CS454 a3 documentation

J27feng 20475044
Xx2lin 20480200

Design choices:

For structure of our binder database, we used a deque and a vector:

```
vector< map<string,pair<string, string> > >VoMap;
deque<pair <string, int> > RR_Queue;
```

VoMap is a vector of map where each map contains a server's functions' info. Inside each map, the key is a string that forms by concatenating a function name and its argtype together as one function's signature and the value is a pair of server identifier and port number.

The RR_Queue is our Round robin queue where the next server should serve the client is at the front of the deque. We append the server identifier and its port together as a string and set it to the first object in our pair. For the second part of the pair, we store the index of the corresponding server map from the vector above. In that way, RR_Queue is connecting to the VoMap.

When registering, if that location is not found in our RR_Queue, we will create a new map containing the function info and insert into VoMap. Immediately after that, we will push a new pair into the back of our RR_Queue to indicate the server location and its index in the vector. If we found the server location in RR_Queue, we will then use the index we found to get our map for this server. Then, we can insert it or overwrite it. Since we use map to record functions and function signature as the key, function overloading can be easily accomplished by calling map[key] = value.

For round-robin scheduling, we will always search the function inside the server at the front of the RR_Queue first. Then, no matter we found the function or not, we will pop this pair back from the front and push it into the back immediately. If we found the function, we will then send back the server location we found. If we can't found the function, we will loop through the whole RR_Queue and do the same thing. At the end, if we really can't find the function among all of the server recorded, we will send a LOC_FAILURE back to the client along with the reason code indicating the function is not found.

For marshalling data, we used reinterpret cast to cast each argument type to a four bytes char array and each argument to a four bytes unsigned char array. And they are ordered as follow

e.g. [# of arg] [at1] [at2] [at3] [a1] [a2] [a3]

When server calls rpcExecute(), it will store the argument arrays first and then parse the argument based on the data type and if it is a scalar or array from its corresponding argument type.

For termination procedure, we record the count of live servers each time accepting. We also decrease the count when knowing a server socket has been hung up. When TERMINATE is received, the binder will run in a loop while the count is larger than 0. Inside the loop, we will loop through our master set and send a TERMINATE message into each socket. In that way, the outside loop will eventually stop after knowing all servers have terminated themselves.

Error codes:

rpcInit

-2 can not connect to binder
-1 can not bind to listenSocket
0 initialize success

rpcCall/rpcExecute

-4 execute failed
-3 failed to communicate with server
-2 location request failed
-1 failed to communicate with binder
0 success
1 unknown argument type
2 termination message is not from binder

rpcRegister

-3 register failed
-2 cannot read from binder
-1 cannot write to binder
0 success
1 register overwrite
2 register duplicate

Functionality:

We provide all functionalities except the bonus part.