CS454 Assignment 3 Manual

Shoubo Wang (sbwang, 20476417) and Xiaochen Liu (xcliu, 20469767)

Data Marshalling/Unmarshalling

We use a character buffer as our data buffer for transmitting information through send and receive calls, so all data are converted to a character array using memcpy() prior to send or receive. The message starts of with a 4-byte integer indicating the length of the whole request, then another 4-byte integer representing the message type (e.g., LOC\_REQUEST), following with the actual message content. The message content format varies with the type of the message.

Binder

We use a map as the database for storing each server’s registered functions. The map key is a FuncSignature object that stores the function name and an array of argument types. The value is a vector of ServerLoc objects containing the server identifier, port as well as the socket file descriptor.

This database design allows us to easily detect function overloading because overloaded function will result in a different FuncSignature object thus producing different map keys. Also with this design, array arguments of different lengths will produce the same FuncSignature object thus occupying the same key in the database map. We have also implemented the comparator methods to both FuncSignature and ServerLoc classes to facilitate searching functions and servers inside the database.

We use a vector of ServerLoc to store the registered servers as our round robin queue.

Upon receiving a register request from the servers, the server information is added to the database and the round robin queue if it is not yet on queue.

Upon receiving a location request from the client, the binder retrieves the list of available servers supporting the specified function by doing a map key lookup. Then it iterates through the list and the round robin queue to search for the next available server.

The binder first checks the server at the front of the queue to see if it supports the specified function; if so, the server information (id and port) is sent to the client and the server is pushed to the back of the queue using a rotate function. However, if the first server does not support the specified function, it is pushed to the back of the queue and move on to the next server. If there is no matching server, then the binder will send a location request failure message back to the client.

If the binder detects a disconnecting server, the binder will remove the server from the queue and the database map.

Upon receiving a terminate request, the binder sends terminate request to all registered servers and then checks the connection with all servers. Once verified that all servers are disconnected, it will then terminate itself.

**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 contains 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 therefore 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 provided to that thread to be processed. 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 parallel 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, except for the bonus functionalities.