Guotian Ye(gye)
Yiyao Liu (y435liu)

CS454 – A3

System Design

1. Marshalling/Unmarshalling of data

   We use message format: Length, Type, Message.
   Where Length indicates Message's length, Type indicate Message's type.
   Both Length and Type are 4 bytes.

   Format:

   Server/Binder: Length, REGISTER, server_id, server_port, name, argTypes
   Length, REGISTER_SUCCESS, indicator
   Length, REGISTER_FAILURE, indicator
   Client/Binder: Length, LOC_REQUEST, name argTypes
   Length, LOC_SUCCESS, server_id, server_port
   Length, LOC_FAILURE, reasoncode
   Client/Server: Length, EXECUTE, name, argTypes, args
   Length, EXECUTE_SUCCESS, name, argTypes, args
   Length, EXECUTE_FAILURE, reasonCode

   - Marshalling:
     Create a consecutive memory block, copy necessary data into this block in designed formats, and then send it out as a whole

     Since "*args*" from clients OR manipulated after server is an array of pointers that point to different local memory location, we use a helper function called *pickle(int* argTypes, void** args)* to pack all memory pointed by *args* into one whole consecutive memory block first. And then send it out.

   - Unmarshalling:
     Extract data from received data block according to designed format. Specially, we use *unpickle(int *arg_types, char* mem_block)* to create an identical new args from the memory block.

2. Helper Classes

   Two helper classes: *Prosig* class and *ServerLoc*
   Prosig represent a 'procedure signature' and here is it definition:

   class Prosig{
   public:

```
    std::string name;
    int argNum;
    int* argTypes;

    Prosig();
    Prosig(std::string name, int argNum, int* argTypes);
    ~Prosig();
    bool operator==(const Prosig &other) const;
};
```

name is procedure name, *argNum* is the number of parameter of the procedure and *argTypes* are parameter types following the rules specified in the assignment. *argNum* is not necessary for us to decide a procedure, but it's stored to help us find / compare them quicker.

ServerLoc represents a 'server location'. Here is the definition of them:

```
class ServerLoc
{
public:
    char identifier[SIZE_IDENTIFIER + 10];
    char portno[SIZE_PORTNO + 10];

    ServerLoc();
    ServerLoc(char* identifier, char* portno);
    bool operator == (const ServerLoc &other) const;
    ~ServerLoc();
};
```

With the help of these classes, the database can be created and used with less efforts, because we can store them as an object, and compare / decided if two objects are identical without any extra efforts.

3. Binder database:

There is a list of servers that have registered to the server, and for each server, it maintain a list of its own procedures (*Prosig*).

In *Tuple*, the *int first* is the socket number that represents the always-open connection between binder and server. With the assumption that the connection is kepe active, socket number ans ServerLoc are identical in identifying a server. Both are recorded for different usage.

```
class Tuple
{
public:
    int first;
    ServerLoc second;
    std::list<Prosig> third;

    Tuple();
    Tuple(int first, ServerLoc second, Prosig function);
};

class BinderDB
{
public:
    std::list<Tuple> database;

    int Register(Prosig function, ServerLoc ser, int sockfd);
    int SearchServer(Prosig function, ServerLoc *ser);
    void Cleanup(int sockfd);
};
```

ServerLoc is used mainly for handling incoming LOC_REQUEST from clients, so that server hostname and server port number can be sent back. *First* (socket number) is used to implement Round-Robin algorithm. Each time a server is retrieved from a LOC_REQUEST, it is pushed back to the end of the list. When a server just serves a client, moving it to the back takes only 1 step.

4. Function overload handling

5. Management of Round-Robin scheduling

6. Multi-threading

   Whenver rpcExecute() in server(s) receives EXECUTION requests from clients, it will pork a new thread call execute(), which handles the desired calculation, including find the correct function in server database.

   When rpcExecute() receives a TERMINATE request, it stops receiving new EXECUTION requests, and waits for currently running threads to terminate, and then exits.

7. RPC functions

Int rpcInit():

1. Servers call this functions.
2. It will first open a connection to the binder, and keep this connection open for later REGISTER
3. It will also creates a connection socket to be used for accepting connections from clients
4. Return 0 for success, negative for failure

int rpcRegister(char name, int* argTypes, skeleton f):*

1. Servers call this function
2. It will register function called name in the binder.
3. Return 0 on success, negative on failure, positive on warnings

int rpcCall(char name, int* argTypes, void** args):*

1. Clients call this function
2. Look for a function called "name" to do some calculation with parameters argTypes and args.
3. Return 0 on success, negative on failure, positive on warnings

int rpcExecute():

1. Servers call this function
2. This function will handle EXECUTION requests from clients and send back the results
3. Return 0 on success, negative on failure, positive on failure

int rpcTerminate():

1. Clients call this function
2. any client can send a TERMINATION request to Binder, after which Binder will send the termination signal to all of its registered servers to shut down. If any server(s) were still processing some existing threads, they would first finish with those threads, and then terminate.

Codes

Success Code

All success codes are represented by 0, which includes:

REGISTER_SUCCESS 0
LOC_SUCCESS 0
RPCCALL_SUCCESS 0

EXECUTE_SUCCESS 0
TERMINATE_SUCCESS 0
INIT_SUCCESS 0;

ERROR Code

1. LOC_FAILURE: -1
   - ✓ Clients "rpcCall" functions that Binder doesn't have
   - ✓ Clients cannot send LOC_REQUEST due to connection failure
   - ✓ Binder has shut down when clients' requests
   - ✓ LOC_REQUEST message is lost during transmission

2. REGISTER_FAILURE: -2
   - ✓ Mostly due to server/Binder connection failure
   - ✓ Server didn't not pass in proper values to the Register request.

3. EXECUTE_FAILURE: -3
   - ✓ Lost connection with the network
   - ✓ Clients cannot send EXECUTE requests to server due to connection failure

4. RPCCALL_FAILURE: -4
   - ✓ No such procedure is registered
   - ✓ Clients' connection to either Binder or servers fails.
   - ✓ Either Binder or servers have already shut down when calls rpcCall().
   - ✓ When EXECURE_FAILURE occurs, clients receive RPCCALL_FAILURE.

5. INIT_FAILURE -6
   - ✓ Environment variable is not properly set
   - ✓ Mostly due to connection problem

6. TERMINATE_FAILURE: -6
   - ✓ Connection breaks when sending TERMINATE requests.

7. UNKNOW_REQUEST: -7
   - ✓ Usually shouldn't happen, jus in case, to let people know maybe the type of service is still unavailable, or the sent data is corrupted

8. NO_RPC: -8
   - ✓ No such RPC is registered on binder

9. NOT_REGISTER: -9
    - ✓ Server didn't call register a single procedure using rpcRegister, but call rpcExecute firstly

10. CANT_CONNECT_BINDER: -16
    - ✓ Problem connecting to Binder, like binder accidentally quit.

Warning Codes

1. REGISTER_DUPLICATE: 100
    - ✓ Server has registered the same function to binder before

2. SERVER_UNAVAILABLE: 101
   (undefined right now)

Accomplishment

Our group implements al the required functionalities as specified in the assignment.
We did not implement any advance feature.