Simple RPC Library Implementation

Boris Kravchenko (bkravche)
Michael Gamble (mlgamble)

Marshalling and unmarshalling of data

The library uses a generic message structure for messages. On the sending side, this structure is populated with appropriate information depending on the message type.

```
struct message {
        int type;
        int reasonCode;
        std::string host;
        std::string port;
        std::string name;
        std::vector<int> argTypes;
        void** args;
};
```

During the marshalling process, the type of the message determines which fields the marshaller will read. All fields are converted into their string representation.

- For integer fields are converted into strings using native string functions.
- Argument types and arguments are copied into raw byte arrays. These byte arrays are converted to strings for convenience.

After marshalling, a vector of these strings is handed to the message passing API. The message passing API then transmits these over the network using the following format:

8 byte length header
4 byte message type
4 byte length header
SEGMENT 1
…
4 byte length header
SEGMENT N

Where N = message type integer / 10

Ex: The Register message type is encoded as the interger 40, therefore there are 4 segments in a register message.

On the receiving side, the message passing API reads the socket and writes the data segments into a vector of strings. These strings are passed to the unmarshaller, where they are written back into a message structure after being cast back into appropriate types. For each argument

type, bit shifts were used to calculate the underlying raw c type and array length, and that information was used to correctly copy the memory into the message structure field.

Binder Database implementation

The binder database is implemented with the following data structure:

> *typedef std::pair<std::string, std::string> stringPair;*
> *std::map<stringPair, std::vector<stringPair> > registrationTable;*

In the implementation, it's used like so:

> registrationTable [pair(name, signature)] = vector of pairs (host, port)

Where:

> name = name of the function
>
> signature = the argument types of the function concatenated into one string. If argument type is an array, the lower 2 bytes are set to all 1's, which ensures two functions that accept the same argument types but have different array lengths are treated the same.

Function Overloading

Because the binder database is keyed by a name and signature pair, overloading comes for free. On lookup, the correct function is selected according to the provided argument types.

The server stub's local database handles overloading a similar manner. It's implemented as:

> SkeletonMap [pair(name, signature)] = skeleton

Round-Robin Scheduling

Round-Robin Scheduling is implemented in the binder with the following data structure:

> *std::deque <stringPair> roundRobin;*

The deque contains host and port pairs for servers that are taking part in the RPC.

Upon server registration, a new arriving server is pushed to the head of the deque.

Upon a location request, the deque is iterated from head to tail, and the first matching server that implements the requested function is returned. This server is then moved to the tail of the deque. This ensures that every server gets an equal opportunity to serve a request.

Termination procedure

The binder keeps a vector of server sockets.

1. Upon receipt of the termination message, the binder validates that the socket it received the message on is not one of the server sockets, which ensures the message came from a client.

2. For each server socket, the binder sends a termination request and waits for a response from each server. This ensures all servers terminate before the binder does.

3. The binder sends a return code of 0 to the client and terminates.

4. Once the server receives the terminate request, it sends a return code 0 to the binder and terminates. The server is only able to receive the termination message type on the binder socket; it's disregarded if it's received on any other socket. This ensures the request came from the binder.

Error Codes

The following error codes are defined:

-1 = *RPC_EXECUTE_FAILURE*
- Server-side function has returned something other than 0

-2 = *RPC_NO_FUNCTION_DEFINED*
- Binder returned location failure on location request; no sever implements this function

-3 = *CONNECTION_FAIL_ON_CONNECT*
- There was a failure when connecting a socket to a host/port

-4 = *CONNECTION_FAIL_DNS_LOOKUP*
- There was a failure doing the DNS lookup stage of socket creation

-5 = *SOCKET_BIND_FAIL*
- Could not bind to a socket

-6 = *NO_SERVER_CONNECTION*
- The client could not connect to the server that the binder returned. This means the server is down

-7 = *NO_BINDER_CONNECTION*
- The client could not connect to the binder to issue a location request. This means the binder is down

-8 = *NO_REGISTERED_FUNCTIONS*
- Server called rpcExecute without registering any functions

-9 = *INVALID_MESSAGE_TYPE*
- A component of the RPC library received an invalid message type

-10 = *INITIALIZATION_FAILED*
- The server didn't initialize properly

1 = *FUNCTION_ALREADY_REGISTERED*
- A warning that's returned to the server on function re declaration