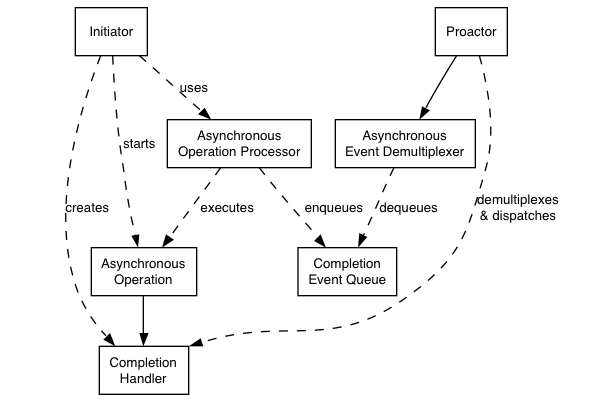
*[Proactor and Boost.Asio](http://www.boost.org/doc/libs/1_53_0/doc/html/boost_asio/overview/core/async.html" \l "boost_asio.overview.core.async.proactor_and_boost_asio)*

Let us examine how the Proactor design pattern is implemented in Boost.Asio, without reference to platform-specific details.



**Proactor design pattern (adapted from [POSA2])**

— Asynchronous Operation

Defines an operation that is executed asynchronously, such as an asynchronous read or write on a socket.

— Asynchronous Operation Processor

Executes asynchronous operations and queues events on a completion event queue when operations complete. From a high-level point of view, services like stream\_socket\_service are asynchronous operation processors.

— Completion Event Queue

Buffers completion events until they are dequeued by an asynchronous event demultiplexer.

— Completion Handler

Processes the result of an asynchronous operation. These are function objects, often created using boost::bind.

— Asynchronous Event Demultiplexer

Blocks waiting for events to occur on the completion event queue, and returns a completed event to its caller.

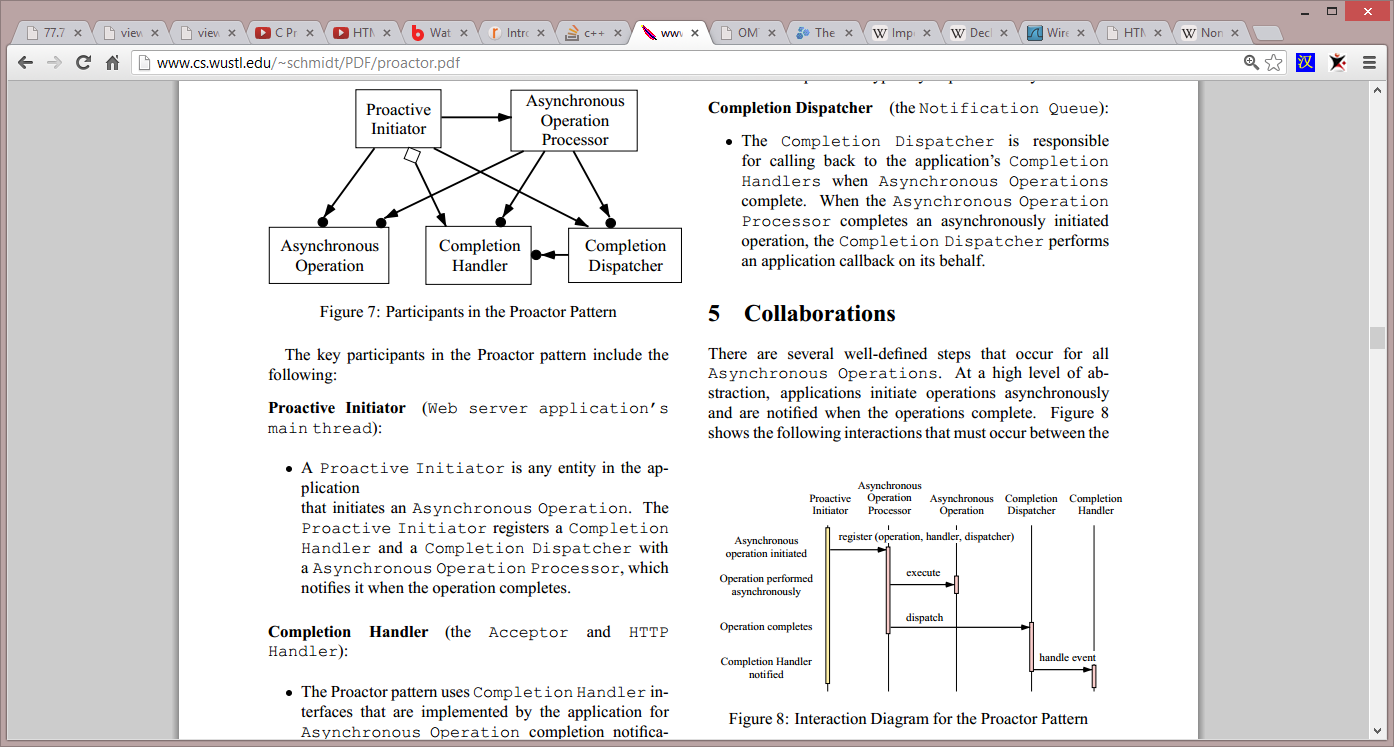
— Proactor

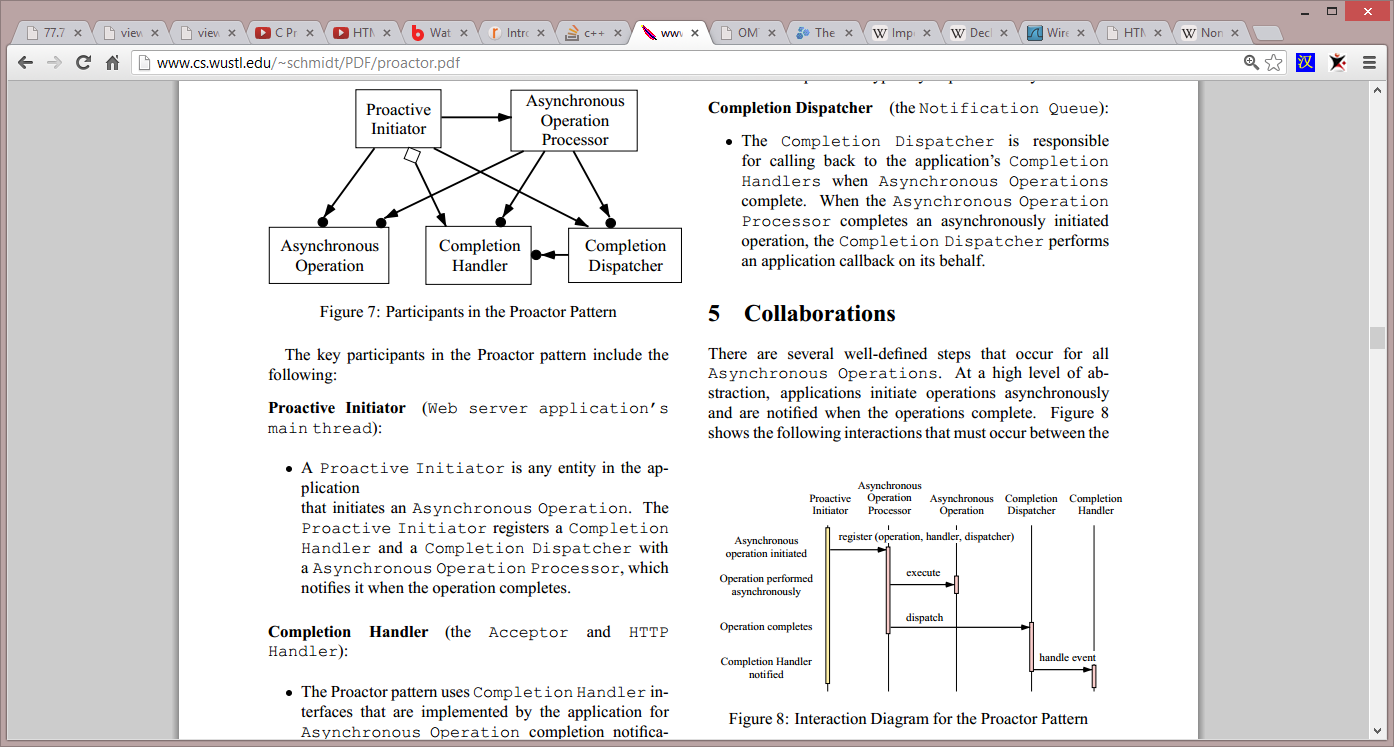
Calls the asynchronous event demultiplexer to dequeue events, and dispatches the completion handler (i.e. invokes the function object) associated with the event. This abstraction is represented by the io\_serviceclass.

— Initiator

Application-specific code that starts asynchronous operations. The initiator interacts with an asynchronous operation processor via a high-level interface such as basic\_stream\_socket, which in turn delegates to a service like stream\_socket\_service.

***DOPS and boost***





Suggestions:

In order to make sure that the architecture of the DOPS system is future flexible, then I suggest that we start by having a central communication unit, a communication server based on the proactive design pattern, we then have **DFD (dataflow diagram) functions connect themselves as clients**, thereby making the transfer of data between web clients and server clients easier. This means in turn that server clients can be related directly to the DFD description and their “address” will be the number they have in the DFD

When a transfer of data is done from a web client, it is handled by the CCS (Central Communication Server) which will find the recipient based on destination address. The CCS will be developed based on the Chat server developed for boost – changes is of course needed, however overall handling of transfer is done using the proactive design pattern

When data is transferred to the destination function, then it will handle the data using its own proprietary protocol, and handshaking/acknowledge to client will be done using that protocol. Typically the function will have an incoming ringbuffer that will receive data in the form of packets, these packets will then be internally transferred using the reactor design pattern ( have one thread looking inside the ringbuffer for packets, then when packets are there send them to relevant sub-functions based on protocol inside the packet ). The protocol inside the packet could be based on the DataEncoderDecoder principle

It would be good to make some sort of security/sanity check inside the CCS, having it know which clients are needed, perhaps also have a list of VIP clients – to make sure that these clients are started and connected. In time of low resources the CCS could disconnect every clients except the ones on the VIP list, this could release resources and change a potentially unstable situation into a more stable situation.

We could also have an administrator connection with the CCS, where we can send administrative commands and receives status on performance, uptime, etc…