CPSC 416 Final Report

ChainChat

Zhihao (Bruce) Li : e7o8
Michelle Chen: r6c8
Chen Hao (John) Hsu: v4l8
Jun Goh: f8f9

CPSC 416 Final Report

~*ChainChat*~

A RPC Chains Library applied to a distributed chat app

Authors:

Zhihao (Bruce) Li : e7o8
Michelle Chen: r6c8
Chen Hao (John) Hsu: v4l8
Jun Goh: f8f9

Abstract:

In our project we have implemented RPC Chain (RPCC) library that can be integrated into any client and server instances. Our goal is to show that RPCC reduces latency on RPC by reducing extra client-server calls within a chain. We used the original RPC library and hacked it to implement RPCC. We then built a Chat Application to demo the RPCC library. The Chat Application features multiple servers around the world that fetch messages from one another and return the compiled messages from the different servers back to the user. Shiviz was used to visualize the RPCC process and the performance of the RPCC was analyzed and compared to the traditional RPC call.

Motivation:

Regular RPC can take a large amount of bandwidth resulting in high latency for clients accessing geographically distributed services. RPCC tries to improve the performance by reducing the number of calls such that it can traverse from server to server without needing to return to the client. This is especially useful when servers are located far from the client and a single call needs to access multiple servers. For example, an email application may need to fetch from a Contacts, Calendars, and Email server. In our application, we have several chat message servers and we call RPCC to retrieve the relevant conversation back to each server, that is outputted back to the end user.

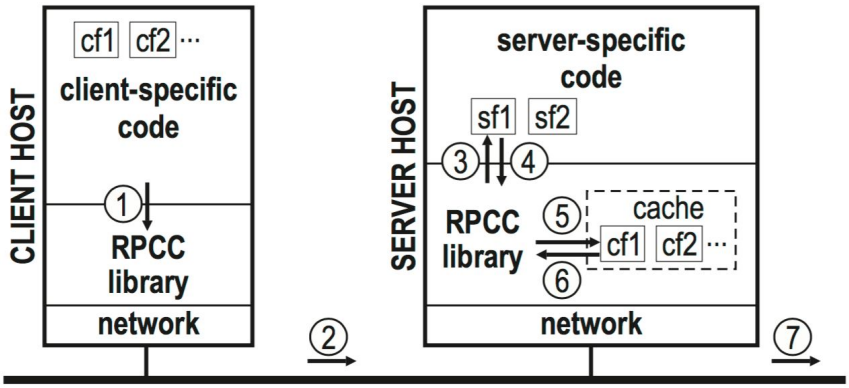
Introduction:

Our RPCC library builds on top of the existing Go RPC framework. We use a series of client-provided chaining functions to determine a path between our servers. This eliminates the need to have the server return to the client after every call, which reduces latency if our servers are geographically distributed. Some key features we have include the use of a remote git repository to store our chaining functions. For error handling, we use a heartbeat feature to detect broken chains as well as handle a number of common exceptions. Finally there is a debugging mode which forces client to execute chaining function and simulates legacy RPC. This report will go through the design of the system and our key features. This is then followed by an explanation of our implementation and a discussion of how we evaluated our system in terms of testing and performance. The report concludes with a look at some of the limitations we faced, a reflection on the development experience, a brief summary of our allocation of work and the references used.

Design:

Our RPCC library design based off of the Microsoft RPCC Research Paper. Each server will have a series of service functions which are similar to regular remote procedure calls. The main difference is the addition of the chaining function that dictates the sequence of service functions to be called. As the chaining takes place, the call will go from server to server without having to return to the client.

The execution sequence of our RPC Chain design will be similar to that described in Figure 2 (see below), taken from the Microsoft Research paper. The chaining function is stored on a remote repository, allowing each server to pull the chaining function logic from this repository instead of having it sent alongside the call. The process starts by calling our RPC Chain library with the initial service function (sf1) and the initial chaining function (cf1) to call. This will execute sf1, followed by cf1 which will tell the RPCC library what the next service function to call is (sf2) and the next chaining function (cf2). This will continue until we reach the end of the chain and the result returns back to the client.



As a part of the design, a heartbeat pulse is passed along with the RPC Chain sent from the server to the client at a fixed interval to indicate that the chain is still alive. The client will listen for the heartbeats through a specific port and if a heartbeat has not been received within a specified timeout period, the client will retry the initial call to restart the same chain once again. If the chain times out once more, the client will make the call with debug mode to detect which server the chain broke at and an exception will be thrown.

A debug mode has also been implemented so that the clients can execute each step of the chaining logic, and each of the servers returns results and logs back to client. This allowed for performance analysis by simulating regular RPC calls and troubleshooting any errors or bugs.

In our design we have accounted for three exceptions: (1) The next server in the RPC chain is down, (2) The repository for the chaining function is down, and (3) The chaining function doesn't have enough information to execute. When any of these errors have occurred, an error struct is created and returned directly to the client to notify them of what has occurred.

To demonstrate the RPC chaining functionalities, a chat application is implemented. In this chat application, there are multiple servers that have messages for the same chat and each server needs to retrieve messages from other servers, compile it, and send the results back to their respective users. The chaining function will allow for the server to retrieve messages in a chain from others, rather than sending multiple RPC calls out to the servers.

The provided API format has the almost all the same interface as RPC. The original functions that register a service, create a service function and others are still included. ServeConn is also included, caveat being the argument is of struct ArgToSF and return type is ReturnValSF. The field in them are both are string based so a struct can be marshalled into binary data and stored as a jsonString to be passed back and forth, with the structs essentially acting as wrappers. There are debug functions that are used when the debug mode is toggled on, and an InitialCall that client uses to dial to initialize the chain, are the additions to the original

RPC interface. To use rpcc.InitialCall, the client must fill in the ChainingFunctionInfo (CFInfo) struct and provide a debugging port if the debugging mode is on.

What we have built has not deviated from our original proposal, with the exception of using GitHub instead of BitBucket to store chaining function in remote repository.

Implementation:

We have taken the existing client and server from RPC library of Go and customized and hacked it to create our RPCC library. The system is built to be cross-platform, and the RPCC design can be integrated with any go client or server implementation. The RPCC library consists of a few main structs that are used in the chaining function: ChainingFunctionInfo and CFReturnVal. The ChainingFunctionInfo struct contains information on the Git repo to retrieve from, the Chaining Function File name, a Chaining Function name (if desired), the debugging port and the client IP address. The CFReturnVal struct contains the ChainingFunctionInfo as well as the service function information, arguments and client information. This allows all the required information to be passed between servers in the chain. We use the package “os/exec” to allow us to copy and update the Git Repo and run the chaining functions.

Evaluation:

As per our evaluation we used we have manually tested the system to ensure that we have all possible exceptions counted for. The tests we have ran include:

service function failed - create a faulty service function that exits with an error

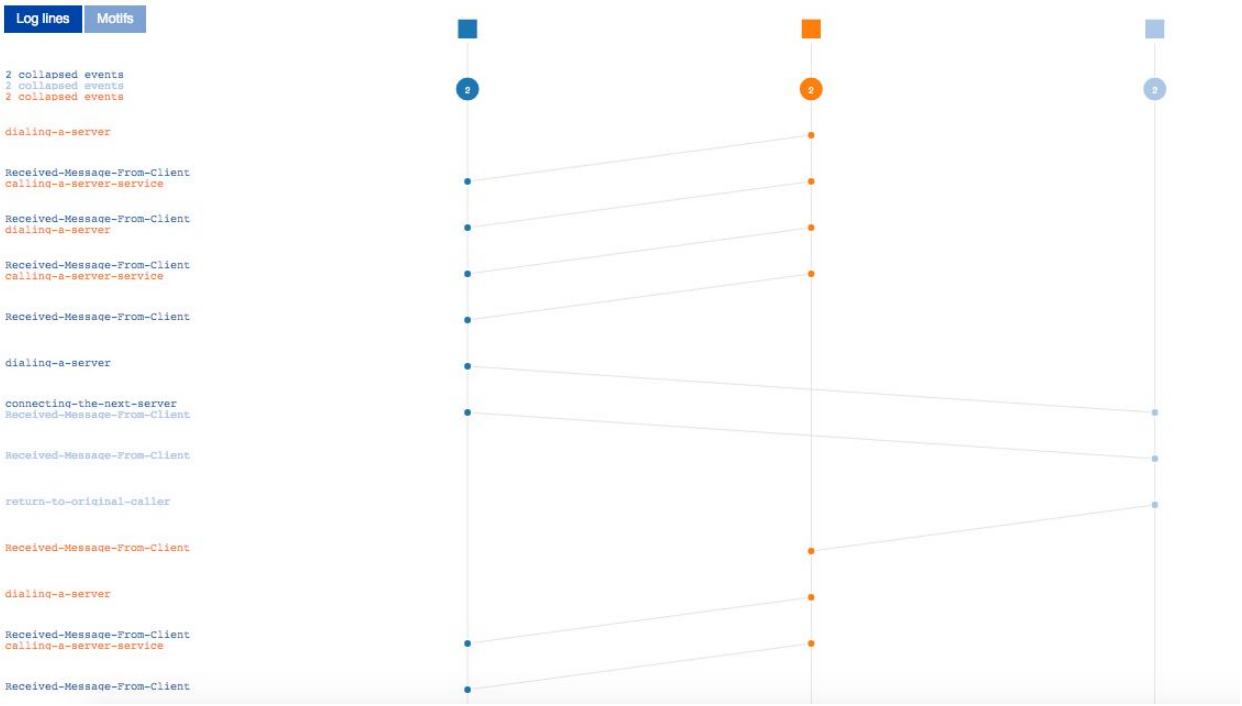
invalid chaining function repository - provided an invalid repository address

failed server in chain - client made a call with one of the servers not started

failed client - client makes call and is stopped

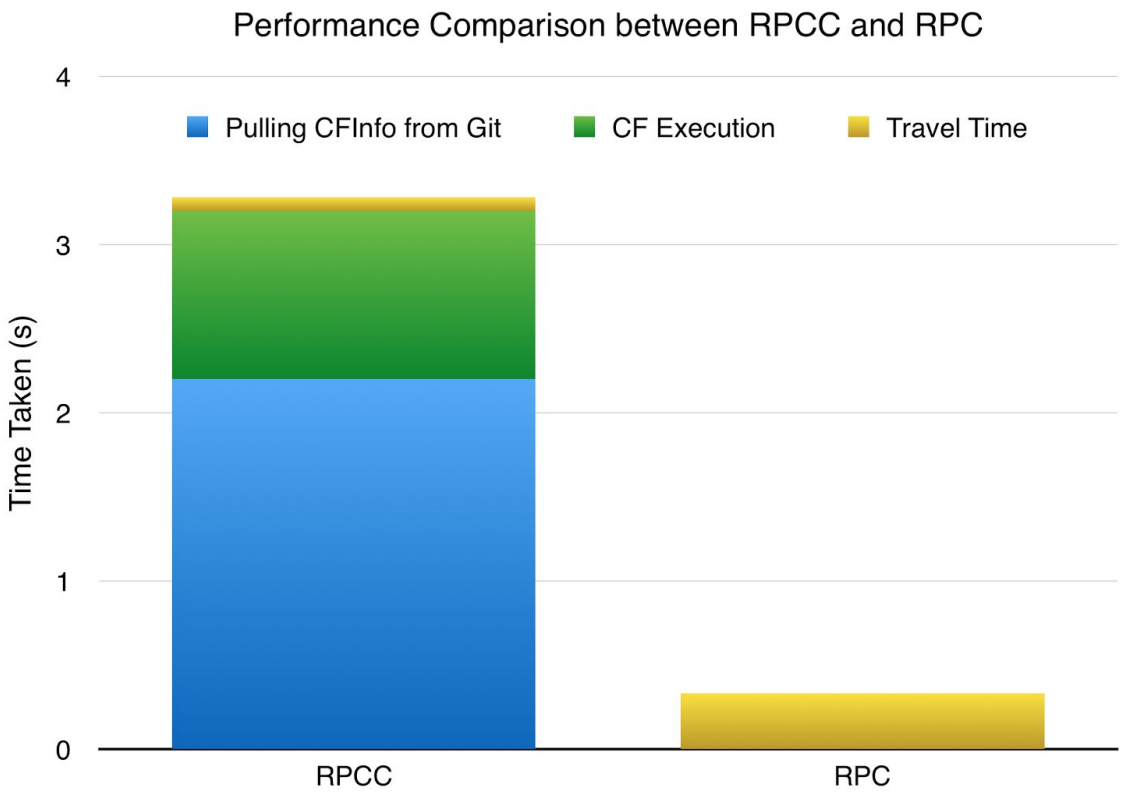
The chat application that is built on top of the RPC Chain was also manually tested to ensure the system is running as expected by checking that all messages were fetched properly and in order.

To ensure that the RPC chain performs as expected, we used Shiviz to visualize the processes taking place throughout a call to the chain (see below)



Here we have a simulation of RPC Chains with 1 client and 2 servers. The left-most one is server 1, the middle line is the client, and the right-most one is server 2. The application starts by calling server 1. Server 1 then executes the chaining function and directly calls server 2 (without needing to return to the client). Server two executes its chaining function which tells it to return back to the client.

We have attempted to analyze the performance of the RPCC we have built to show that it in fact reduces latency on RPC by reducing extra client-server calls within a chain, however after testing, it seems that the RPCC system that we've built takes longer than expected. We tested by using Amazon EC2 instances spun up in three geographically distributed instances (Oregon, Tokyo and Seoul) and recorded overall time it takes to complete a chain and also ran the same test with the regular rpc call and compared the times to see if the latency of the call was reduced by using RPCC. You can observe the performance comparisons on the chart below:



Most of the overhead is caused by Git fetching/pulling, and chaining function executing via os/exec. Git fetching/pulling took approximately 1.1s, while executing the chaining function took us around 500ms. Since we executed these across two hops, we accounted for an average of 3.2 seconds overhead from these two actions.

Limitations:

The RPCC we have implemented is limited to single sequential chains. We have chosen not to build subchaining due to the scope of the project and the time restraint. During the development, we have had issues with escaping characters in json strings through marshalling and unmarshalling and passing escaped characters into the Chaining Function properly. Certain characters such as backslash and quotations will cause issues. The limitation with the application is that it is not fully tolerant/robust because the focus was on the library, and during testing using EC2 instances, a few features such as heartbeat and debugging mode doesn't work as expected due to the way EC2 handles public and private IPs. Our library makes certain assumptions when we initialise heartbeat, debugging and GoVector ports , mainly that the IP address that the client listens on is accessible publicly. However, the EC2 instances in each

region could only listen on a private IP (that is mapped to a public IP) - hence debugging and heartbeat would fail since the library would attempt to reach the private IP and not the public one.

Discussion:

At the start of the project, figuring out how to hack RPC and what each method's purpose was and how it incorporated into what we wanted to do was challenging (ex. What codecs were). We originally began by individually adding what we thought were required methods of the RPC library into our own RPCC library, but then later realized we needed a lot of the pre-existing functions of RPC, so instead we restarted by using the whole client and server code of RPC and customized/added functions as we progressed. One of the roadblocks we kept encountering were marshalling and unmarshalling errors, as mentioned in our limitations sections. This took a lot of time troubleshooting and figuring out workarounds. Another issue that arose was figuring out EC2 and the issues caused by our assumptions of public and private IPs. Otherwise, overall the project went relatively smoothly.

Allocation of Work:

- Bruce: Debugging Mode, Heartbeat
- Michelle: Debugging Mode, Heartbeat
- John: Application Design, EC2 Testing,
- Jun: GoVector Integration,
- Pair Programming: Core RPCC Library Design

References:

- Go Lang Documentation: https://golang.org/doc/
- Go Lang RPC Client: https://golang.org/src/net/rpc/client.go
- Go Lang RPC Server: https://golang.org/src/net/rpc/server.go
- Microsoft RPCC Research Paper: http://research.microsoft.com/pubs/78640/rpcchains-nsdi2009.pdf