**CS454 A3 RPC SYSTEM: MANUAL**

Chi Ming Chou (20309042) and Yufei Guo (20348014)

**DESIGN DECISIONS**

**THE RPC LIBRARY**

Calls used by clients and servers are split into two files, rpcserver.cpp and rpcclient.ccp. In each file, there are variables that are used to save state of that client or server. These state variables include databases, flags, socket file descriptors and thread management variables (only for server). By separating rpc functions based on their usage, there won’t be unused state vriables. For example, had all rpc functions been in the same file, a server database will be unused when a client uses the library.

**DATA MARSHALLING**

All send and recv calls use a character buffer as a data buffer. Thus all messages are converted to a character array before being sent or received. The conversion is done through the use of a union. The largest supported type conversion is 8 bytes (double) so the union is 8 bytes in size. The union has one property of each supported type (including signed and unsigned versions of some integer types) and one character array of size 8.

A custom class called RWBuffer is used for converting data of different types to character arrays. The class supports insertion of data into a given character buffer (writing to the buffer), and extraction of data from character buffers into reference variables of different types (reading from the buffer). Besides supported variable types, the class also supports arrays of each type, given a specific size.

The RWBuffer uses one instance of the above union for all its conversions (it is just reused for each convertion). Each function takes in a buffer pointer, and returns the new buffer pointer after it has been advanced during processing. As an example, we want to convert an integer into a character array and insert it to a buffer. The RWBuffer function will be an insertion function that takes an integer and a buffer pointer. It will:

1. assign the integer to the union
2. extract the first 4 bytes of the character array from the union. These 4 bytes is the character representation of the integer
3. the bytes are copied into the memory specified by the buffer pointer
4. the new buffer pointer (after moving 4 bytes) is returned.

All RWBuffer insertion functions require that enough memory have been allocated for the buffer pointer. It is easier for the caller to manage the memory.

Using RWBuffer, messages encompassing any of the supported types can be easily converted to and from character arrays.

**BINDER**

The binder database for storing each server’s registered functions is implemented as a map where:

* the key is a **struct** that encompasses the function’s name and argTypes array
* the value is the **list** of servers (identified by a hostname and port number) that registered that specific function

This design lends itself nicely to function overloading because the overloaded functions would each have different keys (due to the difference in their argTypes) in the map. The **struct** has a custom comparator operator that is required for it to be used in a map. The comparator defines two **structs** to be equal if:

* the function names are equal
* the argTypes array lengths is equal
* each argument at the same index in argTypes has the same type (includes whether it’s input/output)
* either both arguments are both scalar, or are both arrays.

Thus functions with array parameters of different lengths are equal, a definition provide by the assignment. When we define them to be equal in the comparator, then such structs, and thus such functions, will be seem as equal, and so will be seen as the same key. With this implementation, function overloading is extremely simple, since such functions will correctly replace duplicates when inserted into the map.

When a REGISTER message is received from a server, the server is added to the database and round-robin queue if it had not existed there before.

When a LOC\_REQUEST message is received from a client, the binder simply gets a list of available servers that support the function signature from the database map. Then, a round-robin approach is used to determine which server information to return to the client.

To do round-robin scheduling, a queue is used to maintain a list of active servers. Each time a location request comes in, the binder gets the list of servers that support the requested function from the database. The binder then looks at the server at the front of the round robin queue and if it is able to service the request, the credentials are sent to the client and the server is moved to the back of the queue. If the server at the front cannot service the request, it is pushed to the back of the queue and the next server is evaluated. If the round-robin queue is exhausted, a LOC\_FAILURE message is sent to the client.

If the binder detects that a server is disconnected, the binder removes the server from the round-robin queue and from the binder database by removing it from the server list for each function signature.

When a TERMINATE message is received from a client, the binder sends termination messages to each of the active servers. The binder then periodically polls the servers by trying to connect to each server. If the connection fails for all of the servers, the servers have all closed and therefore the binder can safely close. It keeps polling until all servers are closed. After the binder receives a TERMINATE message, no further messages are handled, so further REGISTER and LOC\_REQUEST messages are not handled.

**SERVER**

The server database for storing the registered functions for that server is similar to the binder database. The difference is that the server only manages its own functions, and its functions are in the form of skeletons. In this version, the database is a map where:

* the key is a **struct** that encompasses the function’s name and argTypes array
* the value is the **skeleton** pointer for that function

The key is the same key struct used in the binder and so uses the same comparator operator.

However, the server does not add every function to its database. It only adds a function if it receives a REGISTER\_SUCCESS response from the binder, to ensure that its database of functions and the binder’s database of its functions are synced.

EXECUTE requests are handled by the server on threads. On the main thread, select is called on the local socket to listen for messages. Once it receives the full message, a new thread is created and the message buffer is sent to that thread to be processed. With this implementation, the main thread will receive all calls sent to the local socket, maintaining synchronous receives and so preventing out of order receives. On the background thread for each message, the message is processed, the database is checked for that function, and if it exists the skeleton is called and the result is sent back in an EXECUTE\_SUCCESS message from the background thread. If no skeleton exists for that function or the call fails, EXECUTE\_FAILURE is returned instead.

A **map** keeps track of each thread that is spawned for the EXECUTE requests. Upon thread termination (after the response message is sent), each thread removes itself from this map. The map is locked using a mutex lock to prevent asychonourous writes. When a server receives a TERMINATE message, it first checks that it’s from the binder socket. If it isn’t, it discards the message. If it is, the server exits out of its select loop and so stops listening for incoming messages. Then it:

1. obtains the mutex lock for the map of EXECUTE threads
2. makes a copy of all living threads at that moment
3. gives up its lock
4. loops through its copy of living threads
5. calls join on each, blocking until all background threads have finished

The reason for the copy is that the map may change while we are calling join, and so we might access freed data. Thus we need to lock the map access. But it will cause a deadlock if threads are waiting on the lock so they can remove themselves from said map. If we make a copy we can ensure our copy never changes, since it is local to our function, so we don’t need the lock as we iterate through our copy.

**CLIENT**

For the bonus **rpcCacheCall**, the client also has a database map. This map stores the retrieved list of servers from the binder. It is the same kind of database as what the binder has (same key and value types), but it is not a duplicate. It stores only the functions where a cache call has been made, and its server list may not be up to date all the time. As in the assignment specifications, the cached server list is re-fetched if all servers for that function in the current cache fail.

As an added note, the binder and client databases do differ slightly in implementation detail, where the binder’s database value is a pointer to a list while the client’s database value is the list itself. The only explanation for the difference is that different programmers worked on each part, and their styles were different.

**ERROR CODES**

FAILURE = -1,

- used for generic errors that don’t fall under any of the other types

FUNCTION\_OVERRIDDEN = 2,

- a warning code returned when a REGISTER message overwrites an existing function

FUNCTION\_NOT\_AVAILABLE = -2,

- when a function lookup is required but it is not found

SOCKET\_SEND\_FAILURE = -3,

- when send() returns an error

SOCKET\_OPEN\_FAILURE = -4,

- when opening a socket returns an error

SOCKET\_UNKNOWN\_HOST = -5,

- when socket host cannot be found

SOCKET\_CONNECTION\_FAILURE = -6,

- when socket connection cannot be established

SOCKET\_LOCAL\_BIND\_FAILURE = -7,

- when binding to a local socket fails

SOCKET\_ACCEPT\_CLIENT\_FAILURE = -8,

- when accepting a new socket connection fails

SOCKET\_RECEIVE\_FAILURE = -9,

- when recv() returns an error

EXECUTE\_UNKNOWN\_SKELETON = -10,

- when a skeleton lookup is required but it is not found

INIT\_UNSET\_BINDER\_ADDRESS = -11,

- BINDER\_ADDRESS environment variable is not set

INIT\_UNSET\_BINDER\_PORT = -12,

- BINDER\_PORT environment variable is not set

INIT\_BINDER\_SOCKET\_FAILURE = -13,

- when connecting to the binder socket fails

INIT\_LOCAL\_SOCKET\_FAILURE = -14,

- when binding to a local socket fails

RECEIVE\_INVALID\_MESSAGE\_TYPE = -15,

- when a message is received but it’s message type is not supported by the listening program

SELECT\_FAILED = -16,

- when select() fails

SELECT\_TIMED\_OUT = -17,

- when select() times out

WRONG\_FUNCTION\_NAME\_RETURNED = -18,

- when function name is returned in a EXECUTE\_SUCCESS call but the sent function name does not match the returned name

RECEIVE\_INVALID\_MESSAGE = -19

- when the length of the message received it not a valid length for that message type

**UNIMPLEMENTED FEATURES**

All features for the assignment were implemented.