FULL SAIL UNIVERSITY

Weight: 9%

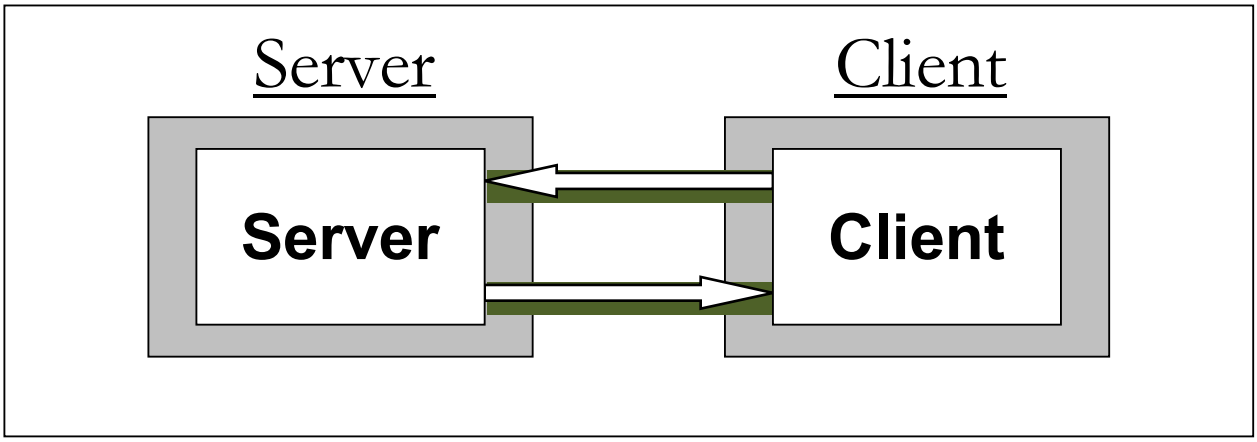
Spaghetti Relay

Table of Contents

Overview....1
Client / Server Behavior....2
Behavior of the client and server....3
Initializing the Connection....3
Message Format....4
Server Class....5
Client Class....7

Overview

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.

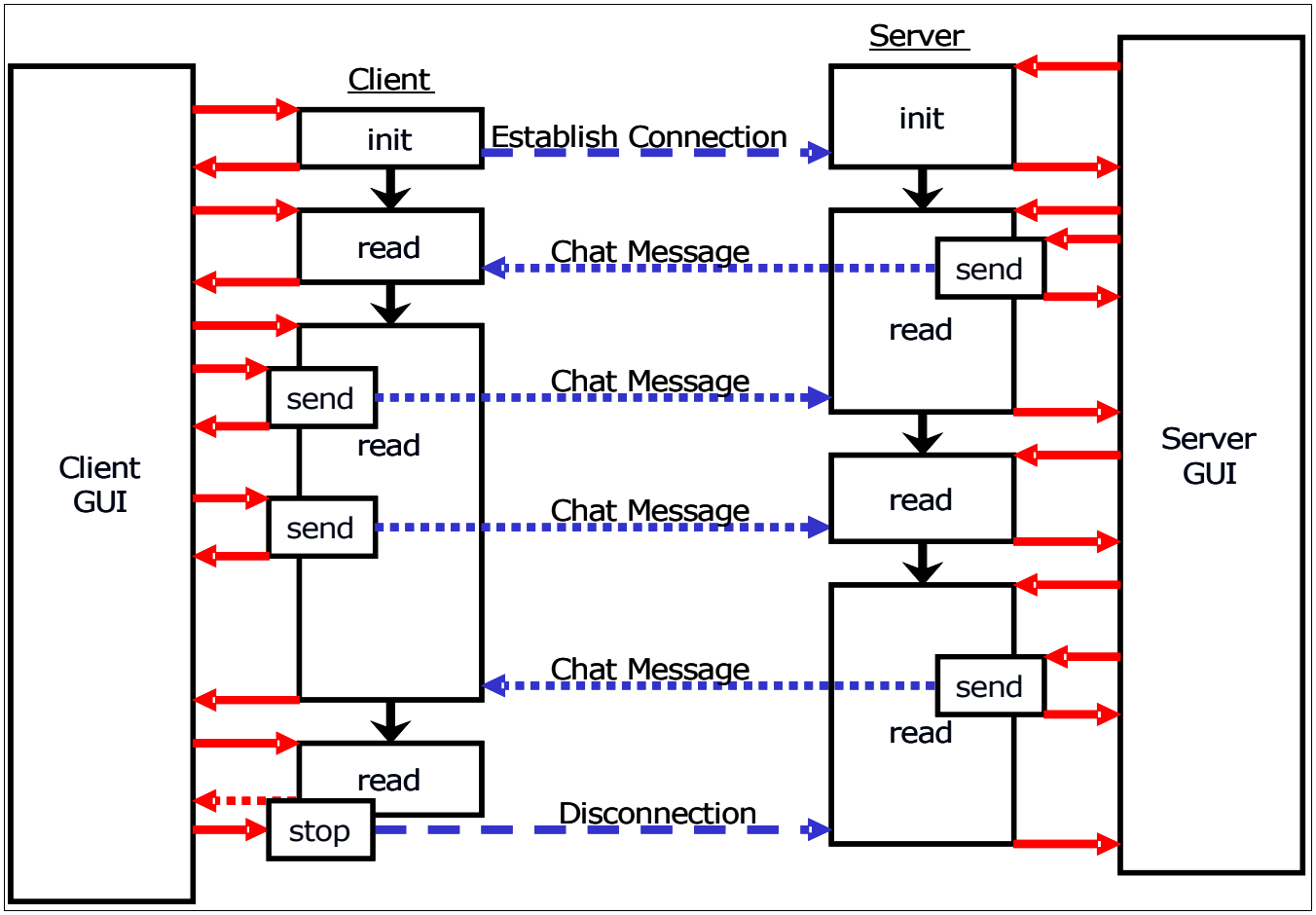


The front-end deals with two dynamic linked libraries (DLLs), Client.dll and Server.dll, through the ClientWrapper and ServerWrapper interfaces. These wrappers instantiate and call the Client and Server classes. Students should write both the Server class and Client class from scratch. If the specification 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.

As the code may be run from a multi threaded environment, take care about the order of your operations (see shutdown methods for client and server.)

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.

Behavior of the client and server

This is the general behavior of the client and server:

- During Server initialization, the class should create a socket (**`socket`**), bind it (**`bind`**), set up a listening queue (**`listen`**), and wait to accept a connection (**`accept`**.)
- During Client initialization, the class should create a socket (**`socket`**) and connect to the server (**`connect`**.)
- Client and Server should end initialization upon a successful connection or failure.
- If Client or Server's sendMessage() method is called, class should send message to the other side.
- If the Client or Server's read() method is called, the class should read until it has a complete message and then return.
- If Client or Server's stop() method is called, it should mark the class as inactive and close the socket in such a way that causes any active methods to terminate gracefully.

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.**

Server Class

The Server class is responsible for listening for and accepting a single connection to a single client. Upon termination of the connection for any reason, the server will be shut down as defined below. The server's primary functions are to accept a connection, send messages, receive messages, and terminate when so instructed.

The Server class shall have the following public methods with the following behaviors:

int init(uint16_t port)

Should listen for and accept a connection from a single client on the specified 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)

Should read in one message from the client and write it to **`buffer`**, not to exceed **`size`**.

Return Values

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)

Should send the message **`data`** of size **`length`** to the client.

Return Values

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()

Should shutdown and close all open sockets and signal the server to gracefully exit (with any method returning `SHUTDOWN` if currently executing.)

HINT: Mark the server as shutting down before shutting down sockets and closing them! Otherwise, other methods cannot tell the difference between a graceful shutdown and an error.

Client Class

The Client class is responsible connecting to a server in order to transfer messages. Upon termination of the connection for any reason, the client will be shut down as defined below. The client's primary functions are to connect to a server, send messages, receive messages, and terminate when so instructed.

The Client class shall have the following public methods with the following behaviors:

int init(uint16_t port, char* address)

Should connect to a server on the specified **port** at the dotted-quadrant format **address**.

Return Values

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)

Should read in one message from the server and write it to **buffer**, not to exceed **size**.

Return Values

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)

Should send the message **data** of size **length** to the server.

Return Values

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()`

Should shutdown and close all open sockets and signal the client to gracefully exit (with any method returning `SHUTDOWN` if currently executing.)

HINT: Mark the client as shutting down before shutting down sockets and closing them! Otherwise, other methods cannot tell the difference between a graceful shutdown and an error.