**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.