RPC Manual

Data Marshalling

In order to send data over the network, we created functions that handle reading and writing standard C types (such as int, short, etc) and their array counterparts into byte buffers, which for simplicity is represented as char arrays in the system. These functions can be found in the file buffer.cc. In general, these functions take in some char*, which represents the starting point of the buffer. From there, the write functions copy data from its arguments into the buffer, while the read functions copy data from the buffer into its arguments (which are references types). Then the read/write function returns a new char*, which represents the next byte in the buffer that's available to read or write.

Note that when converting standard C types into bytes, we make some very strong assumptions about these data types. Specifically, we assume that a character is 1 byte, a short is 2 bytes, an integer is 4 bytes, a float is 4 bytes, a long is 8 bytes, and a double is 8 bytes. We don't assume the endianness of our data and use htonl, htons, ntohl, ntohs to convert short and integer. For float, long, and double, we created out own functions to swap byte orders. We are assuming that all machines that our RPC will run on will use the same binary representation for floating points. This means that if our RPC system is running on machines with incompatible binary representation of floats or doubles (where the incompatibility extends beyond endianness), then we cannot guarantee correctness of results. While the size and identical floating-point representation are strong assumptions, they were not broken while we were testing our code on the linux.student environment.

From these basic functions, we built out the tools that handle communication between server, binder, and client over the network. The format and type of message that are sent between the server, binder, and client follow the recommended protocol detailed in the assignment. One particular assumption that we made that was we assumed that hostname of machines will be at most 255 characters (256 including \0 character). This limit is a macro defined in helpers.h and can be modified if needed.

The code handling communication exists in sender.cc and receiver.cc, which send and receives protocol messages to and from a given socket respectively. Sending and receiving most message types are straightforward applications of the buffer write or read functions proceeded or followed by recv() and send(), respectively. The functions for sending and receiving EXECUTE messages, which need to marshal rpc call arguments, are somewhat complicated. For these, we parse argType[] to figure out the type and length of each argument. We use this information to properly allocate memory and read or write the arguments to and from the buffer in order to transfer them over the network.

Binder

The binder maintains a database of server and function information. It updates the database when a new function is registered and looks up the appropriate servers to handle client rpcCall

requests. The database is implemented using std::map. The key of the map representations a server location, which is captured by its hostname and port. The value of the map is a list of function signatures, which represent the list of all functions registered by the server. Each function signature encapsulates a name and argument types. When a server registers a function, the binder performs a lookup for the server's registered functions. If the function was registered before, the binder overwrites the previous entry in the functions list. If not, the binder adds the new function to this server's registered functions list.

The binder also maintains a queue of server locations to implement round robin scheduling. Whenever a server that the binder has never seen before registers a function, it is added to the front of the queue. When a server is selected to handle a request, it gets moved to the back of the queue. For example, if a client performs a lookup for a function, the binder will examine servers in the order specified by the queue. That is, the binder will first look at the first server in the queue, looks up the database for its registered functions, and checks if there's a function matching what the client is asking. If not, it moves onto the second server in the queue, and so on. If a server has a function that can handle the client's request, then the binder moves the server to the back of the queue and return its name and port back to the client.

In the binder, equality of two functions are defined as follows:

Given function A with name_A and argType_A[] and function B with name_B and argType_B[]

- name_A equals name_B
- argType_A[] and argType_B[] have the same length
- argType_A[i] has the same input bit, output bit and type as argType_B[i]
- argType_A[i] and argType_B[i] are either both scalars or arrays (which may have different lengths)

This allows for function overloading by allowing functions of the same name but different argument types to be considered different. It also allows servers to register functions with variable sized array inputs.

The binder multiplexes requests from multiple servers/clients using select(). We opted to not do multithreaded processing on the binder because we feel that the requests that the binder handle are light weight enough that we don't stand to benefit much from multithreading.

Client

The client-side implementation of our RPC system is relatively straightforward. When a rpcCall is called, it first connects to the binder and does a LOC request using the function name and argument type. If the lookup is successful, the binder returns the server's hostname and port. The client then connects to the server, sends a EXECUTE message, and waits for response. If EXECUTE succeeds, the client copies the server output into its output arguments and returns. If EXECUTE fails, then the client returns the reason code.

Server

When rpcRegister is called, the server connects to binder and registers the function using its name and argument types. It also adds/updates its own local database, implemented as a key-value store, with key being function and argument type and value being the function skeleton. When the server receives an EXECUTE request from a client, it spawn a new thread that extracts the function signature and arguments sent by the client, looks up its local database for the corresponding function skeleton and calls the skeleton. When the local executes completes, the thread either responds back to the client with the updated arguments or informs the client that execution failed.

Termination

Termination occurs when a client calls a rpcTerminate. Upon receiving this request, the binder cycles through all the servers it manages and send all of them a termination request, after which the binder shuts down immediately. The binder will not accept any other requests once it starts to terminate. When a server receives a termination request, it first checks that it came from binder. If so, the server stops accepting requests from clients, waits for any existing requests to finish before shutting down and returning from rpcExecute.

Reason Codes

SOCKET_CREATE_FAIL = -1,

Unable to create a socket to listen to requests. This can occur when calling rpcExecute

SOCKET_CONNECT_FAIL = -2,

Unable to connect to a connect to a remote host. This can happen when client/server fails to connect to binder or client can’t connect to server. Returned by rpcInit and rpcCall.

SOCKET_BIND_FAIL = -3,

Unable to bind a local socket. This can occur when calling rpcExecute.

SOCKET_LISTEN_FAIL = -4,

Unable to listen on local socket. This can occur when calling rpcExecute

GET_SOCKNAME_FAIL = -5,

Could not retrieve the local hostname or port. This can happen on rpcRegister.

NO_BINDER_ADDRESS_ENV = -6,

Occurs if client or server could not get the environment variables for the binder host name. Returned on calling rpcCall and rpcInit.

NO_BINDER_PORT_ENV = -7,

Occurs if client or server could not get the environment variables for the binder port. Returned on calling rpcCall and rpcInit.

RPC_NOT_INIT = -8,

Returned if calling rpcExecute or rpcRegister without rpcInit

SOCKET_SEND_FAILED = -9,

Failure when trying to send messages over the socket. Can be returned by rpcRegister, rpcCall, and rpcExecute

SOCKET_CLOSED = -10,

Socket closed while trying to send messages. Can be returned by rpcRegister, rpcCall, and rpcExecute.

UNEXPECTED_MESSAGE = -11,

Returned if client or server receives a protocol message of unknown type or type is not the one expected.

TERMINATE_NOT_FROM_BINDER = -12,

Returned by rpcExecute if server receives a terminate message not from the binder.

NO_FUNCTIONS_REGISTERED = -13,

Returned if rpcExecute is called without having any functions registered.

SOCKET_RECEIVE_FAIL = -14,

Failure when trying to receive messages over the socket. Can be returned by rpcRegister, rpcCall, and rpcExecute.

RPC_ALREADY_INIT = -15,

Returned when rpcInit is called more than once.

FUNCTION_NOT_FOUND = -16,

Returned by rpcCall if binder could not find a server that can handle the function.

SERVER_EXECUTE_FAILED = -17,

Returned by rpcCall if the server skeleton returned an integer less than 0.

CANNOT_RECEIVE_FROM_BINDER = -18,

Returned by rpcCall if cannot receive LOC response from binder.

SELECT_FAIL = -19,

Returned by rpcExecute when select() fails.

NOT_IMPLEMENTED = -20,

Returned by rpcCacheCall.

FUNCTION_ALREADY_REGISTERED = 1

Returned by rpcRegister if server registers the same function twice.

Unimplemented Feature

The bonus portion of this assignment is not attempted. Calling rpcCacheCall returns NOT_IMPLEMENTED error code