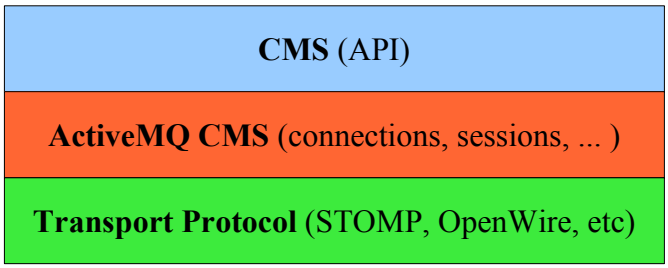
# Introduction

This document is meant to give a high level overview of CMS (C++ Message Service). CMS is a JMS-like API for C++ applications. The idea being that someone who is familiar with JMS can move seamlessly between languages. As those of us who have had the unfortunate luck of being stuck back on a legacy C++ project know – it's really hard to go back to C++. Hopefully this will make it a little less painful.

# Architecture Overview

The implementation of CMS is a 3-layer stack, as shown below.




The client interfaces with the CMS API directly. This API is defined by the interfaces in the cms namespace. These map 1 to 1 (for the most part) to interfaces defined by the JMS spec.

ActiveMQ CMS is an implementation of the CMS interface. This layer is abstracted from the specifics of the protocol being used to talk to the ActiveMQ broker. Instead it is a middle layer between the transport layer and the client. It manages client connections, sessions, subscribers, publishers, etc.

The Transport Protocol layer deals with the specifics of broker communications. It manages connection state, message protocol conversions, and is responsible for notifying the CMS layer when client messages arrive from the broker or when exceptions occur.

# Source Code Overview

The CMS code is currently divided into 2 eclipse projects (eclipse 3.1.1 with CDT 3.0.1): **cms** and **activemqcms**. These projects are configured as managed make projects, so I'm not sure if they will be able to be built outside of eclipse. Also, as it stands now, these projects require pthreads, the linux socket API, and the sys/time.h header (nanosleep, etc). It should be fairly easy to replace these dependencies with an all-in-one library (such as boost) in the future.

The source for both projects is stored under the src directory. Under the src folder, I have maintained a directory structure matching the namespace hierarchy.

The cms project only contains the ***cms*** namespace. No library is generated from this project because it only contains header files. The file/class names in this namespace closely match those of JMS.

The activemqcms project contains one toplevel namespace: ***activemq***. The namespace tree under activemq is as follows:

**-activemq** (main classes that implement the CMS interface)
**-concurrent** (utility classes for thread synchronization)
**-io** (utility classes for IO, including a socket class and java-like IO streams)
**-transport** (interfaces for the transport layer)
**-stomp** (STOMP implementation of the transport layer)

Finally, I have provided another project, called **test** that shows a topic publisher and subscriber talking to each other. This should be helpful in coming up to speed with the code.

## Example of Usage

This is an excerpt taken from the main.cpp in the test project. It illustrates the creation and destruction sequences of many commonly used objects.

```
cms::TopicConnectionFactory* connectionFactory = new activemq::ActiveMQConnectionFactory(
"127.0.0.1:61626" );

cms::TopicConnection* connection = connectionFactory->createTopicConnection();

connection->setExceptionListener( this );

connection->start();

cms::TopicSession* session = connection->createTopicSession( false );

cms::Topic* topic = session->createTopic("mytopic");

cms::TopicSubscriber* subscriber = session->createSubscriber( topic );

subscriber->setMessageListener( this );

cms::TopicPublisher* publisher = session->createPublisher( topic );
```

... (interesting stuff goes here)

```
delete publisher;
subscriber->close();
delete subscriber;
session->close();
delete session;
connection->close();
delete connection;
delete connectionFactory;
```

## Outstanding Items

1) Implement transactions.
2) Implement protocols other than tcp.
3) Implement queues.
4) Implement transport layer for OpenWire.
5) Remove dependency on pthreads, sys/time.h (nanosleep), linux socket API. It should be fairly easy to replace these dependencies with an all-in-one library (such as boost).