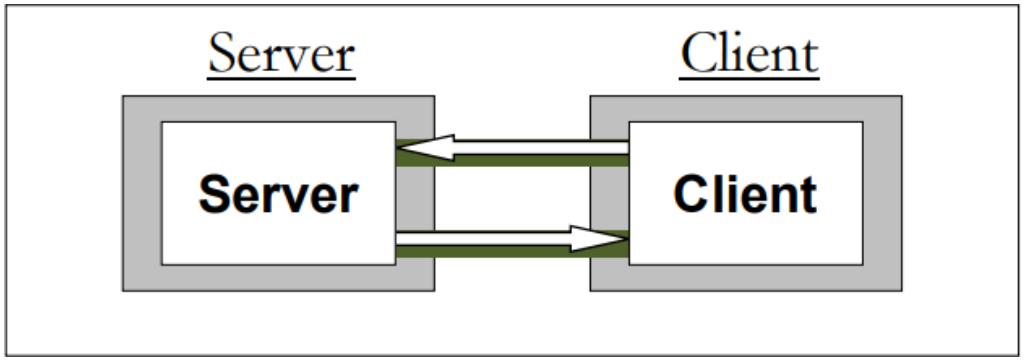
# Lab.1

# Spaghetti Relay

## Objective

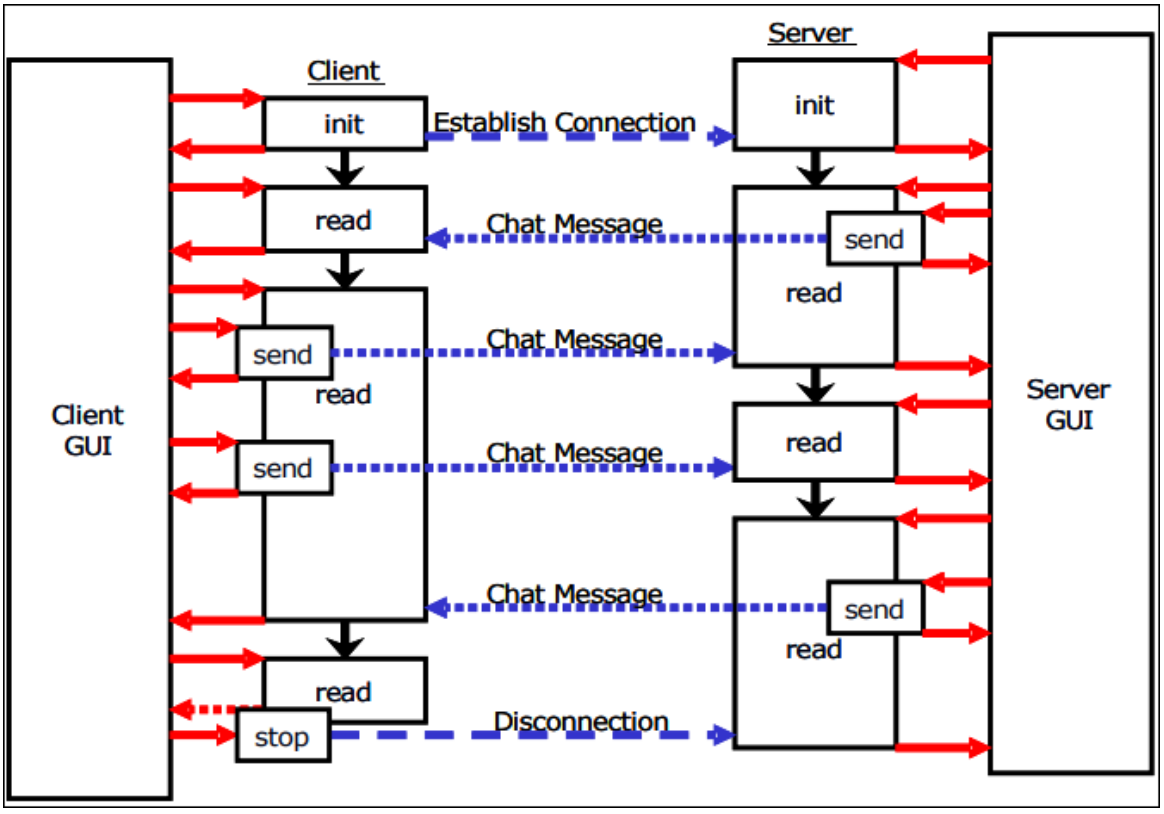
The Spaghetti Relay lab is intended to introduce basic networking concepts involved in binding, listening, and accepting server connections and connecting via clients, as well as sending and receiving simple messages via TCP/IP. This project provides a C# front end; students do not need to work on the C# front end but must create stand-alone classes in C++ which will be called by the GUI. Students' code will be in C++ only.




The front-end deals with two dynamic linked libraries (DLLs), Client.dll and Server.dll, through the ClientWrapper and ServerWrapper interfaces. These wrappers instantiate and call the Client and Server classes. Students should write both the Server class and Client class from scratch. If the specification is follow, the C# front-end should act identically to the example executable. The Client and Server classes need not initialize any platform-specific code as this is all done in the wrapper classes. For cross platform purposes both should include the "platform.h" header file. They should also include the "definitions.h" file to identify network errors.

# Client/Server Behavior

Here is an example of the client & server behavior in action:




- Server initializes and waits for a connection; Client connects to Server, completing initialization of both sides. Both sides start waiting for messages from the other side. (via read() method)
- Server's GUI receives a message from the user and begins a new thread with a **sendMessage()** call. This sends a message to the Client.
- Upon receipt the Client returns to its GUI for display, then begins waiting for messages again.
- Client's GUI receives a message from the user and begins a new thread with a **sendMessage()** call. This sends a message to the Server, which returns to its GUI for display before listening again. This occurs twice
- Server receives another user message and send it to the Client, which displays it.
- Client's stop() method is called, triggering a shutdown. The disconnection of the socket causes the Server to shut down as well.
- Finally, both GUIs are closed and the application terminates.

# Initializing the Connection

Spaghetti Relay is an introductory exercise in networking, so students should try to complete it on their own without too much help, but here are a few pointers for the initialization of the Client and Server

## The Client

A client initialization method might look something like this:

```
init(address, port)
{
      Create a socket (socket().)
      Convert the address and port into a sock_addr structure.
      Connect the socket to the sock_addr (connect().)
      NOTE: If there are errors in any call, return the correct error
code. Otherwise, return SUCCESS.
 }
```

## The Server

A server initialization method might look something like this:

```
init(port)
{
      Create a socket (socket().)
      Bind the socket to the specified port (bind().)
      Set up a listening queue for connections (listen().)
      Wait for and accept a single connection from a client (accept().)
      NOTE: If there are errors in any call, return the correct error
code. Otherwise, return SUCCESS.
}
```

# Message Format

The message format for both client and server is simple. The first byte is the length of the message (from 0 to 255.) This byte is followed by the message in its entirety. **The length must be handled**.

# Error Checking

Every networking function must be error checked. If the function returns a SOCKET then the error check would be to compare it with the INVAILD_SOCKET macro. If the function returns a number then the error check would be to compare it with the SOCKET_ERROR macro. There are some other special cases that you can look up on the MSDN.

# Server

Server shall have the following public methods with the following behaviors for error checking:

***int init(uint16_t port)***

Return Values On a successful connection, returns **SUCCESS**.

If error was encountered when binding the socket, returns **BIND_ERROR**.

If error was encountered when creating a socket or listening, returns **SETUP_ERROR**.

If error appeared during accept and was not caused by shutdown, returns **CONNECT_ERROR**.

If error appeared during accept and WAS caused by shutdown, returns **SHUTDOWN**.

***int readMessage(char* buffer, int32_t size)***

On a successful receive and write to buffer, returns **SUCCESS**.

If error appeared during receipt and was not caused by shutdown, returns **DISCONNECT**.

If error appeared during receipt and WAS caused by shutdown, returns **SHUTDOWN**.

If the message is longer than size, returns **PARAMETER_ERROR**.

***int sendMessage(char* data, int32_t length)***

On a successful write to stream, returns **SUCCESS**.

If error appeared during send and was not caused by shutdown, returns **DISCONNECT**.

If error appeared during send and WAS caused by shutdown, returns **SHUTDOWN**.

If length is less than 0 or greater than 255, returns **PARAMETER_ERROR**.

***void stop() // no returns .***

# Client

Client shall have the following public methods with the following behaviors for error checking:

***int init(uint16_t port, char* address)***

On a successful connection, returns **SUCCESS**.

If address is not in dotted-quadrant format, returns **ADDRESS_ERROR**.

If error was encountered when creating a socket, returns **SETUP_ERROR**.

If error appeared during connect and was not caused by shutdown, returns **CONNECT_ERROR**.

If error appeared during connect and WAS caused by shutdown, returns **SHUTDOWN**.

***int readMessage(char* buffer, int32_t size)***

On a successful receive and write to buffer, returns **SUCCESS**.

If error appeared during receipt and was not caused by shutdown, returns **DISCONNECT**.

If error appeared during receipt and WAS caused by shutdown, returns **SHUTDOWN**.

If the message is longer than size, returns **PARAMETER_ERROR**

***int sendMessage(char* data, int32_t length)***

On a successful write to stream, returns **SUCCESS**.

If error appeared during send and was not caused by shutdown, returns **DISCONNECT**.

If error appeared during send and WAS caused by shutdown, returns **SHUTDOWN**.

If length is less than 0 or greater than 255, returns **PARAMETER_ERROR**.

***void stop() // no returns .***

# Examples and notes

Example of an error check on the send function with graceful disconnects:

```
int sizeOrError = send( socket, buffer , 2 , 0 );

if(sizeOrError == 0) // graceful disconnect
      return SHUTDOWN;

if(sizeOrError == SOCKET_ERROR) //  ungraceful disconnect
      return DISCONECT;
```

When cleaning up a Socket you must call shutdown then closeSocket:

```
shutdown( socket, SD_BOTH);
closeSocket(socket);
```

***Tip**: Always clean up every socket that is created. Even if it is only a listening socket.*

***Tip**: TCP with stream requires partial message loops see the slides for an example.*

***Tip**: Don't forget the send the length. Without the length in front of the message you won't know how much to receive. That also means you should receive the length first before you receive the body of amessage.*

***Note:** readMessage() - the size parameter isn't the size of the message but rather the number of characters that can be sent in a single message and it is set to 256*

**What to Submit?**

Header and source files:
- *Client.h, Server.h, Client.cpp, Server.cpp*

A *Wireshark capture* showing the packets exchanged between the client and the server for the TCP connection initiation and termination, as well as the handling of the message length