

**1. General Information and specification**

The product is an intermediate echo server and client that covers the following features and specifications.

1. An Echo server that echoes back the message sent to the server.
2. Non GUI implementation in C++. This project utilizes C++ 14.
3. Not using 3rd party components such as Boost ASIO. This product can be built from the get go.
4. Works on Windows and can be open with Visual Studio 2017 as well as Visual Studio 2019 (Community Edition).
5. Can be built in Linux-like platform. Utilizes CMake 3.10 for build (default with Ubuntu 18.04)
6. Supports binary based protocol with mandatory header and variable sized body.
7. Server supports multiple client sessions.
8. Server utilizes thread pool.
9. Server supports multiple instances and connects with each other.
10. Server messages are broadcasted to all clients connected and to the clients connected to its peers as well.
11. A very flexible and configurable Client application.
12. Client can connect to any server by providing a server list thru a file as argument or as a command line parameter.
13. Client reconnects to a server when connection is lost.

**Updates for V2.0**

Version v2.0 contains fixes and update to prevent data error and crashes;

1. Client can now run as listener (--listener), it will listen to messages publish across the network. It will overrule the other message parameter and will not send a message to the server.
2. Server peers are only connected thru a single connection, server either connects to (Active) or listen from (Listener) its peers. Duplicated connections are ignored or dropped. Server utilizes the server ids to do the checking. **TO BE UPDATED WITH SERVER CONNECT TYPE PARAMETER, DISPLAY SCREENSHOT ON SAMPLE**
3. Reads and write for each socket connections are divided into two sockets, where the read thread supplies the packets to the Write thread to speed up reading from socket buffer and consumes arriving data faster.
4. Converted the sockets to non-blocking.
5. Supported select() and FD\_SET() to check whether sockets are ready for read and/or write operations to prevent previous limitation wherein it ran out of network buffer and caused crash.

**2. Build and environment**

For Windows, Using Visual Studio 2017 or 2019 Community Edition;

1. Double click the solution file CrytekCppTest.sln
2. In the Solution Explorer Window, right click the Solution and select ‘Build’ or ‘Rebuild Solution’ for either Debug and/or Release Configuration.
3. Build result can be found in Demo/Windows folder in the main directory in preparation for demo run.

For Linux-like platform,

The applications are compiled with atleast *CMake 3.10 with gcc-7.5.0* both are default for Ubuntu 18.04, however the package includes a build script that installs the build essentials of the OS, run the script to setup the environment and build the application.

*./build.sh*

After a successful build, both binaries will be available in Demo/Linux folder ready for execution.

**3. Run time configuration**

When built properly, the project generates two files

*./Client.exe* or *./Client* in Linux-like platform

./Server.exe or ./Server in Linux-like platform

Each program provides a customizable options for its run time execution. These options are settable thru its command line.

**Server Command Line Options**

|  |  |
| --- | --- |
| -h,--help | Display Help (this information) |
| --peers <filename> | loads the peers server list from file (optional)  Server can run even without specifying peers. |
| -i,--id <string> | Server identity (REQUIRED) |
| -p,--port <number> | Server port to listens to (required) |
| -thread-count <number> | Max number of threads in the thread pool to. (default 100) |
| --expand <true/false> | Will the server expand and create more thread if it runs out (default false). |

Example: *./Server.exe –id server1 –port 9000 –peers peersconfig.txt*

**Client Command Line Options**

|  |  |
| --- | --- |
| -h,--help | Display Help (this information) |
| --frequency <number> | Number of times this client will send the message, Set to (-1) to continuously sends forever, (default is 1). |
| --reconnect-retry <number> | Number of times to retry reconnect when disconnected from server (default is 3) |
| --reconnect-delay <number> | Number of seconds to wait before next reconnect attempt in seconds (default is 5 seconds) |
| --message <string> | Set the Message to send to the server (default is 'Hello World'). Note to place the value in between quotes (“) |
| --delay <integer> | Delay in sending the next message in seconds (default is 5 seconds). |
| --host <address> | Ip address or host name of the server to connect to |
| --port <number> | Port of the host name of the server to connect to |
| --servers <filename> | Specify the file that contains the list of servers to connect to. The client connects to the first address in the list, if connection fails then it will move to the next one. |
| -i, --id <string> | Client identity in the network (REQUIRED) |
| -l,--listener | Run as listener (overrule other parameters; frequency, delay and message) |

Guideline for setting up server list:

When --host and --port is specified, it takes higher precedence than --servers. That means the client will attempt to connect to the specified host first, if connection fails then it will go thru the list provided in the server list file.

Example: *./Client.exe –id client\_1 –host localhost –port 9000 —frequency -1 #spams forever*

**Config files**

Both the Server and Client applications accepts files as configuration especially for peers (Server) and server list to connect to (Client).

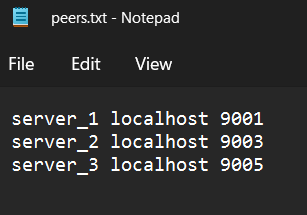
This application implemented a very basic configuration file loader and parser where values are separated via a white space and loaded each line.  
See bellow information for each configuration file.

Server Peers list (--peers)

Contains the list of peers to connect to

Syntax:

*ServerID(space)host address(space)port(newline)*

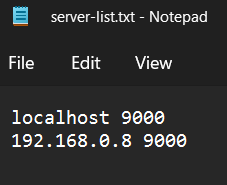
******

See Section 4. Guideline for setting up port for peer servers.

Client’s Server List (--servers)

Contains the list of server to connect to

Syntax

 host address(space)port(newline)

**4. Guideline in running the applications**

**Server and Client Identity (ID) guideline**

Both Server and Client require an id (-i, --id) as a parameter when running the applications. The ID serves as the login and identity of each actors in the network. The ID is also used to identify separation between a server and client specially in routing to avoid cyclic sending of packets by marking the packet with the origin information.   
The ID for servers is also use for basic handshake, servers and its peers must have the prefix “**server\_**” to identify itself as a server and a peer, when an application connects to the port specified for peers (see next section (Port Guideline) but without the prefix, the server will ignore and deny further transaction thus failing the handshake.

For client application connecting to the basic server port, any client ID can be use.

A basic ID checking is implemented (CEchoDistributedServer::ValidateIdentity) to check ID duplicates connected to one server and does not support checking across the network , supporting it might need persistent data information such as storing in a Database which is out of scope for this exercise.

ID should not contain white space as well, validation is done in server and client.

**Port Setting Guideline**

It can be observed in the demo that a single server is using two ports for listening (Ex: for server 1 listens to port 9000 and 9001).   
Servers are launched at specific port and automatically opens a port for listening to its peers (port+1).  
This is to establish a clear separation for server and clients for our basic handshake implementation. Applications connecting at the basic port (ex: port 9000) do not go thru the server identity handshake check.  
This is to simulate basic security checks and handshaking, an application must have the correct ID and must connect to the correct port to be allowed in the network as a peer.

In real world network architecture, a more advance handshake and checks implementation is needed for both client and server peers.

**5. Demo**

The package together with this documentation contains a demo environment for both Windows and Linux. It is pre-requisite to build the project first in either platforms before running the demo (see Section 2 of this document).

Windows *./run-demo.bat*

Linux *./run-demo.sh*

For Linux, it is recommended to use Gnome based OS like Ubuntu as it utilizes gnome-terminal (command) to open new window to launched the programs.

**z**

**Setup**

Running 3 Servers, 1 client connected to each servers, except for Server 2 which has 2 clients.

1. Server 1

- No peers connected at boot

- 100 thread count in pool

- Not expanding

1. Server 2

- Connect to Server 1 and Server 3 at boot

- 100 thread count in pool

- Not expanding

1. Server 3

- Connect to Server 1 and Server 2 at boot

- 100 thread count in pool

- Not expanding

1. Client 1

- Frequency -1 (sends forever)

- connects to Server 1

- send delay 1 seconds

- sends message “Hello from client 1”

1. Client 2 and Client 4

- Frequency -1 (sends forever)

- connects to Server 2

- send delay 1 seconds

- sends message “Hello from client 2” and “Client 4 says hi”

1. Client 3

- *Frequency 10*

- connects to Server 3

- send delay 1 seconds

- sends message “Hello from client 3”

Expected Result

1. Each client will receive their own message sent to the server they are connected to
2. Each client will receive the messages from other clients.
3. Client 3 will disconnects and close after 10th message sent.

By editing the scripts in the demo, an experiment can be made to test various features of the programs.

Reconnection Test

1. Run 1 server and 1 client with client sending forever (frequency -1), after few seconds, close the server window. Client will detect disconnection, it will pause the Send thread and will attempt reconnection.
2. Run the server again (using same port), unless the retry attempt in the client side runs out, it should reconnect and resume send operation.

Thread pool expansion test.

1. Run 1 server and set number of thread to only 2 (numOfThreads 2) in the server config file.
2. Run client connecting to this server. Server should run properly and client should be receiving messages as expected.
3. Run another client that connects to the same server, Server should show Constant creation and deletion of new thread, both clients should work normally.
4. Close either one of the Client. Server should stop expanding/de allocating thread and both application behaves normally.

**4. Software Design**

**General Setup Overview**

Below is structure from the demo.

5

Server 1

port+1

port

3

3

Server 2

port+1

port

Server 3

port+1

port

4

4

2

1

Client 1

Client 2

Client 3

Client 4

Overview

* (Blue Arrow) Clients connect to a server at basic port, peers connects to each other thru peers port (port+1).
* (Read Arrow) Client sends message to the server;
  1. Server accepts the message, adds an origin information by appending it server ID and some arbitrary information.
  2. Server broadcasts the message to its connected clients
  3. Server broadcasts the messages to its connected peers
  4. Server receiving message from peer sends it to its own clients.
  5. Server 3 will still send the message to Server 1, even though it already received and processed it earlier, he won’t processed it any longer. See section **Preventing Cyclic and duplicate data.**

To prevent cyclic sending, all servers checks for the origin information before broadcasting it to its peers or connected clients. In the above example, Server 1 and 3 won’t be sending the same message back to Server 2 as the message originated from it.

**Identity check**

As described in the earlier sections, server and client applications needs an ID to be accepted in the network (See Section 4). Application’s Identity (ID) should be immediately sent after initial connection.

The server confirms the identity by sending the ID back to the sender (client/peers) if It passed necessary checks. Receiving party checks reply from server if It matched the ID it sent previously.

**Server design**

The core of the server application is the thread pool and queue. A task can be any operation in the server, including handling active socket connections, Identity validation and broadcasting messages.

Thread

Main Thread  
(CEchoDistributedServer)

Thread

Thread

Task Queue

…

Thread

See implementation at ThreadPool.h and ThreadPool.cpp.

**Thread Pool Expansion**

After every new connection (from a client or to/from a peer) a task is created which will eventually be picked up by a thread in the pool, this task will last until the socket connection is terminated. Since clients have an option for a persistent connection and sends messages until it is closed, there is a possibility of running out of threads.

This implementation of thread pool supports temporary thread expansion, this option is not enable by default and must be explicitly enable (See Section 3).

1. All thread is launched with an ID/index for easy tracking.
2. An atomic counter monitors the number of active thread doing work on a task.
3. When a new task is queued and there is no available thread, a new thread is created.
4. After a thread completes a task, it check the total thread count, if it exceed the normal count, it will kill itself by ending its loop and marking itself by adding its ID to the finished list.
5. The list is then iterated to deallocate dead thread in the next call to QueueTasks().

See implementation at CThreadPool::QueueTask(),CThreadPool::ProcessTask() and CThreadPool::Execute() implementation.

In Summary, when thread pool expansion options is enabled, it creates and kills extra thread almost everytime it exceeds the normal count.

**Handling Socket Connections**

The server handles connections from 3 different cases

1. Client connections from basic port (ex: 9000)
2. Connections from peers port(ex: 9001)
3. Connections to peers

All socketing connections are stored in a single map with the ID as the key, the server differentiates between client and peer connections using this key.

**Preventing Cyclic and duplicate data**

The following algorithm was implemented to prevent Cyclic sending and duplicate of data.

1. A variablem\_msgId was created to keep track for every unique transactions sent from client to a server.
2. When packet arrives to the server, the server appends its own ID and latest transaction count (m\_msgId) of the message separated by a space (Ex: “server\_1 0”) it becomes the transaction ID of the packet.
3. In CEchoDistributedServer::BroadCastMessage, it checks whether the transaction ID has been processed by the originating server before. If it’s a new transaction, it cached the transaction ID to a map (m\_finishedTransactions). The map uses the transaction ID as the key and the time it was added.
4. When a new transaction has been cached, the Cleanup task is triggered (CEchoDistributedServer::CleanUpTransaction). It will remove transactions older than FINISHED\_TRANSACTION\_SECS, which is defaulted to 60 seconds.
5. The CEchoDistributedServer::BroadCastMessage checks for the origin information (transaction Id) of the packet against the current socket connection ID to send or discard the sending of data.