CS 454

Assignment 3

Shao Yu Zhang

Hanpei Zhang

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()`, we add a new entry in `rpcHandlerMapping()`. We will then register this function with the binder by calling `registerServer` function in `BinderClient` (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 the 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). 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 uses the delimiters ";" and ":" to break up the string to elements. FThe server will then unmarshall the string by taking each element and build them back into an array.

After invoking the required function on the server, the server will re-marshall the function in a way similar to what the client did. However, the server will only marshall arguments that are labelled as 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`.

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` class contain functions which both the clients (e.g. `locateServer`) and the servers (e.g. `registerServer`) to call. The `BinderClient` class is responsible for taking request from clients/servers, and send the processed request to the `Binder` through sockets.

Binder database

The Binder class keep two mappings. One mapping, named "`socketHostPortMap`", keeps track of all server by mapping an `int`(which contains the socket) to a `HostPort` struct (which contains the address and port of the server). This is used for terminating all servers when required. The second map, named "`mapping`", keeps tracks of which server can handle which request. 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.

The function signature contain both the function and the argument types. Therefore, in 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 function for that server. 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 on error code indicating the same function have been registered.

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 remove the server in the first position and put it last in the same list. This is how we achieved the round robin scheduling.

Termination

The binder is responsible for handling termination of the system. When a client send a terminate all message to the binder, the binder will loop through all servers stored in `socketHostPortMap` and sent termination message to each server. Each server is responsible for their own termination. After sending out the message, the binder will terminate 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
UNINITIALIZED_BINDER	-2	Attempt to execute Server or register function without initializing Binder
BINDER_UNREACHEABLE	-3	Cannot communicate to Binder (socket operation to Binder failed)
UNINITIALIZED_SERVER	-4	Attempt to execute Server or register function without initializing Server
NO_BINDER_ADDRESS	-5	No binder address specified
INVALID_NAME	-6	Attempt to run or register a function with empty name
INVALID_ARGTYPES	-7	Attempt to run or register a function with empty argument type
NO_SERVER_WITH_ARGTYPE	-9	No server have registered a function that handles given function signature