CS454 Assignment 3 Document

Alfred Chan (20392255)

December 2, 2017

Overview

This document is seperated into 6 major sections: the first halves talks about what has been done, and the second halves talks about how I handle `rpcCacheCall`. Since I attempted the bonus, the system does more work to maintain a certain degree of synchronization in every request, though these extra works do not change `rpcCall`'s behavior. In particular, each node, which can be the binder, servers and clients, contains a local name directory that maps a machine identifier (ipv4 address and port number) to unique integer ids generated by the binder.

1 Class Definitions

This section describes each major class in the system. In addition to classes, a file named "common.hpp" contains utility methods that is shared amongst most classes. These utility methods are for converting between C++ structures and raw data.

- **struct** `Name`: this simple structure contains two integers that represent an ipv4 address and a port number. Obviously, this structure is used as a key type for `std::map` in the name directory, so there is a operator overload of "<" that is based on `std::pair`'s "<" overload.
- **struct** `Function`: similarly to **struct** `Name`, this structure is also used as a key type in the name directory. This structure is composed of a function's name (in `std::string`), and a list of argument types

(in `std::vector<int>`). Again, the comparison operator is almost free, but with one catch: the equivalence of two function signitures disregard array cardinality (i.e. an array of size 1 is treated the same as an array of size 100), so to address this problem the comparison operator call `Function::to_signiture()` to create another copies of the signitures whose array size can only be 0 (scalar) or 1 (array).

Notice that overloading `operator<` for `Function` effectively solves function overloading for the RPC system, since equivalence of argument types is the same as item-by-item comparison of `std::vector<int>`, given that the argument types are inserted to the vector in the same order as in `argTypes`.

Of course, when the **server** calls a skeleton it needs to know the actual cardinality of the array types. This means the server needs to to store the original function signiture along with the modified version.

- **struct** `Postman::Message`: this structure is a quadruple that contains
  - the version number (`unsigned int`) of the name directory
  - the size of the message (`unsigned int`), which is used to allocate buffers
  - the message type from the **enum** `Postman::MessageType`
  - and finally the message contents (`std::string`)

  All send methods in `Postman` are basically message composers, and they uniformly pass the created message to the function `Postman::send(int, Message)`.

- **struct** `Postman::Request`: this structure is a pair of a file descriptor and a `Postman::Message`. The purpose of this structure is to be packed within a `std::queue`, so that the caller can poll request from `Postman`.

  In retrospect, storing the fd was an oversight, because my original intention was to buffer requests and allow the program to handle them at any time. This was true in my original codes, because connections last for the whole program execution. However, I later changed the duration of connections to be based on the scope that the connection

was established (thanks to C++'s RAII), so all requests must be handled while connections are still valid. The intention is to minimize the number of active connections for the binder.

- **class** `Sockets`: this class provide type-safe C++ bindings for the C-based network library. All methods are unsynchronized, because it is intended to be a private object of another class. Most of the codes in this class are inherited from assignment 2. I wanted this class to be a silver bullet for approaching Unix network sockets, but obviously it's flawed. In any case, this class is much more robust than the one I wrote for CS456, since I did it in C.

- **class** `Postman`: this major class has 2 functionalities:

  - to buffer raw messages from `Sockets` and to handle marshalling of these raw messages by turning them into `Postman::Message`, and then later into `Postman::Request`
  - to provide synchronized public access to `Sockets` for connections, to the request queue, and to create messages and feed them to the internal method `Postman::Send(int, Message)`.

  In other words, this class is the "workhorse" of network-related codes.

- **class** `NameService`: this class is responsible for maintaining various mappings that allows callers to resolve names and to get suggestions about available servers. The mappings include:

  - a bidirectional mapping of `Name`'s and `int` ids. The mapping is simply based on two separate maps: `std::map<Name,int>` and `std::map<int,Name>`
  - a mapping of `Function`'s to pairs of `std::set<int>` ids, and an integer pivot, where the pivot decides scheduling using round robin. Note that each function has its own pivot. Also, the pivots are **local** values that are **not** sychronized along with the name directory, so the binder can have different pivots than the server and the clients.

  In additional to name resolving, `NameService` has an internal log which can only affect by the binder. It uses a very simple version

of timestamp ordering, such that there is no conflict and no abort. Clients and servers update their own name directory through the logs that come from every reply (even not from the binder).

- **class** `Global`: this class is located in `rpc.cpp`, which takes advantages of RAII to serve initialized global variables for the C-functions in `rpc.h`. This class is shared between the client codes and the server codes. For the server, `Global` manages a mapping of `Function` (modified) to pairs of a actual `Function` object and a skeleton function.

- **class** `Task`: this structure has a simple `run()` method that allows servers to run a call request. All data are immutable copies of many struct/class described earliar.

- **class** `Tasks`: this is basically a synchronized queue of `Task`'s. Each server has one. It manages worker threads which basically wait for new `Task`'s by a semaphore.

  In addition, `Tasks` provides a `terminate` function, which changes the `is_terminate` flag and raise the semaphore by `MAX_THREADS`, allowing every threads to wake up and terminate by themselves; of course, `terminate()` blocks until it finished joining the threads.

- **class** `ScopedConnection`: this is a simple class that establishes a connection through `Postman`, and later when the object is out of scope, the destructor disconnects the connection through `Postman`.

- **class** `ScopedLock`: this is similar to `ScopedConnection`, except this class handles the locking/unlocking of a `pthread` mutex.

2 Protocol

As described earlier, every `Postman::Message` is of the form

```
nameservice_version msg_size msg_type msg_content
```

The following subsections describe `msg_type` and the contents within `msg_content`. All `msg_type`'s are defined within `Postman` in an enum called `MessageType`. Anything related to the name directory will be explained in a later section,

though in a nutshell, nameservice_version allow the receiver to determine which portion of the logs should be attached in a reply. **Notice that all replies except** CONFIRM_TERMINATE **has partial logs attached**, I will refer them as log_delta.

2.1 Request: ASK_NS_UPDATE

This request can happen for the servers when a new server joins. This is discussed in details in NEW_SERVER_EXECUTE. The message content is empty, because all message already has a local version number of name directory.

2.2 Reply: NS_UPDATE_SENT

The message content contains log_delta, which is used by servers/clients to update their local name directory.

2.3 Request: I_AM_SERVER

This request is sent within rpcInit(), and it is delivered to the binder. The message content is the port number that servers use to listening incoming messages. This request is needed because I decided to make connections last within a scope, and the binder needs to start connections to servers. In effect, this request register a server in the binder's name directory.

2.4 Reply: SERVER_OK

This is binder's reply to I_AM_SERVER. The message contains

```
server_id log_delta
```

The server_id is generated by the binder, which the server uses to set its global variable Global::server_id. Effectively, a server is a server if and only if server_id $\neq$ static_cast<unsigned int>(-1).

2.5 Request: Register

Only the server can send this message. This request register a **unique** Function to the binder. All subsequent register for methods that has the

same signiture, determined by `operator<`, will cause the server to update the skeleton locally – no message is sent. The message contents contain

```
server_id fn_name_len fn_type_len fn_name fn_types
```

where `fn_name` is the name of the function and `fn_types` are the types of function (as integers), `fn_name_len` and `fn_type_len` are lengths of `fn_name` and `fn_type`, respectively – I will refer to this quadruple as the binary representation of `Function`.

2.6 Request: `LOC_REQUEST`

This message can only be sent by a client, who wants to ask the binder for a server suggestion. The message contents contain just the binary representation of a `Function`.

2.7 Reply: `LOC_REQUEST_REPLY`

This is binder's reply to a client's location request. If the binder has a suggestion, then the message contents contain

```
true log_delta server_id
```

The client will update the name directory, as always for all replies, with `log_delta`, and then it resolves `server_id` using its local name directory – this must succeed since `log_delta` brings client to the same version of the name directory as the binder's. If the binder doesn't have a suggestion, then the message conents contain

```
false log_delta NO_SERVER_AVAILABLE
```

where `NO_SERVER_AVAILABLE` is an error number (arguably unecessary).

2.8 Request: `EXECUTE`

This is called by the client to the server for a task execution. The message contents contain

```
func args
```

where `func` is the binary representation of `Function`, and `args` contains 0 or more of arrays (scalars are treated as arrays of size 1) **that are input arguments**.

2.9 Reply: EXECUTE_REPLY

This is the server's execution reply to the client. The message contains

```
retval args
```

where `retval` is the integer return value of the RPC call. If the skeleton returns an error, `retval` will be `SKELETON_FAILURE` (see later sections). On the other hand, `args` contains the **output arguments**, which overwrite the corresponding items in the same `args` that the client used to send the execute request.

2.10 Request: TERMINATE

This request is sent by the client to the binder. When the binder gets this message, it send the a terminate request to the servers. Then the servers will ask for confirmation to the binder. The message contents contain an empty string.

2.11 Request/Reply: CONFIRM_TERMINATE

This message is a bit more interesting. It serves as both request and reply, both ways, from servers to the binder as a request, and then from the binder to servers as a reply. The message contains

```
is_terminate
```

Notice `is_terminate` is only used when servers get the binder's reply. Since the reply comes from the binder, if `is_terminate` is true, then the terminate request is geniune, and servers will proceed to termination. **However, the binder doesn't follow the specification about terminating after all servers have terminated; the binder terminates when it replies** `CONFIRM_TERMINATE` **to all servers.** I guess this is design decision due to `ScopedConnection`, because every connection must disconnect at the end of any RPC methods in `rpc.h`, if not earlier. I chose to do it this way because I would like to keep the number active connections low, so that the binder and servers can serve more requests. After all, there is an upper limit in the number of connections that a socket can listen to. The downside is that the binder no longer knows if a server is active anymore, unless the binder tries to connect to a potentially dead server.

2.12 Request/Broadcast: NEW_SERVER_EXECUTE

The message content is an empty string. The server sent this request to the binder when `rpcExecute()` runs. When the binder gets this request, the binder will broadcast this request to all servers. If a server is dead, the the binder would remove it from the binder's name directory. If servers are alive, then the servers would send a ASK_UPDATE_NS request to the binder. Therefore, the servers' name directory will become **mostly** (not completely) up-to-date.

3 System Flow

The flow of the system goes as follows.

3.1 Client

Ignoring `rpcCacheCall`, the system goes as follows.

- The client initialize `Global` variables.
- The client calls `rpcCall`.
- `rpcCall` asks the binder for a server suggestion.
- If the binder finds a good suggestion, then reply the client with the server's id. Otherwise, send a failure. In either cases, the binder attaches partial logs of the name directory based on client's version in the request header.
- If the client doesn't get a valid suggestion, then end `rpcCall`. Otherwise, the client send an execute request to the server.
- The server gets the request, run it, and then reply to the client. The server also send partial logs to the client, usually of size 0, because the client should have up-to-date name directory due to interactions with the binder.
- The client applies to logs, get the result, and then end `rpcCall`.
- It is possible that later the client calls for a terminate request to the binder.

3.2 Server

Generally, the flow goes as follows.

- The server initialize `Global` variables and call `rpcInit` separately.
- During `rpcInit`, the server sends a `I_AM_SERVER` request to the binder and wait for its id in a `SERVER_OK` reply.
- Then the server registers functions.
- Then the server run `rpcExecute`. At this point, the server sends a `NEW_SERVER_EXECUTE` to the binder, and the binder will broadcast this request to all servers, forcing them to send a `ASK_NS_UPDATE`, which updates all servers' name directory. Notice that it is possible for the binder to catch some dead servers, which causes the binder to remove them from its name directory, effectively incrementing the version. Thus, it is a possibility that **not every server will be synchronized to the same, latest version**. However, we will see in the last section that this is ok.
- Meanwhile, the server may get a terminate request from **anyone**. Thus, it will ask the binder for confirmation through a `CONFIRM_TERMINATE` request. If the binder agrees the confirmation, then the server will call `Tasks::terminate()`, which is a blocking call that **may not necessarily halt. This happens when a thread runs a long-running process or an infinite loop**. Once all tasks end, the server will exit gracefully.

3.3 The Binder

Unlikely clients/servers, requests can arrive to the binder in any order, but the binder handles them in a gigantic switch statement in a sequential manner. In particular, the binder handles the following requests:

- `I_AM_SERVER`
- `REGISTER`
- `LOC_REQUEST`

- NEW_SERVER_EXECUTE
- CONFIRM_TERMINATE
- ASK_NS_UPDATE

4 Error Numbers

The following errors/warnings are defined in an enum in `common.hpp`.

- EXECUTE_WITHOUT_REGISTER (2): this is a warning that the server runs `rpcExecute` without registering any functions. In case this happens, `rpcExecute` immediately terminates and return this warning.
- SKELETON_UPDATED (1): this warnings that the server re-registers a method, and only the skeleton is updated.
- OK (0): this means successful.
- BAD_FD (-1): this happens when `Sockets` couldn't create a socket.
- BINDER_UNAVAILABLE (-2): the binder is down.
- CANNOT_ACCEPT_CONNECTION (-3): this happens when `Sockets` cannot accept a connection.
- CANNOT_BIND_PORT (-4): this happens when `Sockets` cannot bind a port.
- CANNOT_CONNECT_TO_SERVER (-5): this happens when a client cannot connect to a server.
- CANNOT_LISTEN_PORT (-6): the binder/server cannot set a port to listen to incoming messages.
- CANNOT_RESOLVE_HOSTNAME (-7): the local name directory (binder/-clients/servers) cannot find the hostname.
- CANNOT_START_CONNECTION (-8): binder/clients/servers cannot start a connection.
- CANNOT_WRITE_TO_SOCKET (-9): cannot write to a socket.

- FUNCTION_ARGTYPES_ARE_INVALID (-10): `argTypes` is `NULL`
- FUNCTION_NAME_IS_INVALID (-11): functions whose name is null, or **not** $0 <$ length ≤ 63 (this excludes the null terminator). Notice valid names must have a length that is strictly greater than 0.
- FUNCTION_NOT_REGISTERED (-12): client tries to call a function that a server did not register.
- HAS_ALREADY_INIT_SERVER (-13): the server calls `rpcInit` more than once. This error number is returned by `rpcInit`.
- HAS_RUN_EXECUTE (-14): the server attempts to run `rpcExecute` more than once.
- NOTHING_TO_RECEIVE (-15): `select()` tells `Sockets` to fill up the read buffer, but there's nothing to fill.
- NOTHING_TO_SEND (-16): `select()` tells `Sockets` to clear the write buffer, but there's nothing to write.
- NOT_A_CLIENT (-17): a server (i.e. `rpcInit()` has run) tries to run client methods: `rpcCall`, `rpcCacheCall`, and `rpcTerminate`
- NOT_A_SERVER (-18): a client tries to run server methods: `rpcInit`, `rpcRegister`, and `rpcExecute`.
- NO_SERVER_AVAILABLE (-19): the name directory cannot find a server to complete requests.
- REMOTE_DISCONNECTED (-20): the remote machine disconnected before the calling machine gets a reply.
- SKELETON_FAILURE (-21): the skeleton function returns a negative value.
- SKELETON_IS_NULL (-22): the provided skeleton is a `NULL` pointer.
- SERVER_HAS_NO_AVAIL_THREADS (-23): the server rejects a request because it doesn't have any free worker threads. This only happens when you call `rpcCacheCall`, because it defies round-robin.

- `TERMINATING` (-24): this represents the server is terminating – this is probably not a "public-facing" errno.
- `UNREACHABLE` (-100): unreachable codes reached; in other words, gg.

5 Name Directory And `rpcCacheCall`

To (mostly) synchronize name directories, each machine contains a local in-memory log, versioned by timestamp ordering. The log is represented by a `std::vector(LogEntry)`. Thus, the version of the name directory is simply the size of the vector. `LogEntry` is one of the following:

- `NEW_NODE id ip_addr listen_port`

  where `id` is an unique incremental id generated by the binder, `ip_addr` is the ipv4 address of a server machine, and `listen_port` is the port number that the machine uses to listen incoming messages.

- `KILL_NODE id`

  This entry removes entries that are related to the server machine of `id`.

- `NEW_FUNC id func`

  This entry associates a function definition to the server machine of `id`.

Notice that a server always has up-to-date information about itself, because each request that affects the binder's name directory has a reply with the changes attached. Also, when a server runs `rpcExecute`, it forces the binder to broadcast `NEW_SERVER_EXECUTE`, which causes all servers to ask the binder for the latest updates.

5.1 The Simple `rpcCacheCall`

The algorithm is the same as the one described by the specification. The algorithm goes like this

```
1 let S be an empty set of integer ids
2 while local name directory has suggestions
3   // round-robin, same code as the binder
4   let i be the suggestion
5
```

```
6    if i is in S then break // reaching a cycle
7    otherwise let S = S union {i}
8
9    let server be resolve(i) from the name directory
10
11   start a scoped connection to server
12     if server is connected then
13       send EXECUTE request
14       return if got either OK or SKELETON_FAILURE
15     end if
16   end connection // happens automatically by RAII
17
18 end while loop
19
20 // local cache failed, so use the binder as fallback
21 run rpcCall and return whatever from it
```

Notice that the a set is used to detect cycles. That's right, it is possible that there alive servers but they reject the execute request due to having no available worker threads (see optimization, the last section). To keen readers, you may argue that returning the pivot is more efficient, which I just realize at the time writing. That's true, since the pivot wraps around the container, so comparing i to the pivot is an easy way to detect a cycle. But, but, but, `NameService::suggest()` is amortized $\Theta(n)$ (number of candadates) anyways, to due `std::advance`, and running an extra $\Theta(\log n)$ for `std::set::find` and `std::set:insert` won't hurt anyways, right? In fact, had I store the iterator instead of a pivot the cost of `suggest` may become amortized $O(1)$. Well, I am just lazy to the change the codes.
Ok, I admit I am just high, since I haven't used the listing environment since taking my last algorithm course.

5.2 Cases When Name Directory Is Not Up-to-date

Recall my design decision that all remote connections are scoped. This means the binder doesn't know immediately when a server dies (for whatever unnatural reasons). Thus, `KILL_NODE` entries are added lazily to the logs when the binder realizes a server is down. This means some entries actually can be zombies. The binder knows whether a server live in the following circumstances:

- The binder broadcast `NEW_SERVER_EXECUTE` to every server.
- The client ask for a `LOC_REQUEST`, which causes the binder to look for a suggestion. To make a suggestion, the binder needs to probe candadate servers by openning connections.
- The binder broadcast `TERMINATE` to every server. However, there is no point to update the name server because the system is terminating.

Though, zombie entries are not problematic, because they only cause `rpcCacheCall` to probe more servers, but not the binder. If a server tries to register a `Name` that is already a zombie (with the same ip address and port), the binder assigns a new id and replaces all old entries. For clients, there are 2 cases in a successful probe:

- All records related to the server is still up-to-date (i.e. not zombies). In this case, the client proceed with the call.
- Records related to the server are no longer up-to-date (i.e. zombies). This **only** happens when a server is dead and it reboots, such that the listen port is the same as before. There are 2 subcases:
  - The server re-registers functions before the client tries to call them. In this case, the server will accept the request and will run the call, which is fine because the specification guarantees same functionality for each unique method across servers.
  - The server is slow and the client catch on before registration is complete. In this case, the server reply with the error `FUNCTION_NOT_REGISTERED`. The client moves on to the next candadate.

In any case, I think it is reasonable to assume that servers stay up with little down time, which makes the logs grow very little in rare occassion. So the size of logs in each message is amortized $O(1)$ – mostly consist of an integer (zero) that represents the number of log entries and nothing follow after.

6 Optimization

In addition to the scoped connections, I did simple load balancing in `rpcCacheCall`. If a client tries to probe servers that does not have free worker threads, the

servers reject the call. The client needs to move on until either 1) it finds a server that has free worker threads, or 2) all candadates have been exhausted. The latter case is the same case when all servers are perceived to be dead, so `rpcCall` will be used as fallback. Notice `rpcCall` always forces the server to add a task regardless of having free worker threads. In a sense, this optimization may potentially increase the number connections, but it is a fair tradeoff to reduce the convoy effect as much as possible.