# **RELEASE NOTES**

This project serves to demonstrate the concepts required in building process communications applications. It has not yet reached a matured and stable release.

This project contains 3 sub-projects:

* cppSockets – this is a shared library containing common classes and utilities used by server and client.
* cppEchoServer – this is the executable for the server application.
* cppEchoClient – this is the executable for the client application.

These projects are built using Eclipse C/C++ Development Tool (in particular MSYS64, MinGW GCC)

## **Running the Server and Client**

libCppSocket.dll must be in runtime path.

E.g.:

set PATH=%PATH%;C:\lma\git\codingforfun\cppSockets\Debug

To run the server, the command is “*cppEchoServer <port>*”

E.g *cppEchoServer 3100*

To run the client, the command is “*cppEchoClient <host:port> <host:port>*”

E.g *cppEchoClient localhost:3100 localHost:3101*

1. cppEchoServer starts listening on localhost. Port is be passed in as command line argument.
2. On accepting connection from client, an independent thread will be started for the client socket.
3. The client thread will receive message and broadcast the message to all connected clients via the Sessions singleton.
4. Sessions is a thread-safe singleton holding a map of IDs and Client Sockets.

## **LIMITATIONS**

### Server Limitations

1. Server Shutdown currently using CTRL-C to kill this process.
2. Graceful shutdown not yet implemented, e.g. a client to send a shutdown signal to server or via Windows Services control.
3. Upon client lost detection, Server should remove invalid socket entries from Sessions.
4. **Message structure is now limited to string format "<LicenceID> <Message>", with a white-space in between.**
5. In theory, server should validate LicenceID from client, i.e. client is allowed to use the server operation.
6. No synchronization of states between servers.

### Client Limitations

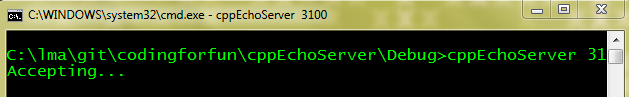
1. Client cppEchoClient has not been refactored to individual classes.
2. Client limited to 2 connections strings for now.
3. Client establishes real connection on first send.
4. Client detects lost connection on receive thread. Currently it does not notify lost connection to main sending thread.
5. Client would prompt user to re-send message on lost connection.

## **Screenshots Examples**

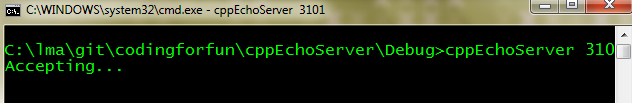
### Starting the Servers

Two servers are started locally, and waiting to accept connections.

Server 1 listening on port 3100:



Server 2 listening on port 3101:

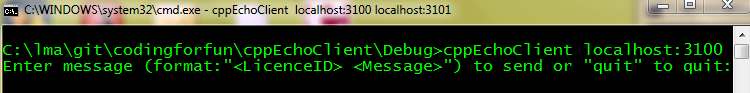


### Starting the Clients

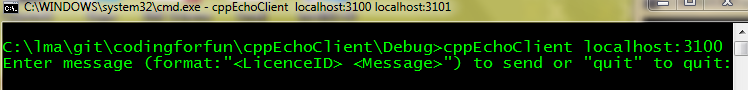
Clients can be started passing in 2 connection strings:



Client 1 started



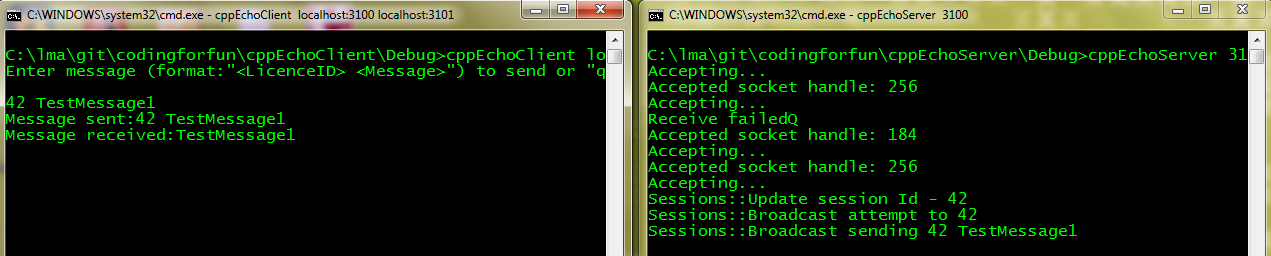
Client 2 started



### Sending messages and broadcasting

Client 1 sends “42 TestMessage1” to server 1.

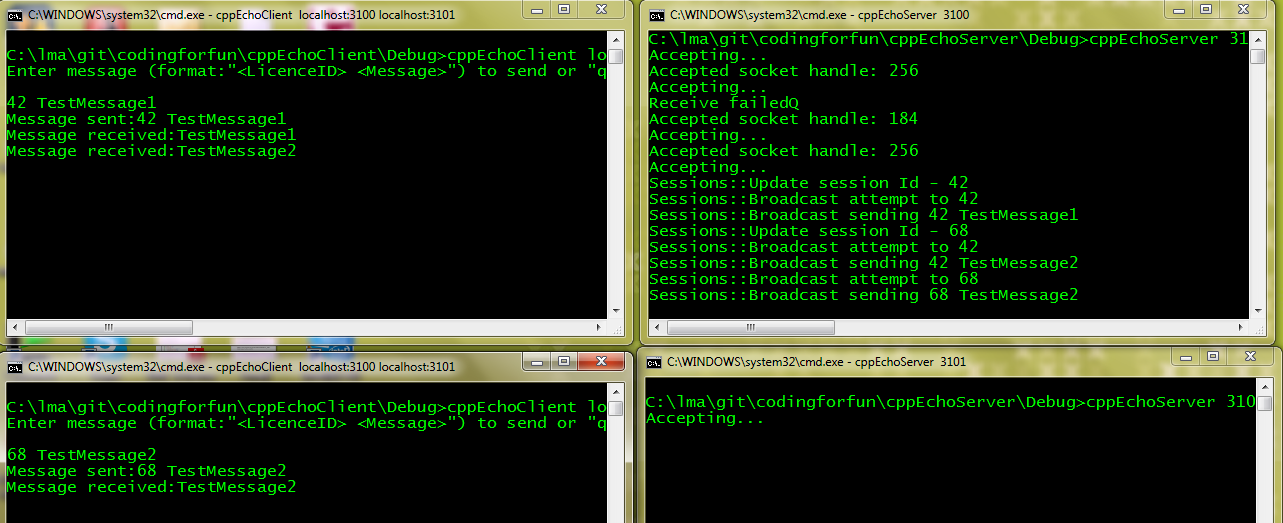
Client 1 receives echoed message “TestMessage1”



Client 2 sends “68 TestMessage2” to Server 1.

Client 2 receives echoed message “TestMessage2”.

Client 1 also receives echoed message “TestMessage2”.

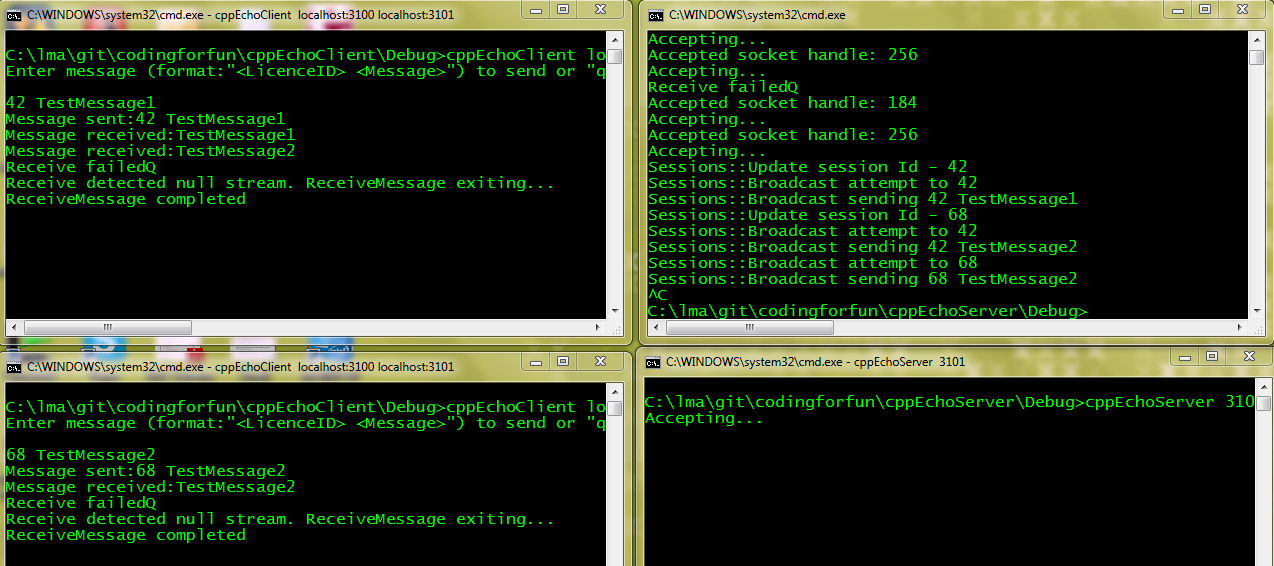


### High Availability Simulation

Kill Server 1 via CTRL-C.

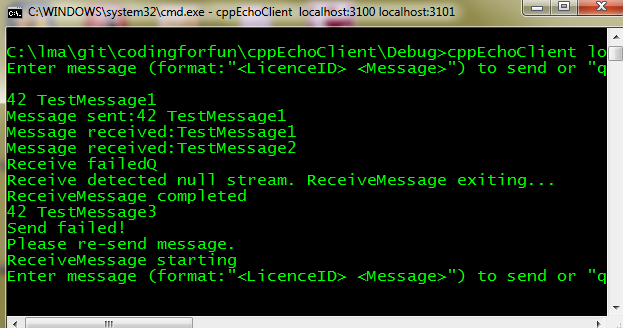
Client 1 receiver thread detects lost connection and the thread ends.

Client 2 receiver thread detects lost connection and the thread ends.

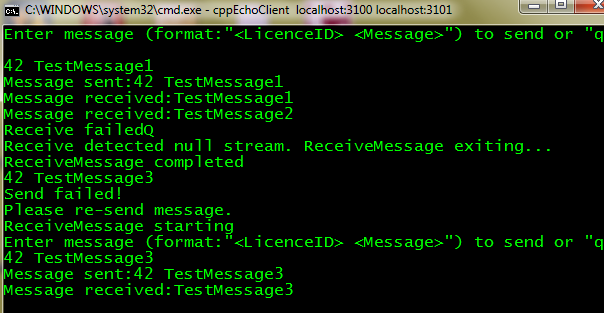


On next send message “42 TestMessage3”, Client 1 detects previous server was lost and attempt to connects to Server 2.

Error message is displayed to prompt user to re-send message.

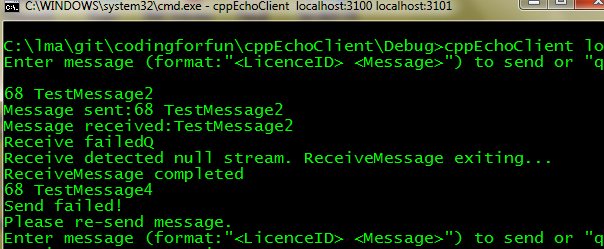


Client 1 receives echoed message “TestMessage3” as normal.



Similarly, on next send message “68 TestMessage4”, Client 2 detects previous server was lost and attempt to connects to Server 2.

Error message is displayed to prompt user to re-send message.

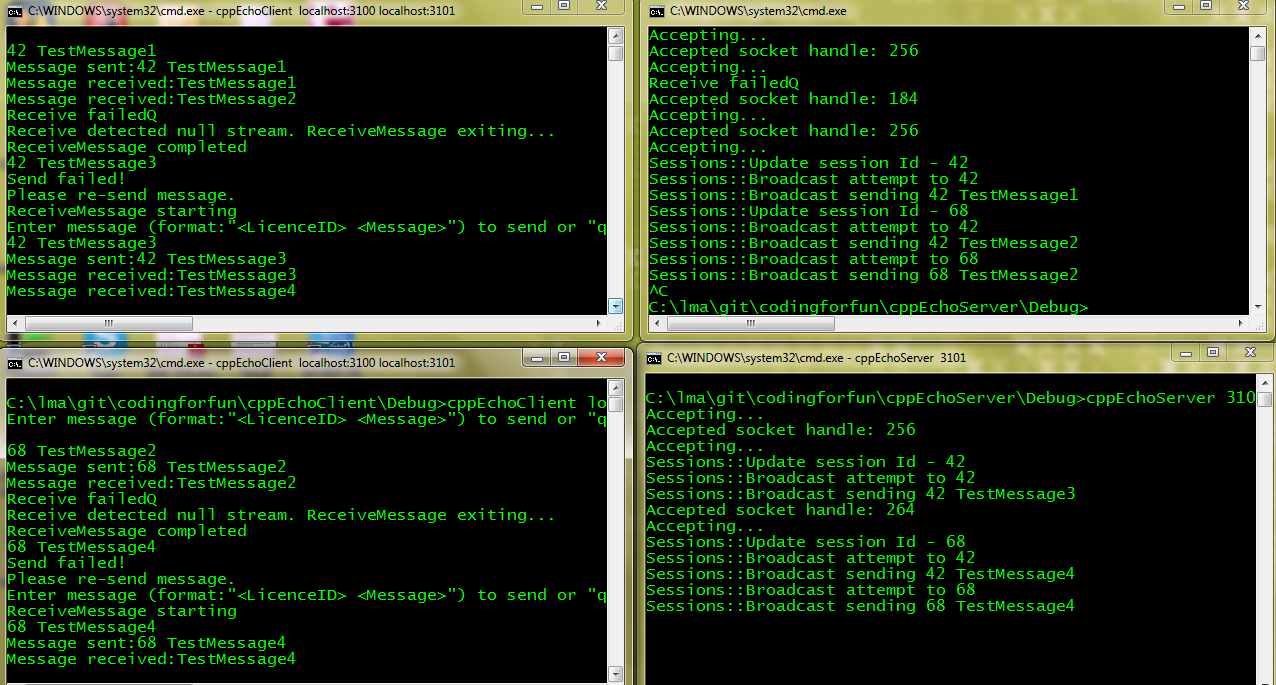


Client 2 re-sends “68 TestMessage4” to Server 2.

Client 2 receives echoed message “TestMessage4”.

Client 1 also receives echoed message “TestMessage4”.

We are back in business ☺



### Quiting the Client

Finally, enter “quit” to quit the client.

