C++ Bridge for NaradaBrokering (JNI-based)

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. Final section explains the architecture and how to utilize the C++ Bridge to implement communication channels.

Section 1: Configuration

Broker Configuration

JDK –Version: Both the Broker and the Bridge require JDK1.5 or above

1. Download and unzip the Naradabrokering to some local directory (say **NB_HOME**)
2. 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&
```

A benchmark test, where the broker is fired with 64kB of data at a rate of ~4.5MB, shows that <max value> of 512 is a good heap size.

3. 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 a must.

Compiling the C++ Bridge

1. Download and unzip the **`cppbridge.tar.gz`** to a local directory (say **`BRIDGE_HOME`**)

2. Inside **`BRIDGE_HOME`** you will find a **`src`** directory which contains both Java and the C++ code.

   Note: Java classes are pre-compiled and are in the **`nbcppbridge.jar`** located in **`BRIDGE_HOME/lib`**

3. Set the **`JAVA_HOME`** variable in the make file (located in the **`BRIDGE_HOME/src`** directory) to point to the appropriate location.

4. Default goal will perform the necessary compilation, build chat executable and move it to **`BRIDGE_HOME/build`** directory.

Section2: Simple Chat Client

1. Once the C++ Bridge is compiled go to **`BRIDGE_HOME/build`** directory and run the **`chat.sh`** to start a chat client. e.g. **`./chat.sh 5000`**

2. The integer argument is the entity Id which identifies this client in a given broker network. To start the second chat client run it again in a new shell with different entity id. (say **`./chat.sh 6000`**)

3. Once the two chat windows shows the line “Happy Chatting”, you can type any text to be sent to the other.

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

Section3: The Architecture

The C++ Bridge uses JNI technology to communicate with Naradabrokeing. The following diagram shows the high-level architecture.

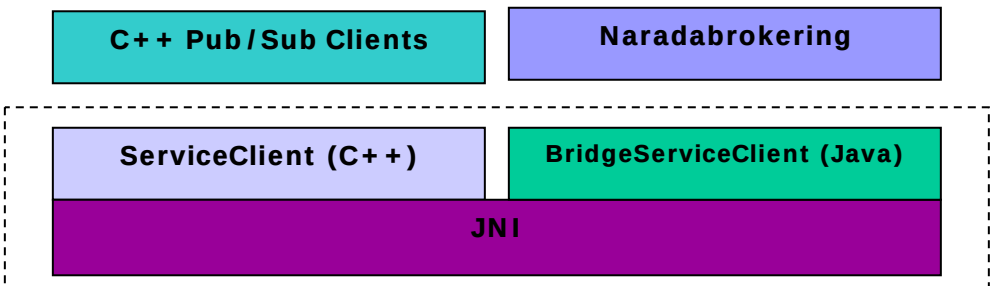

Figure1: Architecture of the C++ Bridge for Naradabrokering.

How to Use the Bridge

The API that the C++ programmer needs to work with comprises of one class - **`ServiceClient`** and the **`Callback`** interface.

`ServiceClient` contains the following public methods and perform the tasks as described below.

1.
```
bool init (int entity_id, char *config_file_path, char
*host_name,  int port_num, char *transport);
```

Description:

This will initializes the **`ServiceClient`** where it will load the JVM and initializes the communication with the broker.

Parameters:

`entity_id` – ID of this service instance
`config_file_path` - Path to the ServiceConfiguration.txt
`host_name` - Host Name where the Broker is running
`port_num` - Port Number of the Broker
`transport` - Transport type to be used. (Default uses TCP) possible options “niotcp” , “udp”

2.
```
bool subscribe (char *topic, long callbackId);
```

Description:
This is used to subscribe to any topic using this **`ServiceClient`** instance.

Parameters:

`Topic` – Topic to which the messages are sent. **e.g. "/topics/nbcpp"**
`callbackId` - This is the reference to the Callback object used for this topic.

User can provide different callback objects for different topics or use the same callback object. The `callbackId` should be a pointer to any implementation of **`Callback`** interface. Please see the **`chat_client.cc`** for a sample.

3.

```
bool publish (char *topic, char *transfer_bytes);
```

Description:

This will publish a given set of bytes to a given topic using this **`ServiceClient`** instance.
`topic` - Topic to which the message is published. **e.g. "/topic/nbcpp**"
`transfer_bytes` - Set of bytes to be transferred.

> Note: If multiple publishers and subscribers need to be run in a single process they should all share a single instance of **`ServiceClient`** as it is not possible to create multiple JVMs in a single process.

Simple Publisher Example

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

```
int
main (int argc, char *argv[])
{
  //Input parameters for publisher
  int entity_id = atoi (argv[1]);
  char *service_config_path
=/test/abc/ServiceConfiguration.txt";
  char *host_name = "gf6.ucs.indiana.edu";
  int port_num = 3075;
  char *transport = "niotcp";
  char *topic = "/publish/mytopic";
  ServiceClient sClient;


 //Initialize the service_client
  if (!sClient.
```

```
      init (entity_id, service_config_path, host_name, port_num,
    transport))
    {
      cout << "Error:Initialization Failed \n";
    }

   //Publish a given set of bytes.
   char *buffer=”This is my test message”;
      if (!sClient.publish (topic, buffer))
      {
        cout << "Error:Publishing Failed \n";
      }

  return 0;
}
```

Simple Subscriber Example

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

```
//Callback class for the receiving events.
class MyCallback:public Callback
{
public:
  MyCallback ()
  {
  }
   ~MyCallback ()
  {
  }
  virtual int on_event (char *received_data)
  {
    cout << received_data << endl;
  }
};

int
main (int argc, char *argv[])
{
  //Input parameters for publisher
  int entity_id = atoi (argv[1]);
  char *service_config_path
=/test/abc/ServiceConfiguration.txt";
  char *host_name = "gf6.ucs.indiana.edu";
  int port_num = 3075;
  char *transport = "niotcp";
  char *topic = "/publish/mytopic";
  ServiceClient sClient;


 //Initialize the service_client
```

```
  if (!sClient.
      init (entity_id, service_config_path, host_name, port_num,
    transport))
    {
      cout << "Error:Initialization Failed \n";
    }

  //Callback handles the events to me.
  Callback *callback = new MyCallback ();
  long cbid = (long) callback;

   //Subscribing to the topic
   if (!sClient.subscribe (topic, cbid))
    {
      cout << "Error:Subscription Failed \n";
    }

   While(1){sleep(2);}//Need to wait to receive events.


  return 0;
}
```

Other Configurations

`ServiceClient` assumes the following parameters for the JVM and currently they are placed in **`native_calls.cc`**

```
/* define it to be ':' on Solaris */
#define PATH_SEPARATOR ';'

/*java classes are here */
#define USER_CLASSPATH "../lib/NaradaBrokering.jar:../lib/log4j-
1.2.8.jar:../lib/jug-uuid.jar:../lib/nbcppbridge.jar"

/* libcppbridge.so should be here */
#define USER_LIBPATH "../lib"
```

If you need to change these directory locations please update this file and re-compile the source.