CPSC416 Project Proposal - RPC Chains

Hanson Chun, Tony Shen, Neda Tofighi, Farzad Daei

February 29, 2016

Introduction:

Remote Procedure Call (RPC) allows one program to request a service from programs in other computer networks without knowing the detail of the foreign networks. The general RPC structure requires each remote call result to return to the caller before another remote call can be made. To avoid this back and forth over multiple sites which introduces latency, we propose a RPC chain so computation can proceed to the next server without returning to the client/first server for each call. The RPC chain, or RPCC, permits data flow from server to server without returning to the client in the process and reduce the negative performance effects of wide-area links applications that span multiple sites. Thus, RPC chains can significantly reduce end-to-end latency and network bandwidth in an application. We will apply the features of RPCC to our distributed storage system to demonstrate its performance benefits.

To use RPCC a client will need to provide a list of server hops it plans to make for a particular request its making so that each of the server's ip's are known. For communication between servers, a structure (ex. rpcChain in Fig 1.) will be passed through them with all the necessary info, explained in Parameters and State section. The chain order is dictated by the array field in the chain struct. This array can be updated by each server once it is their turn in the chain, as long as the addition or removal of a server hop does not conflict with our required information, further explained in Server modularity and composability section. To create the actual rpc calls, there will be an asynchronous call within one service function to the next desired service function, and this will be done before the return statement. Since it is asynchronous there is no need to wait for all the chain calls to be finished for the replies to make its way back to the first caller. Instead, once the end of the chain is reached, the last server dials directly to the first caller and sends it the final result.

The protocol will be implemented as a Go library which extends the existing Go RPC library and builds wrappers around its functionality. It will mostly call the parents functions in the RPC library but automate the RPCC functionality while still providing customizability for the application.

Figure 1:

```
type entryFunction func(interface{})interface{}

type serverEntity struct {
        connection_info string
        entry entryFunction
}
```

```
type rpcChain struct {
        id int
        currrentPosition int
        args interface{}
        reply interface{}
        chain serverEntity[]
}

func (client *Client) Call(chain rpcChain,
timeout int) error

func Callback(chain rpcChain)
```

Specific RPCC features:

Server modularity and composability (sub chains):

RPC chains allow the servers to be developed independently of each other. A server does not need to know about the other servers internal structures and functionalities. However, the servers do need to which server to dial next. So dependency between servers is only the RPC invocation and is relatively low.

The article RPC Chains: Efficient Client-Server Communication in geo-distributed Systems (Reference at bottom), states that chaining functions provided by clients can be stored in a repository in source code format and a reference is sent to the server which will compile the codes at runtime in order to save bandwidth. To scope down the difficulty level for cpsc 416, we decided to omit the chaining function repository. Instead, individual servers should implement the chaining logic and dial to the next server in line based on arguments (e.g. the rpcChain struct in figure1) passed in. In order to implement this, the servers must know about the service and function entry function (refer to figure 1) that a potential chaining server provides. Consider the following example where an RPC call can traverse two paths:

1) A -> B -> C
2) A -> B -> D -> C

In this case, server B has two potential chaining servers, C and D. Server B must know about the addresses, services and RPC entry functions of C and D. This logic is implemented in the respective servers. A server will only know the RPC entry functions of potential chaining servers, so in this case, server A will not know about C nor D.

Using our system as an example, server A represents the front-end in which the clients communicate with, server B represents the metadata store, server C stores the files, and server D handles file authentication. In the case where the client decides to store a file without credentials validation, the chain will only call servers B and C. Server D will not be called in this case. However, if the client decides to include file authentication, the chain will first traverse through authentication server D before storing the file in server C. So RPCC composability allows D to dial directly to C instead of returning back to B then B dials to C. Performance is enhanced using chain composability.

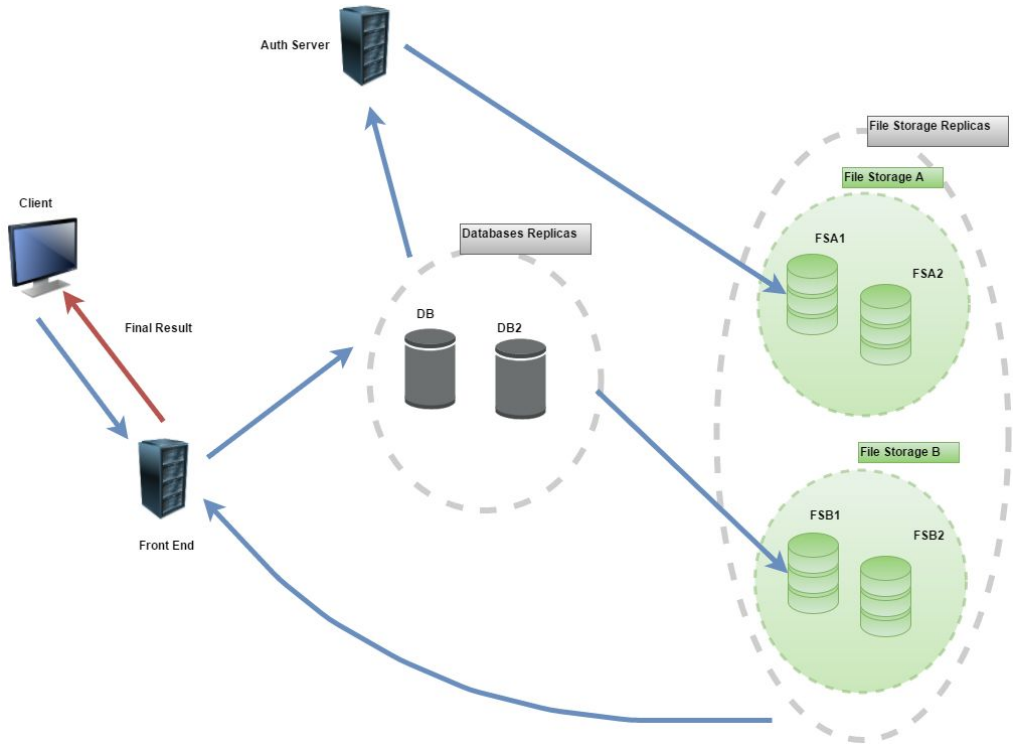
Chain dynamicity and broken chains:
The servers that are called by a client can vary dynamically during execution. This is most relevant to provide error checking in the case of a broken chain. There exists two ways a chain can be broken and cause the procedure call to fail:
1) In the case that a server X can not contact the next server, it will dial back to the very first server to notify failure. At this point, the first server will analyze the results it has received. Since the last contacted server X is not the final server in the chain, it will handle the recovery through complete retransmission, or it will ignore it.
2) A chain can also fail during computation of a request if the server crashes midway. This will be detected by a timeout (whose duration is set by the application based on expected time for return) running on the first server. Upon triggering the timeout, the procedure call is assumed dead. At this point, the application will decide if it wants to retransmit from the beginning of the chain or ignore the request.

Parameters and State:
The chaining structure may depend on run-time variables and other server states as it gets updated along the chain. Our design allows chains to have a header with some necessary information and optional parameters and output. Each chain has a unique id for identification since multiple chains can have the same parameters and headers, and the first server must be able to identify the results. The state of the chain is maintained by a variable in the chain header which keeps track of the current server being executed in the chain. Furthermore, each server in the chain can produce results and can pass arguments to the next server in the chain, in which the last server in the chain must provide the final results to the first server that created the chain.

References

RPC Chains: Efficient Client-Server Communication in Geodistributed Systems
http://research.microsoft.com/pubs/78640/rpcchains-nsdi2009.pdf

System Design Diagram



System Description

Our system has three unique types of servers:

1) Authentication servers containing file credentials
2) Metadata servers containing file storage location information
3) Two clusters of file storage servers containing files with and without credentials respectively

Main RPCC dial path components:

1) Client-> Front-End-> Metadata Server -> Authentication Server -> File Storage A
2) Client-> Front-End-> Metadata Server -> File Storage B

The distributed storage system demonstrates good performance, consistency and fault tolerance. RPCC is used to enhance performance, server replicas enables fault tolerance and consistency is managed by front end logic. We assume single point failures have high availability and fast recovery.

The file system allows users to list, store, and retrieve text files contents. Much like file hosting services such as Mediafire, files will be stored in a file storage server. All file contents are open to public for listing and retrieving; however, correct credentials will be needed in order to overwrite the files stored with a secret.

SWOT Analysis

Strength:

- Entire team has proficient go programming experience
- Team members have positive attitudes and are able to communicate effectively
- Project is similar to assignment 4 and all team members have a general idea of implementation; one member implemented simple RPC chain (two RPC invocation) in assignment 4
- Team members have good understanding of RPCC and went of some implementation details during design

Weakness:

- Most of the team members have not worked with each other
- Unsure of implementation details
- Limited experience with adding to an existing API

Opportunities:

- Unsure

Threats:

- Other academic deadlines
- Scheduling conflicts with other course project meetings

Timeline

Feb 29	Proposal deadline, planning and task distribution
Mar 4	RPCC API basic support done (Farzad)
Mar 7	Front-end/Client set up (Tony + Hanson) File Storage set up (Neda) RPCC API updated (Farzad)
Mar 11	A file is able to be received & store to/from the file storage (All)
March 21	Fault tolerance and consistency algorithms/recovery added (All)
March 25	Testing of multiple clients calling our file app (Farzad + Hanson) Start Report (Neda + Tony)
April 4	Application finished (All)
April 8	Construct final report (All)
April 11	Project and final report deadline (Al)