System Manual

RPC System for CS454 Assignment 3

Winter 2015

Section 002

Developers:
Meng Yang 20402357
James Cui 20378921

Table of Contents

Design

RPC Methods

RPC methods in the provided interface are divided into two separate files. **rpc_server.cc** contains the subset of the given RPC methods called by the server, and **rpc_client.cc** contains the rest of the methods which are called by the client.

Server RPC methods (rpc_server.cc)	Client RPC methods (rpc_client.cc)

`rpcInit();`
`rpcRegister();`
`rpcExecute();`
	`rpcCall();`
	`rpcCacheCall();`
	`rpcTerminate();`

RPC Server

Aside from the implementation of server RPC methods, the main design features implemented on the server end are:

- **Skeleton database:** a local database used for storing skeleton records
    - A **skeleton record** has 2 parts: signature and skeleton. The signature contains the function name and the argument types associated with the function

```
typedef struct _SKEL_RECORD_ {
    char *fct_name;
    int *arg_types;
    skeleton skel;
} SKEL_RECORD;
```

- **Thread programming used to handle client requests**
    - The server needs to process each client execute request, and each execute request is a connection from client to the server. Therefore, threading is important to maintain great server performance when responding to these calls.
    - 1 new **thread** for each **client request**
        - Server waits for all client processing threads to finish before exiting

RPC Client

Aside from the implementation of client RPC methods, the main design features implemented on the client end are:

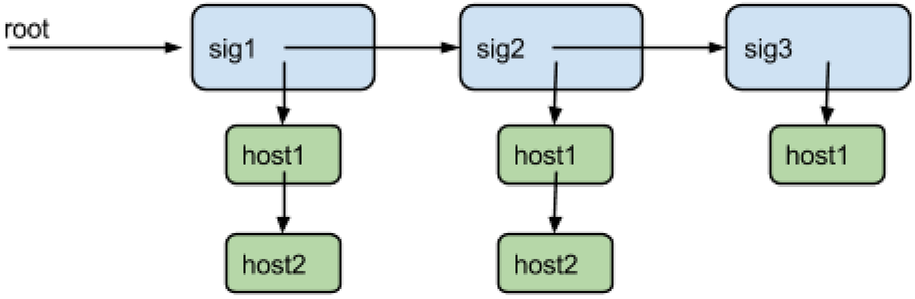
- **Client Cache Database:** needed by rpcCacheCall()
    - This database is a wrapper on the host database (see binder section)
    - Database operations are multi-thread-safe and is implemented using Singleton Design Pattern

Binder

Binder uses select() to multiplex among incoming connections. It is using a database called Host Database to save the pairs of <host,sig> registered by the servers. When server terminates (received a buffer of zero length from read()), it will remove all the host nodes of the server from the database.

- Host Database:
    - Structure: linked list of signatures, each contain a list of hosts

Figure 1



```
sig node {
    - fct name
    - arg types
    - next sig
    - root host
}
```

```
host node {
    - socket id
    - ip
    - port
    - next host
}
```

-
    - ensures that each hosts appears in a signature **only once**
    - ensures **round robin scheduling** (details below)
    - alternate design: our alternate design is to do a **reverse index** data structure, where root points to a link list of host nodes and each host node has a list of signatures, but we choose our design because it consumes less memory (because host nodes name are shorter than signature nodes)

Round Robin Scheduling

When the binder requests for a signature, host database will return the first host if any. Then, it will loop all signatures, and loop through all hosts, and if it finds the host, it will put the host at the end of the linked list. Take the example in figure 1: if a client request for signature 1, then the database will return host1, and will put host1 at the end, so the end result for signature 1 would be [sig1]-->[host2]-->[host1] and similarly for signature 2 [sig2]-->[host2]-->[host1] .

Protocol

Here below is the underlying protocol that we used to pass messages. It is a modified version of the recommended protocol.

Message Format

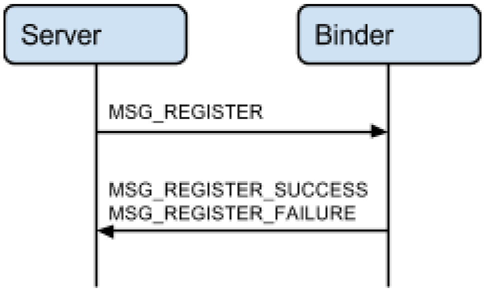
All messages used in the RPC system follows a standard general format, as such:

```
message buffer: [     4     ||    1    ||   ...  ]
                   msg_len     msg_type   message

( numbers: the amount of bytes; triple dots: a block of variable length )
where message is dependent on the msg_type:

MSG_LOC_REQUEST MSG_LOC_CACHE_REQUEST
message: [   4   ||   ...    ||   4     ||   ...    ]
            name      name      argTypes    argTypes
           length               length


MSG_REGISTER_SUCCESS MSG_REGISTER_FAILURE
MSG_LOC_CACHE_FAILURE MSG_LOC_FAILURE MSG_EXECUTE_FAILURE
message: [     4     ]
          reason_code

MSG_LOC_SUCCESS
message: [   4   ||  2   ]
            serv    serv
             ip      port

MSG_LOC_CACHE_SUCCESS
message: [    4    ||    ...    ||    ...     ]
           number       server       server
          of hosts       ips         ports

MSG_REGISTER
message: [   4   ||  2  ||   4   ||   ...    ||    4    ||   ...      ]
           server  server   name      name      argTypes    argTypes
             ip      port    length              length

MSG_EXECUTE MSG_EXECUTE_SUCCESS
message: [   4   ||  2  ||   4   ||   ...    ||    4    ||   ...    ||   ...   ]
           server  server   name      name      argTypes    argTypes      args
             ip      port    length              length

MSG_TERMINATE
message: [] (empty)
```

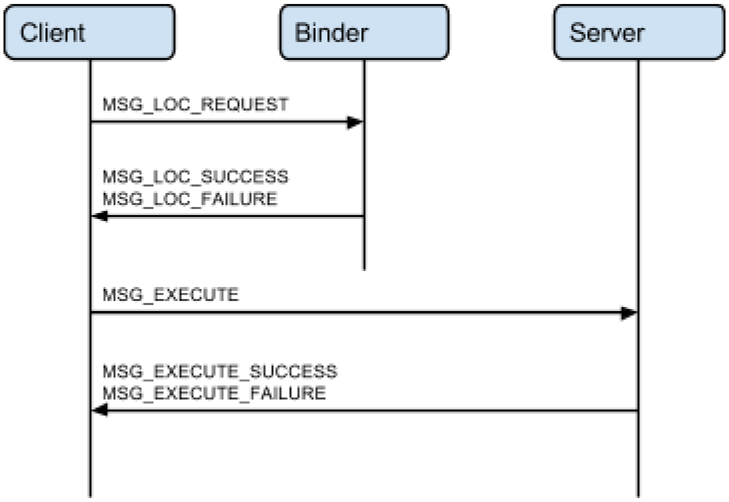
Communication

The diagrams below illustrate interaction between server/binder/client for the corresponding RPC methods.

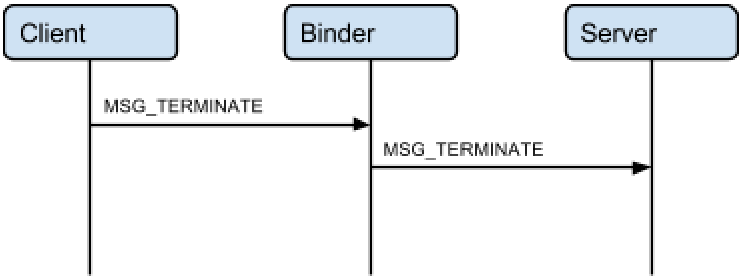
1. rpcRegister()



2. rpcCall() and rpcCacheCall()



3. rpcTerminate()



Error Codes

List below are all the error/return codes used in the RPC system

```
/*
 * Summary of RPC related error/return codes
 * - for a list of full error codes, please see defines.h
 */

// BINDER environment variables not set
#define RPC_ENVR_VARIABLES_NOT_SET              -100

// common parameter errors
#define RPC_NULL_PARAMETERS                     -200
#define RPC_INVALID_ARGTYPES                    -201

// if socket() failed/not called
#define RPC_SOCKET_UNINITIALIZED                -1

// server codes
#define RPC_SERVER_CREATE_SOCKET_SUCCESS         0
#define RPC_SERVER_CREATE_SOCKET_FAIL           -10
#define RPC_SERVER_BIND_SOCKET_FAIL             -11
#define RPC_SERVER_GET_SOCK_NAME_FAIL           -12

// rpcRegister() codes
#define RPC_REGISTER_SUCCESS                     0
#define RPC_REGISTER_OVERRIDE_PREVIOUS           1
#define RPC_REGISTER_SERVER_SETUP_ERROR         -13
#define RPC_REGISTER_INVALID_FCT_NAME           -14
#define RPC_REGISTER_INVALID_ARGTYPES           -15
#define RPC_REGISTER_UNKNOWN_ERROR              -16


// rpcCall() codes
#define RPC_CALL_SUCCESS                         0
#define RPC_CALL_NO_HOSTS                       -17
#define RPC_CALL_INTERNAL_DB_ERROR              -18

// RPC method success codes
#define RPC_INIT_SUCCESS                         0
#define RPC_CACHE_CALL_SUCCESS                   0
#define RPC_TERMINATE_SUCCESS                    0
#define RPC_EXECUTE_SUCCESS                      0

// select() failure
#define RPC_CONNECTION_SELECT_FAIL              -21

// R/W interaction with server or binder
#define RPC_CONNECT_TO_BINDER_FAIL              -19
#define RPC_CONNECT_TO_SERVER_FAIL              -20

#define RPC_WRITE_TO_BINDER_FAIL                -22
#define RPC_READ_FROM_BINDER_FAIL               -23

#define RPC_WRITE_TO_SERVER_FAIL                -24
#define RPC_READ_FROM_SERVER_FAIL               -25
```

Unimplemented Features

All features (including the bonus portion) required by the assignment have been implemented!

Testing Done

Major tests performed in the RPC system includes, but not limited to:

- ✓ Test with many clients and one server, the server was able to answer all the clients concurrently
- ✓ When three server registered the same function, the binder answers rpcCalls for the same function from client in a round robin fashion.
- ✓ Test with binder, clients, and servers all running on different hosts (tested 3 clients on 006,004, and 002), they were able to communicate with each other
- ✓ Test with client which has 10 threads doing rpcCalls, performance was great.
- ✓ Test rpcCacheCall with one client that has many threads (no deadlocks)
- ✓ Error checking in client calls / server register with invalid argTypes, appropriate error code returned from rpc methods
- ✓ Test the case where the server waits for all client processing threads to finish before exiting
- ✓ Test with valgrind to make sure there are no memory problems in our code