CS 454 A3 - Documentation

Nihal Pednekar (`npedneka`)
Renato Ferreira Pinto Junior (`r4ferrei`)

July 12, 2017

1 Design choices

1.1 Marshalling and unmarshalling

Each of the messages in our system is represented by a C++ `class` or `struct` containing the relevant fields for the message such as argument types, function name, or status code as C++ data types.

To marshal and unmarshal messages, we use a generic `Serializer` class which, using C++ templates, we specialize to each message type. After the message length, the first byte of each message indicates which message type (and consequently which serialization scheme) should be used.

Using this scheme, sending any message over the network can be done transparently: call its `Serializer`, and pass the returned buffer to the TCP layer. Conversely, a buffer can be translated into a message type that can be handled by higher-level code.

1.2 Binder database and round-robin scheduling

A C++ data type uniquely determines a *function signature* by its function name and ordered list of argument types, where each argument type consists of its data type, input/output nature, and scalar/array nature (regardless of array size).

The binder keeps a database associating existing function signatures to live servers. In particular, it keeps the following mappings:

- For each server, its associated *timestamp*

- For each function signature, a list of live servers
- A queue of live servers sorted by timestamps

The *timestamp* for a server is a sequence number that is incremented every time the binder registers or locates a procedure in the system.

On a *locate* request, the binder chooses the server with earliest timestamp that serves the desired function, and responds to the client with that server. It then updates the timestamp for that server. This ensures that at any moment, the server that has been idle for the longest time possible is chosen to serve a query, as required for the round-robin strategy.

When the connection to a server is closed, that server is atomically remove from all data structures so that it is not served to any clients.

1.3 Server database and execution

We also keep a simple database on the server side associating function signatures with the corresponding skeletons. This database allows the server-side library to route incoming client requests to the right skeleton.

Once the correct skeleton is detected, it is invoked with the given arguments. Any output arguments are copied back to the argument array, which is marshalled and sent back to the client-side library.

1.4 Function overloading

Since the binder data structures outlined above deal with *function signatures*, overloading is automatically resolved – we always match on the entire signature when serving a client.

Similarly, if a server registers the exact same signature twice, that is easily detected by the server-side database, so that the new skeleton takes the place of the old one on incoming queries.

1.5 Non-blocking server

The server-side library has a main thread that accepts connections from clients. Since we are dealing with a relatively small-scale problem, we chose to spawn a new thread to process each request. Hence, each request is processed and responded to without preventing the main thread from accepting and processing new queries.

If more scalability were required, we would consider using a thread pool instead of spawning a new thread for every incoming request.

1.6 Termination

When the binder receives a termination request, it redirects that request to all servers and waits for them all to acknowledge termination. Finally, the binder itself acknowledges termination to the client-side library and exits.

On the server side, we have an additional thread that is responsible for listening to the binder for the termination request. This allows the server to detect termination and respond to it even if worker threads are blocked on TCP operations. Once the termination message is received, other threads are finalized and the server exits.

By only processing termination requests that come from the binder socket, we provide the required level of authentication.

1.7 TCP abstraction layer

We built a small TCP library to prevent low-level network code from polluting too much higher-level logic. The three abstractions we have are sockets (which can send and receive data), clients (which can connect to a remote server and obtain a socket), and servers (which can accept incoming connections and manage multiple sockets concurrently).

Together with the marshalling and unmarshalling layers outlined above, this layer allows most message-passing code to consist of a few lines of code expressing clear intent concisely at call site.

2 Error codes

We have determined the following error codes to be relevant for users of the RPC library:

Alias	*Numeric code*	*Returned by*	*Meaning*
SERVER_SOCKET_CREATION_ERROR	-1001	rpcInit	A server socket could not be created, e.g. no port available.
BINDER_CONNECTION_ERROR	-1002	rpcInit, rpcCall, rpcRegister, rpcExecute	The connection to the binder has failed.
UNEXPECTED_MESSAGE	-1003	rpcCall, rpcRegister, rpcExecute, rpcTerminate	A node received a message that does not respect the protocol. In particular, a server received a call to a function it does not serve.
SERVER_CONNECTION_ERROR	-1004	rpcCall, rpcTerminate	The connection to a server has failed.
LOCATE_FAILURE	-1005	rpcCall	No live servers can serve the requested function.

In addition to these error messages, if a given skeleton returns non-zero status, that status is propagated back to the return value of `rpcCall`. We chose numbers below -1000 for library error codes so that most skeleton error codes (which are usually single bytes) could be read back at call site transparently. Depending on the use case, other similar strategies could be used, such as using a high bit to determine whether the error code is a library error or a user-function error.