1 Introduction

Over the past several months, there have been several disparate attempts at creating a C++ ActiveMQ client - each was developed with a certain set of goals in mind. The purpose of this document is to initiate some discussion on architecture to meet the needs of the disparate groups. The hope is that merging these needs into a unified architecture will result in a common client design that could be used for any language (e.g. C#).

2 ActiveMQ Client Requirements

In order to start this discussion, I'd like to list out what I believe to be the list of goals from the disparate efforts.

1) Client must be easy to use and familiar to those used to the JMS interface.
2) Client must be able to be used in single or multi-threaded "modes". In other words, it should provide an event-based as well as a polling model.
3) Client must support swappable messaging protocols such that the client application could switch between STOMP and Openwire (for example) without changing code.
4) Developing/adding a new messaging protocol must be trivial.

3 High-level Architecture

The proposed architecture is divided into a 4-layer stack as shown below.

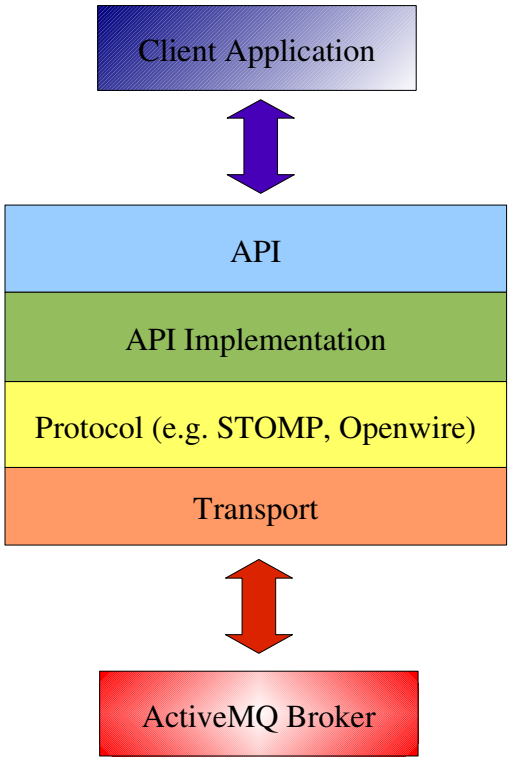

Illustration 1: Proposed ActiveMQ Client Architecture (High-Level)

3.1 API

The API is simply that – the set of interfaces that define the functionality for the client. These interfaces mimic those of JMS.

3.2 API Implementation

This layer is a set of classes that implement the JMS-like API. They don't contain any protocol-specific knowledge. Instead, they just use invoke abstract requests through an protocol interface. The main purpose of this layer is to bridge the gap between the protocol and the API (e.g. managing sessions, publishers, subscribers, etc).

3.3 Protocol

This layer contains all the specifics to the particular protocol that has been selected. This layer is responsible for marshaling messages between the API and the transport layer. It is also responsible for any particulars specific to the protocol, such as keep-alive messages and data compression.

3.4 Transport

This layer is the most fundamental. It is a simple layer for sending/receiving data. For now, we will assume this is done via TCP sockets.

4 Architecture Part Deux (Slightly Less High-Level)

In this section, I'll show the general model of the classes that make up each of the layers. I'm omitting the API because we can assume it closely matches JMS. Also, I'll describe the layers from the bottom up, that is the lowest level of abstraction to the highest.

4.1 Transport Layer

Illustration 2: Transport Layer Class Diagram

This layer is comprised of the Transport interface that has basic methods for IO, as well as a factory for creating transport objects (based on a url string). We implement a TcpTransport that owns a socket and communicates with the socket via a Java-like buffered stream model. The purpose for the buffered stream model is for efficiency when reading/writing to the socket, since socket IO can somewhat slow if done in small chunks.

4.2 Protocol Layer

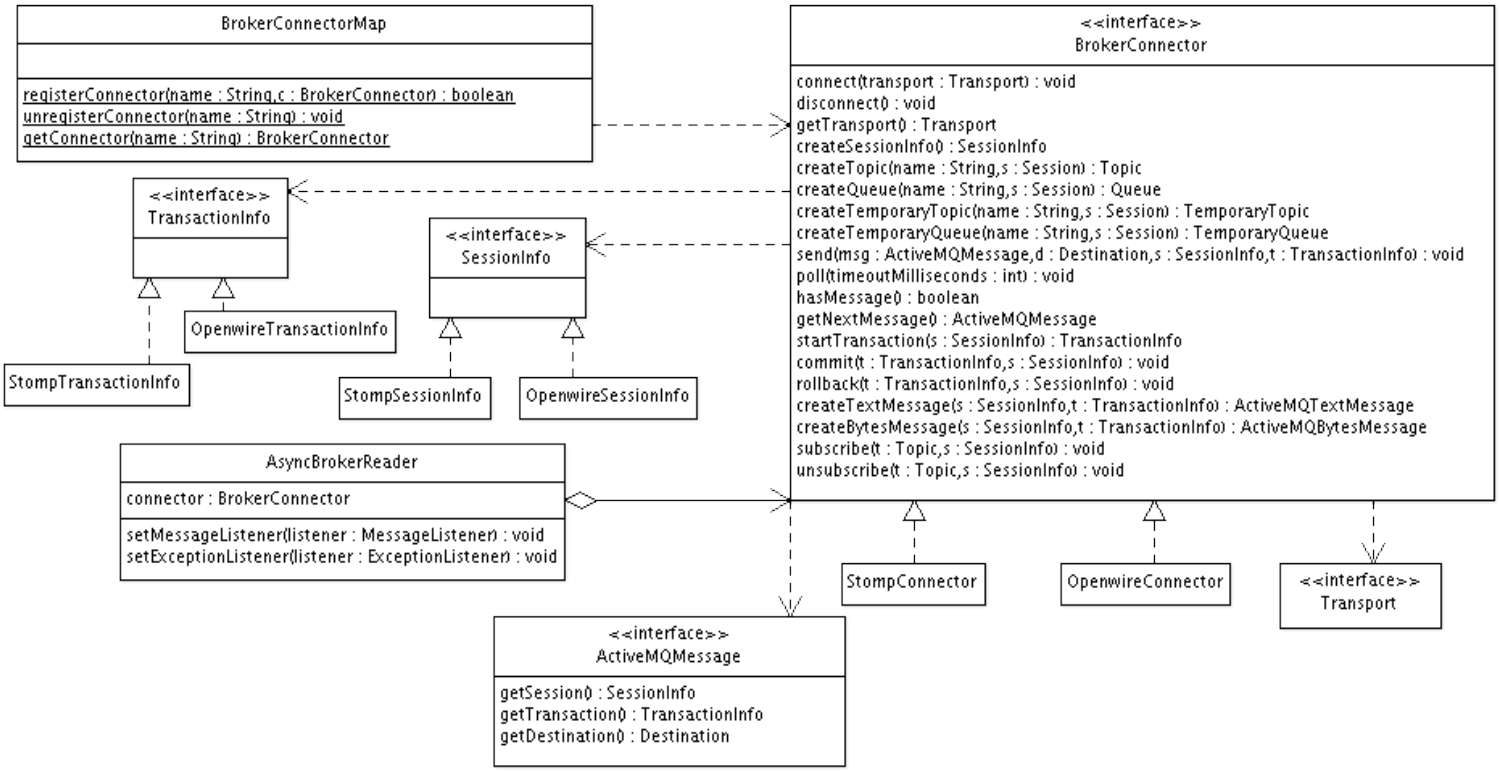

Illustration 3: Protocol Layer Class Diagram

This layer contains the BrokerConnector interface which represents a protocol. In the diagram we have two implementations of the connector: STOMP and Openwire. The BrokerConnector interface contains all of the methods that the API Implementation layer needs in order to do its job (e.g. sessions, pub/sub, transactions, etc.). The BrokerConnectorMap is an object that contains the collection of registered connectors. This class allows the user to extend the model by dynamically adding a protocol, if desired. In the same way, this allows the ActiveMQ development team to extend the model.

By itself, the BrokerConnector interface only provides a poll-style single-threaded manner for reading messages. The AsyncBrokerReader provides an event-oriented layer. It contains a thread that automatically polls the BrokerConnector and fires events when user messages are received or an exception occurs.

4.3 API Implementation

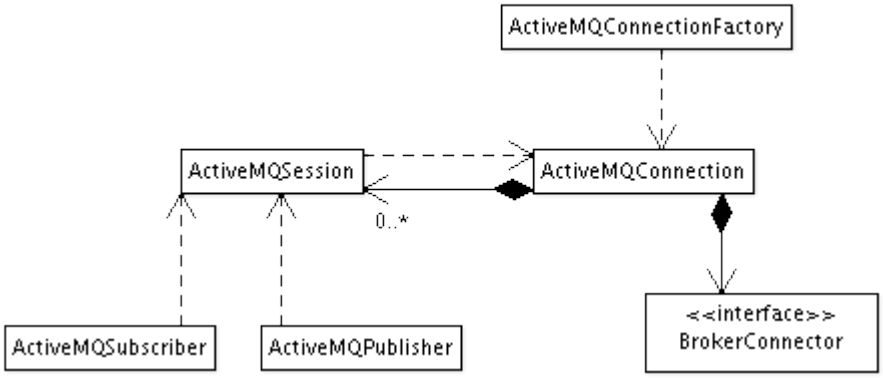

Illustration 4: API Implementation Class Diagram

This section is just provided as a sketch of how the API Implementation might look. The connection owns the BrokerConnector, and uses it for creating sessions, publishers, subscribers, etc. I'll leave the details of this layer for future discussion.

4.4 Synchronization

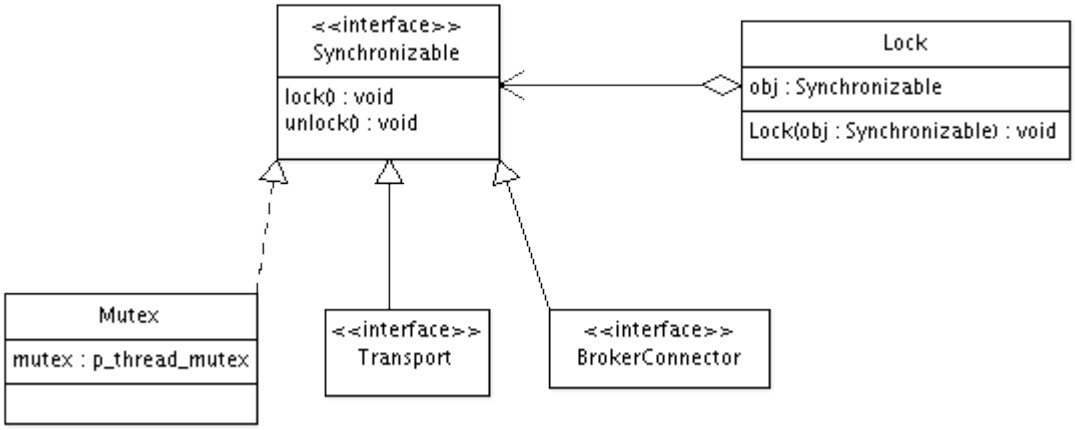

Illustration 5: Synchronization Class Diagram

This is not a layer, exactly, but I thought would be worthy of discussion. The attempt here is to replicate the Java synchronization model in C++. We start with a Synchronizable interface which just provides lock/unlock functionality. We then create a Lock class that takes a Synchronizable, locks it in its constructor, and unlocks it in its destructor. This is a common model used for locking/unlocking mutexes to ensure that the mutex is always released when control leaves scope (even if an exception occurs). We then extend the Transport and BrokerConnector interfaces from Synchronizable, allowing us to lock these objects just as if they were mutexes. What this achieves is clean Java-like synchronization blocks where we can simply lock on the object that we want to protect, as shown below.

```
{
        TcpTransport* transport;

        //... set the transport to something

        Lock lock( transport);

        // Update the transport object

} // when we leave the scope of this block, the class will be automatically unlocked.
```

The reason why I feel this model is superior to just locking/unlocking in all of the interface methods, is that it allows the programmer to decide for him/herself when a lock should occur, rather than being forced to lock every time a method is called. For example, if we're in a single-threaded model, there is no reason to lock a mutex when we call the read() method of the TcpTransport class. On the other hand, if we're in a multi-threaded model, we probably would want to lock on that method.

5 Summary

As I stated previously, the hope of this document is to provoke discussion and to provide direction for the merger of our disparate goals for this project. Comments are welcome and encouraged.