... did you get anywhere with this yourself, or are you still just reading around it?
>
> If you have the Acceptor work done, you may as well add it to the branch.

Threading

A discussion took place on JIRA bug DEFT-139. The solution was inspired by the Tornado model. AWF use an infinite IOLoop model where one thread is responsible for all operation on his sockets (Read/Write/Accept/Connect). Each IOLoop have its own Selector and run in its own Thread. Currently the server socket is registered in every single Selector.

Pros:

- Don't bother with threading while processing request
- Almost no locks on socket IO
- Simple and fast

Cons:

- Server socket Selector is shared accross all IOLoops
- Accepting Socket locks the IOLoops waiting for the Selector
- Long processing for a request impacts other client

> **elecharny**
>
> There are many possible options here, but using one single selector is a vast waste of spared CPU, as you won't use the extra ones, unless you add a thread pool behing the selector (not necessarily a good idea...)
>
> The main problem, even if you use more than one selector, is that once a socket has been assigned to a selector, depending on how long it will take to process it, all the other incoming requests will be suspended.
>
> The best solution would be to decouple the request execution from the request decoding : once the request has been decoded, we can free the selector and process the next incoming data. That means you have to add a threadPool somewhere after the request decoding.
>
> For long requests (for instance, when the client is pushing a huge bit of data), there is still a problem : where do we store the data temporarilly, and when do we consider that the request is completed...
>
> One point, you can perfectly have at least two selectors : one to accept the incoming connection, and another one to process the read and writes. Also note that it could be a good idea to separate the reads and writes by creating a selector for the read, and another for the writes.
>
> Also note that, if you want to have AWF compatible with Java 6, there is a very nasty bug in the way the selector handles some disconnection corner case. You may get a CPU spinning at 100%. What happens is that the select() method will returns a value > 0 but with no SelectionKey to process, because the remote peer has closed the socket. In this case, the select() will always return a value >0, and as it's an infinite IOLoop, you'll get a 100% CPU. The only way to workaround this is to detect this situation (by measuring how long you staid in the select() before getting a value > 0, and if it's 0, you can suspect that there is something going wild. Repeat this check 5 times, and then you can decide the selector is getting crazy) and by creating a new selector, registering all the SelectionKey on this new selector. Painfull, especially if you have one single selector for all the channels...

We surely can implement a workaround for crazy selectors.

IO Strategies

Talking about IO and threading is a good point and seeing what others do is required.
Grizzly support different Strategies for handling IO see IOStrategies. The idea of supporting different IO strategies can be a good option though it tends to make things harder.
In Netty, there a boss executor responsible for accepting and connecting sockets, once socket is accepted, it goes to a worker executor which hold multiple selector and process connected sockets IO.

Expectations

Emmanuel suggested that we put down the expectations and requirements for the system as a whole. TODO.

> **elecharny**
>
> Not that I want to push any framework here, but if you are considering rewriting the network layer, just know that this has already been done many times. I would suggest you focus on the important part atm, ie the request decoding and the protocol handling, and use a framework (or at least a part of it) to speed up the developement.
>
> You can decouple this by designing a facade that abstract the network layer from the rest of the server, allowing you to tests some frameworks, and eventually redesign one that fits your exact needs, when you'll have agreed.