Members: Zhihao (Bruce) Li, Chen Hao Hsu, Cunn Yong (Jun) Goh, Michelle Chen

CPSC 416 Project Proposal

Table of Contents:

1. Introduction

RPC is a standard method to allow clients to call on functions and obtain results from remote servers. One of the issues with RPC is that it may lead to large bandwidth costs and high latency if the client is accessing services that are geographically distributed. A relatively recent paper[1] from Microsoft introduces the concept of RPC chains. Instead of having a client call on a server, have the result returned, call on the next one and so forth, RPC chains allows these RPC calls to be chained together, and have the final result be returned to the original client.
A comparison of RPC chains and standard RPC is seen below.

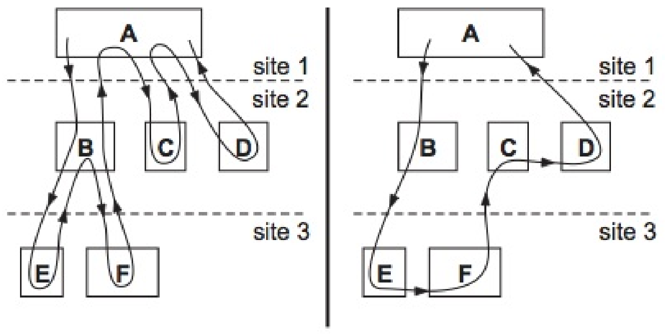

Figure 1: **(Left)** Standard RPCs. **(Right)** RPC chain. [2]

[1] http://research.microsoft.com/pubs/78640/rpcchains-nsdi2009.pdf
[2] http://research.microsoft.com/pubs/78640/rpcchains-nsdi2009.pdf

2. Project Problem

We want to measure the reduction in latency by comparing services using RPC chains versus ones using multiple RPC calls. We will distribute at least two services across different geographic locations, using either Amazon Web Services or Microsoft Azure, and have a client accessing these services using regular RPC calls and our implementation of RPC chains. By measuring the difference in execution times from both methods we hope to quantify the success of our implementation.

3. Proposed Solution Design

3.1 Chaining Design

Our design of RPC Chains will be based on the design described by the Microsoft 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, taken from the Microsoft Research paper. 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.

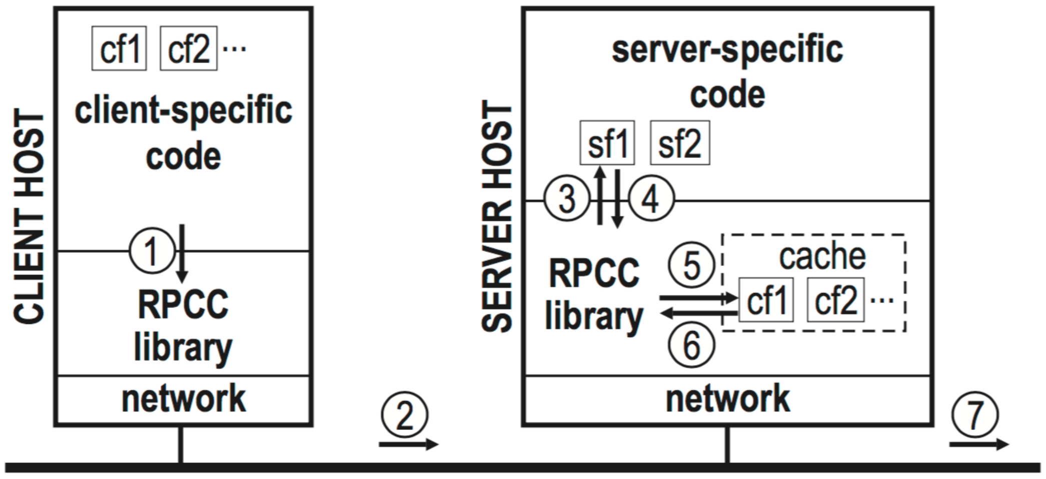

Figure 2: RPC Chain Execution[3]

[3] http://research.microsoft.com/pubs/78640/rpcchains-nsdi2009.pdf

3.2 Remote Code Repository

This feature allows the server to pull the chaining function logic from a remote repository instead of having the client sending it alongside the initial call. The server will then run the chaining function as necessary in order to decide what the chain should do next. We will set up a private BitBucket repository that will store of all our chaining functions. This will simplify the process of passing chaining functions and will allow support for more complex chaining functions. For the scope of this project, we will not support server-side caching of chaining functions pulled from the code repository.

3.3 Debugging Mode

There will be an option to toggle on and off the debug mode. In the debug mode, the servers in the chain will each log their progress and send the result back to the client after each service function call. The client uses this result and executes the next step of the chaining logic. This will make it easier to find bugs during development and also make it possible to simulate the effect of using a series of regular RPC calls. The latter will allow us to make performance comparisons between RPC and RPC Chains.

3.4 Performance Analysis

As stated in Section 3.3, the design will include a debug mode feature that will allow the application to switch between uninterrupted RPC chain and standard RPC. The standard RPC will return the result to the client after each service function. We will measure and compare the latency of a call that uses RPC with the same call using RPC Chains. We do acknowledge that certain feature omissions such as chaining function caching may reduce the performance of RPC Chains.

3.5 Error Detection

The chaining function sends a heartbeat pulse to the client at a fixed time interval from server which is working on the RPC chain request at that moment to indicate that the RPC chain is still alive. The heartbeat pulse would be sent via a TCP connection between each server and the client. The client will have a timeout that is double the heartbeat time interval. When the timeout is reached with no heartbeat response from the chain function, the client will have detected an error.

3.6 Error Recovery

The design covers error recovery for when a server in the RPC chain is down. When this error is detected, the client will retry its request again and the same request will once again go through the chain. If the server that caused the previous chain to break is still down, an exception will be thrown. The handling of this exception is captured in 3.7 Exception Handling.

3.7 Exception Handling

We implement the exception cases mentioned in the paper which can happen in the following 3 situations:

(1) The next server in the RPC chain is down
(2) The repository for the chaining function is down
(3) The chaining function doesn't have enough information to execute

The client receives the exception name and parameters. Since the exception doesn't contain the result produced so far, the client would need to resend the request to the same RPC chain service with the debug mode on to help identify the source of the error.

3.8 Out of Scope Features

To keep this project in the appropriate scope for CPSC 416, we have chosen to leave out some of the more advanced features that may be found in a robust RPC Chain implementation. Firstly, our implementation will only allow a sequential RPC Chain and we will not support parallel RPC Chain execution. We also chose to not implement subchains. We won't be modularizing the library to the extent of allowing it to gracefully fall back to standard RPC and handling legacy RPC services. Finally, we won't include any of the extensions described in Section 7 of the Microsoft Research Paper such as: intermediate chain results, handling large chaining states, and chaining proxy.

3.9 Data Visualization

We will utilize Go-Vector in our implementation to support logging of the distributed aspects of our system. These logs will then be inputted into Shi-viz to visualize the distributed logic of RPC Chains.

3.10 RPC Chain Application

We will apply our RPC Chain library onto a sample distributed chat application. We will have several servers in different geographic regions that will generate chat messages. The RPC

Chain will have to go through these servers to retrieve the entire conversation and send it back to the client.

4. SWOT Analysis

Strengths	**Weaknesses**
● A subset of us have worked together with a subset of us ● All have worked with Go ● Some of us has used EC2 instances ● All of us have experience with Git ● Project is based on a well written RPC Chains paper	● Different schedules ● Not everyone has worked with each member ● We don't understand the paper that well ● We don't have experience with complex Go projects or building of libraries ● We haven't worked with the translation API before ● We don't have experience GoVector library and ShiViz
Opportunities ● RPC package exists for reference ● Good translation or dictionary libraries	**Threats** ● Not enough time. ● Other courses, projects and exams

5. Timeline

Milestone 1: Project Setup (1 week)
Milestone 1.1: Set up EC2 or Azure instances and get Golang Running (John)
Milestone 1.2: Have the instance be able to pull code from Git and run sample code mimicking chaining function logic (Section 3.2, everyone)
Milestone 1.3: Project Design and Architecture: design of the library (configurables and exposed methods) (everyone)
Milestone 1.4: Learning GoVector and Shiviz (Section 3.9, everyone)

Milestone 2: Complete basic sequential chaining (2 weeks)
Milestone 2.1: Implementation of Client (Section 3.1) (everyone)
Milestone 2.2: Implementation of Chaining Servers (Section 3.1) (everyone)

Milestone 3: Metrics and Testing (1 week)
Milestone 3.1: Add support for debugging mode (Section 3.3) (Bruce, Michelle)
Milestone 3.2: Compare performance between RPCC and RPC using debugging mode (Section 3.4) (Jun, John)

Milestone 4: Error Handling (1 week)
Milestone 4.1: Add Heartbeat Error Detection (Section 3.5) (Bruce, Michelle)
Milestone 4.2: Handle exceptions and recover from errors in the chain (Section 3.6, 3.7) (Jun, John)

Milestone 5: Project Completion (1 week)
Milestone 5.1: Work on application that leverages our implementation (3.10) (everyone)
Milestone 5.2: Complete Final Report (everyone)

Date	**Deliverable**
Feb 29	Project Proposal Due
Feb 29 - Mar 7	Milestone 1
Mar 7 - Mar 21	Milestone 2
Mar 18	Meeting with TA regarding project status
Mar 21 - Mar 28	Milestone 3
Mar 28 - Apr 4	Milestone 4
Apr 4 - Apr 11	Milestone 5
Apr 11	Project + Final Report Due