CS 454 Assignment 3

Alex Lim (a7lim)
Ravindra Ashar-Bryant (rklashar)

1. Design Choices

Handling Data

All data is exchanged in the form of messages, which are a simple flat buffer beginning with an integer for length, followed by an integer for type, followed by a buffer containing the other arguments. Messages are assembled from their arguments using utility functions contained in $ROOT/src/common/message.c . Message de-marshalling is slightly more complex, since the type of the message is not immediately known. In both the binder and server receiver, each received message is first processed by a switch statement of message types. This switch statement then dispatches the message to a local message handler within either the binder or the server. This local message handler is then responsible for unpacking and handling the message.

Binder Database

In our code, the binder database is defined as:

typedef pair<string, string> CompleteFunction;
typedef pair<string, string> ServerLocation;

map< CompleteFunction, set<ServerLocation> > function_locations

The binder database is function_locations, which maps functions and their arguments to sets of servers which can serve them.

Note that in our implementation, arguments are also a string. We accomplish this by turning function arguments into a c-style declaration. Examples:

A long array argument which is input:
long*:i

A char which is input and output:
char:i:o

We then simply concatenate all the arguments together, then add them to a pair along with the procedure name to create a complete function. This ensures function names with different arguments end up with different entries. Then we simply add the server (which is a pair<hostname, port>) to the set of servers capable of supplying that function.

Function Overloading

Since function names and arguments are used as a compound key in the binder, this is a non-issue there. On the server side, if a function is registered with the exact same name and arguments, and the same skeleton, we return a warning without ever calling the binder. Same name and arguments but different skeleton does not produce a warning, but does not call the binder either.

Round Robin

This is handled by a simple queue in the binder, with the servers arranged in decreasing priority. When a client makes a request function, the server iterates over the servers in the queue and checks to see if they are found in the set of servers matching that function. The binder then returns the first such matching function to the client, and puts it at the end of the list. Now, this server is only called again when it moves far enough up the list that it is the first one capable of serving a function, OR if it is the only server in the list capable of serving a function.

Note that when a server is first added to the queue, it is added at the front.

Termination

Every time the binder receives an rpcInit call on a socket, it adds that socket to a list of servers sockets. When the binder gets a terminate message from a client, it forwards that message to every socket on the list of server sockets it has, then exits.

All servers maintain an open connection to the binder when rpcInit is called. When rpcExecute is called, a child process is forked off and listens for a terminate message on the socket the server originally opened to the binder (and thus the only messages which can be received on this socket are from the binder). When this message is received by the child process, it sends SIGTERM to it's parent server process (which is busy in a receive loop for clients), then shuts down. The parent server process has a special sigterm handler, which simply calls exit(0) when it receives a sigterm. Thus all servers shut down cleanly (upon main process death, any thread processing a client request also dies, and any client waiting on a result receives a NETWORK_ERROR because the server is no longer available).

2. Error Codes

Binder

To make the binder as “fault tolerant” as possible, the binder will not return any error codes except core assertion failures.

Common

NETWORK ERROR: This error is used for any send or recv calls which fail to complete. This could be due to a terminated server or binder, or simply a actual network error.

UNRECOGNIZED MESSAGE TYPE: This error is thrown by the client or the server if a message is received with an unknown type. This error will never occur under normal circumstances and is only applicable if network data were garbled somehow, or if bad messages were sent on purpose.

Client

NO MATCHING SERVERS: This error is used when the binder cannot locate a server matching the function and argument types. This error is returned when a LOC_FAILURE message is received

FUNCTION NOT FOUND: This error is returned by the server as the reason in an EXECUTE_FAILURE message, and occurs if the server cannot find the function specified by the client

EXECUTION FAILURE: This error is used when a function executed by the server has a negative return code. This error is returned as the reason in an EXECUTE_FAILURE message.

UNKNOWN REASON: This error is returned from a EXECUTE_FAILURE server message with an unknown reason.

Server

UNINITIALIZED NETWORK HANDLERS: This error is returned if rpcRegister is called without calling rpcInit

REGISTRATION AFTER INITIALIZATION: This error is returned if the server attempts to register more functions with the binder, after starting execution. In practice, this should never happen unless the server forks another thread to keep registering or something equally silly, since rpcExecute is a blocking call.

DUPLICATE FUNCTION REGISTRATION: This warning is thrown when a server attempts to re-register a function with the same name and arguments.

PREVIOUSLY INITIALIZED NETWORK HANDLER: This warning is returned if rpcInit is called twice

3,4. Implementation

All aspects of the original assignment except the bonus have been implemented