**1 Introduction**

This project is a basic implementation of the client/server model. The only difference in this system is that servers may also initiate communication with other servers, effectively acting as clients. The services that the servers offer are computations of functions with numerical parameters. Clients may request these services by giving the server a function to perform and the correct amount of parameters for that function.

**2 Client**

The client’s job is to allow users to initiate communication with servers and request a service. In order to do this, the client provides the user with a command prompt interface where a given number of commands may be entered. The commands a user may enter are the following:

*CONNECT*

This allows users to change the address of the server they are going to connect to. By default the client will automatically try to connect to the address that was given as a command line argument on startup. Once this command is entered, the user will have to enter the name of the server, in the format *hostname:port,* that they would like to connect to next to.

*QUIT*

This will shut down the client process.

*[function name] [param1] [param2] […]*

This allows users to specify the name of the function they would like the server to perform, along with the parameters for that function. When the user enters this type of command, the client will send a client request message directly to the server and wait for a response. The full details of the client request message will be explained in section 2.3. The server can reply back with one of the five messages, which will be described in the next section (2.1). When the client receives the server’s response message, it will display the entire message in the console for the user to read.

**2.1 Interpreting server reply messages**

Depending on the command the client sent to the server, the server can reply back with any of the following messages:

*SERVER/UNKOWN*

When a server sends a message with this header, it means that the server does not know how to perform the function that the client requested, but it does know which servers can perform this function. Along with this header, the following information will be sent: *function name, number of parameters, and list of servers that can perform this function.*

*SERVER/MISMATCH*

When a server sends a message with this header, it means that server knows the function that the client has requested, but the number of parameters for that function does not match the number of parameters the client sent to the server. Along with this header, the following information will be sent: *function name, number of parameters the client sent, and the number of parameters the actual function requires.*

*SERVER/BUSY*

When a server sends a message with this header, it means that the server knows the function that the client has requested and the server is capable of performing that function, but it is currently busy. Along with this header, the following information will be sent: *function name, number of parameters, and list of other servers that can perform this function.*

*SERVER/RESULT*

When a server sends a message with this header, it means that everything went fine and the server was able to perform the function. Along with this header, the following information will be sent: *function name, number of parameters, number of return values, the result of the function, and the delimeter character that separates return values.*

*SERVER/ERROR*

When a server sends a message with this header, it means that the server did not recognize this function at all.

**2.2 Running and testing**

Before you try to run and test the client, at least one server must be up and running. When this condition is fulfilled, the client may be started with the name of the running server as the first and only command line argument.

**Ex:**

./client localhost:4005

The name of the server must be in the following format *host:port*. Once the client starts up, it will give the user instructions on the three possible types of commands (*CONNECT, QUIT, function*). The user may enter any of these three types and commands; and it is recommended that if you are testing the system, you should try to see what functions the server you are connected to “knows” about but cant perform. If the server “knows” the function but can’t perform it, it will give a list of other servers that can perform it. After getting the list of other servers that can perform the function, you can try to request that function on those servers by using the *CONNECT* command to switch to any of those servers.

**2.3 Protocol for contact with servers**

The client has only one type of message it can pass to the servers as opposed to the five different types it can receive from the servers. The message that the client sends to the server is a request message for a function computation. The message the client can send is the following:

CLIENT/REQUEST\n

[function name] [param1] [param2] […] \n

(NOTE: There are spaces between all of the values in the second line of the client message, even between the last parameter and the ‘\n’ character! This makes it easier for the server to process the message)

**2.4 Implementation and basic algorithm**

The client will keep looping and taking user input until the user input is *QUIT*. Every time the client loops, it asks the user for its input and it will process one of the three possible commands mentioned in Section 2 (*CONNECT, QUIT, function).*

**3 Server**

The server’s job is to take process messages from clients and other servers. If the message is from a server (*SERVER/HELLO*), it will process the message and store the name of the server along with its functions. If the message is from a client (*CLIENT/REQUEST)*, it will process the message and send back one of the replies mentioned in Section 2.1.

**3.1 Running and testing**

If there are no servers currently running, you may run the server by only passing it the path of its config.txt file as a command line argument.

**Ex:**

./server /Users/Config/configB.txt

The config file must contain the following information:

Hostname:Portnumber

[function1] [number of parameters]

[function 2] [number of parameters]

[…] […]

If there are already servers up and running, you must specify them as the next command line arguments with a space separating the different servers.

**Ex:**

./server /Users/Config/configB.txt localhost:4000 localhost:4005

If a server is started when other servers are already up and running, the newly started server will send a *SERVER/HELLO* message to servers it just received as command line arguments. In reply, those receiving servers will send their own SERVER/HELLO message to the newly started server. Once the send/reply HELLO messages are done, the server will then loop forever waiting for new connections from either clients (*CLIENT/REQUEST*) or new servers (*SERVER/HELLO*)

**3.2 Protocol for contact with servers and clients**

The server can receive any of the following types of messages:

*CLIENT/REQUEST*

This is a message from a client, in order to request a service of a function. Details of this message were included in Section 2.3.

*SERVER/HELLO*

This message is sent from other servers in order to let the receiving server know their name and what functions they can perform. When a server receives this message, it will store the information of the server name and functions in its list of servers and functions.

The actual message format is the following:

SERVER/HELLO\n

hostname:port\n

[function1] [number of parameters]\n

[function2] [number of parameters]\n

[…] […]\n

\r

(NOTE: The ‘\n’ and ‘\r’ character are used by the receiving server to indicate a end of line and a end of message)

**3.3 Implementation and basic algorithm**

**main(int argc, const char \* argv[])**

When a server first starts, it will look into its config file, which is the path of the first parameter you gave it on startup, and begin to set itself up. It will store its name and list of functions inside its global variables (*whoami,* and *functionList* which will be described soon*)*. If the server received names of other servers as startup parameters, (in the correct format *hostname:port)* then for each of these servers, it will send a *SERVER/HELLO* message. After sending the *SERVER/HELLO* message, the server will also receive a *SERVER/HELLO* message back from all of the original receiving servers. This happens because we not only want to tell the servers what the newly created server knows, but we also want to tell the new server what the existing servers know. When it gets a *SERVER/HELLO* message in reply, the original sender will process this message by calling *processServerHello*. The details of the *SERVER/HELLO* message were included in Section 3.2. Once the server is done processing all of the reply *SERVER/HELLO* messages, it will begin to loop forever waiting for processes to connect to it. In order to do this, it will *bind* to a socket, *listen* to that socket, and wait to *accept* a connection from that socket. Once a process connects to the server, it will spawn a new thread to handle the request and that new thread will continue its execution in the *request\_handler* method.

**request\_handler(int clientSock)**

The request\_handler method takes one parameter, which is the socket descriptor return from the call tothe *accept* method. Request handler will begin reading from the socket that was created from the *accept* call. It will process and store the first line of the message one character at a time until it hits a ‘\n’ character. This ‘\n’ character will let the server know that this is the end of the message header. Once it has stored this message header, it will check to see if it is a *SERVER/HELLO* header or a *CLIENT/REQUEST* header. If it is a *SERVER/HELLO* header, it will call on the *processServerHello(clientSock, 1)* method.

**processServerHello(int clientSock, int reply)**

The processServerHello method takes two parameters: socket descriptor, and a reply value that should either be 1 or 0. This method begins by creating a mutex in order to protect the global variables it will soon change (*functionList and serverList).*

*functionList* is a struct that holds the following values: a string *name*, a int *parameterNumber*, a int *canDo*, and a array of strings *otherServers*.

*name*: The name of the function

*parameterNumber*: The number of parameters required for the function

*canDo:* A value always set to 1 or 0 depending on if the server can perform this function

*otherServers:* A array of strings that holds the name of all other servers that can perform this function

The method will then go through its global variable *serverList* and add the name of the server that sent the *SERVER/HELLO* message to the end of this list. Once it does that, the server knows that all of the following lines it reads from *clientSock* will either be a function (with its number of parameters) or the ‘\r’ character which signals the end of the message. The server will begin to process each of these functions (separated by the ‘\n’ character). It will store the function name and number of parameters and check to see if it “knows” of that function in its global variable *functionList*. If it does know the function, then it will add the server name to the end of that functions *otherServers* array (if it isn’t already there). If it doesn’t know the function, then it will add the function to the end of *functionList* and add the name of the server to that newly created function’s *otherServers* array. Now that the global variables are not being accessed anymore, the method will destroy the mutex. After this, the method will check to see if the parameter *reply* is set to 1. If it is, then the server knows that it was not the one that first sent the *SERVER/HELLO* message and it should reply back with its own *SERVER/HELLO* message. In order to make debugging easier and to see the interaction between the different servers, the following message will be printed at the end of this method:

“Successfully received a Server hello message from *[name of the server]*”

**processClientRequest(int clientSock)**

This method will begin by printing “Client connected” in order to make debugging easier. After that, it will read an entire line from the *clientSock*. The server knows that this line will be formatted in the way described in Section 2.3, so it will break the line up by using strtok(line, “ “). It will store the function name in a string, and the following parameters in an integer array. It will then go through *functionList* and check if it even knows about the function. If it does know of the function, it will check if the number of parameters it received match that functions *parameterNumber*. If it doesn’t match, then it will send the *SERVER/MISMATCH* message to the client. If the parameters do match and the functions *canDo* variable is set to 1 (meaning the server has the capability to perform the function), it will create a random number between 0 and 1. If the number is 1, then the server will send a *SERVER/BUSY* message, that includes the names of all the other servers that can perform this function. It gets this list by going through the functions array *otherServers*. If the random number is 0, then it will execute the function name with the given parameters and will send the results back in a *SERVER/RESULT* message. The way the functions are executed is discussed in Section 3.4. Now, if the function’s *canDo* variable is set to 0, this means the server does not have the capability to perform this function and it will send a *SERVER/UNKNOWN* message that includes a list of the names of all the other servers that can perform this function (found in the exact same way as the *SERVER/BUSY* message). Finally, if neither of these things happen, then the server will send a *SERVER/ERROR* message meaning the request was invalid or the server has never heard of the function.

**3.4 Functions**

All servers actually know of all of the functions and are realistically able to perform them all, but in order to comply with the assignment requirements we only let servers perform a function if its *canDo* variable is set to 1. The functions are defined in the server normally, and the names of the functions are stored in an array of strings. The function pointers are also stored in an array in the same matching order (in order to provide a easy way to “map” a function name to the actual function pointer). Now because the functions take different parameters, we must create a generic function pointer and cast them all to this typedef. This was implemented in Lines 74-80 in the code. The global list of functions include the following:

**int multiply(int a, int b, int c)** (returns a\*b\*c)

**int plus(int a)** (returns a+1)

**int subtract(int a, int b)** (returns a-b)

**int add(int a, int b)** (returns a+b)

**int minus(int a)** (returns a-1)

**void special(int a, int b, char result)** (sets result to be “a\*10 b\*10”)

**4 Additional Information**

In order to keep this project simple and focused on the task of getting a distributed system up, it assumes good intentions from the user. It does not fully check for malformed user input that does not follow the protocols of the system. The system also assumes that the config file is in proper format.