CS 454

Assignment 3

Shao Yu Zhang, 20308263

Hanpei Zhang, 20292868

1. Server

Servers are responsible for handling RPC requests. Each server stores a mapping between a function signature and a pointer to the function that matches the signature in `rpcHandlerMapping`. This mapping contains all the function which a particular server can handle. To simplify the finding of the correct server/function to call for `rpcCall` and to handle function overloading in the `Binder`, we store all function signatures as strings. We have a function `normalizeArgs()` in `util.cpp` that will take the name of the function and the `argType` that it will take, and convert them into a string as the function signature.

Function registration

When a server calls `rpcRegister()`, the function is registered locally and in `rpcHandlerMapping()`. This is recorded so server knows the skeleton to use when a client approaches. Then the `BinderClient` is used to register this function with the binder (more on `BinderClient` in section 2).

Data marshalling

When a client makes an `rpcCall`, we are required to build a string that contains the function name, function argument type, and the argument values to send to the server through sockets. We have a function `serializeCall()` in util.cpp for this purpose. This function takes all required parameters mentioned above and builds a string to be send through sockets.

The function will first serialize the function signature with the aforementioned `normalizeArgs()` function in `util.cpp`. Then the function add a "#" as a separator, and then it will serialize each argument. The function loops through each argument and inspects the type (and therefore, the size of each element) the argument, and the length of the argument (if it is an array). The length of the array in ascii. We will loop through the array of argument, convert each element into a string, and increment the pointer to the next element (the increment will depends on the type of the argument). We insert a ":" between each argument and a ";" between each element of an array.

When the server receives the string, it reads the arguments incrementally. For each argument it reads the length, then for each byte, it will read the input data tokenized by “;”. For each argument the server also allocates appropriate amount of space for the arguments.

After invoking the required function on the server, the server will serialize the function in a way similar to what the client did. However, the client will only deserialize server will outputs. After receiving the marshalled message from the server, the client will unmarshall the message in the same way the server did and modify the `args` that are indicated as outputs in `argTypes`. Allocated memory are then freed.

Termination

The server will stop running when it receives the "terminate" message from the Binder. The server will simply terminate after receiving this message.

2. Binder

The binder system consists of two classes: `Binder` and `BinderClient`. `Binder` is "backend" class responsible for storing the binder database and processing the actual request. The `BinderClient` provides high level abstraction for both the client (e.g. `locateServer`) and the server (e.g. `registerServer`). The `BinderClient` class is responsible for taking request from clients/servers serialize the request, sending request and parsing response through underlying sockets.

Binder database and Function overloading

The Binder database consists of two maps. One mapping, named "`socketHostPortMap`", keeps track of all server by mapping an `int`(which contains the socket descriptor) to a `HostPort` struct (which contains the address and port of the server). This is used for removing server apis when a server disconnects. The second map, named "`apiMapping`", keeps tracks of which server can handle which function. It maps a string to a list of `HostPorts`. The string is the function signature, while the list of `HostPorts` contain the list of servers that can handle a function with that signature. Note that the signature does not keep details of array length; scalar and array is still distinguished however.

The function signature contain both the function name and the argument types. Therefore, from the `Binder`’s point of view, functions with same name and different arguments are unrelated (i.e. they are stored as separate entries in the mapping). This is how we achieve function overloading.

When a server request to register a function, the Binder will loop through all currently registered servers for that function. If there's no duplicate, the Binder will add a new entry for the function to be registered to the front of the list. Otherwise, we will send back a warning indicating the same function have been registered by the server already.

Scheduling

In the binder database, we store which server can handle which request by mapping a function signature to list of servers. When a request is sent from a client to the binder to get a server to handle a function with a particular signature, the binder will find the list which the signature maps to, and then return the first server in the list. The binder will then move the server from the first position to the last position within the same list. This is how we achieved the round robin scheduling.

Termination

The binder is responsible for coordinating termination of the system. When a client initiates rpcTerminate, the binder will loop through all servers stored in `socketHostPortMap` and broadcast termination message to each server. If there are no such servers, binder short circuits its termination. When a server receives a termination request, it waits for all of its worker threads to stop working. It is possible for worker threads to be in deadlock and this is not handled. Once all threads have terminated, server terminates also. In the meantime, binder stop listening for incoming connections and waits for all servers to disconnect. Once all server disconnects it informs the original client that the entire system will terminate. Then binder terminates itself.

3. Error Codes:

The list of error code are:

Error Code	Value	Description
MISSING_ENV_VAR	-1	Environment variables that suppose to store BINDER_ADDRESS and BINDER_PORT are missing
	-	Attempt to rpcExecute or rpcRegister function without rpcInit in the server.
BINDER_UNREACHEABLE	-3	Cannot communicate to Binder (socket operation to Binder failed)
UNINITIALIZED_SERVER	-4	Attempt to rpcExecute or rpcRegister without rpcInit.
INVALID_NAME	-6	Attempt to run or register a function with null name pointer.
INVALID_ARGTYPES	-7	Attempt to run or register a function

		with null argument type.
NO_SERVER_WITH_ARGTYPE	-9	No server have registered a function that handles given function signature
SERVER_DOES_NOT_SUPPOR T_CALL	-10	Called a rpc on a server that does not actually process it.
SERVER_UNREACHEABLE	-11	Cannot communicate to Server (socket operation to Server failed)
INVALID_SKELETON	-12	Skeleton cannot be null pointer.
NO FUNCTION TO SERVE	-13	rpcExecute without any registered functions.

Warning Code	Value	Description
REREGISTER SAME INTERFACE	1	Server has already registered a function with the same argument types. Previous register's skeleton is replaced.
EXCEED MAX ARRAY LEN	2	An array argument contains more elements than is needed by the server's function. Extra elements may be ignored.

3. Implementation.

All functionality are implemented, as is rpcCacheCall.

4. Advanced Functionality

During our testing, it was very tedious to test/debug having to set host/port every time. There was a command line flag to servers to allow servers to read a hostport file to public_html directory.
This is **disabled** at submission.

rpcCacheCall is implemented also. This is fairly straightforward. Binder returns a serialized format of functions and hostports that is able to service them. The client deserialize them by tokenization and reconstruct the mapping. Logic to find hostports is wrapped around calls to rpcCallHelper which is used for both rpcCall and rpcCacheCall.