CS454 Assignment 3 System Manual

By Shuopeng Zhang and Taiyue Liu

Index

Student ID and Userid:
Name: Taiyue Liu ID: 20441344 userid: t67liu
Name: Shuopeng Zhang ID: 20395527 userid: s89zhang

Design Choices:

- **Marshalling/Unmarshalling**

There are two methods for marshalling and unmarshalling and they are applied to different cases.

The First Method for marshalling and unmarshalling:

The marshalling and unmarshalling between the Binder and a Client:

The marshalling processes on the Client's side are:

- Sends a "1" to the binder to indicate that this is a client.
- Sends an integer that is the size of the function name (including '\0') to the binder. For example, the size of function name "f0" is 3 ('f', '0', '\0').
- Sends the function name to the binder.
- Sends the types of arguments to the binder separately until the client get a zero and then send zero out (If there are N arguments, then for the types of them and the ending zero, the client needs to send N+1 times).

The unmarshalling processes on the Client's side are:

- After sending messages to the binder, the first data received from the binder is an integer that indicates whether the binder found a server for the client or not. If the number is "0", which means the binder did get a server, then the client will be ready for receiving the information of that server. If the number is "1", the client will know there is no server can help it and stop receiving messages.
- Receives the size of the address of the server, the address of the server, the size of the port number of the server and the port number of the server separately from the binder in order. The size of the address and the size of the port number received are for allocating space for storing the address and the port.

The unmarshalling processes on the Binder's side are:

- Receives an indicator, if it is "1", the binder knows it is the client who is asking for support.
- Receives an integer N, the binder prepares a char array whose size is N to store the incoming function name.
- Receives the function name and puts it into the char array.
- Receives the types of arguments separately and stores them in a vector.

The marshalling processes on the Binder's side are:

- Based on the types that the binder received and the information in its database, the binder decides the address and the port number of a server that can handle this request to the client. Then, the binder sends a “0” to inform the client that there is a server can support that service. Otherwise, the binder sends a “1” to inform the client that there is no server can help it.

- If the binder did find a server for the client. Then the binder will send the size of the address of that server, the address of that server, the size of the port number of that server and the port number of that server separately to the client in order.

The marshalling and unmarshalling between a Server and a Client:

The marshalling processes on the Client’s side are:

- Based on the information gotten from the binder, the client try to connect to the server that can serve it.
- Sends “1” to the server to inform the server that this is a request from the client.
- Sends an integer that is the size of the function name (including ‘\0’) to the server. For example, the size of function name “f0” is 3 (‘f’, ‘0’, ‘\0’).
- Sends the function name to the server.
- Sends the types of arguments to the server separately until the client get a zero and then send zero out (If there are N arguments, then for the types of them and the ending zero, the client needs to send N+1 times).
- Sends the arguments to the server separately. If an argument is an array and the size of it is N, then for this argument, the client needs to send out the N elements in that array separately. Since the client stores the size of that array in the corresponding arguments type, the client knows when all the elements in that array have been sent out and then continue to send next argument.

The unmarshalling processes on the Server’s side are:

- Receives a “1” from the client and know that the client is asking for service.
- Receives an integer that is the size of the function name (including ‘\0’) from the client. For example, the size of function name “f0” is 3 (‘f’, ‘0’, ‘\0’).
- Receives the function name from the client.
- Receives the types of arguments from the client separately until the server gets a zero (If there are N arguments, then for the types of them and the ending zero, the server needs to receive N+1 times).
- Receives the arguments from the clients separately. If an argument is an array and the size of it is N, then for this argument, the server needs to receive the N elements in that array separately. Since the server has the size of that array in the corresponding arguments type, the server knows when all the elements in that array have been received and then continue to receive next argument.

The Second Method for marshalling and unmarshalling:

In this case, all message will be convert to a string first, regardless it is an integer, char, float, or double. Then, we will calculate the string size and send. After that, the string will be send.
The receiver needs to receive the size first, and then allocate a char buffer of the received size. Then, it will receive the actual data.

The marshalling and unmarshalling Between Server and Binder:

The marshalling processes on the Server's side and the unmarshalling process on the Binder's side are:

The primary message from Server to Binder is to register functions of the server into the database of binder. For the sake of simplicity, all message are break down as much as we can. For example, suppose we have a function
int foo(int a, int b[]) where b is an array. The server will first send a number 0 to the binder to tell the binder that the server wants to talk to him. Then, the server will send the total number of parameters the foo needs to receive. In this case, the number of parameters is 2 (i.e. a and b). Initially, instead of send total number of parameters, we sent total number of time the binder needs to receive. However, since all other information we needs to send, such as hostname, function name, are deterministic, we change it to the number of parameters. This is the reason why we send total number of parameters first. After that, server sends a string "REGISTER" to the binder to tell the binder that it wants to register functions. Later, the server will send hostname, port number, and function name. After that, the server will send the parameter types. Note that the parameters data type are sent independently. That is, each time, we only send one parameter's data type. For example, we will send data type int first for parameter a, and send int array for parameter b after that.
The binder just receives the all messages in the send order of server.
All the messages send from server to Binder through method 2.

The marshalling processes on the Binder's side and the unmarshalling process on the Server's side are:
The binder normally does not talk to server, except that it tells servers to terminate.
When the binder wants servers to terminate, it will first send a 0 to all the servers to tell them that binder want to talk to them. Then, the binder will send a string "terminate" to all the servers. The servers will then terminate.
All the messages send from Binder to Server through method 2.
Marshalling/Unmarshalling Between Server and Client
The marshalling processes on the Server's side and the unmarshalling process on the Client's side are:
This happens when a server want to return a function result to a client. The server

will first send to a number to client to indicate that whether the function running is success or not. Then, the server will check if there is anything needs to return. If there is, the server will send 1 or a number greater than 1 to the client to tell the server that the function's return value is a scalar or an array. If it is scalar, it will return 1. If it is an array, it will return a number greater than 1. After that, the server will send the data type index to the client. Then, the server will sends all the actual return value(s) to the client. The client will receive these messages in the same order as the server sends them.

All the messages sent from server to client through method 2.

- **Binder Database Structure**

The Database structure for storing servers' information:

We use a vector of "struct locSturct" to store the information of all servers. Each server is stored in exactly one "struct locSturct" and is put into the vector. The "struct locStruct" contains a string "hostname, which is the address of the corresponding server, a integer "portnum", which is the port number of that server and a Boolean "exist" to show that this server is still open. The server with an ID n will be stored in the n-th position in the servers' information vector.

The Database structure for storing functions' information:

We use a vector of "struct proce_info" to store the information of all functions. Each function is stored in exactly one "struct proce_info" and is put into the vector. Several functions with the same name but different argument types will be stored in only one "struct proce_info" (Actual implementation for this structure is in "Binder.h" file).

The "struct proce_info" contains a string "name", which is the name of the function, a vector of "proce_para" called "parameter" where "proce_para" is vector of "struct datatype" and a vector of "loc_list" called "loc_id" where "loc_list" is a vector of integer. Each "struct datatype" stores the type information of one argument, where the information includes whether the argument is Input or Output or both, the type of the argument and whether it is an array or not. Each "proce_para" is a vector of "struct datatype", which represents a set of argument types and the "parameter" represents multi-sets of argument types (since we put the functions with the same name but different argument types in one "struct proce_info", so we use "parameter" to store all sets of argument types that functions use). Each vector in "loc_id" contains the IDs of a set of servers that can support the corresponding function with certain set of argument types. If server "5" is in the second vector in "loc_id", then the server "5" can support the corresponding function with the argument types in the second vector of "parameter".

- **Round-robin Scheduling**

In the binder's database, we maintain a vector of all servers' IDs including those servers that is closed called "round_robin_list". When a client is asking for service, the binder will first find out the vector of servers' IDs called "it" where these servers can support this service. Then we run the helper function "round_robin" in file "Binder.c" with "it". The function "round_robin" will return the server in "it" with the highest order, that is in the most front position in "round_robin_list" around the servers in "it", erase that server ID from the "round_robin_list" and push it at the back of "round_robin_list". By using this algorithm, we can ensure that servers can share working load wisely.

- **Function Overloading**

As what we mentioned in binder's database structure, several functions with the same name but different argument types will be stored in only one "struct proce_info". According to their different sets of arguments, we put their sets of arguments in different positions in "parameter" and put their IDs in different positions that are same with their positions in "parameter" into "loc_id". When a server tries to register a new function, the binder needs to check whether there is a function that has the same name with the new function or not. If there is, then the binder checks whether the argument types of the new function have been registered. If there is no function has the same with the new one, then the binder registers a new function name associated with its argument types and server ID and adds it into the database.

If the function name has been registered, then binder needs to check whether the new function has the same argument types with an existing one. If there is, the binder just add the server id to the "loc_id" to indicate that this server can support the function with these arguments types. If there no such set of arguments types, then the binder needs to push the new vector of arguments types at the end of "parameter" and push the new vector containing only this server ID at the end of "loc_id".

This multi-sets design can efficiently support function overloading.

- **Termination procedure**
  - A client sends a termination message to the binder.
  - The binder receives the request and sends shut down messages to servers through the sockets associated with servers.
  - The servers receive the shut down messages from the binder and close themselves

Error Code Checklist

CLIENT_FAIL_TO_CONNECT_TO_BINDER (-4)
: The requesting client cannot connect to the binder.

CLIENT_FAIL_TO_CONNECT_TO_SERVER (-5)
: The requesting client cannot connect to the server.

NO_SUCH_SERVIE (-3)
: The binder cannot find a server to support the service requested by the client.

FAIL_TO_SEND_MEG (-1)
: The message cannot be sent out.

FAIL_TO_RECE_MEG (-2)
: The incoming message cannot be received.

FAIL_TO_GET_ADDR_INFO (-6)
: Cannot put server’s information in a packet.

UNKNOWN_VARABLE_TYPE (-7)
: Wrong variable types.

BINDER_GET_SOCKFD_FAIL (-8)
: The binder cannot create a socket.

BINDER_LISTEN_FAIL (-9)
: The binder cannot listen messages from clients.

SOCKET_BIND_GETSOCKNAME_LISTEN_CONNECT_GETADDRINFO_FAIL(-11)
: Creating socket, binding socket, getting socket name, listening, connecting or get address information fails.

REGISTER_DATA_IN_BINDER_FAIL(-14)
: The server cannot register a function in the binder’s database successfully.

SELECT_FAIL(-12)
: Selecting file descriptor fails.

```
SERVER_UNMARSHALL_FAIL(-13)
: Server cannot unmarshall the messages from the client successfully.
```

Functionality Description

- All the functionalities (except rpcCacheCall ()) that are descripted in assignment 3 can be implemented well by our library.
- The rpc functions can still work well in the cases that Multi-clients, Multi-servers, or servers, clients and binder are not on the same machines.