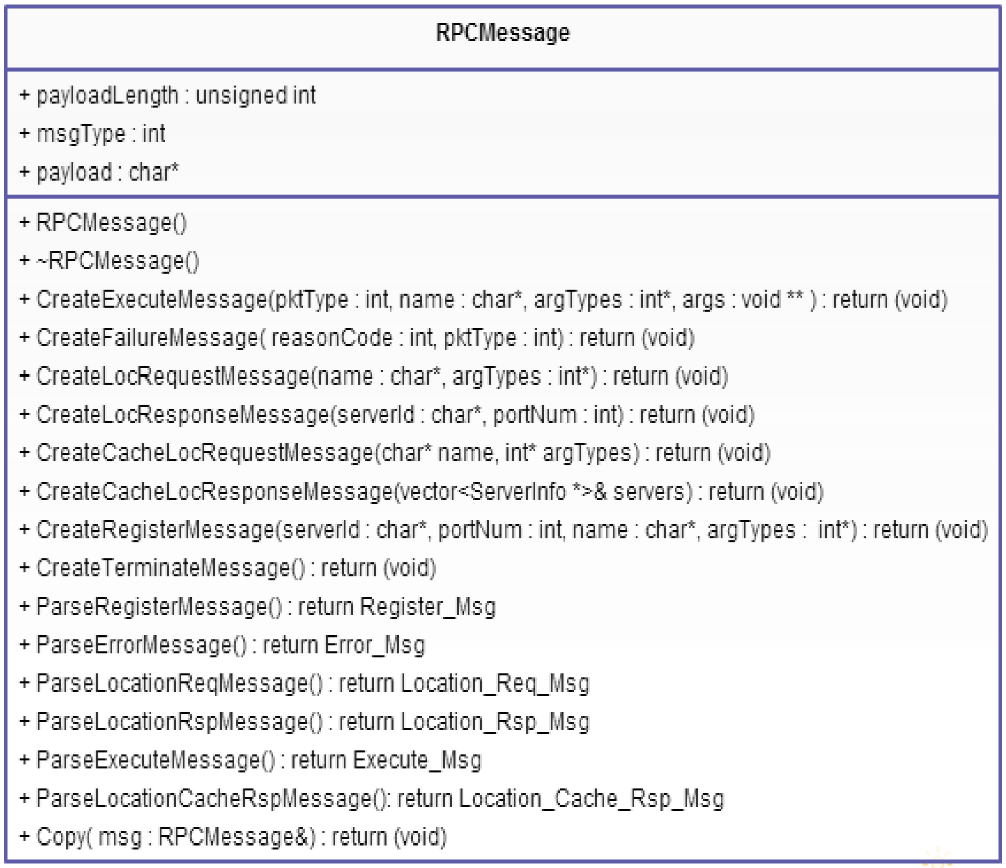
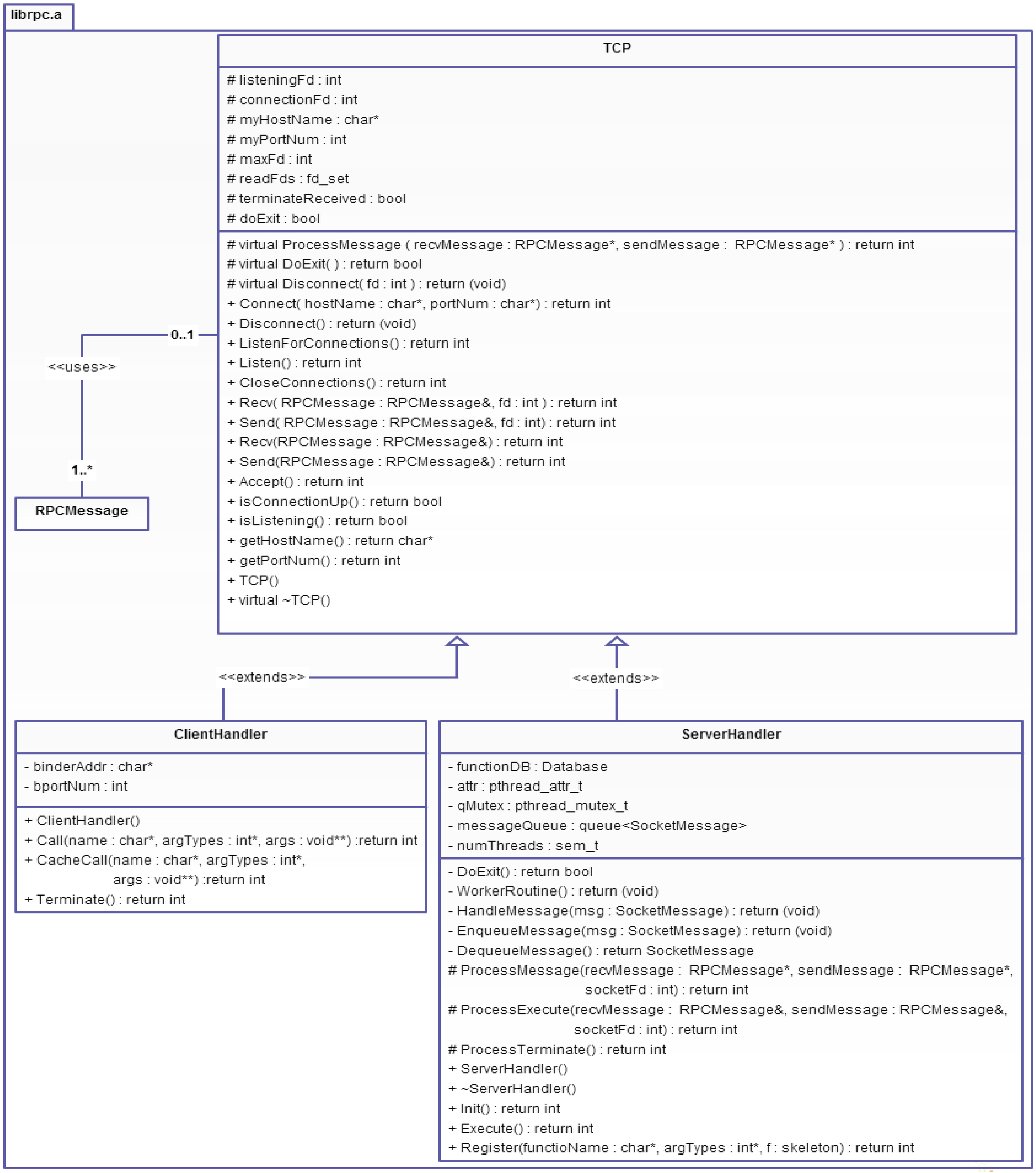
Introduction:

The design aspects used were separated according to the functionality required. Illustrated below are the class diagrams for each of the classes used to design the Remote Procedure Calls' functionality for this assignment. These class diagrams are provided to assist in explaining the overall design choices that were considered during the implementation. Each of the class diagrams are separated into packages to help in distinguishing them for the deliverables, i.e. librpc.a library, and binder binary executable.

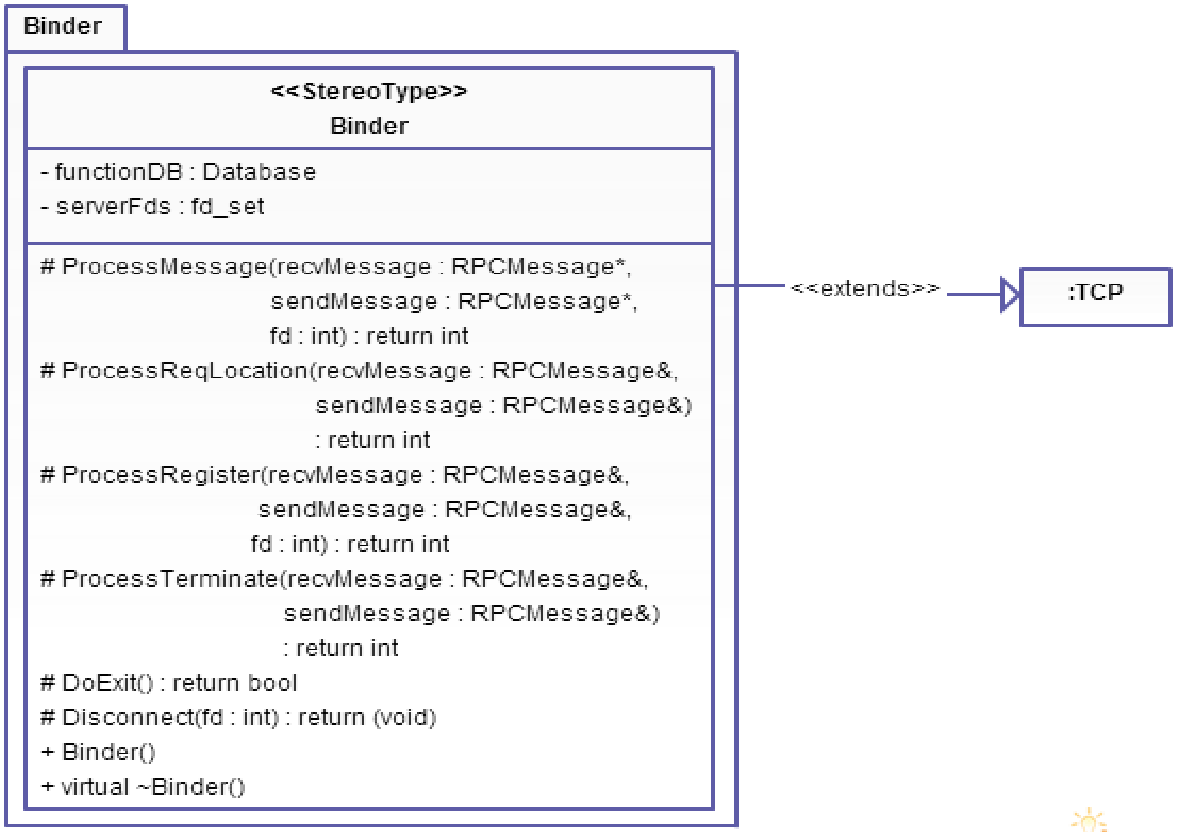
Class Diagrams:



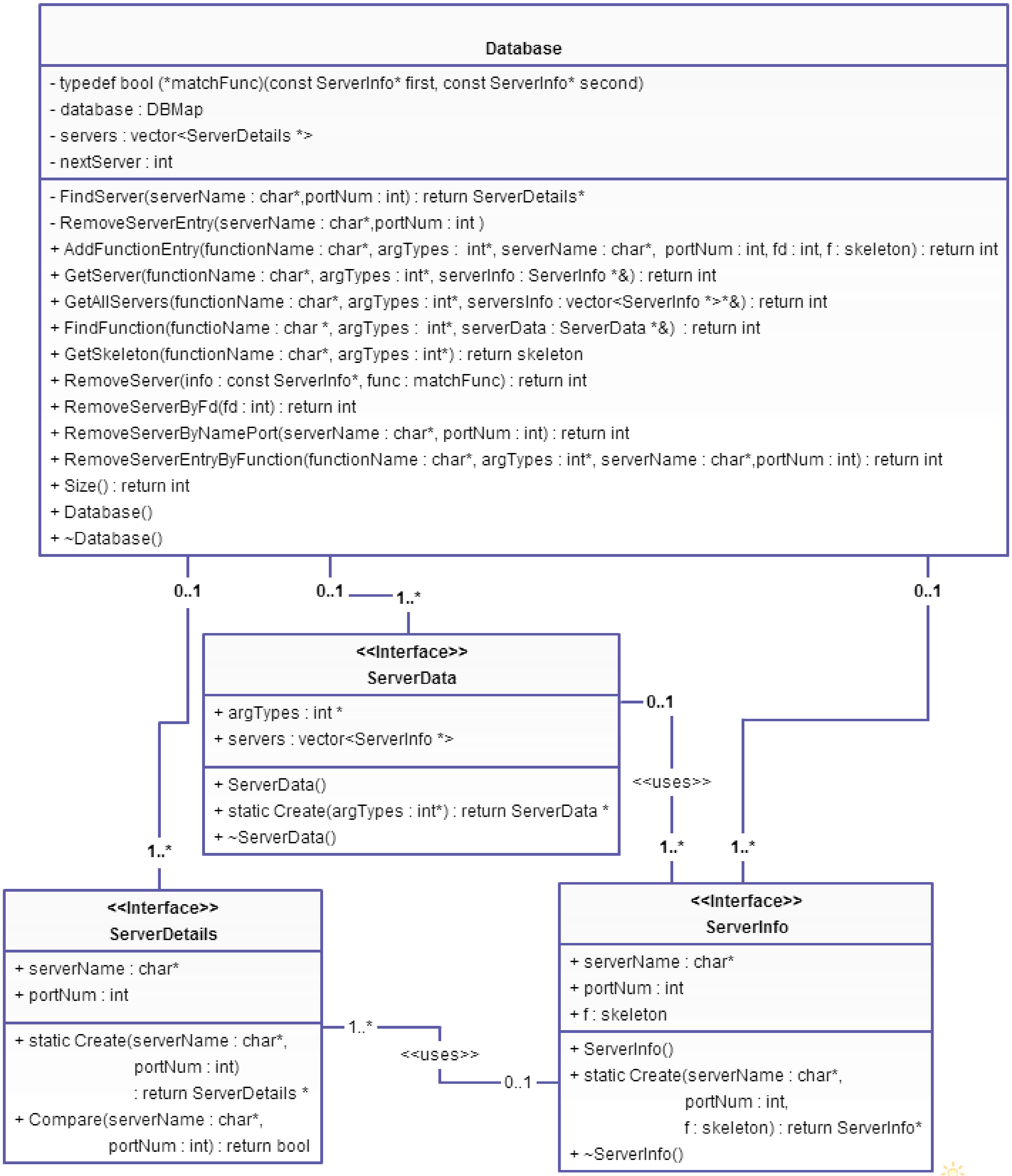
```
RPCMessage
+ payloadLength : unsigned int
+ msgType : int
+ payload : char*
+ RPCMessage()
+ ~RPCMessage()
+ CreateExecuteMessage(pktType : int, name : char*, argTypes : int*, args : void ** ) : return (void)
+ CreateFailureMessage( reasonCode : int, pktType : int) : return (void)
+ CreateLocRequestMessage(name : char*, argTypes : int*) : return (void)
+ CreateLocResponseMessage(serverId : char*, portNum : int) : return (void)
+ CreateCacheLocRequestMessage(char* name, int* argTypes) : return (void)
+ CreateCacheLocResponseMessage(vector<ServerInfo *>& servers) : return (void)
+ CreateRegisterMessage(serverId : char*, portNum : int, name : char*, argTypes :  int*) : return (void)
+ CreateTerminateMessage() : return (void)
+ ParseRegisterMessage() : return Register_Msg
+ ParseErrorMessage() : return Error_Msg
+ ParseLocationReqMessage() : return Location_Req_Msg
+ ParseLocationRspMessage() : return Location_Rsp_Msg
+ ParseExecuteMessage() : return Execute_Msg
+ ParseLocationCacheRspMessage(): return Location_Cache_Rsp_Msg
+ Copy( msg : RPCMessage&) : return (void)
```


librpc.a
TCP
# listeningFd : int
# connectionFd : int
# myHostName : char*
# myPortNum : int
# maxFd : int
# readFds : fd_set
# terminateReceived : bool
# doExit : bool
# virtual ProcessMessage (recvMessage : RPCMessage*, sendMessage : RPCMessage*) : return int
# virtual DoExit( ) : return bool
# virtual Disconnect(fd : int) : return (void)
+ Connect(hostName : char*, portNum : char*) : return int
+ Disconnect() : return (void)
+ ListenForConnections() : return int
+ Listen() : return int
+ CloseConnections() : return int
+ Recv(RPCMessage : RPCMessage&, fd : int) : return int
+ Send(RPCMessage : RPCMessage&, fd : int) : return int
+ Recv(RPCMessage : RPCMessage&) : return int
+ Send(RPCMessage : RPCMessage&) : return int
+ Accept() : return int
+ isConnectionUp() : return bool
+ isListening() : return bool
+ getHostName() : return char*
+ getPortNum() : return int
+ TCP()
+ virtual ~TCP()
0..1
<<uses>>
1..*
RPCMessage
<<extends>>
<<extends>>
ClientHandler
- binderAddr : char*
- bportNum : int
+ ClientHandler()
+ Call(name : char*, argTypes : int*, args : void**) :return int
+ CacheCall(name : char*, argTypes : int*,
args : void**) :return int
+ Terminate() : return int
ServerHandler
- functionDB : Database
- attr : pthread_attr_t
- qMutex : pthread_mutex_t
- messageQueue : queue<SocketMessage>
- numThreads : sem_t
- DoExit() : return bool
- WorkerRoutine() : return (void)
- HandleMessage(msg : SocketMessage) : return (void)
- EnqueueMessage(msg : SocketMessage) : return (void)
- DequeueMessage() : return SocketMessage
# ProcessMessage(recvMessage : RPCMessage*, sendMessage : RPCMessage*,
socketFd : int) : return int
# ProcessExecute(recvMessage : RPCMessage&, sendMessage : RPCMessage&,
socketFd : int) : return int
# ProcessTerminate() : return int
+ ServerHandler()
+ ~ServerHandler()
+ Init() : return int
+ Execute() : return int
+ Register(functioName : char*, argTypes : int*, f : skeleton) : return int

Binder:


Binder
<<StereoType>>
Binder
- functionDB : Database
- serverFds : fd_set
# ProcessMessage(recvMessage : RPCMessage*, sendMessage : RPCMessage*, fd : int) : return int
# ProcessReqLocation(recvMessage : RPCMessage&, sendMessage : RPCMessage&) : return int
# ProcessRegister(recvMessage : RPCMessage&, sendMessage : RPCMessage&, fd : int) : return int
# ProcessTerminate(recvMessage : RPCMessage&, sendMessage : RPCMessage&) : return int
# DoExit() : return bool
# Disconnect(fd : int) : return (void)
+ Binder()
+ virtual ~Binder()
<<extends>>
:TCP

Database:


Database
- typedef bool (*matchFunc)(const ServerInfo* first, const ServerInfo* second)
- database : DBMap
- servers : vector<ServerDetails *>
- nextServer : int
- FindServer(serverName : char*,portNum : int) : return ServerDetails*
- RemoveServerEntry(serverName : char*,portNum : int)
+ AddFunctionEntry(functionName : char*, argTypes : int*, serverName : char*, portNum : int, fd : int, f : skeleton) : return int
+ GetServer(functionName : char*, argTypes : int*, serverInfo : ServerInfo *&) : return int
+ GetAllServers(functionName : char*, argTypes : int*, serversInfo : vector<ServerInfo *>*&) : return int
+ FindFunction(functioName : char *, argTypes : int*, serverData : ServerData *&) : return int
+ GetSkeleton(functionName : char*, argTypes : int*) : return skeleton
+ RemoveServer(info : const ServerInfo*, func : matchFunc) : return int
+ RemoveServerByFd(fd : int) : return int
+ RemoveServerByNamePort(serverName : char*, portNum : int) : return int
+ RemoveServerEntryByFunction(functionName : char*, argTypes : int*, serverName : char*,portNum : int) : return int
+ Size() : return int
+ Database()
+ ~Database()
0..1
0..1
1..*
0..1
<<Interface>>
ServerData
+ argTypes : int *
+ servers : vector<ServerInfo *>
+ ServerData()
+ static Create(argTypes : int*) : return ServerData *
+ ~ServerData()
0..1
<<uses>>
1..*
1..*
1..*
<<Interface>>
ServerDetails
+ serverName : char*
+ portNum : int
+ static Create(serverName : char*,
portNum : int)
: return ServerDetails *
+ Compare(serverName : char*,
portNum : int) : return bool
1..*
<<uses>>
0..1
<<Interface>>
ServerInfo
+ serverName : char*
+ portNum : int
+ f : skeleton
+ ServerInfo()
+ static Create(serverName : char*,
portNum : int,
f : skeleton) : return ServerInfo*
+ ~ServerInfo()

Brief Class Descriptions:

TCP:

TCP is the major class used by client, server and binder for communication between each other. The TCP class is capable of doing multiple things concurrently. It can connect to to the hostname and port number provided and simultaneously create a socket for listening. This behaviour of TCP class makes it a flexible choice to be extended by the client, server and the binder. The client needs to connect to the binder and a server, consequently can extend the TCP class. The server needs to connect to binder and also listen for any client connections, therefore can extend the TCP class. Similarly, the binder needs to listen for any client and server connections, and by extending the TCP class, this task can be accomplished easily.

The TCP class provides a simple interface to send and receive messages between each of the entities. In order to perform these tasks it needs a formated RPC message and a socket connection open to send or receive the messages. Formated RPC messages can be created using RPCMessage class (Please refer to the description of RPCMessage described below).

The TCP class also provides a functionality to disconnect any socket file descriptor that was previously opened by each of the entities. So provided a socket id, it can close and disconnect that socket connection.
The logic for communication is separated from the system in TCP class. This is was done to make sure any sort of communication is done through a common interface throughout the system. Therefore, TCP class was implemented to send and received specific formated RPC messages. This made it easier to implement the client, server and the binder. For any received message, TCP class calls its own ProcessMessage function and propagating the actual message for it to deal with. Using this functionality, the server and the binder can override the ProcessMessage function in their own classes and define their own behaviours of each type of message that they receive.

ClientHandler:

The ClientHandler class manages the RPC calls for the client (e.g. rpcCall, rpcTerminate). The ClientHandler class extends from TCP to use its base functionality of connecting, sending, receiving messages between the client and the binder/server.

ServerHandler:

The ServerHandler class manages any RPC calls for the server (e.g. rpcRegister, rpcExecute). It provides a public interface for making rpcInit, rpcRegister and rpcExecute calls. It extends from TCP and overrides the ProcessMessage function to define its own behavior for all the messages it receives from the binder or the client.

ServerHandler handles each message without blocking any other calls that are made to the server. When ProcessMessage is called, the ServerHandler class pushes this message into a queue and spawns off a thread to deal with it. The thread runs a worker routine, which then actually looks into the message, services it and sends the response back. In this way, the original thread listening for messages it kept unblocked.

For each type of expected message like Execute or Terminate, ServerHandler has a procedure with the same name. For instance, ProcessExecute is called by the worker routine when it sees that the RPCMessage is an execute message. ProcessExecute then unmarshalls the content of the message, calls the function and sends the results back. In this way multiple further messages can be added to the system without actually worrying about how messages are sent and received.

Binder:

The Binder class extends from TCP. It opens up a socket for listening. Similar to the ServerHandler class, it overrides the ProcessMessage function and defines its own behavior for each message type.

Binder deals with both client and server messages. It holds a database of functions registered by the servers that is used when a client asks for a server which can fulfill the client's requests. It also holds a set of server socket file descriptors so that it can send them a terminate message when it receives one from the client.

RPCMessage:

RPCMessage class is the one that provides the functionality to marshall and unmarshall data. For each type of RPC message, it takes the raw arguments and creates a message of that type. For instance, it provides a function called CreateRegisterMessage which takes in server identifier, port, function name and argTypes and creates a register RPCMessage which can be sent to the binder by the server.

For almost every Create* method, RPCMessage class also provides functions for unmarshalling it. Each of these function names, for unmarshalling, start with Parse. Parse* functions return a unmarshalled form of data which can be used later when processing the request.

Database:

Database class used by all client, server and binder. Client only uses it for cacheCall. Database holds the information for registered function names, their argument types and server identifiers which provides these function services. This class also maintains the round-robin algorithm. So anyone can call the GetServer function without worrying about who's turn it is to fulfill the client's request.

Marshalling and Unmarshalling:

The basic concept between marshalling and unmarshalling of data is to incorporate all the data sent and received between the binder, the server and the client respectively into a RPC message. To be specific, each RPC message contains three fields, the payload length (unsigned int payloadLength), a message type (int msgType) indicating the type of the payload and the actual data contained in a character array (char* payload). This idea is based on the suggested message protocol specified in the assignment. The message type is differentiated according to the type of message being passed around between each of the entities. The binder and the servers will have 4 unique message types that are passed around between them. Similarly the binder and the clients also has 7 unique message types that could be passed between them. A server and its clients have 3 unique message types that are passed between them. (Please refer to the codes below for a more detailed description).

binder ←→ server messages

REQ_REGISTER	1	Server sending a request register message to the binder
REQ_REGISTER_SUCCESS	2	Binder’s response of a successful registration
REQ_REGISTER_FAILURE	3	Binder’s response of a unsuccessful registration

binder ←→ client

REQ_LOC_REQUEST	4	Client sending a request location message to the binder
REQ_LOC_SUCCESS	5	Binder’s response of a successful location requested function
REQ_LOC_FAILURE	6	Binder’s response of a unsuccessful location requested
REQ_TERMINATE	10	Client sending a terminate message to the binder which propagates it to all the servers
REQ_CACHE_LOC_REQUEST	11	Client sending a cache request location message to the binder
REQ_CACHE_LOC_SUCCESS	12	Binder’s response of a successful cache location request
REQ_CACHE_LOC_FAILURE	13	Binder’s response of a unsuccessful cache location request

server ←→ client

REQ_EXECUTE	7	Client sending a execute request message to the server for function
REQ_EXECUTE_SUCCESS	8	Server's response of a successful execute request
REQ_EXECUTE_FAILURE	9	Server's response of a unsuccessful execute request

The 'char* payload' variable holds the data regarding the request being made between the binder, the server and the client and assists in marshalling and unmarshalling of data between each of the entities. Marshalling and unmarshalling for each of the different request and response types is performed based on only the essential information required at each end of the execution step. For instance, a server registering itself with a binder will marshal all its data into the character array and, once the binder unmarshals the data and processes the register request, it may marshal different data back to the server.

As mentioned in the description of RPCMessage class, there are pair functions to marshall and unmarshall each message. Each marshalling function (e.g. CreateRegisterMessage) is paired with another unmarshalling function (e.g. ParseRegisterMessage). Marshalling function creates a buffer of the required length and fills in that buffer with the message contents. For instance CreateRegisterMessage takes server id, port, function name and argtypes and create a buffer of length that can hold all these 4 pieces of information sequentially. It also sets payloadLength of the message to be equal to this length. The unmarshalling function (e.g. ParseRegisterMessage) takes the buffer and parse it out. Each unmarshalling function knows at what buffer location will each information will be located. For example, CreateRegisterMessage puts in argTypes after 100 characters (this is fixed in our code) after start of the function name in the buffer. ParseRegister knows that after 100 chars, argTypes can be found. So each unmarshalling method returns a struct of the message type (e.g. struct Register_Msg) which holds the information which was marshalled earlier. For instance, Register_Msg struct holds serverId, port, name and argTypes. This struct can then be used for further processing of the request.

Unmarshalling functions also look for values in the message and parse accordingly. For instance, parsing execute message, the unmarshalling function first parses the argTypes. Based on that it knows that after argTypes, args will be in the buffer. But from this point on, it looks at each argType, looks for the length and type, and based on that it parses out the buffer to create new args array.

Binder Database Design:

Binder database is designed for quick and easy retrieval of server information given function name and argument types, yet maintaining aspects like function overloading. Essentially the database design can be broken down in three classes ServerData, ServerInfo and ServerDetails. These three classes are integrated into a single interface class called Database.

ServerData class holds argument types for a registered function. It also holds a collection of ServerInfo. ServerInfo class holds information about skeleton and file descriptor with which the function was registered. ServerDetails class hold information about the server's name and port number with which the function was registered.

So to put it together, the database class holds a map which takes function name as the key and value as a vector of ServerData. Since there can be multiple functions having the same name, it was redundant to hold same function names therefore function names were hashed as keys for the map. The vector of ServerData holds an instance of ServerData which has argument types information in it. It is possible that there can be multiple functions having the same name and different argument types, therefore each function is mapped to multiple argument types. Each ServerData then points to multiple ServerInfo which holds the information about the server name, port, skeleton etc related to that particular function name and argument types. There is a one-to-many relationship between function names and ServerData (argument types) and a one-to-many relationship between ServerData and ServerInfo (server name, port, etc). ServerInfo separated the server name and port information into a different struct called ServerDetails. In reality, ServerInfo holds a pointer to ServerDetails. This was done so that memory can be saved. Since a server can register multiple functions, it was redundant to create multiple ServerDetails structures for each function and argument type and store it. Therefore the system keeps a vector of ServerDetails, which essentially is a list of all the servers that have registered at least one function with the binder. Everytime a register calls come in, the respective ServerDetails entry is picked up from the collection and ServerInfo holds a pointer to it, in case it cannot find a ServerDetails entry (e.g. registering for the first time), then it would just add a new ServerDetails entry to the collection. This collection is maintained separately in the database class. This collection is also helpful in implementing the round-robin algorithm. More details on round robin are provided later.

When a request comes in from a client to the binder, requesting for the server information related to a particular function name and argument types, the database class first looks into the map for that function name. If it finds an entry, it then further looks into the collection of ServerDatas for matching argument types. Once it finds a matching argument type, it then goes through the ServerInfo collection. Based on the round robin algorithm, it then returns the next server from the ServerInfo collection. In case any of the match fails, that means there are no servers available and therefore appropriate error code is returned.

The design decision to implement the database class was done to ensure that the same database can be used by the server (as a local database) and by the client (as a cache). To add a function entry, the function AddFunctionEntry of the database class can be called. It requires function name, argument types, server name and port. It also allows optional parameters like file descriptor of the current connection to server and skeleton f. Using this database for client cache, it would not need to add file descriptor or skeleton, and therefore these are optional but the binder and the server can use them.

There are multiple functions to remove function entries like RemoveServerEntryByFunction. Depending on the nature of the information, the client, server and the binder can call any Remove* function to remove the entries from the database. RemoveServer is the main function which removes an entry. It takes in a matching function pointer which matches two entries, and if both the parameters are matched, those entries are removed. Due to the implementation of several Remove functions, it was redundant to copy that logic over and over, therefore it was put it in one place at RemoveServer. Other Remove functions now just call this function with right matching function to match the entries for removal. For instance RemoveServerEntryByFunction will call RemoveServer with a matching function that will match server name and port, whereas RemoveServerByFd will call RemoveServer with a matching function that will match file descriptors.

When a server unexpectedly disconnects from the Binder, the Binder needs to Remove all function entries from that particular Server. For this reason it can use RemoveServerEntryByFd which will remove all server entries for fd. Since on disconnect, the Binder knows the Fd, it can easily remove function entries for that server using that fd. But if it is used as a Cache on the client side, then the client might want to remove server entries based on the server name and port which it can do using RemoveServerEntryByFunction.

<u>Handling Function Overloading:</u>

As discussed in the previous section, the database is designed in a way that it can map a single function name to multiple argument types, which are then further mapped to multiple server details. The database class handles function overloading when the AddFunctionEntry function is invoked.

When AddFunctionEntry fucntion is called, database class first tries to find the function name in the hash map. If an entry exists, it then looks into the collection of argument types to check if any of the previous registered argument type matches are matched. If it finds a match, it then looks through the collection of server details to find a server that can serve the client’s request. If the current server name with which AddFunctionEntry function is called, is found, the database just returns a warning of duplicate registration and does nothing. But if it does not find a server entry, it creates and adds one. In addition, if AddFunctionEntry cannot find a particular argument type, it then adds a new argument type with server details to it. In this way, multiple functions with same name can be registered with different argument types, essentially overloading.

So now if a request comes in for a function, it is first matched based on the name, then argument type, and from the servers list registered, round robin algorithm is applied.

Managing Round-Robin:

Earlier on, it was mentioned that the Database class collects registering server information (server name and port) into a separate vector. Each server entry is used as a pointer by the actual function database for mapping function and argument types.

The database class maintains a variable nextServer which holds the index to the next server in the server collection. Now when a request comes in and it is matched to a list of servers who can fulfill this request, it will need to find the right server to return based on the round robin policy. Lets call this collection serverPointers and the original collection as serverOriginals. Now starting at nextServer, it tries to see if serverOriginals[nextServer] is pointed by an entry in serverPointers. If it does, then it means that our next server in the round robin queue can fulfill the request, therefore returning it. Otherwise it sets nextServer = (nextServer + 1) % size(serverOriginals). In this way, the simple round robin algorithm will end up at a server which satisfies the round robin condition and also exists in the serverPointers. This round-robin design allows O(1) runtime efficiency as we are using an array of servers, with nextServer holding the index of the server in the array.

Termination Procedure:

In order to gracefully terminate both the binder as well as all the servers that have been registered with the binder, the binder, once receiving a terminate message will set a flag terminateReceived to true. In the main loop where the binder listens for connections and data receiving on the sockets, it checks it terminateReceived is set. If it is set then it does not accept any new client and server requests. It will send all the servers that have been registered with it, a terminate RPC message. For this it holds a fd_set called serverFds. It will essentially sent the terminate message to all the sockets that are in serverFds. serverFds is populated when a server registers with the binder and is cleared when the server disconnects.

As soon as the server gets the Terminate message from the binder. It first validates that it is coming from the same socket as the socket connected to the binder. If it is not, the message is ignored. When the Teminate message is received, terminateReceived flag is set by the server which also blocks it from receiving any new connections. Each execute message is served in a different thread. So once all execute messages are sent back and all the clients disconnects, the server will terminate. Instead of keep a count of running threads etc and making server shutdown decision, it was better to depend on the client connections. Essentially after each client request is fulfilled, the client closes the connection. Since because of the terminateReceived flag, it won’t accept any new connections but would wait for all current clients to terminate. Doing this the server makes sure that it deals with all the requests it has received before it can shutdown.

On the binder side, the binder waits for all the servers to shutdown. After terminateReceived flag is set, it will not accept any new connections but will listen for disconnect messages from the current server connections. As soon as all servers terminate, the binder gracefully terminates.

Optimizations:

An optimization was done in the Database class. Instead of creating separate memory objects for each server details entry (server name and port), the database class just creates one object and then references this memory object from multiple function entries. This saves memory for whoever uses the database.

Another optimization was done in the ServerHandler to perform non-blocking calls. Essentially the servers should be capable of handling multiple requests at a time and for this reason threading was used. For each message received, the message is put into a queue and a separate thread is spawned off to work on it. This new thread dequeues the message from the queue, services it and sends the result back. Using this technique, multiple requests can be fulfilled at the same time and messages can also be received concurrently.

Due to the complex nature of the argument types and args, a separate block of helper functions were defined in utility.cpp. The functions include the functionality to copy arguments, check arguments matching, etc. In this way, these functions can be utilized anywhere by anyone using the system.

Protocol.h includes structures which are essentially the messages for communication. This are used by RPCMessage class for marshalling and unmarshalling. Keeping them separate makes it easier to use it in different part of the system.

Building all communication functionality in the TCP class helped in reducing any redundant code. It is used as a base class by the client, server and the binder. If it was to be built separately in all these three class, the code would be difficult to follow and comprehend and tedious to perform any bug fixes. Inheriting from the TCP class not only made the implementation of the system easy to develop, but also has helped a lot during debugging.

ProcessMessage is a function which the essential entry point to handle any message. Binder and ServerHandler specifically depend on this function. At this point, the type of message can be checked and passed onto right procedures. This makes it easier to developing new message types for communication and handling.

Error Codes:

Error codes are located in “lib_src/common.h” as definition variables

Define Variable Name	Code	Description
ERR_BINDER_ADDRESS_MISSING	-300	Binder’s environment variables are not set for the client and the server
ERR_BINDER_UNREACHABLE	-301	Binder is unreachable or it has crashed
ERR_SOCKET_LISTENING_FAILED	-302	Opening a socket connection failed
ERR_BINDER_REGISTRATION_FAILED	-312	Server registration with the binder has failed
ERR_NOT_INITIALIZED	-313	Server cannot initialize itself with binder due to connection issues
ERR_COMMUNICATION_FAILED	-314	Send or Receive has failed during communication
ERR_NO_REGISTERED_FUNCTIONS	-315	No functions are registered and rpcExecute is called
ERR_MESSAGE_SEND_FAILED	-310	‘send()’ system call returned an error
ERR_MESSAGE_RECV_FAILED	-311	‘recv()’ system call returned an error
ERR_FUNCTION_MISSING	-320	No server is available to serve the requested function call
ERR_SERVER_UNREACHABLE	-321	Client’s request to a server has failed
ERR_BINDER_SHUTDOWN	-322	Binder has shut down unexpectedly
ERR_SERVER_SHUTDOWN	-323	Server has shut down unexpectedly
ERR_SERVER_FUNCTION_RETURNED_ERROR	-330	Client’s requested function returned error
WARN_ALREADY_INITIALIZED	300	Server trying to reinitialize with binder
WARN_DUPLICATION_FUNCTION_REGISTRATION	310	Server has already registered this function with the binder
WARN_UNAUTHORIZED_TERMINATE_REQUEST	320	Unauthorized server termination request
SUCCESS	0	Everything went as expected

Other Functionalities:

The cache system for the client was implemented as a bonus. It fulfills the requirements specified in the assignment description. It caches the server entries and use it in a round robin fashion for call to servers. If call to server fails or server disconnects in a middle of execution, the server entry is removed from the cache. If the cache becomes empty, a new request is sent to binder to get the server details.

It can be tested by first starting the binder, followed by a test server and a test client which uses rpcCacheCall. Put the client to sleep after first rpcCacheCall, and then kill the binder. Once doing so run the rpcCacheCall again, this will essentially show that the client has performed caching of the server information.