1. C++ Client for Naradabrokering

The first section of this user guide will take you through the installation process and the next section shows how to use the simple chat client. The final section explains the architecture and how to utilize the C++ Client to implement communication channels.

1.1 Configuration

1.1.1 Broker Configuration

> Note: The current implementation of the C++ Client works on **Intel**-based architectures. The differences in the **endianness** of various architectures require different compilations. More explanation about this will follow in section three.

Download and unzip the Naradabrokering from http://www.naradabrokering.org/ to some local directory (say **NB_HOME**)

Start the Broker using the **startbr.sh** shell scripts in the bin directory inside **NB_HOME.**

> Note: If you need to handle larger payloads, please change the line in the startbr.sh

```
java -classpath $cp cgl.narada.node.BrokerNode $brokerConfigFile
$serviceConfigFile $brokerCommunicatorPort&
```

> to

```
java -Xmx<max value>m -Xms<min value>m -classpath $cp
cgl.narada.node.BrokerNode $brokerConfigFile $serviceConfigFile
$brokerCommunicatorPort&
```

Use the BrokerConfiguration.txt found in the **config** directory inside **NB_HOME** to change the ports that the broker used for communication. Please note that this step is not mandatory, using the default ports is fine.

1.1.2 Compiling the C++ Client

Download and unzip the **nbcpp.tar.gz** to a local directory (say **CLIENT_HOME**)

Inside **CLIENT_HOME** you will find a **src** directory, which contains the C++ code.

Compile the simple chat client using the *make* tool. Use the following command.

```
make pubsub
```

1. This will create an executable **pubsub** in the **src** directory itself.

1.2 Section2: Simple Chat Client

Once the C++ Client is compiled go to **`CLIENT_HOME/src`** directory and run **`pubsub`** to start one chat client. This will require few input parameters as explained below.

```
./pubsub 7799 44567 /test/topic 127.0.0.1 5045
```

The first integer argument is the **entityId**, which identifies this client in a given broker network.
The, next integer argument is the **templateId** which is a unique integer associated with a given topic.
The third parameter is the **topic** for which this client publishes and subscribes. This can be any string without intemediate spaces.
The fourth and the fifth arguments are the host address and the port number of the broker. Please note that we need to use the TCP port of the broker. This would be port 5045 if you are using the default port numbers.

Once the chat client is started, start another chat client with different **entityId**. The next step is to see the Chat program in action by typing in few messages.

To exit from the chat client type **`$<return>.`**

1.3 Section3: The Architecture

The C++ Client establishes a TCP connection with a given broker and supports exchanging of pub/sub messages. The following diagram shows the architecture of the C++ Client.

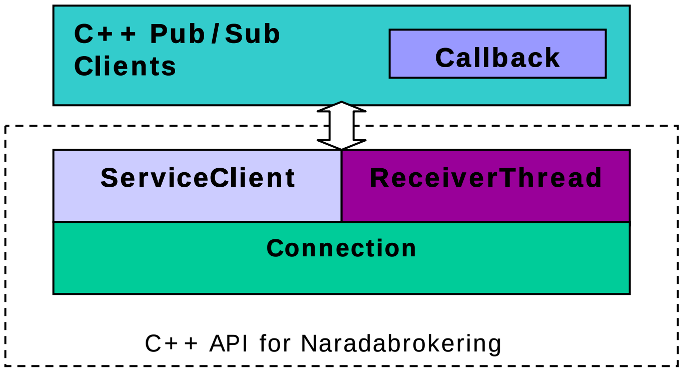

Figure1: Architecture of the C++ Client for Naradabrokering.

The API that the C++ programmer needs to work with has one class - **`ServiceClient`** and the **`Callback`** interface. The **`ServiceClient`** hides the rest of the components shown in the above architecture diagram from the user, and provides four basic methods that give a publish/subscribe interface for the C++ clients. These methods are listed below:

```
void init(string host,int port,int entityId,int templateId);
void subscribe(string topic,Callback *callback);
void publish(string topic,char* bytes,int length);
void close();
```

First, the client needs to establish a connection using the **`init(..)`** method shown above which takes four input parameters.

`Host`	host address of the broker
`Port`	TCP port of the broker(default is 5045)
`entityId`	This will identify the client uniquely in a broker network
`templateId`	A template Id for this connection

If the client needs to subscribe to a specific topic, then the method to use is: **`subscribe(..)`**. This method takes a topic and a Callback object as input parameters.

`topic`	String parameter, which specifies the topic to which the client needs to subscribe.
`callback`	This should be an implementation of the Callback interface provide by the C++ API. The client is expected to implement the onEvent(NBEvent *nbEvent) method of the callback. The C++ API will call this method for any event received for a topic that this client is subscribed.

To publish messages to a topic, the client can utilize the **`publish(..)`** method. This method takes three parameters as explained below.

`topic`	String parameter which specifies the topic to which the client needs to publish events.
`Bytes`	This is the content payload of the message and can be any number of bytes
`length`	Length of the byte array of the content payload

Finally if the client needs to close the broker connection, then it can use the **`close()`** method of the **`ServiceClient`**.

1.3.1 The Endianness

The current implementation supports Intel-based machines that use Little Endian ordering when storing bytes. This affects the way we store multi-byte data types and send them over the communication channels. Java handle bytes in BigEndian format as inherited from its Solaris roots. However, the Intel based architectures use LittleEndian format, and hence a conversion is required when exchanging messages between these architectures. The current implementation assumes a 32-bit value for integers and 16-bit values for short data types. This part requires little more research to make it generic for both 32-bit and 64-bit architectures. However, this difference does not affect the usage since the C++ client accepts a byte array as the content payload which is unique across the above platforms.

1.3.2 Simple Pub/Sub Example

The following code fragment shows the methods that need to be used in order to write a pub/sub client using the above API.

```
ServiceClient serviceClient;
/*Establishes a connection*
serviceClient.init(host,port,entityId,templateId);

MyCallback callback;
```

```
/*Subscribed to a topic*/
serviceClient.subscribe(contentSynopsis,&callback);

/*Publishes a message*/
serviceClient.publish(contentSynopsis,msg,strlen(msg));

/*Close the connection*/
serviceClient.close();
```