A Reliable Broadcast System

Yuchen Dai, Xiayi Huang, Diansan Zhou
Department of Computer Sciences
and Engineering
Santa Clara University
December 10 2013
## Table of Contents

2 Introduction ........................................................................................................................................3  
   2.1 Objective ................................................................................................................................3  
   2.2 What is the Problem ................................................................................................................3  
   2.3 Why this is a project related with class ......................................................................................3  
   2.4 Why Other Approach is No Good ..........................................................................................3  
   2.5 Why you think your approach is better .................................................................................3  

3 Theoretical Bases and Literature Review .........................................................................................4  
   3.1 Theoretical Bases and Literature Review .................................................................................4  
   3.2 Definition of the Problem ........................................................................................................4  
   3.3 Theoretical Background of this Problem ..................................................................................4  
   3.4 Research Related to the Problem ............................................................................................4  
   3.5 Advantage and Disadvantage of Those Researches ...............................................................5  
   3.6 Solution ..................................................................................................................................5  
   3.7 Why Your Solution is Different From others ..........................................................................6  
   3.8 Advantage of Your Solution ......................................................................................................6  

4 Hypothesis .....................................................................................................................................6  

5 Methodology ....................................................................................................................................7  
   5.1 Input Data Generation ..............................................................................................................7  
   5.2 How to Solve the Problem ........................................................................................................7  
   5.3 Output Generation ....................................................................................................................7  
   5.4 Hypothesis Verification ............................................................................................................8  

6 Implementation .............................................................................................................................8  
   6.1 Code .......................................................................................................................................8  
   6.2 Design Document and Flowchart .............................................................................................8  

7 Data Analysis and Discussion .........................................................................................................11  

8 Conclusion and Recommendations ..................................................................................................12  

9 Bibliographies ...............................................................................................................................12  

10 Appendices ..................................................................................................................................13
2 INTRODUCTION

2.1 Objective
We are trying to build a reliable broadcast transportation layer, and test its reliability of data synchronization in a distributed storage system with harsh network environment.

2.2 What is the Problem
The network and servers (nodes) for transmitting and storing data are not always reliable, so a reliable broadcast system is desired. However, the consistency of storing data in different servers (nodes) becomes an issue. On the network side, when packages being broadcasting to servers through different routes, some of them can be lost, delayed or corrupted due to the harsh network environment. The data arrived at each server can be different. One the server side, during the data transmission, one or more servers has the possibility to be disconnected from the network for a while. After they are back online, the data stored in each single server is inconsistent. Therefore, how to synchronize data within multiple servers becomes a challenge.

2.3 Why This is a Project Related to the This Class
This project is about developing a transportation layer program. The network environment used for simulation is based on the network socket programming. Our test cases will cover the main transporting problems occur in the real world.

2.4 Why Other Approach is No Good
There are few high performance broadcast transportation protocols in the world. The widely used protocol are TCP and UDP. TCP is reliable but has nothing to do with broadcast. The UDP broadcast does not guarantee reliable delivery. Some other protocol before 1990's works only there is no server failure. So when server fails or network partition occurs, these protocols have to wait. The dominant consensus algorithm, Paxos, which is highly efficient and reliable, is not designed to solve the broadcast problem. It does not guarantee the causal order, which highly limited the adoption.

2.5 Why You Think Your Approach is Better
The algorithm behind our transportation layer is called “raft”. Compared with the current mainstream -- Paxos, Raft is able to provide equivalent result with the identical efficiency as Paxos. But the theory of Raft is much less difficult to understand, and its architecture is more feasible for building a practical system. The raft can be implement as a order-keeping broadcast protocol. Our protocol based on raft supplies fundamental broadcast APIs and tackle the problem of common network failure and
host failure. It can be adopted by distributed replicate systems without any modification.

3 THEORETICAL BASES AND LITERATURE REVIEW

3.1 Theoretical bases and literature review.

The network environment control and data transmission parts are based on the socket programming technique, and the theory behind the data synchronization of multiple servers is called Raft.

3.2 Definition of the problem

A fundamental problem in distributed computing is to achieve overall system reliability in the presence of a number of faulty processes. This often requires processes to agree on some data value that is needed during computation. The consensus problem requires agreement among a number of processes for a single data value. Some of the processes may fail or be unreliable in other ways, so consensus protocols must be fault tolerant. The processes must put forth their candidate values, communicate with one another, and agree on a single consensus value. In the broadcast system, a common agreement should be achieved can be defined as, "which is the next message".

3.3 Theoretical background of the problem

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final. Typical consensus algorithms make progress when any majority of their servers are available; for example, a cluster of 5 servers can continue to operate even if 2 servers fail. If more servers fail, they stop making progress (but will never return an incorrect result).

Consensus typically arises in the context of replicated state machines, a general approach to build fault-tolerant systems. Each server has a state machine and a log. The state machine is the component that we want to make fault-tolerant, such as a hash table. It will appear to clients that they are interacting with a single, reliable state machine, even if a minority of the servers in the cluster fail. Each state machine takes as input commands from its log. In our hash table example, the log would include commands like set x to 3. A consensus algorithm is used to agree on the commands in the servers’ logs. The consensus algorithm must ensure that if any state machine applies set x to 3 as the nth command, no other state machine will ever apply a different nth command. As a result, each state machine processes the same series of commands and thus produces the same series of results and arrives at the same series of states.
3.4 Related research to solve the problem

Bitcoin protocol solves distributed consensus without centralized authority. Clients could submit a work request to any server. The server uses a proof of work to compile work requests into an ordered sequence of command execution in the blockchain. After a certain number of blocks/confirmations, the state engines would start executing the commands in the order of the ledger/blockchain. All server nodes could agree on the state/order of the blockchain through the proof of work system after a certain number of confirmations based on network latency. In a local network, block confirmations could happen very rapidly, within seconds. In a larger distributed network, confirmations would have to be spread out for convergence of the blockchain between nodes.

Another example of consensus protocols is a polynomial time binary consensus protocol that tolerates Byzantine failures is the Phase King algorithm by Garay and Berman. The algorithm solves consensus in an synchronous message passing model with n processes and up to f failures, provided n>4f. In the Phase King algorithm, there are f+1 phases, with 2 rounds per phase. Each process keeps track of its preferred output (initially equal to the process’s own input value).

Google has implemented this distributed lock service library called Chubby. It maintains lock information in small files which are stored in a replicated database to achieve high availability in the face of failures. The database is implemented on top of a fault-tolerant log layer which is based on the Paxos consensus algorithm. In this scheme, Chubby clients communicate with the Paxos master in order to access/update the replicated log; ie, read/write to the files.

3.5 Advantage/disadvantage of those research

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members which play a key role in building reliable large-scale software systems. Most implementations of consensus are based on Paxos or influenced by it.

Paxos first defines a protocol capable of reaching agreement on a single decision, such as a single replicated log entry. Then it combines multiple instances of this protocol to facilitate a series of decisions such as a log.

Unfortunately, Paxos has two drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The foundation of Paxos is single-decree subset which is very dense and subtle. It is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. And the composition rules for multi-Paxos add significant additional complexity and subtlety which can be decomposed in other ways that are more direct and obvious instead.

The second problem with Paxos is that is does provide a good foundation for building practical implementations because there is no widely agreed-upon algorithm for multi-Paxos.

Furthermore, the Paxos architecture is a poor one for building practical systems; this is another consequence of the single-decree decomposition. For example there is little benefit to choosing a collection of log entries independently and then melding them into a sequential log; this just adds complexity. It is simpler and more efficient to design a system around a log, where new entries are
appended sequentially in a constrained order.

Another problem is that Paxos uses a symmetric peer-to-peer approach at its core which is very difficult to implement in practical systems because when a series of decisions are made, it is simpler and faster to first elect a leader, and then have the leader coordinate the decisions.

### 3.6 Solution

One approach to generating consensus is for all processes to agree on a majority value. For \( n \) processes, a majority value will require at least \( n/2 + 1 \) votes to win.

The new approach should provide a complete and appropriate foundation for system building and reduced the amount of design work required of developers; it must be safe under all conditions and available under typical operating conditions; and it must be efficient for common operations.

Raft algorithm is employed in this project to synchronize data stored in multiple servers. It provides improved understandability by reducing nondeterminism. The approach is to simplify the state space by reducing the number of states to consider, making the system more coherent and eliminating nondeterminism where possible. For example, the common agreement is a sequential log records which are not allowed to have holes, and Raft limits the ways in which logs can become inconsistent with each other.

### 3.7 Where your solution different from others

Paxos consensus algorithm is designed as synchronizing the states of multiple state machine. However, it only guarantee that the result would be same, which is total order message passing. We want to add a causal ordering semantics on our broadcast protocol so that the sending and receiving orders are the same among all the servers.

### 3.8 Advantage of our solution

From the client side, they can send messages concurrently and get response whether previous messages are commit or not. Therefore, the throughput is very high. On the other hand, the protocol is integrated without modification by replicated storage systems which require log synchronization, and it is easy to understand and verify. The master-slave pattern make the message overhead is less than many other solutions.

### 4 HYPOTHESIS (GOAL)

Under any non-Byzantine network conditions including network delays, partitions, and packet loss, duplication, and disordering, a distributed broadcast system based on the Raft algorithm is still reliable. The goal of this project is to use Raft algorithm developing a distributed broadcast system, and test its reliability with multiple different network environments.
5 METHODOLOGY

5.1 Input Data Generation

The input data for testing the reliability of the system is the file “name_list.txt”. The program will scan the name table of the file, and store in the transmission buffer of the super client. For setting up the network environment, each route between servers is configured individually through the super client. The configurations (route type) are typed in the super client’s terminal by the tester.

5.2 How to Solve the Problem

- Algorithm Design
  - we will use Raft algorithm to implement a broadcasting protocol

- Language Used
  - Erlang for Raft algorithm
    Erlang is a functional language. It is well deployed in the distributed domain. The feature of integrating send/receive semantics and a gen_fsm behavior library make the coding easy and robust.
  - C++ for simulating realistic network environment
    We want to simulate all the network failures that might encountered in the real world. To make the failures controllable, the middleware should maintain a reliable connection with its ends. The middleware should accept command so that it can drop or delay packages from any direction.
  - C for peripheral test (super client) and verify data
    C is a common glue language.

- Tools Used
  Eclipse, GCC, Erlang compiler, Visual Studio etc..

5.3 Output Generation

Firstly, through the super client, each route type of the network is specified. Then the super client needs to find the master server for data input. It will keep sending consequential packet with sequence number to each server until a server (master) send an ACK back. After the distributed system finishing synchronizing, the super client will request each server to send the data stored in their own. Then the super client will compare the data from individual server with the original data it initially sent to the master server. Any functioning server should have the same log stream in the right order as the master’s. Several cases have been designed to test the system.
5.4 Hypothesis verification

Having received each server’s data, the super client compares the data from individual server with the original data it initially sent to the master server. Due to the network problems, the data in one or more servers may not be identical to the originals. However, as long as the instantaneous master server should have the committed and valid data. With this goal achieving, it is safe to say the hypothesis of the project is verified.

6 implementation

6.1 Code (refer programming requirements)

- Language and framework.
  The main project is written by Erlang/OTP. The OTP framework is the main utility which highly reduce the code work.

- The building environment
  The code is written under the Erlang/OTP version R16B02. We can not guarantee that it can be run in another version. But we do not use the latest language feature so that a lower version should also be available.

6.2 Design Document and Flowchart

Erlang/OTP is a language well known by its light-weight processes and message passing interface. The processes in our project are divided into three most important modules the servers, the transportation channel, storage process and a client.

- Servers
  The code of server is a set of large finite state machines. It has three states: the leader, the candidate and the follower. The state changes only these events occurs: a timer preset by itself or by an incoming event. We use gen_fsm behaviour to implement it.

- Channels
  To simulate the unstable channel we described above, we design our own channel module. The channel modules are the only access through which the servers can communicate. It is a gen_server behaviour module. Apart from being used by servers, it can also accept commands sent by controllers to simulate the unpredictable link.

- Storage
  In this algorithm, we need separate persistent media to keep the local data of individual servers. In most of the other systems, this part is actually a local disk. In our implementation, we use long life processes to store the data for our servers. We carefully make sure that these stateful processes would not fail. In fact, if any one of them fails, the whole system is consider as failure. Our main object of this project is to handle the network failure. If any one wants to apply this algorithm on a real world project,
a hard disk and a serialization/deserialization should be added. In our implementation, a individual storage process maintains

- a process dictionary with only several entries
- a general balanced tree which can keep ordered data.

**Client**

A client is utility which can send command to channel and simulate a blocking broadcasting RPC to servers. It is also used as a tracer for it can access the internal states of servers.

![Figure 1. The network (three routes) controlled by the super client.](image)
Figure 2. The data for testing the system controlled by the super client.

Figure 3. The master server broadcasts (synchronizes) testing data through specified problematic network.
7. Data Analysis and Discussion

Since the consensus algorithm we use is based on log replication and leader server election, the data input will be logs stored in all servers, and the goal is to make sure all the logs stored in the all the servers are consistent. The initial input sample data is attached is the appendix. After running all the input data with our algorithm, we can output new series of log data, the we will compare the correctness of the new output with the original input, if they are the same, then we can conclude that the system based on raft consensus algorithm is reliable.

However, there are some abnormal situations in the network, for example, data duplicate, data disorder, request delay and data lost. The consensus algorithm is supposed to be reliable even under
these abnormal environments. In this case, we will need to simulate these environments using client/server sockets programming. The system is reliable if all these test cases works. That is to say, the output data should be consistent with all the other servers under these environments.

8 Conclusions and Recommendations

The goal of our project is to build a reliable client/server distributed system using Raft consensus algorithm by simulating client and server communications. The clients send all of their requests to the leader server. When a client first starts up, it connects to a randomly chosen server. If the client’s first choice is not the leader, that server will reject the client’s request and supply information about the most recent leader it has heard from. If the leader crashes, client requests will time out; clients then try again with randomly chosen servers.

Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final. Typical consensus algorithms make progress when any majorities of their servers are available; for example, a cluster of 5 servers can continue to operate even if 2 servers fail. If more servers fail, they stop making progress (but will never return an incorrect result).

Consensus typically arises in the context of replicated state machines, a general approach to build fault-tolerant systems. Each server has a state machine and a log. The state machine is the component that we want to make fault-tolerant, such as a hash table. It will appear to clients that they are interacting with a single, reliable state machine, even if a minority of the servers in the cluster fail. Each state machine takes as input commands from its log.

The implementation of reliable broadcast system is achievable using Raft consensus algorithm and the application is based on normal and abnormal test cases, like data duplicate, data lost, data disorder and request delay. More work can be done by increasing the number of servers and implementing more complicated network environment.

9 BIBLIOGRAPHY
