CS 454 Assignment 3 System Manual

Chao Chen (c99chen)
Honghao Zhang (h344zhan)

Design choices:

Overview of overall remote procedure call system

First is system configuration. Usually, there will be a central binder, which severs certain number of servers and clients. The calls between each component is shown in Fig1.

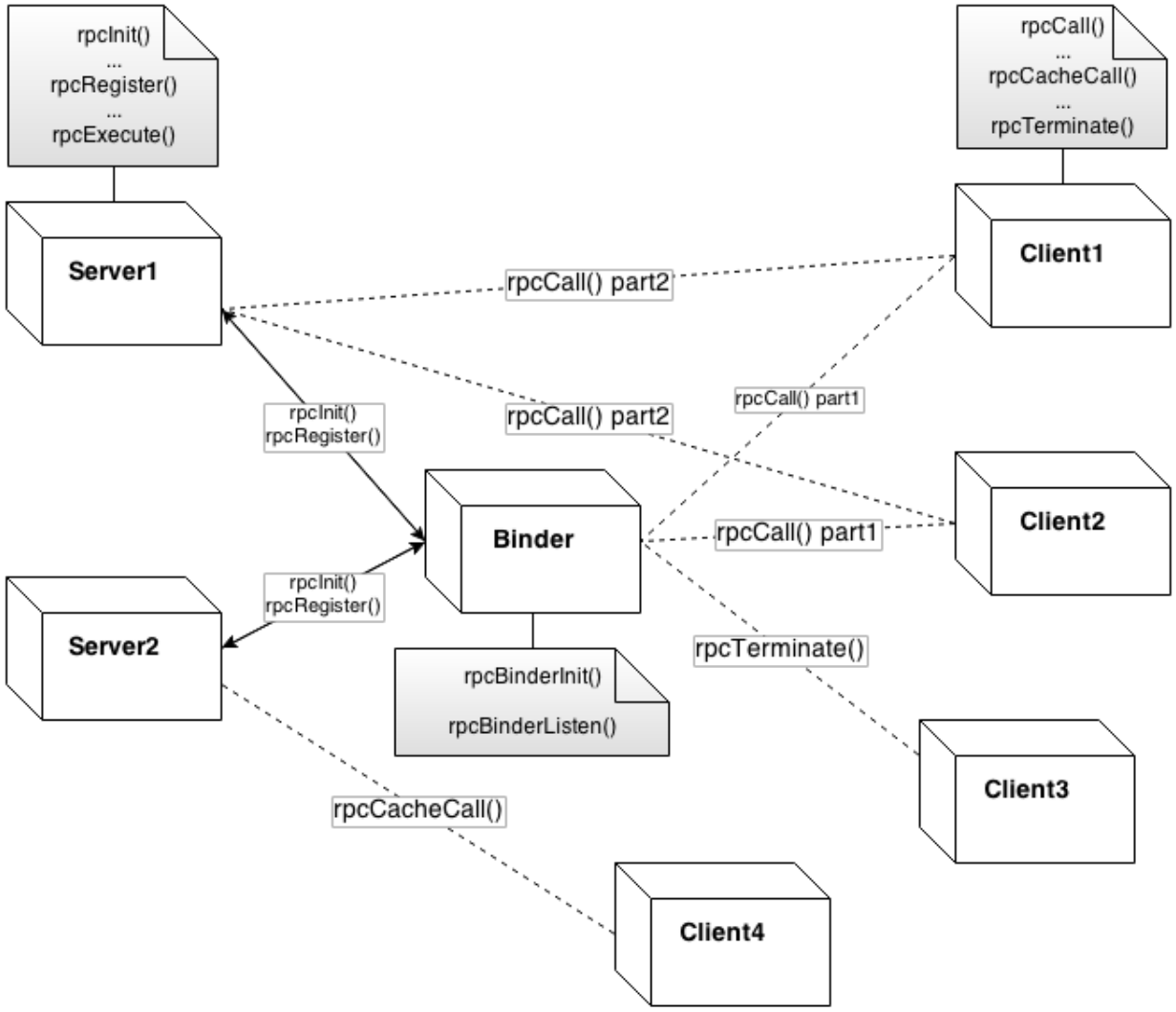

Fig.1 System configuration

Life cycle of the remote procedure system is as follow: First, binder will initialize itself, set up socket that will handle connections from servers and clients. After successfully initialized, binder start to listen any calls. Second, servers will start to initialize themselves, set up connection socket to binder and listen socket for clients. Then, each server will register it's procedures in it's local database and in binder's database. After successfully registered, servers start to execute and is ready to server any calls from clients. Third, for clients, each client will use rpcCall to call

a remote procedure, server will first get an IP/Port of an available server and use this address to call server. Client can also use rpcCacheCall to cache a copy of current available servers for a procedure. Once those cached servers can be called and execute procedure successfully, clients will never contact with binder. If all cached servers cannot handle a procedure call, new copy of server list will be retrieved from binder. Finally, client will send a terminate message to binder, then binder will inform all it's connected servers to terminate. Once all servers are terminated successfully, binder will terminate itself.

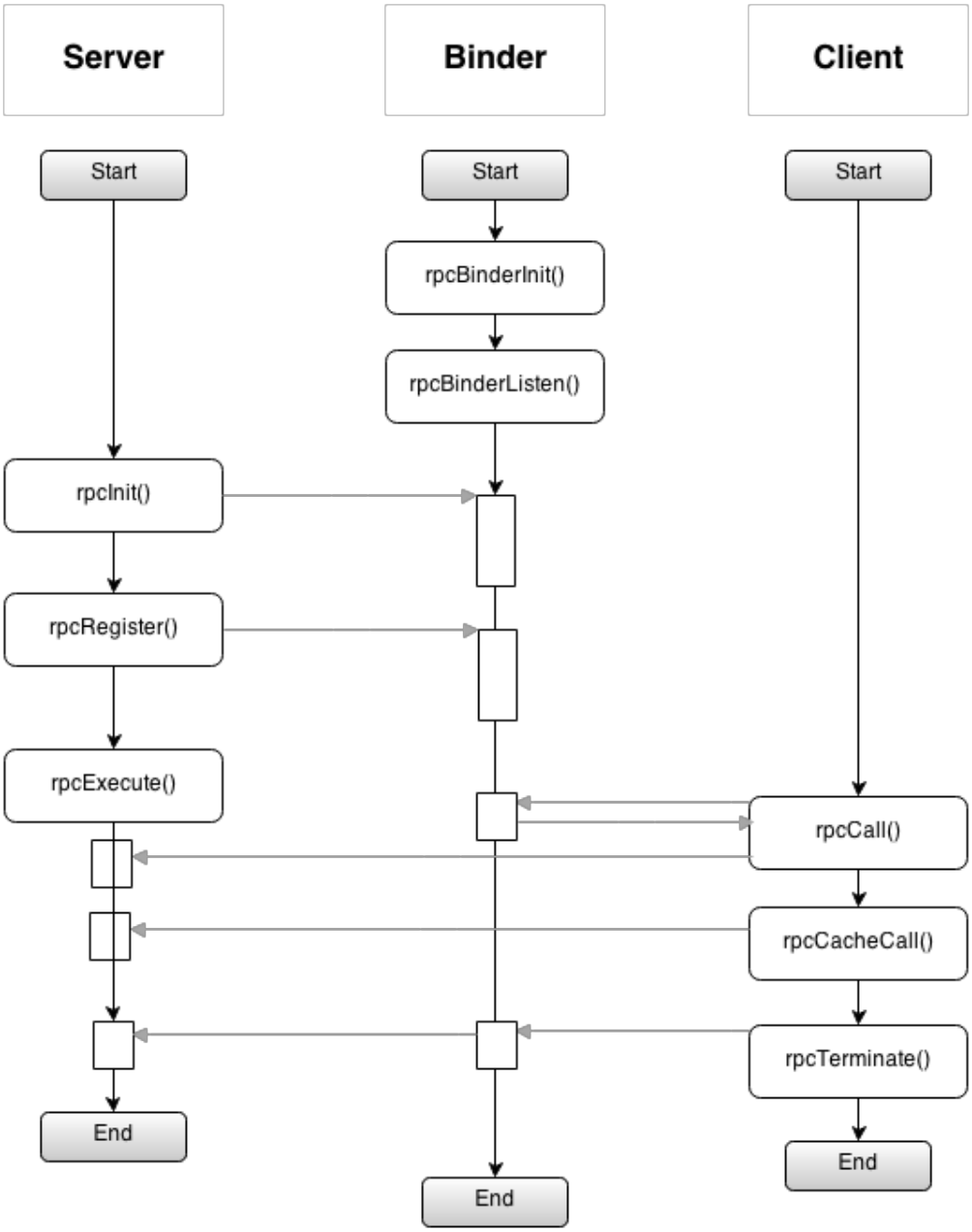

Fig.2 System Life Cycle

Overview of RPC library

We split rpc library into 4 files, rpc_binder, rpc_server, rpc_client and rpc_helper. Despite these four files, we have another class named pmap, which is an data structure support of this library.

Design Choice

Message Protocol

In our system, we convert data into bytes to send among binder, server, and client. In communication between client and binder, binder and server, client and server; Most of messages will meet the standard message protocol, which is

LENGTH | TYPE | BODY

This will involve three message sendings. The first is message body length, second is message type and the last one is the message body. Receiver will receive it's message according its expecting message protocol. Which will first receive message length, then type and last message body (will dynamically allocate memory space according received message length). Each sending and receiving will have size error checking.

Both message length and message type will be exactly 4 bytes in all cases. Possible message types are:

REGISTER	REGISTER_SUCCESS	REGISTER_FAILURE
LOC_REQUEST	LOC_SUCCESS	LOC_FAILURE
EXECUTE	EXECUTE_SUCCESS	EXECUTE_FAILURE
TERMINATE		
LOC_CACHED_REQUEST	LOC_CACHED_SUCCESS	LOC_CACHED_FAILURE

Marshalling/Unmarshalling

All message body is byte stream. Despite it's variable type, either pointer or simple type like int. In sender side, All of them will be assembled into a pre-allocated BYTE messageBody[messageLength]. In receiver side, according different message type, message body will be received and processed to it's original data type as sender side.
For pointers, the original data will be assembled (memcpy).

The marshalling between Client and Binder

1. Client sends a Location Request to binder after connecting to binder. On client side, the information will be converted to bytes and appended into one message body as format [name, argType,].
2. Client sends binder message length first, and then, send message type. Client sends the message body after those two packets.
3. Client sends message length, message type, and message body (function name, function argTypes) to server to request an function execution
4. Binder sends response type(e.g LOC_SUCCESS) and a reason code to client.
5. Binder append the server IP and port into one single message and send back to client.

The unmarshalling between Client and Binder

1. Binder receives client's location request first; from location request to know the sender is a client.
2. Binder receives message body from client by using message length that sent by client before to get the information and data in message body [name, argtype]
3. Client receives the response type and reason code.
4. Client receives message body contained server information from binder, and disassemble the message body to get server IP and port.

The marshalling between Client and Server

1. Client assemble a message by using function name, function argTypes, and arguments. Client will send the message length to server first. Sending the message type (EXEC_REQUEST) after message length. The last one is message body. (follow the protocol)
2. Server assemble a message by using function name, function argTypes, and the arguments after executed the skeleton functions. Server sends the message back to client.

The unmarshalling between Client and Server

1. Server receives message length, message type and message body from client. Server will disassemble the message and pass the arguments into skeleton functions to get the result.
2. Client receives message from server after server done the execute. Client will disassemble the message to get the result.

The marshalling and unmarshalling between Server and Binder

1. Server will send REGISTER type of message to binder, message body will be in this format [name,argTypes,IP,Port,], each different sector will be separated by ','. In binder side, it will allocated the precise amount of buffer for message body, detects the message type is REGISTER and handle the data use the corresponding method. In this method, it will split the message body according predefined separator ',' into 4 parts: name bytes, argTypes bytes, IP bytes and port bytes. These four parts will be used to

initialize the real data: name, argTypes, IP and port bytes. Binder will respond either REGISTER_SUCCESS or REGISTER_FAILURE, to inform the result of this procedure registration.

The structure of binder database

On binder side, there is a data structure named pmap. There are several vectors in pmap as private members to store data as a binder database.

Binder procedure main table

<<name, type>, socket>	<IP, Port>
<<f0, int (int, int)>, 2>	<129.60.2.15, 58763>
<<f0, int (short, char)>, 3>	<29.87.2.49, 39754>

Binder round robin list

vector<socket>	vector<socket>	vector<socket>
1	**1**	3
2	3	4
3	4	2
4	2	1
1	2	3

The case1 is after all server registered same functions, there is a sequence of them in vector. Assume this time the number 2 server is the first server can handle the request(maybe function f0) from client. We will remove the #2 server and put it into the bottom of vector. In case2, a request of f1 sent from client side and #1 server will handle it since it is the first one can handle f1 in vector. After, we pick #1 to handle the request, #1 will be removed and push onto the bottom of vector.

Handling of function overloading

On binder side, there is a vector of pairs of (function name, argTypes) and skeleton. It is used to store the function registration information. Since the same server registers a same function with different arg types. The vector will put the new pair into the itself. The old function will be still in vector.

Managing round-robin

In the binder's database, we maintain a vector of all servers' socket ids. When a client is asking for service, the binder will first find out the vector of servers' ID. Then we will pick the first ID we find in vector as the executing function ID; and then, put it to the back of vector (so, next time it will be the last one to handle this same situation).

Termination procedure

- A client sends a termination message to binder
- The binder will send the termination messages to all servers registered on it through the sockets.
- Servers will shut down after receive the message if there is no process on server side. **Otherwise, waiting the process done first; and then, shut down themselves.**

Error Code List

Binder:

rpcBinderInit():

1	BINDER WARNING: server re-register, the old one will be replaced by new one
0	BINDER SUCCESS
-1	BINDER ERROR: Get addr info error
-2	BINDER ERROR: Could not create socket
-3	BINDER ERROR: Listen socket bind failed
-4	BINDER ERROR: Get port error

rpcBinderListen():

0	BINDER SUCCESS
-1	BINDER ERROR: Select error
-2	BINDER ERROR: Accept new connection failed
-3	BINDER ERROR: No room left for new connection
-4	BINDER ERROR: Message length error
-5	BINDER ERROR: Message type length error
-6	BINDER ERROR: Message body length error
-7	BINDER ERROR: Received unknown message type
-8	BINDER ERROR: Send response type failed
-9	BINDER ERROR: Send response errorCode failed
-10	BINDER ERROR: REGISTER Wrong IP size (>16)
-11	BINDER ERROR: REGISTER Wrong port size (!= 4)
-12	BINDER ERROR: REGISTER Wrong message body
-13	BINDER ERROR: LOC Wrong message body
-14	BINDER ERROR: LOC Procedure not found
-15	BINDER ERROR: LOC Send IP+Port failed
-16	BINDER ERROR: LOC_CACHED Wrong message body

-17 BINDER ERROR: LOC_CACHED Procedure not found
-18 BINDER ERROR: LOC_CACHED Send Ip+Ports length error
-19 BINDER ERROR: LOC_CACHED Send Ip+Ports body failed
-20 BINDER ERROR: TERMINATE Send message to servers length error
-21 BINDER ERROR: TERMINATE Send message to servers type error
-22 BINDER ERROR: TERMINATE Send message to servers body error

Server:

rpcInit():

1 WARNING, procedure is overided.
0 SERVER SUCCESS
-31 SERVER ERROR: Get addr info for listen socket for clients
-32 SERVER ERROR: Create listen socket for clients failed
-33 SERVER ERROR: Bind listen socket for clients failed
-34 SERVER ERROR: Get listen socket for clients port number failed
-35 SERVER ERROR: Create socket to binder failed
-151 SERVER ERROR: Get BINDER_ADDRESS failed
-152 SERVER ERROR: Get BINDER_PORT failed
-36 SERVER ERROR: Connect socket to binder failed

rpcRegister():

-37 SERVER ERROR: Send registration message length error
-38 SERVER ERROR: Send registration message type error
-39 SERVER ERROR: Send registration message body error
-40 SERVER ERROR: Receive registration response type length error
-41 SERVER ERROR: Receive registration response errorCode error
-42 SERVER ERROR: Received wrong registration response type

rpcExecute():

-43 SERVER ERROR: Select error
-44 SERVER ERROR: Server accpet new client failed
-45 SERVER ERROR: No room left for new client
-46 SERVER ERROR: Send response type failed
-47 SERVER ERROR: Send errorCode failed
-48 SERVER ERROR: Received wrong message length
-49 SERVER ERROR: Received wrong message type length
-50 SERVER ERROR: Received wrong message type
-51 SERVER ERROR: Received wrong message body
-52 SERVER ERROR: EXECUTE Create new thread failed
-101 SERVER ERROR: EXECUTE arg type error
-53 SERVER ERROR: EXECUTE argTypes size error
-54 SERVER ERROR: EXECUTE Message body size error
-55 SERVER ERROR: EXECUTE Init args failed
-56 SERVER ERROR: EXECUTE Skeleton not found
-57 SERVER ERROR: EXECUTE Execute failed
-101 SERVER ERROR: EXECUTE arg type error

-58 SERVER ERROR: EXECUTE Send back execution message length failed
-59 SERVER ERROR: EXECUTE Send back execution result failed

Client:

-61 connection lost between binder and client
-62 wrong length of message from binder to client
-63 wrong response type from binder
-64 wrong response code from binder
-65 connection lost between server and client
-66 send binder message length failed from client
-67 send binder message type failed from client
-68 send message failed from client
-69 wrong response type from server to client
-70 terminate failed on client side
-71 binder host is null
-72 binder port is invalid
-73 wrong socket id on client side
-74 response error from server
-75 null ip address
-76 null argtypes
-77 null arguments
-78 wrong message received
-79 wrong argtype

Functionality Description

- The rpc functions can still work well in a large number of clients and multi-servers.

Advanced Functionality

- We implemented rpcCacheCall() in rpc library that can keep a cache of server information on client side database.
- The rpcCacheCall can do exactly same functionalities as rpcCall() in our system with less communications between client and binder.
- The rpcCacheCall() can do round-robin well when client sends a execute request to server.
- The functionality can be tested by registered several servers on binder and one client to call the rpcCacheCall(), and we can see the sequence of server execution.
- Once a server received a TERMINATE message and it still in execution of a heavy loading procedure, it will wait for this execution complete and then binder can terminate.

Client procedure main table
(cache call)

<name, type>	<IP, Port>
<f0, int (int, int)>	<129.60.2.15, 58763>
<f0, int (short, char)>	<29.87.2.49, 39754>

Client round robin list

vector<IP, Port>
S3 <128.20.65.49, 88723>
S1 <129.60.2.15, 58763>
S2 <29.87.2.49, 39754>
Sentinel

1

vector<IP, Port>
S1 <129.60.2.15, 58763>
S2 <29.87.2.49, 39754>
Sentinel
S3 <128.20.65.49, 88723>

2

vector<IP, Port>
S1 <129.60.2.15, 58763>
S2 <29.87.2.49, 39754>
S4 <128.20.65.49, 88723>
S5 <128.21.65.129, 76432>
Sentinel
S3 <128.20.65.49, 88723>

3

Server procedure main table

<name, type>	<skeleton>
<f0, int (int *, void**)>	int f0_Skel(int* a, void** b)
<f0, int (int *, void**)>	int f1_Skel(int* a, void** b)

In our client side database, we have a vector to store IP, Port pair. In this vector, there is a “Sentinel” in vector to separate the used pair and new pair. Since a server just handled a request from client. The server’s IP,Port pair will be push onto the bottom of vector. If there is a new server register the same function on binder, the new server’s IP,Port pair will be insert into the position just before “Sentinel” but after all other not executed servers’ pairs. In this case, the

rpcCacheCall() will perform an excellent round-robin selection for server to handle client RPC call.