CS 454

Assignment# 3

Nurlan Masimli (nmasimli 20490844)

Rasul Rasulzade (rrasulza 20488806)

Documentation and System Manual

RPC

Overview:

Client side of the RPC is single-threaded. Basically, the client calls rpcCall () to execute the desired function. It makes a request to the binder and after the result (location) is received a request to execute the function is made to the server. Only one client calls rpcTerminate().

Server side of the RPC is multi-threaded. In rpcInit(), main thread of the server initializes all static data that needs to be initialized and creates a thread pool to be able to handle multiple requests from the clients. After call to rpcInit(), rpcRegister() is called for a few times to register procedures that the server serves. Then, rpcExecute() is called to handle the requests from the clients.

Data structures and static data:

- static struct location my_loc. This static variable is used in server side and initialized in rpcInit(). After call to rpcInit(), this data is no longer modified and only read by the server for later use such as registering procedures with the binder.
- static int listening_sock, static int binder_sock. These static variable is setup in rpcInit().After call to rpcInit(), this data is no longer modified and only read by the main thread of the server for later use.
- static struct proc_node *proc_list. This is a head pointer to the linked list that contains all of the procedure signature to skeleton mappings.
- static intQueue intQ. The queue of sockets those wait to be handled by the worker threads. This queue is atomic data structure. Mutex locks are used internally in the queue.
- static pthread_t worker_threads[NUM_THREADS]. Thread pool to be able to handle multiple requests from the clients.

Implementation details:

Multithreading in the server:

In rpcExecute(), Main thread of the server selects FD_SET that contains 2 sockets (listening socket for client connections and binder socket for the binder to know the server is up and for termination synchronization). When select() returns, if the listening socket is set in FD_SET, the main thread accepts the new connection and pushes the socket number into the queue of socket that workers are handling. If binder socket is set in FD_SET, the main thread checks the termination protocol and if TERMINATION request is properly sent, the main thread pushes socket number -1 into the queue equal times to the number of workers to let them cleanly exit. Workers pop()s the next element from the queue and if it is -1, then it is termination signal and therefore the worker returns. Otherwise, the worker receives the request sent to the socket and returns the response after it is handled.

Function overloading:

Mappings of procedure signatures to skeletons are stored in procedure list. When there is a new registration, the update() function first checks whether there is a function with this signature or not (name and argTypes need to match including scalar to scalar and array to array i.e. ignoring the length for arrays). If it matches the skeleton is overwritten, otherwise, new procedure node is created.

Marshalling/Unmarshalling of the data:

In order to marshall data, first the total size is computed to find the size of the buffer to be created. Then the buffer is created and first the length of the message and the type of the message and then the actual message is placed into the buffer. Also, for unmarshalling of the data, first the length and the type of the message is received and a buffer of size received is created for the rest of the message. Depending on the type of the message, the data is placed into the buffer accordingly. Marshalling/Unmarshalling for fixed size parts are done by calling memcpy() on the offset of the buffer and the data directly. For non-fixed size, parts it is not differently. For argument types, there is a argument length placed in the buffer that indicates how much memory from the offset refers to argument types to be able to use the mempy() directly. Arguments are the trickiest ones. For unmarshalling arguments, first the array of pointers of length argument len (read before the argTypes) is allocated in the heap. After that size of each argument is computed and equal amount of memory is allocated in related index of pointer array. For each index of the array memcpy() is called for size of the argument related. For marshalling arguments first the total size and size of each argument is computed. Total size is needed to know how much space is needed in the send buffer. Size of each argument is needed to know how much space will be copied from which pointer and also to update the offset for each argument to be copied.

Messaging:

All messages are sent and received in recommended form in the assignment: First 4 bytes represent the length of the message, the next 4 bytes represent the type of the message and the following bytes represent the message. Also we use the protocols as described in the assignment for client/binder, server/binder and client/server communication.

Before sending a message, char array is created then the length of the message, the type of the message and the message is copied to this array by using memcpy(). When receiving this message, first its length is received then the same size array is allocated on the heap to store the message. After the message is received and handled this array is deleted from heap.

Binder:

The binder maintains database to store pairs of registered procedure and the list of server identifiers, which send registration request for the procedure. This database is implemented as a map such that:

- **key** is ProcSignature structure that contains procedure name, argTypes and the length of argTypes
- **value** is a list of ProcLocation structures that contains hostname, socket and port number of the server

It also maintains list of servers (struct location) for round robin, and vector of active server sockets that will be closed if the server terminates.

Procedure overloading is handled implicitly by overloading equality and comparison operators for **ProcSignature** and **ProcLocation** structures. Hence, the same procedure with different argTypes are considered as function overloading and added to the database as a new entry.

If message type REGISTER is received, binder registers the procedure that server has sent. If the same server tries to register procedure with the same name and the same arguments in argTypes, then binder sends REGISTER_FAILURE with warning code ERR_RPC_PROC_RE_REG to the server. Procedures with the same name and different arguments in argTypes are allowed be registered. In this case REGISTER_SUCCESS with warning code ERR_RPC_SUCCESS will be sent to server. However, procedures with the same name and the same arguments in argTypes that differs only in their array lengths are considered as re-registration and REGISTER _FAILURE with warning code ERR_RPC_PROC_RE_REG is sent to the server. The binder sends error code only if memory allocation in the heap fails during message reading. In this case REGISTER _FAILURE with error code ERR_BINDER_OUT_OF_MEMORY is sent to the server and binder rejects registration.

If message type LOC_REQUEST is received, binder searches for the requested procedure in the mapping table and if it is there, round robin algorithm will determine the next server for this procedure. Then binder sends LOC_SUCCESS and server information to the client. Otherwise, binder sends LOC_FAILURE with error code ERR_RPC_NO_SERVER_AVAIL to indicate that this is invalid procedure that has not been registered yet. If memory allocation in the heap fails during message reading, LOC_FAILURE with error code ERR_BINDER_OUT_OF_MEMORY is sent to the client.

Binder adds a server to the list of servers for round robin only if there is more than one server that registered the same procedure. This is because there is no need for round robin algorithm to choose the next server for the procedure, which is only registered by single server. In this case that server is always chosen. Otherwise, round robin algorithm finds the next server and moves it to the end of the list.

If one of the active servers is terminated or disconnected, it will be removed from mapping table of procedure and servers, list of server identifiers and vector of active server sockets.

If message type TERMINATE is received, binder sends terminate message to all servers in the vector of active server sockets. Then binder closes its socket and terminates. Also, if the binder suddenly disconnects, all servers that are connected to the binder will be terminated.

Error codes:

Return code	Value	Description
ERR_RPC_PROC_RE_REG	1	Warning: procedure has already been registered.
ERR_RPC_SUCCESS	0	Success
ERR_RPC_OUT_OF_MEMORY	-1	Error: no memory available on heap of the rpc
ERR_RPC_PROC_EXEC_FAILED	-2	Error: procedure returns an error
ERR_RPC_EXEC_BEFORE_REG	-3	Error: procedure is executed before registration
ERR_RPC_UNREGISTERED_PROC	-4	Error: invalid procedure
ERR_RPC_NO_SERVER_AVAIL	-5	Error: invalid procedure request (no available server)
ERR_RPC_UNEXPECTED_MSG_TYPE	-6	Error: invalid message type
ERR_RPC_SOCKS_INACTIVE	-9	Error: no active sockets
ERR_RPC_SOCKET_FAILED	-10	Error: invalid socket
ERR_RPC_ENV_ADDR_NULL	-11	Error: invalid binder hostname
ERR_RPC_ENV_PORT_NULL	-12	Error: invalid binder port
ERR_RPC_HOSTENT_NULL	-13	Error: invalid binder hostent
ERR_RPC_THREAD_NOT_CREATED	-14	Error: pthread_create() is failed
ERR_BINDER_OUT_OF_MEMORY	-15	Error: no memory available on heap of the binder
ERR_RPC_BINDER_SOCK_CLOSED	-17	Error: binder socket is closed
ERR_RPC_BINDER_SOCK_FAILED	-18	Error: invalid binder socket
ERR_RPC_SERVER_SOCK_FAILED	-19	Error: invalid server socket
ERR_BINDER_TERMINATE_SIG	-50	Error: terminate message is received, stop binder
ERR_RPC_INCOMPLETE_MSG	-99	Error: message size is not properly aligned

Unimplemented functionalities:

Bonus functionality: rpcCacheCall() for the client.