Assignment 3 Documentation

Junxing Mao

Zean Zhao

Marshalling and Unmarshalling

- With an argType variable, spit it up into 4 properties of input, output, type, and length using masks
    - -where input on but 31, output on bit 30, type between bit 23-16, length on bits 15-0
    - When using, the first number seen is the number of argTypes followed by all the argType variable properties, separated by spaces
- With each argValue, marshalling it by taking the value and string it all together separated by spaces
- To unmarshalling, just do the exact opposite of when marshalling
    - Group each argType back together bitwise
    - Apply argType properties correctly to their corresponding argValues

- **RPC Library**
- rpcInit
    - -open server to binder connection and client to server port
        - -return value of 0, everything is good
        - -return value of -1, can't find available address
        - -return value of -2, can't find available port
        - -return value of -3, fail to start listening on opened to client socket

- rpcCall
    - -creates a new client connection to the binder
    - -process each argument into input, output, type and length
    - -store them all in a vector of unsigned int
    - -send this to the binder
    - -ASSUMING all provided args match their associated types in argsType
    - -3 types of binder response
        - -call_success, we got server info as part of the response, open a connection to it and send to that server everything, return 0
        - -call_warning, something happened, we don't really care, return 1
        - -call_fail, function doesn't exist, return -1
    - -going assume here the binder does all the type checking so that when i get call_success, i just use the provided arguments and run the associated function

- rpcRegister
  - -process each argument into input, output, type and length
  - -store them all in a vector of unsigned int
  - -call send with the server's addr and client portnum using the toBinderSocket opened during init
  - -if send fails then clearly we cant connect to the binder, return value of -1
  - -since there is only 2 options left thus
    - -if binder response if register_success then we store a local copy of the function, return value of 0
    - -else the response will be register_warning meaning a copy already exist so we don't have to care about it, return value of 1

- rpcExecute
  - -infinite loop, 2 threads for the 2 different ports
  - -if toBinderSocket receives a message of "terminate", this should be the only communication between server to binder
    - -step1: respond with "terminated"
    - -step2: close all opened ports by the server
    - -step3: kill all other threads
    - -step4: exit
  - -if toClientSocket receives a connection, need to process the message back to char* name, int* argTypes, and void** args then pass this to the correct skeleton within a new thread
    - -respond with execute_success, going assume here everything is set during the skeleton execution, so that i just have to exit and close the connection after wards
    - -if the execution still fails, even though it should fail ever now, return to the client whatever the return value was from the skeleton
  - Unfortunately, we could not get the function to return the skeleton result in the correct format,

- rpcTerminate
  - -creates a new client connection to the binder
  - -send a terminate msg
  - -close the connection
  - -everything else is handled by the binder
  - Takes awhile to terminate due to too many threads

- processArgTypes
  - -calls a mask to help process all the arguments
  - -store each argument's input, output, type, and length within a vector

- getBits
    - -a mask from bit a to bit b

- clientSocketSetup
    - -opens a client connections to a provided addr and port

- **Binder:**
    - Database:
    - There were several data structures used to store the registered functions of each server.
    - A class, “sig”, was used to store in, out, argtype, and arglength of each function parameter.
    - A separate class, “proc” contained the procedure name, “name”, the server location as well as port, “svrname” and “svrport” respectively, and finally a vector of the function parameters (“sigs”) for the procedure.
    - The database itself is a map mapping the socket file descriptor that the binder uses to connect to a server to a vector of the server's supported procedures.
    - A separate vector of socket descriptors was created to facilitate a round robin scheduling system.

    - Round Robin scheduling:
    - There is a vector that contains the identifier for every server which has registered functions to the binder
    - when an rpcCall is received by the binder, it searches through the vector, searching if the procedure is available for the current server identifier
    - when a server is found to have the procedure, it is the first one in the vector to have it.
    - There server identifier will be moved from it's current position in the vector to the back of the vector as a priority change implementation.

    - Handling rpcTerminate:
    - The binder, when receiving a terminate command from any client, will proceed to
        - close the connection to the client
        - loop through all open connections (to the servers) and send "terminate" to each server
        - for each connection wait for servers to send "terminated" back
        - close that connection and clear it from the list of connections
        - close the listener socket

- Handling rpcRegister:
- The binder, when receiving a register command from any server, will proceed to
    - read in the name of the procedure
    - read in the machine name and port of the server's listening socket
    - read in the function signatures of the procedure
    - store the name, machine name and port, as well as the signatures in the proc class
    - check if the database already has the same procedure listed with the current server
    - register the procedure under the current server and send "register_success" to the server if it is a new procedure
    - do not register the procedure and instead return "register_fail" to the server if it is already registered

- Handling rpcCall:
- The binder, when receiving a call command from any client, will proceed to
    - read the name, as well as the function signatures of the procedure
    - find if there exists a server with the procedure registered in the database
    - if found, send back "call_success, <machine name>, <port>" to the client and close the connection
    - else, send back "call_fail" to the client and close the connection