Analysis of Optimization Ideas Of Basic Raft Consensus Algorithm

Yaojian Wang, Tingting Bao, ZheJun Liang, Changyuan Zhang

Acknowledgement

First,this project was done based on the ideas from the paper named “In Search of an Understandable Consensus Algorithm” written by Stanford University faculties, therefore, we would like to thank them for the inspirations.

In addition,we would like to express our special appreciation and gratitude to our Professor, Dr.Ming-Hwa Wang for giving us this opportunity and encouraging our research.

Furthermore,we would also like to thank the Santa Clara University library for providing us with group discussion rooms in the college, which helped us in concentrating on the task.

Last but not least, we would like to thank our family and friends who supported us during our work.

Abstract

In the context of network agents, consensus means that a group of machines reach a common decision on a certain issue. Hence,consensus is a fundamental problem in any fault-tolerant distributed systems and it plays a key role in building reliable large-scale software systems. One very important Consensus algorithm is Raft. Raft decomposes the consensus problem into three relatively independent subproblems, which are leader election, log replication and safety. In the basic raft algorithm, there are actually a lot of aspects which can be improved to let raft to be more efficient. In our project, we will analyze the improvement of raft in the four following steps, such as sending the request to nodes by batch instead of one by one, applying pipeline to forward request, parallelly executing appending logs and forwarding requests, Asynchronous apply to state machine. Batch processing is the execution of a series of jobs in a program on a computer without manual intervention (no-intervention). One important improvement is batching, here batching is a set or "batch" of inputs, rather than a single input. The pipeline means a chain of data-processing stages. Asynchrony refers to the occurrence of events independently of the main program flow and ways to deal with such events. Therefore, our goal is to understand and analyze ideas behind optimizing the efficiency of Raft Algorithm, which including using batching, pipelining, appending logs in parallelism and asynchronous apply to get higher throughput, lower latency and higher performing speed.

1.Introduction

As an important consensus algorithm, raft implements consensus by first electing a distinguished *leader*, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. However, the basic raft algorithm still has a lot of aspects which can be improved. In our project, we will study and analyse the most major improvement which is using batching, pipelining, parallelly appending logs and sending logs, and asynchronous apply, then analyse the result of each improvement and get a deep understanding of raft.

1.1 Objective

The purpose of our project is to understand and analyze ideas behind optimizing the efficiency of Raft Algorithm. Since Raft is widely used in industry and academia, and we wanted to study if there are ways to improve raft's efficiency. Motivated by this, we read lots of research papers and studied several big open-source projects to summarize those most common resolutions. Last but not least, we want to compare the performances between the basic raft and the extended raft.

1.2 What is the problem

In the basic Raft algorithm, there is no upper bound existing for the message delays or the time taken to perform computation. Firstly, The leader deals with requests one by one, which is not very efficient. Secondly, leader sends logs to followers after appending logs to itself. Thirdly,

The leader deal with one request after another request. At last, each server use its main process to apply logs to its state machine. All the four parts can be improved by batching, pipelining, parallelism and asynchronism to improve its throughput, performance, and latency.

1.3 Why this is a project related the this class

Cloud computing deals mainly with big data storage, processing, and serving. While these are mostly embarrassingly parallel tasks, coordination still plays a major role in cloud computing systems. Coordination is needed for leader election, group membership, cluster management, service discovery, resource/access management, consistent replication of the master nodes in services, and finally for barrier-orchestration when running large analytic tasks.

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems.

Last but not least, it is an excellent study path for us to gain better understandings of consensus and build up our problem-solving skills, whilst accomplishing our project.

1.4 Why other approach is no good

In most of practical applications, Paxos is widely used to solve the consensus problem.However, Paxos is absurdly complex to understand,in spite of numerous attempts to make it more approachable.As a result, both system builders and students struggle with Paxos.

Raft is the simplified version of Paxos,so it fits better for the scope of our project. In designing Raft, the author applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety) and state space

reduction. But raft still have some issue about the performance because of asynchronization. Hence, we wanted to study and analyze some improvement ideas.

1.5 Why you think your approach is better

As we know, generally, using batch can improve performance efficiently. RocksDB I/O is an typical example. Usually, it will write multiple values in a WriteBatch buffer then write to the storage device as a whole, instead of writing the values one by one into the storage devices. For Raft, Leader can collect multiple requests together, then send them as a whole package to followers. We can define the maximum size to limit the number of the request sent as a whole.

According to simple request flow of Raft, step 2 and step 3 can be handled by parallelism. In another word, it doesn't matter whether the leader appends the log before forwarding the request or forwarding the request before appending the log. Why ? Because, in Raft, if a log is appended by most nodes, it is asserted that the log has been committed, which mean the committed log must be applied successfully. Because after this log has been committed, this log status will not affect data consensus, we can create a new thread to apply committed logs to state machine. Why we need to create a new thread to do such things? It's because applying committed logs will involve IO, which is time-consuming.

1.6 Statement of the problem

Basic Raft still have some issue about the performance during log replication. We want to study and examine some common ideas of improving the performance by using batch, pipeline, append log in parallelism and asynchronous apply.

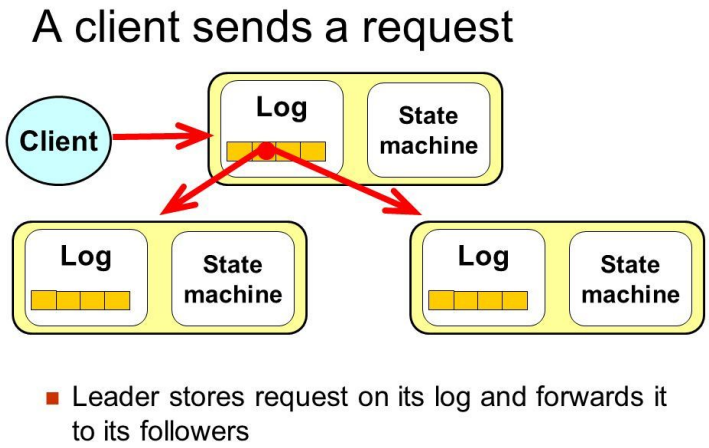
1.7 Area or scope of investigation

In this project, we are studying and analyzing an extended Raft algorithm including the following aspects:

- Basic Raft Algorithm
- Batch
- Pipeline
- Parallelism
- Asynchronous

2. Theoretical bases and literature review

2.1 Definition of the problem



In General, Raft handle a simple Request Flow works by this way:

(1) The leader receives a request from a client

(2) The leader appends the request in its log file

(3) The leader sends the corresponding log entry to its followers.

(4) The leader will wait for a response from followers. If most nodes have submitted one common log, then apply this log.

(5) The leader sends the results back to client

(6) The leader waits for the next request and repeat step (1) ~ (5)

From the flowchart, we can see the leader dealing with requests one by one, which is not very efficient. And also in the basic raft, leader operate step 3 after step 2. However, leader can actually send corresponding log entry to its followers when it appending request in its log file, which means that we will do step 2 and step 3 in parallelism. When leader wait for response from followers after sending the first request, it can also send the second request, which means we can send requests using pipeline. At last, since if the log has been committed, it will not affect the data consensus, we can create a new thread to apply the log to state machine and let the main thread to do the following log duplication.

2.2 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. Because of this, they play a key role in building reliable large-scale software systems.

Paxos has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it. Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. That's why Raft comes to light.

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi) Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes

Raft more understandable than Paxos and also provides a better foundation for building practical systems. Raft implements consensus by first electing a distinguished *leader*, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines.

Batch processing is the execution of a series of jobs in a program on a computer without manual intervention (no-intervention). In our project, our batch is a set or "batch" of inputs, rather than a *single* input.

The pipeline in our project means a chain of data-processing stages.

Asynchrony refers to the occurrence of events independently of the main program flow and ways to deal with such events.

2.3 Related research to solve the problem

For every command from the client, append to local log and start replicating that log entry, in case of replication on at least a majority of the servers, commit, apply commited entry to its own leader state machine, and then return the result to the client. If log Index is higher than the nextIndex of a follower, append all log entries at the follower using RPC, starting from the his nextIndex.

2.4 Advantage/disadvantage of those research

Raft is a new protocol that is based on insights from various previous consensus algorithms and their actual implementations. It recombines and reduces the ideas to the essential, in order to get a fully functional protocol without compromises that is still more optimal relative to previous

algorithms for consensus that do not have understandability and implementability as a primary goal.

All of these roles have a randomized time-out, on the elapse of which all roles assume that the leader has crashed and convert to be candidates, triggering a new election and incrementing the current term. So there is no upper bound about the time consuming. We need to analyze its efficiency.

2.5 Our solution to solve this problem

Based on the procedures of Raft Request Flow, the leader handles the requests sequentially. When leader handles one request, the leader is locked from next request until the leader reply the feedback to client after receiving the feedback from other nodes. However, this procedure loop is not efficient enough, and this motivated us to analyze the performances by following steps:

(1) Analyze the performance when sending requests concurrently by using batch instead of sending requests sequentially

Sending the request to nodes one by one is slow if there are a large number of nodes needed sending. In order solve the problem that the next request has to wait until the current request is committed. We propose to apply batch idea for sending the request to nodes. The leader collects the requests into a batch depending size and time limitation. After that, the leader send the whole batch as a whole message. So, the leader and nodes can handle a whole package of the request.

(2) Analyze pipeline method to forward requests

After sending the requests by batches, leader will send the next request after the current batch request is committed. However, sending batch request can be enhanced by pipeline. After

sending a batch request to nodes, the leader can send next batch request to nodes. In order to keep the consistency, leader can maintain a NextIndex variable to keep the position of the log of the next follower. In general, it is assumed that the network would be stable after the leader build the connection with the followers, so that the leader don't have to wait for the response from the followers. The leader can adjust the NextIndex and resend the log when the network is down.

(**3) Analyze the parallel execution of logging (step 2) and forwarding (step 3)**

Because it doesn't matter the sequence of completion step 2 and step 3, applying parallelism can improve the performance of the consensus algorithm. As we know, logging, a type of I/O task, is very costly. This method can only be applied to leader, instead of followers. Because if the follower tells the leader before appending the log successfully, even though the log appending fails, the leader will think that log has been committed, which increase the risk.

(4) Analyze Asynchronization

When the log is committed, the log being applied doesn't impact the consistency of data. So, a new thread can be created to apply the log asynchronously after a log is committed. One of the most essential advantages of using asynchronous apply is that we are capable of achieving concurrent processing for appending log and applying log. As for a single client, it still has to accomplish the entire process for each request, however, the concurrency and the quantity of request have been optimized as an entity.

2.6 Where your solution different from others

In the basic raft consensus algorithm, leader sends requests to followers sequentially. In our solution, we try to collect requests first and send to followers all at once by batch. In general, leader only begins sending corresponding requests after committing the previous request.

However, in our project, we will analyze the applications of Pipeline to forward requests to followers. Generally, leader only operates step 3 after finishing step 2. But in our way, we can parallelly execute logging (step 2) and forwarding (step 3). In the usual way, a server would apply the committed log to its own state machine in the final step of main process. But in our solution, we will use asynchronous apply to apply the committed logs.

2.7 Why your solution is better

Since consensus is one of the most essential problems in distributed systems. People have proposed several solutions in recent years.In most of practical applications, Paxos is widely used to solve the consensus problem.However, Paxos is absurdly complex to understand,in spite of numerous attempts to make it more approachable.As a result, both system builders and students struggle with Paxos.

Raft is the simplified version of Paxos,so it fits better for the scope of our project. In designing Raft,raft still has some issue about the performance because of asynchronous.Therefore, we proposed the idea of analyzing adding batch, pipeline and parallelism methods to gain a deep insight of raft algorithm.

3. Hypothesis

Using batch to store a certain number of log, and then set them in bulk. Append and commit operations can be done in parallel. Using bath and pipeline can improve efficiency of raft.

Possible failures may happen during sending or receiving information. We need to roll back or find appropriate solution to deal with this situation.

Since size of the batch has a great impact on the performance of the program, we need to do some test before define the batch size.

These above improvement ideas are what we want to examine for this project.

4. Methodology

4.1 How to generate/collect input data

Design an algorithm to simulate client to send a lot of requests. we test different number of requests such as 1000, 2000, 3000, 4000, 5000. Then based on time to evaluate performance of our proposal and original way.

4.2 How to solve the problem

(1) Study the basic Raft program

(2) Analyze batch for forwarding requests.

Batch: As we know, generally, using batch can improve performance efficiently. RocksDB I/O is an typical example. Usually, it will write multiple values in a WriteBatch buffer then write to the storage device as a whole, instead of writing the values one by one into the storage devices. For Raft, Leader can collect multiple requests together, then send them as a whole package to followers. We can define the maximum size to limit the number of the request sent as a whole.

(3) Analyze by applying Pipeline for forwarding requests:

Pipeline: if only batch is applied, the leader has to wait until followers return the feedback. Pipeline can improve this efficiency. Leader can maintain a NextIndex variable to represent the position of the log of the next follower. Generally, once the leader build the connection with the

follower, it is assumed that the network is stable such that leader don't have to wait for the response from the followers. If the network is down, the follower return error, the leader will adjust the NextIndex and resend the log.

(4) Analyze the parallelism of logging and forwarding

According to simple request flow of Raft, step 2 and step 3 can be handled by parallelism. In another word, it doesn't matter whether the leader appends the log before forwarding the request or forwarding the request before appending the log. Why ? Because, in Raft, if a log is appended by most nodes, it is asserted that the log has been committed, which mean the committed log must be applied successfully.

The reason why applying this approach is that appending log is costly task, so that leader can forward the request and append the request to its log simultaneously.

To notice, although leader can forward request to followers before logging, follower can't append log before telling leader if it has successfully appended the log. If the follower tells the leader before appended the log successfully, even though the log appending fails, the leader will think that log has been committed, which increase the risk in the system.

(5) Analyze Asynchronous

When the log is committed, when the log being applied doesn't impact the consistency of data. So, a new thread can be created to apply the log asynchronously after a log is committed. One of the most essential advantages of using asynchronous apply is that we are capable of achieving concurrent processing for appending log and applying log. As for a single client, it still has to accomplish the entire process for each request, however, the concurrency and the quantity of request have been optimized as an entity.

(6) Compare the performance improvement based on rare Raft and improved Raft.

4.3 Language used

Java, go

4.4 How to proof correctness

Evaluate the performance about original raft and the raft we have improved. Based on our test, we get runtime of raft algorithm under different kind of situation.

For batch, we test different batch size based on same number of request.

For pipeline, we can change parameter MaxInflightMsgs to change size of pipeline to check whether it make a big difference.

For parallelism, we can check the runtime under different situations such as single thread and multi thread.

5. Implementation

5.1 Code

5.1.1 code for multi-threading and single thread

```
static class CommittingThread implements Runnable{

        private RaftServer server;
        private Object conditionalLock;

        CommittingThread(RaftServer server){
            this.server = server;
            this.conditionalLock = new Object();
        }
```

```
        void moreToCommit(){
            synchronized(this.conditionalLock){
                this.conditionalLock.notify();
            }
        }

        @Override
        public void run() {
            while(true){
                try{
                    long currentCommitIndex = server.state.getCommitIndex();
                    while(server.quickCommitIndex <= currentCommitIndex
                            || currentCommitIndex >= server.logStore.getFirstAvailableIndex()
- 1){
                        synchronized(this.conditionalLock){
                            this.conditionalLock.wait();
                        }

                        currentCommitIndex = server.state.getCommitIndex();
                    }

                    while(currentCommitIndex < server.quickCommitIndex && currentCommitIndex <
server.logStore.getFirstAvailableIndex() - 1){
                        currentCommitIndex += 1;
                        LogEntry logEntry = server.logStore.getLogEntryAt(currentCommitIndex);
                        if(logEntry.getValueType() == LogValueType.Application){
                            server.stateMachine.commit(currentCommitIndex,
logEntry.getValue());
                        }else if(logEntry.getValueType() == LogValueType.Configuration){
                            synchronized(server){
                                ClusterConfiguration newConfig =
ClusterConfiguration.fromBytes(logEntry.getValue());
                                server.logger.info("configuration at index %d is committed",
newConfig.getLogIndex());

server.context.getServerStateManager().saveClusterConfiguration(newConfig);
                                server.configChanging = false;
                                if(server.config.getLogIndex() < newConfig.getLogIndex()){
                                    server.reconfigure(newConfig);
                                }

                                if(server.catchingUp && newConfig.getServer(server.id) !=
null){
                                    server.logger.info("this server is committed as one of
cluster members");
                                    server.catchingUp = false;
                                }
                            }
                        }
```

```
                    server.state.setCommitIndex(currentCommitIndex);
                    server.snapshotAndCompact(currentCommitIndex);
                }

                server.context.getServerStateManager().persistState(server.state);
            }catch(Throwable error){
                server.logger.error("error %s encountered for committing thread, which
should not happen, according to this, state machine may not have further progress, stop the
system", error, error.getMessage());
                server.stateMachine.exit(-1);
            }
        }
    }

    }

private void moreToCommit() {
        try{
            long currentCommitIndex = this.state.getCommitIndex();

            if (currentCommitIndex < this.quickCommitIndex && currentCommitIndex <
this.logStore.getFirstAvailableIndex() - 1){
                currentCommitIndex += 1;
                LogEntry logEntry = this.logStore.getLogEntryAt(currentCommitIndex);
                if(logEntry.getValueType() == LogValueType.Application){
                    this.stateMachine.commit(currentCommitIndex, logEntry.getValue());
                }else if(logEntry.getValueType() == LogValueType.Configuration){
                    ClusterConfiguration newConfig =
ClusterConfiguration.fromBytes(logEntry.getValue());
                    this.logger.info("configuration at index %d is committed",
newConfig.getLogIndex());
                    this.context.getServerStateManager().saveClusterConfiguration(newConfig);
                    this.configChanging = false;
                    if(this.config.getLogIndex() < newConfig.getLogIndex()){
                        this.reconfigure(newConfig);
                    }

                    if(this.catchingUp && newConfig.getServer(this.id) != null){
                        this.logger.info("this server is committed as one of cluster
members");
                        this.catchingUp = false;
                    }

                }

                this.state.setCommitIndex(currentCommitIndex);
                this.snapshotAndCompact(currentCommitIndex);
            }
```

```
            this.context.getServerStateManager().persistState(this.state);
        }catch(Throwable error){
            this.logger.error("error %s encountered for committing thread, which should not
happen, according to this, state machine may not have further progress, stop the system",
error, error.getMessage());
            this.stateMachine.exit(-1);
        }
    }
```

5.1.2 Code for batching:

```
// sendAppend sends RPC, with entries to the given peer.
func (r *raft) sendAppend(to uint64) {
        pr := r.prs[to]
        if pr.IsPaused() {
                return
        }
        m := pb.Message{}
        m.To = to

        term, errt := r.raftLog.term(pr.Next - 1)
        ents, erre := r.raftLog.entries(pr.Next, r.maxMsgSize)

        if errt != nil || erre != nil { // send snapshot if we failed to get term or entries
                if !pr.RecentActive {
                        r.logger.Debugf("ignore sending snapshot to %x since it is not recently
active", to)
                        return
                }

                m.Type = pb.MsgSnap
                snapshot, err := r.raftLog.snapshot()
                if err != nil {
                        if err == ErrSnapshotTemporarilyUnavailable {
                                r.logger.Debugf("%x failed to send snapshot to %x because
snapshot is temporarily unavailable", r.id, to)
                                return
                        }
                        panic(err) // TODO(bdarnell)
                }
                if IsEmptySnap(snapshot) {
                        panic("need non-empty snapshot")
                }
                m.Snapshot = snapshot
                sindex, sterm := snapshot.Metadata.Index, snapshot.Metadata.Term
                r.logger.Debugf("%x [firstindex: %d, commit: %d] sent snapshot[index: %d, term:
%d] to %x [%s]",
                        r.id, r.raftLog.firstIndex(), r.raftLog.committed, sindex, sterm, to,
pr)
```

```
                pr.becomeSnapshot(sindex)
                r.logger.Debugf("%x paused sending replication messages to %x [%s]", r.id, to,
pr)
        } else {
                m.Type = pb.MsgApp
                m.Index = pr.Next - 1
                m.LogTerm = term
                m.Entries = ents
                m.Commit = r.raftLog.committed
                if n := len(m.Entries); n != 0 {
                        switch pr.State {
                        // optimistically increase the next when in ProgressStateReplicate
                        case ProgressStateReplicate:
                                last := m.Entries[n-1].Index
                                pr.optimisticUpdate(last)
                                pr.ins.add(last)
                        case ProgressStateProbe:
                                pr.pause()
                        default:
                                r.logger.Panicf("%x is sending append in unhandled state %s",
r.id, pr.State)
                        }
                }
        }
        r.send(m)
}
```

5.1.3 code for grouping batch requests

```
private static List<String> reqList = new ArrayList<>();


   private static void executeAsClient(ClusterConfiguration configuration, ExecutorService
executor) throws Exception{
        RaftClient client = new RaftClient(new RpcTcpClientFactory(executor), configuration,
new Log4jLoggerFactory());
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        while(true){
            System.out.print("Message:");
            String message = reader.readLine();
            if(message.startsWith("addsrv")){
                StringTokenizer tokenizer = new StringTokenizer(message, ";");
                ArrayList<String> values = new ArrayList<String>();
                while(tokenizer.hasMoreTokens()){
                    values.add(tokenizer.nextToken());
                }

                if(values.size() == 3){
```

```
                    ClusterServer server = new ClusterServer();
                    server.setEndpoint(values.get(2));
                    server.setId(Integer.parseInt(values.get(1)));
                    boolean accepted = client.addServer(server).get();
                    System.out.println("Accepted: " + String.valueOf(accepted));
                    continue;
                }
            }else if(message.startsWith("fmt:")){
                long start = System.currentTimeMillis();
                String format = message.substring(4);
                System.out.print("How many?");
                String countValue = reader.readLine();
                int count = Integer.parseInt(countValue.trim());
// jeff s
                int batch_size = 10;
// jeff e
                for(int i = 1; i <= count; ++i){
// jeff s
                    reqList.add(format);
                    if (reqList.size() < batch_size) {
                        continue;
                    } else {
                        StringBuilder sb = new StringBuilder();
                        for (String reqValue : reqList) {
                            sb.append(reqValue);
                        }
                        byte[][] byteArr = new byte[][]{sb.toString().getBytes()};
                        System.out.println("byteArr.length = " + byteArr.length);
                        for (int j = 0; j < byteArr.length; j++) {
                            System.out.println("byteArr[" + j + "]length = " +
byteArr[j].length);
                            for (int k = 0; k < byteArr[j].length; k++) {
                                System.out.print(byteArr[j][k]);
                            }
                        }
                        boolean accepted = client.appendEntries(new
byte[][]{sb.toString().getBytes()}).get();
                        System.out.println("Accepted: " + String.valueOf(accepted));
                        reqList.removeAll(reqList);
                    }

                    //String msg = String.format(format, i);
/*                     for (int j = 10; j > 0; j--) {
                        String msg = String.format(format, i);
                        sb.append(msg);
                        i++;
                    }
                    i--;
*/
```

```
                    //boolean accepted = client.appendEntries(new byte[][]{ msg.getBytes()
}).get();

                    //System.out.println("Accepted: " + String.valueOf(accepted));
// jeff e
                }
                long end = System.currentTimeMillis();
                long elapse = end - start;
                System.out.println("time elapse: end - start = " +  (end - start));
                continue;
            }else if(message.startsWith("rmsrv:")){
                String text = message.substring(6);
                int serverId = Integer.parseInt(text.trim());
                boolean accepted = client.removeServer(serverId).get();
                System.out.println("Accepted: " + String.valueOf(accepted));
                continue;
        }

            boolean accepted = client.appendEntries(new byte[][]{ message.getBytes() }).get();
            System.out.println("Accepted: " + String.valueOf(accepted));
        }
    }
```

5.1.4 the entry of the process to handle client requests in server

```
public RaftResponseMessage processRequest(RaftRequestMessage request) {

    this.logger.debug(
            "Receive a %s message from %d with LastLogIndex=%d, LastLogTerm=%d,
EntriesLength=%d, CommitIndex=%d and Term=%d",
            request.getMessageType().toString(),
            request.getSource(),
            request.getLastLogIndex(),
            request.getLastLogTerm(),
            request.getLogEntries() == null ? 0 : request.getLogEntries().length,
            request.getCommitIndex(),
            request.getTerm());

    RaftResponseMessage response = null;
    if(request.getMessageType() == RaftMessageType.AppendEntriesRequest){
        response = this.handleAppendEntriesRequest(request);
    }else if(request.getMessageType() == RaftMessageType.RequestVoteRequest){
        response = this.handleVoteRequest(request);
    }else if(request.getMessageType() == RaftMessageType.ClientRequest){
        System.out.println("test only: receive client request in RaftServer.java");    // jeff
test only
        response = this.handleClientRequest(request);
    }else{
        // extended requests
```

```
        response = this.handleExtendedMessages(request);
    }

    if(response != null){
        this.logger.debug(
                "Response back a %s message to %d with Accepted=%s, Term=%d, NextIndex=%d",
                response.getMessageType().toString(),
                response.getDestination(),
                String.valueOf(response.isAccepted()),
                response.getTerm(),
                response.getNextIndex());
    }

    return response;
}
```

5.1.5 client send requests to servers

```
private RaftResponseMessage handleClientRequest(RaftRequestMessage request){
    RaftResponseMessage response = new RaftResponseMessage();
    response.setMessageType(RaftMessageType.AppendEntriesResponse);
    response.setSource(this.id);
    response.setDestination(this.leader);
    response.setTerm(this.state.getTerm());

    long term;
    synchronized(this){
        if(this.role != ServerRole.Leader){
            response.setAccepted(false);
            return response;
        }

        term = this.state.getTerm();
    }

    LogEntry[] logEntries = request.getLogEntries();
    if(logEntries != null && logEntries.length > 0){
        System.out.println("logEntries.length = " + logEntries.length);    // test only
        for(int i = 0; i < logEntries.length; ++i){

            this.stateMachine.preCommit(this.logStore.append(new LogEntry(term,
logEntries[i].getValue())), logEntries[i].getValue());
        }
    }

    // Urgent commit, so that the commit will not depend on heartbeat
    this.requestAppendEntries();
    response.setAccepted(true);
    response.setNextIndex(this.logStore.getFirstAvailableIndex());
```

```
    return response;
}
```

5.1.6 commit the log to the state machine

```
static class CommittingThread implements Runnable{

    private RaftServer server;
    private Object conditionalLock;
    private long _start; // jeff test

    CommittingThread(RaftServer server){
        this.server = server;
        this.conditionalLock = new Object();
    }

    void moreToCommit(){
        synchronized(this.conditionalLock){
            this.conditionalLock.notify();
        }
    }
    // jeff start
    void setStartTime(long start){
        System.out.println("start = " + _start);
        this._start = start;
    }
    // jeff end
    @Override
    public void run() {
        while(true){
            try{
                long currentCommitIndex = server.state.getCommitIndex();
                while(server.quickCommitIndex <= currentCommitIndex
                        || currentCommitIndex >= server.logStore.getFirstAvailableIndex() - 1){
                    synchronized(this.conditionalLock){
                        this.conditionalLock.wait();
                    }

                    currentCommitIndex = server.state.getCommitIndex();
                }

                while(currentCommitIndex < server.quickCommitIndex && currentCommitIndex <
server.logStore.getFirstAvailableIndex() - 1){
                    currentCommitIndex += 1;
                    LogEntry logEntry = server.logStore.getLogEntryAt(currentCommitIndex);
                    if(logEntry.getValueType() == LogValueType.Application){
                        server.stateMachine.commit(currentCommitIndex, logEntry.getValue());
```

```
                    long end = System.currentTimeMillis();
                    System.out.println("end time = " + end);    // test only
                    System.out.println("end - start = " + (end - _start));
                }else if(logEntry.getValueType() == LogValueType.Configuration){
                    synchronized(server){
                        ClusterConfiguration newConfig =
ClusterConfiguration.fromBytes(logEntry.getValue());
                        server.logger.info("configuration at index %d is committed",
newConfig.getLogIndex());

server.context.getServerStateManager().saveClusterConfiguration(newConfig);
                        server.configChanging = false;
                        if(server.config.getLogIndex() < newConfig.getLogIndex()){
                            server.reconfigure(newConfig);
                        }

                        if(server.catchingUp && newConfig.getServer(server.id) != null){
                            server.logger.info("this server is committed as one of cluster
members");
                            server.catchingUp = false;
                        }
                    }
                }

                server.state.setCommitIndex(currentCommitIndex);
                server.snapshotAndCompact(currentCommitIndex);
            }

            server.context.getServerStateManager().persistState(server.state);
        }catch(Throwable error){
            server.logger.error("error %s encountered for committing thread, which should
not happen, according to this, state machine may not have further progress, stop the system",
error, error.getMessage());
            server.stateMachine.exit(-1);
        }
    }
}

}
```

5.1.8 append requests to entries

```
public CompletableFuture<Boolean> appendEntries(byte[][] values){
   if(values == null || values.length == 0){
       throw new IllegalArgumentException("values cannot be null or empty");
   }

   LogEntry[] logEntries = new LogEntry[values.length];
   for(int i = 0; i < values.length; ++i){
```

```
        logEntries[i] = new LogEntry(0, values[i]);
    }

    RaftRequestMessage request = new RaftRequestMessage();
    request.setMessageType(RaftMessageType.ClientRequest);
    request.setLogEntries(logEntries);

    CompletableFuture<Boolean> result = new CompletableFuture<Boolean>();
    this.tryCurrentLeader(request, result, 0, 0);
    return result;
}
private void tryCurrentLeader(RaftRequestMessage request, CompletableFuture<Boolean> future,
int rpcBackoff, int retry){
    logger.debug("trying request to %d as current leader", this.leaderId);
    getOrCreateRpcClient().send(request).whenCompleteAsync((RaftResponseMessage response,
Throwable error) -> {
            if(error == null){
            logger.debug("response from remote server, leader: %d, accepted: %s",
response.getDestination(), String.valueOf(response.isAccepted()));
            if(response.isAccepted()){
                future.complete(true);
            }else{
                // set the leader return from the server
                if(this.leaderId == response.getDestination() && !this.randomLeader){
                    future.complete(false);
                }else{
                    this.randomLeader = false;
                    this.leaderId = response.getDestination();
                    tryCurrentLeader(request, future, rpcBackoff, retry);
                }
            }
        }else{
            logger.info("rpc error, failed to send request to remote server (%s)",
error.getMessage());
            if(retry > configuration.getServers().size()){
                future.complete(false);
                return;
            }

            // try a random server as leader
            this.leaderId =
this.configuration.getServers().get(this.random.nextInt(this.configuration.getServers().size()
)).getId();
            this.randomLeader = true;
            refreshRpcClient();

            if(rpcBackoff > 0){
                timer.schedule(new TimerTask(){

                    @Override
```

```
                    public void run() {
                        tryCurrentLeader(request, future, rpcBackoff + 50, retry + 1);

                    }}, rpcBackoff);
            }else{
                tryCurrentLeader(request, future, rpcBackoff + 50, retry + 1);
            }
        }
    });
}
private RpcClient getOrCreateRpcClient(){
    synchronized(this.rpcClients){
        if(this.rpcClients.containsKey(this.leaderId)){
            return this.rpcClients.get(this.leaderId);
        }

        RpcClient client = this.rpcClientFactory.createRpcClient(getLeaderEndpoint());
        this.rpcClients.put(this.leaderId, client);
        return client;
    }
}
```

5.1.8 communication between client to server for multiple requests

```
    private AsynchronousSocketChannel connection;
    private AsynchronousChannelGroup channelGroup;
    private ConcurrentLinkedQueue<AsyncTask<ByteBuffer>> readTasks;
    private ConcurrentLinkedQueue<AsyncTask<RaftRequestMessage>> writeTasks;
    private AtomicInteger readers;
    private AtomicInteger writers;
    private InetSocketAddress remote;
    private Logger logger;

    public RpcTcpClient(InetSocketAddress remote, ExecutorService executorService){
        this.remote = remote;
        this.logger = LogManager.getLogger(getClass());
        this.readTasks = new ConcurrentLinkedQueue<AsyncTask<ByteBuffer>>();
        this.writeTasks = new ConcurrentLinkedQueue<AsyncTask<RaftRequestMessage>>();
        this.readers = new AtomicInteger(0);
        this.writers = new AtomicInteger(0);
        try{
            this.channelGroup = AsynchronousChannelGroup.withThreadPool(executorService);
        }catch(IOException err){
            this.logger.error("failed to create channel group", err);
            throw new RuntimeException("failed to create the channel group due to errors.");
        }
    }

    @Override
```

```
    public synchronized CompletableFuture<RaftResponseMessage> send(final RaftRequestMessage
request) {
        this.logger.debug(String.format("trying to send message %s to server %d at endpoint
%s", request.getMessageType().toString(), request.getDestination(), this.remote.toString()));
        CompletableFuture<RaftResponseMessage> result = new
CompletableFuture<RaftResponseMessage>();
        if(this.connection == null || !this.connection.isOpen()){
            try{
                this.connection = AsynchronousSocketChannel.open(this.channelGroup);
                this.connection.connect(this.remote, new
AsyncTask<RaftRequestMessage>(request, result), handlerFrom((Void v,
AsyncTask<RaftRequestMessage> task) -> {
                    sendAndRead(task, false);
                }));
            }catch(Throwable error){
                closeSocket();
                result.completeExceptionally(error);
            }
        }else{
            this.sendAndRead(new AsyncTask<RaftRequestMessage>(request, result), false);
        }

        return result;
    }

    private void sendAndRead(AsyncTask<RaftRequestMessage> task, boolean skipQueueing){
        if(!skipQueueing){
            int writerCount = this.writers.getAndIncrement();
            if(writerCount > 0){
                this.logger.debug("there is a pending write, queue this write task");
                this.writeTasks.add(task);
                return;
            }
        }

        ByteBuffer buffer = ByteBuffer.wrap(BinaryUtils.messageToBytes(task.input));
        try{
            AsyncUtility.writeToChannel(this.connection, buffer, task, handlerFrom((Integer
bytesSent, AsyncTask<RaftRequestMessage> context) -> {
                if(bytesSent.intValue() < buffer.limit()){
                    logger.info("failed to sent the request to remote server.");
                    context.future.completeExceptionally(new IOException("Only partial of the
data could be sent"));
                    closeSocket();
                }else{
                    // read the response
                    ByteBuffer responseBuffer =
ByteBuffer.allocate(BinaryUtils.RAFT_RESPONSE_HEADER_SIZE);
                    this.readResponse(new AsyncTask<ByteBuffer>(responseBuffer,
context.future), false);
```

```
                }

                int waitingWriters = this.writers.decrementAndGet();
                if(waitingWriters > 0){
                    this.logger.debug("there are pending writers in queue, will try to process
them");
                    AsyncTask<RaftRequestMessage> pendingTask = null;
                    while((pendingTask = this.writeTasks.poll()) == null);
                    this.sendAndRead(pendingTask, true);
                }
            }));
        }catch(Exception writeError){
            logger.info("failed to write the socket", writeError);
            task.future.completeExceptionally(writeError);
            closeSocket();
        }
    }

    private void readResponse(AsyncTask<ByteBuffer> task, boolean skipQueueing){
        if(!skipQueueing){
            int readerCount = this.readers.getAndIncrement();
            if(readerCount > 0){
                this.logger.debug("there is a pending read, queue this read task");
                this.readTasks.add(task);
                return;
            }
        }

        CompletionHandler<Integer, AsyncTask<ByteBuffer>> handler = handlerFrom((Integer
bytesRead, AsyncTask<ByteBuffer> context) -> {
            if(bytesRead.intValue() < BinaryUtils.RAFT_RESPONSE_HEADER_SIZE){
                logger.info("failed to read response from remote server.");
                context.future.completeExceptionally(new IOException("Only part of the
response data could be read"));
                closeSocket();
            }else{
                RaftResponseMessage response =
BinaryUtils.bytesToResponseMessage(context.input.array());
                context.future.complete(response);
            }

            int waitingReaders = this.readers.decrementAndGet();
            if(waitingReaders > 0){
                this.logger.debug("there are pending readers in queue, will try to process
them");
                AsyncTask<ByteBuffer> pendingTask = null;
                while((pendingTask = this.readTasks.poll()) == null);
                this.readResponse(pendingTask, true);
            }
        });
```

```
        try{
            this.logger.debug("reading response from socket...");
            AsyncUtility.readFromChannel(this.connection, task.input, task, handler);
        }catch(Exception readError){
            logger.info("failed to read from socket", readError);
            task.future.completeExceptionally(readError);
            closeSocket();
        }
    }

    private <V, I> CompletionHandler<V, AsyncTask<I>> handlerFrom(BiConsumer<V, AsyncTask<I>>
completed) {
        return AsyncUtility.handlerFrom(completed, (Throwable error, AsyncTask<I> context) ->
{
                        this.logger.info("socket error", error);
                        context.future.completeExceptionally(error);
                        closeSocket();
                    });
    }
```

5.1.9 servers accept requests

```
    private void acceptRequests(RaftMessageHandler messageHandler){
        try{
            this.listener.accept(messageHandler, AsyncUtility.handlerFrom(
                    (AsynchronousSocketChannel connection, RaftMessageHandler handler) -> {
                        connections.add(connection);
                        acceptRequests(handler);
                        readRequest(connection, handler);
                    },
                    (Throwable error, RaftMessageHandler handler) -> {
                        logger.error("accepting a new connection failed, will still keep
accepting more requests", error);
                        acceptRequests(handler);
                    }));
        }catch(Exception exception){
            logger.error("failed to accept new requests, will retry", exception);
            this.acceptRequests(messageHandler);
        }
    }
```

5.1.10 servers read requests

```
    private void readRequest(final AsynchronousSocketChannel connection, RaftMessageHandler
messageHandler){
        ByteBuffer buffer = ByteBuffer.allocate(BinaryUtils.RAFT_REQUEST_HEADER_SIZE);
        try{
```

```
            AsyncUtility.readFromChannel(connection, buffer, messageHandler,
handlerFrom((Integer bytesRead, final RaftMessageHandler handler) -> {
                if(bytesRead.intValue() < BinaryUtils.RAFT_REQUEST_HEADER_SIZE){
                    logger.info("failed to read the request header from client socket");
                    closeSocket(connection);
                }else{
                    try{
                        logger.debug("request header read, try to see if there is a request
body");
                        final Pair<RaftRequestMessage, Integer> requestInfo =
BinaryUtils.bytesToRequestMessage(buffer.array());
                        if(requestInfo.getSecond().intValue() > 0){
                            ByteBuffer logBuffer =
ByteBuffer.allocate(requestInfo.getSecond().intValue());
                            AsyncUtility.readFromChannel(connection, logBuffer, null,
handlerFrom((Integer size, Object attachment) -> {
                                if(size.intValue() < requestInfo.getSecond().intValue()){
                                    logger.info("failed to read the log entries data from
client socket");
                                    closeSocket(connection);
                                }else{
                                    try{

requestInfo.getFirst().setLogEntries(BinaryUtils.bytesToLogEntries(logBuffer.array()));
                                        processRequest(connection, requestInfo.getFirst(),
handler);
                                    }catch(Throwable error){
                                        logger.info("log entries parsing error", error);
                                        closeSocket(connection);
                                    }
                                }
                            }, connection));
                        }else{
                            processRequest(connection, requestInfo.getFirst(), handler);
                        }
                    }catch(Throwable runtimeError){
                        // if there are any conversion errors, we need to close the client
socket to prevent more errors
                        closeSocket(connection);
                        logger.info("message reading/parsing error", runtimeError);
                    }
                }
            }, connection));
        }catch(Exception readError){
            logger.info("failed to read more request from client socket", readError);
            closeSocket(connection);
        }
    }
```

5.1.11 Server process requests

```
    private void processRequest(AsynchronousSocketChannel connection, RaftRequestMessage
request, RaftMessageHandler messageHandler){
        try{
            RaftResponseMessage response = messageHandler.processRequest(request);
            final ByteBuffer buffer = ByteBuffer.wrap(BinaryUtils.messageToBytes(response));
            AsyncUtility.writeToChannel(connection, buffer, null, handlerFrom((Integer
bytesSent, Object attachment) -> {
                if(bytesSent.intValue() < buffer.limit()){
                    logger.info("failed to completely send the response.");
                    closeSocket(connection);
                }else{
                    logger.debug("response message sent.");
                    if(connection.isOpen()){
                        logger.debug("try to read next request");
                        readRequest(connection, messageHandler);
                    }
                }
            }, connection));
        }catch(Throwable error){
            // for any errors, we will close the socket to prevent more errors
            closeSocket(connection);
            logger.error("failed to process the request or send the response", error);
        }
    }
```

5.1.12 Vote for Leader

```
    private void requestVote(){
        // vote for self
        this.logger.info("requestVote started with term %d", this.state.getTerm());
        this.state.setVotedFor(this.id);
        this.context.getServerStateManager().persistState(this.state);
        this.votesGranted += 1;
        this.votesResponded += 1;

        // this is the only server?
        if(this.votesGranted > (this.peers.size() + 1) / 2){
            this.electionCompleted = true;
            this.becomeLeader();
            return;
        }

        for(PeerServer peer : this.peers.values()){
            RaftRequestMessage request = new RaftRequestMessage();
            request.setMessageType(RaftMessageType.RequestVoteRequest);
            request.setDestination(peer.getId());
```

```
            request.setSource(this.id);
            request.setLastLogIndex(this.logStore.getFirstAvailableIndex() - 1);
            request.setLastLogTerm(this.termForLastLog(this.logStore.getFirstAvailableIndex()
- 1));
            request.setTerm(this.state.getTerm());
            this.logger.debug("send %s to server %d with term %d",
RaftMessageType.RequestVoteRequest.toString(), peer.getId(), this.state.getTerm());
            peer.SendRequest(request).whenCompleteAsync((RaftResponseMessage response,
Throwable error) -> {
                handlePeerResponse(response, error);
            }, this.context.getScheduledExecutor());
        }
    }

    private void requestAppendEntries(){
        if(this.peers.size() == 0){
            this.commit(this.logStore.getFirstAvailableIndex() - 1);
            return;
        }

        for(PeerServer peer : this.peers.values()){
            this.requestAppendEntries(peer);
        }
    }
```

5.1.13 Leader ask followers to append entries

```
    private void requestAppendEntries(){
        if(this.peers.size() == 0){
            this.commit(this.logStore.getFirstAvailableIndex() - 1);
            return;
        }

        for(PeerServer peer : this.peers.values()){
            this.requestAppendEntries(peer);
        }
    }

    private boolean requestAppendEntries(PeerServer peer){
        if(peer.makeBusy()){
            peer.SendRequest(this.createAppendEntriesRequest(peer))
                .whenCompleteAsync((RaftResponseMessage response, Throwable error) -> {
                    try{
                        handlePeerResponse(response, error);
                    }catch(Throwable err){
                        this.logger.error("Uncaught exception %s", err.toString());
                    }
                }, this.context.getScheduledExecutor());
            return true;
```

```
        }

        this.logger.debug("Server %d is busy, skip the request", peer.getId());
        return false;
    }
```

5.1.14 Followers handle and reply responses

```
    private synchronized void handlePeerResponse(RaftResponseMessage response, Throwable
error){
        if(error != null){
            this.logger.info("peer response error: %s", error.getMessage());
            return;
        }

        this.logger.debug(
                "Receive a %s message from peer %d with Result=%s, Term=%d, NextIndex=%d",
                response.getMessageType().toString(),
                response.getSource(),
                String.valueOf(response.isAccepted()),
                response.getTerm(),
                response.getNextIndex());
        // If term is updated no need to proceed
        if(this.updateTerm(response.getTerm())){
            return;
        }

        // Ignore the response that with lower term for safety
        if(response.getTerm() < this.state.getTerm()){
            this.logger.info("Received a peer response from %d that with lower term value %d
v.s. %d", response.getSource(), response.getTerm(), this.state.getTerm());
            return;
        }

        if(response.getMessageType() == RaftMessageType.RequestVoteResponse){
            this.handleVotingResponse(response);
        }else if(response.getMessageType() == RaftMessageType.AppendEntriesResponse){
            this.handleAppendEntriesResponse(response);
        }else if(response.getMessageType() == RaftMessageType.InstallSnapshotResponse){
            this.handleInstallSnapshotResponse(response);
        }else{
            this.logger.error("Received an unexpected message %s for response, system exits.",
response.getMessageType().toString());
            this.stateMachine.exit(-1);
        }
    }

    private void handleAppendEntriesResponse(RaftResponseMessage response){
        PeerServer peer = this.peers.get(response.getSource());
        if(peer == null){
```

```
            this.logger.info("the response is from an unkonw peer %d", response.getSource());
            return;
        }

        // If there are pending logs to be synced or commit index need to be advanced,
continue to send appendEntries to this peer
        boolean needToCatchup = true;
        if(response.isAccepted()){
            synchronized(peer){
                peer.setNextLogIndex(response.getNextIndex());
                peer.setMatchedIndex(response.getNextIndex() - 1);
            }

            // try to commit with this response
            ArrayList<Long> matchedIndexes = new ArrayList<Long>(this.peers.size() + 1);
            matchedIndexes.add(this.logStore.getFirstAvailableIndex() - 1);
            for(PeerServer p : this.peers.values()){
                matchedIndexes.add(p.getMatchedIndex());
            }

            matchedIndexes.sort(indexComparator);
            this.commit(matchedIndexes.get((this.peers.size() + 1) / 2));
            needToCatchup = peer.clearPendingCommit() || response.getNextIndex() <
this.logStore.getFirstAvailableIndex();
        }else{
            synchronized(peer){
                // Improvement: if peer's real log length is less than was assumed, reset to
that length directly
                if(response.getNextIndex() > 0 && peer.getNextLogIndex() >
response.getNextIndex()){
                    peer.setNextLogIndex(response.getNextIndex());
                }else{
                    peer.setNextLogIndex(peer.getNextLogIndex() - 1);
                }
            }
        }

        // This may not be a leader anymore, such as the response was sent out long time ago
        // and the role was updated by UpdateTerm call
        // Try to match up the logs for this peer
        if(this.role == ServerRole.Leader && needToCatchup){
            this.requestAppendEntries(peer);
        }
    }
```

5.1.15 Input and Entries of the raft program

```
    public static void main( String[] args ) throws Exception
    {
```

```
        if(args.length < 2){
            System.out.println("Please specify execution mode and a base directory for this
instance.");
            return;
        }

        if(!"server".equalsIgnoreCase(args[0]) && !"client".equalsIgnoreCase(args[0]) &&
!"dummy".equalsIgnoreCase(args[0])){
            System.out.println("only client and server modes are supported");
            return;
        }

        ScheduledThreadPoolExecutor executor = new
ScheduledThreadPoolExecutor(Runtime.getRuntime().availableProcessors() * 2);
        if("dummy".equalsIgnoreCase(args[0])){
            executeInDummyMode(args[1], executor);
            return;
        }

        Path baseDir = Paths.get(args[1]);
        if(!Files.isDirectory(baseDir)){
            System.out.printf("%s does not exist as a directory\n", args[1]);
            return;
        }

        FileBasedServerStateManager stateManager = new FileBasedServerStateManager(args[1]);
        ClusterConfiguration config = stateManager.loadClusterConfiguration();

        if("client".equalsIgnoreCase(args[0])){
            executeAsClient(config, executor);
            return;
        }

        // Server mode
        int port = 8000;
        if(args.length >= 3){
            port = Integer.parseInt(args[2]);
        }
        URI localEndpoint = new
URI(config.getServer(stateManager.getServerId()).getEndpoint());
        RaftParameters raftParameters = new RaftParameters()
                .withElectionTimeoutUpper(5000)
                .withElectionTimeoutLower(3000)
                .withHeartbeatInterval(1500)
                .withRpcFailureBackoff(500)
                .withMaximumAppendingSize(200)
                .withLogSyncBatchSize(5)
                .withLogSyncStoppingGap(5)
                .withSnapshotEnabled(5000)
                .withSyncSnapshotBlockSize(0);
```

```
        MessagePrinter mp = new MessagePrinter(baseDir, port);
        RaftContext context = new RaftContext(
                stateManager,
                mp,
                raftParameters,
                new RpcTcpListener(localEndpoint.getPort(), executor),
                new Log4jLoggerFactory(),
                new RpcTcpClientFactory(executor),
                executor);
        RaftConsensus.run(context);
        System.out.println( "Press Enter to exit." );
        System.in.read();
        mp.stop();
    }
```

5.1.16 Client starts

```
    private static void executeAsClient(ClusterConfiguration configuration, ExecutorService
executor) throws Exception{
        RaftClient client = new RaftClient(new RpcTcpClientFactory(executor), configuration,
new Log4jLoggerFactory());
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        while(true){
            System.out.print("Message:");
            String message = reader.readLine();
            if(message.startsWith("addsrv")){
                StringTokenizer tokenizer = new StringTokenizer(message, ";");
                ArrayList<String> values = new ArrayList<String>();
                while(tokenizer.hasMoreTokens()){
                    values.add(tokenizer.nextToken());
                }

                if(values.size() == 3){
                    ClusterServer server = new ClusterServer();
                    server.setEndpoint(values.get(2));
                    server.setId(Integer.parseInt(values.get(1)));
                    boolean accepted = client.addServer(server).get();
                    System.out.println("Accepted: " + String.valueOf(accepted));
                    continue;
                }
            }else if(message.startsWith("fmt:")){
                String format = message.substring(4);
                System.out.print("How many?");
                String countValue = reader.readLine();
                int count = Integer.parseInt(countValue.trim());
                for(int i = 1; i <= count; ++i){
                    String msg = String.format(format, i);
                    boolean accepted = client.appendEntries(new byte[][]{ msg.getBytes()
}).get();
```

```
                System.out.println("Accepted: " + String.valueOf(accepted));
            }
            continue;
        }else if(message.startsWith("rmsrv:")){
            String text = message.substring(6);
            int serverId = Integer.parseInt(text.trim());
            boolean accepted = client.removeServer(serverId).get();
            System.out.println("Accepted: " + String.valueOf(accepted));
            continue;
        }

        boolean accepted = client.appendEntries(new byte[][]{ message.getBytes() }).get();
        System.out.println("Accepted: " + String.valueOf(accepted));
    }
}
```

5.1.17 Generate a client

```
public class RpcTcpClientFactory implements RpcClientFactory {
    private ExecutorService executorService;

    public RpcTcpClientFactory(ExecutorService executorService){
        this.executorService = executorService;
    }

    @Override
    public RpcClient createRpcClient(String endpoint) {
        try {
            URI uri = new URI(endpoint);
            return new RpcTcpClient(new InetSocketAddress(uri.getHost(), uri.getPort()),
this.executorService);
        } catch (URISyntaxException e) {
            LogManager.getLogger(getClass()).error(String.format("%s is not a valid uri",
endpoint));
            throw new IllegalArgumentException("invalid uri for endpoint");
        }
    }

}
```

6. Data analysis and discussion

6.1 Output generation

6.1.1 output for different batch size

Start 3 Raft-HTTP servers:

```
raft2017/06/12 14:33:17 INFO: 2 is starting a new election at term 1
raft2017/06/12 14:33:17 INFO: 2 became candidate at term 2
raft2017/06/12 14:33:17 INFO: 2 received MsgVoteResp from 2 at term 2
raft2017/06/12 14:33:17 INFO: 2 [logterm: 1, index: 3] sent MsgVote request to 1 at term 2
raft2017/06/12 14:33:17 INFO: 2 [logterm: 1, index: 3] sent MsgVote request to 3 at term 2
raft2017/06/12 14:33:17 INFO: 1 [term: 1] received a MsgVote message with higher term from 2 [term: 2]
raft2017/06/12 14:33:17 INFO: 1 became follower at term 2
raft2017/06/12 14:33:17 INFO: 1 [logterm: 1, index: 3, vote: 0] cast MsgVote for 2 [logterm: 1, index: 3] at term 2
raft2017/06/12 14:33:17 INFO: 3 [term: 1] received a MsgVote message with higher term from 2 [term: 2]
raft2017/06/12 14:33:17 INFO: 3 became follower at term 2
raft2017/06/12 14:33:17 INFO: 3 [logterm: 1, index: 3, vote: 0] cast MsgVote for 2 [logterm: 1, index: 3] at term 2
raft2017/06/12 14:33:17 INFO: 2 received MsgVoteResp from 1 at term 2
raft2017/06/12 14:33:17 INFO: 2 [quorum:2] has received 2 MsgVoteResp votes and 0 vote rejections
raft2017/06/12 14:33:17 INFO: 2 became leader at term 2
raft2017/06/12 14:33:17 INFO: raft.node: 2 elected leader 2 at term 2
raft2017/06/12 14:33:17 INFO: raft.node: 3 elected leader 2 at term 2
raft2017/06/12 14:33:17 INFO: raft.node: 1 elected leader 2 at term 2
```

Store 5000 requests with batch size of 64

```
05:36:03 raftexample2 | 2017-06-12 05:36:03.840376 I | storing 5000 key-vals took 939.641223ms
05:36:03 raftexample1 | 2017-06-12 05:36:03.845578 I | storing 5000 key-vals took 942.141036ms
05:36:03 raftexample3 | 2017-06-12 05:36:03.855077 I | storing 5000 key-vals took 952.478853ms
```

Store 5000 requests with batch size of 128

```
05:37:53 raftexample3 | 2017-06-12 05:37:53.022803 I | storing 5000 key-vals took 1.375025944s
05:37:53 raftexample1 | 2017-06-12 05:37:53.033431 I | storing 5000 key-vals took 1.383080767s
05:37:53 raftexample2 | 2017-06-12 05:37:53.065981 I | storing 5000 key-vals took 1.415680606s
```

Store 5000 requests with batch size of 256

```
05:40:10 raftexample1 | 2017-06-12 05:40:10.126791 I | storing 5000 key-vals took 857.011171ms
05:40:10 raftexample3 | 2017-06-12 05:40:10.128630 I | storing 5000 key-vals took 856.341491ms
05:40:10 raftexample2 | 2017-06-12 05:40:10.150352 I | storing 5000 key-vals took 878.614888ms
```

Store 5000 requests with batch size of 512

```
05:41:38 raftexample3 | 2017-06-12 05:41:38.153212 I | storing 5000 key-vals took 1.635460575s
05:41:38 raftexample1 | 2017-06-12 05:41:38.162465 I | storing 5000 key-vals took 1.641334128s
05:41:38 raftexample2 | 2017-06-12 05:41:38.174407 I | storing 5000 key-vals took 1.65449608s
```

Store 5000 requests with batch size of 1024

```
05:43:08 raftexample3 | 2017-06-12 05:43:08.294799 I | storing 5000 key-vals took 1.050651742s
05:43:08 raftexample1 | 2017-06-12 05:43:08.299137 I | storing 5000 key-vals took 1.051665777s
05:43:08 raftexample2 | 2017-06-12 05:43:08.299631 I | storing 5000 key-vals took 1.052939761s
```

6.1.2 output for same request with batching and without batching

Store 1000 requests with batch

```
05:26:31 raftexample1 | 2017-06-12 05:26:31.992383 I | storing 1000 key-vals took 80.247934ms
05:26:31 raftexample2 | 2017-06-12 05:26:31.995054 I | storing 1000 key-vals took 81.103407ms
05:26:31 raftexample3 | 2017-06-12 05:26:31.998143 I | storing 1000 key-vals took 83.181628ms
```

Store 1000 requests without batch

```
05:12:08 raftexample3 | 2017-06-12 05:12:08.322672 I | storing 1000 key-vals took 1.705090354s
05:12:08 raftexample1 | 2017-06-12 05:12:08.323086 I | storing 1000 key-vals took 1.703652739s
05:12:08 raftexample2 | 2017-06-12 05:12:08.324449 I | storing 1000 key-vals took 1.705786053s
```

Store 2000 requests with batch

```
05:28:16 raftexample2 | 2017-06-12 05:28:16.154869 I | storing 2000 key-vals took 204.414879ms
05:28:16 raftexample1 | 2017-06-12 05:28:16.157906 I | storing 2000 key-vals took 204.968736ms
05:28:16 raftexample3 | 2017-06-12 05:28:16.165717 I | storing 2000 key-vals took 214.784658ms
```

Store 2000 requests without batch

```
05:15:57 raftexample3 | 2017-06-12 05:15:57.873707 I | storing 2000 key-vals took 3.720259209s
05:15:57 raftexample1 | 2017-06-12 05:15:57.873888 I | storing 2000 key-vals took 3.71920521s
05:15:57 raftexample2 | 2017-06-12 05:15:57.881530 I | storing 2000 key-vals took 3.727112285s
```

Store 3000 requests with batch

```
05:29:27 raftexample3 | 2017-06-12 05:29:27.216757 I | storing 3000 key-vals took 400.9326ms
05:29:27 raftexample1 | 2017-06-12 05:29:27.218408 I | storing 3000 key-vals took 400.375728ms
05:29:27 raftexample2 | 2017-06-12 05:29:27.241410 I | storing 3000 key-vals took 423.418645ms
```

Store 3000 requests without batch

```
05:17:48 raftexample1 | 2017-06-12 05:17:48.940392 I | storing 3000 key-vals took 5.420312744s
05:17:48 raftexample3 | 2017-06-12 05:17:48.940635 I | storing 3000 key-vals took 5.419657812s
05:17:48 raftexample2 | 2017-06-12 05:17:48.947103 I | storing 3000 key-vals took 5.425242355s
```

Store 4000 requests with batch

```
05:30:48 raftexample1 | 2017-06-12 05:30:48.717028 I | storing 4000 key-vals took 732.361757ms
05:30:48 raftexample3 | 2017-06-12 05:30:48.719866 I | storing 4000 key-vals took 734.082236ms
05:30:48 raftexample2 | 2017-06-12 05:30:48.727507 I | storing 4000 key-vals took 741.837768ms
```

Store 4000 requests without batch

```
05:19:54 raftexample1 | 2017-06-12 05:19:54.136550 I | storing 4000 key-vals took 7.122190093s
05:19:54 raftexample2 | 2017-06-12 05:19:54.136865 I | storing 4000 key-vals took 7.121617899s
05:19:54 raftexample3 | 2017-06-12 05:19:54.145024 I | storing 4000 key-vals took 7.128500357s
```

Store 5000 requests with batch

```
05:31:49 raftexample1 | 2017-06-12 05:31:49.863582 I | storing 5000 key-vals took 1.12605017s
05:31:49 raftexample2 | 2017-06-12 05:31:49.871323 I | storing 5000 key-vals took 1.130989486s
05:31:49 raftexample3 | 2017-06-12 05:31:49.872007 I | storing 5000 key-vals took 1.13078253s
```

Store 5000 requests without batch

```
05:21:35 raftexample1 | 2017-06-12 05:21:35.573175 I | storing 5000 key-vals took 9.025090902s
05:21:35 raftexample2 | 2017-06-12 05:21:35.573396 I | storing 5000 key-vals took 9.023368035s
05:21:35 raftexample3 | 2017-06-12 05:21:35.585159 I | storing 5000 key-vals took 9.03517988s
```

6.1.3 Output for different size of pipeline

Store 5000 requests without pipeline:

```
^@14:17:11 raftexample3 | 2017-06-12 14:17:11.240682 I | storing 5000 key-vals took 9.022396798s
14:17:11 raftexample2 | 2017-06-12 14:17:11.240845 I | storing 5000 key-vals took 9.021657624s
14:17:11 raftexample1 | 2017-06-12 14:17:11.254730 I | storing 5000 key-vals took 9.035498925s
```

Store 5000 request with pipeline size 2:

```
14:18:57 raftexample3 | 2017-06-12 14:18:57.053156 I | storing 5000 key-vals took 8.53310308s
14:18:57 raftexample2 | 2017-06-12 14:18:57.053673 I | storing 5000 key-vals took 8.531965004s
14:18:57 raftexample1 | 2017-06-12 14:18:57.059380 I | storing 5000 key-vals took 8.536946638s
```

Store 5000 request with pipeline size 4:

```
14:20:07 raftexample1 | 2017-06-12 14:20:07.983995 I | storing 5000 key-vals took 4.510029333s
14:20:07 raftexample2 | 2017-06-12 14:20:07.984901 I | storing 5000 key-vals took 4.510106121s
14:20:07 raftexample3 | 2017-06-12 14:20:07.985624 I | storing 5000 key-vals took 4.510760159s
```

Store 5000 request with pipeline size 8:

```
14:21:01 raftexample3 | 2017-06-12 14:21:01.683776 I | storing 5000 key-vals took 2.510828301s
14:21:01 raftexample2 | 2017-06-12 14:21:01.684866 I | storing 5000 key-vals took 2.509308159s
14:21:01 raftexample1 | 2017-06-12 14:21:01.687227 I | storing 5000 key-vals took 2.511744883s
```

Store 5000 request with pipeline size 16:

```
14:22:01 raftexample2 | 2017-06-12 14:22:01.223353 I | storing 5000 key-vals took 1.119255403s
14:22:01 raftexample3 | 2017-06-12 14:22:01.224804 I | storing 5000 key-vals took 1.119121001s
14:22:01 raftexample1 | 2017-06-12 14:22:01.225040 I | storing 5000 key-vals took 1.118290442s
```

Store 5000 request with pipeline size 32:

```
14:24:07 raftexample2 | 2017-06-12 14:24:07.416237 I | storing 5000 key-vals took 1.249166594s
14:24:07 raftexample1 | 2017-06-12 14:24:07.428811 I | storing 5000 key-vals took 1.260002052s
14:24:07 raftexample3 | 2017-06-12 14:24:07.436495 I | storing 5000 key-vals took 1.268513219s
```

Store 5000 request with pipeline size 64:

```
14:23:16 raftexample2 | 2017-06-12 14:23:16.133613 I | storing 5000 key-vals took 1.372869878s
14:23:16 raftexample3 | 2017-06-12 14:23:16.142248 I | storing 5000 key-vals took 1.379878245s
14:23:16 raftexample1 | 2017-06-12 14:23:16.161259 I | storing 5000 key-vals took 1.398006414s
```

6.1.4 output for multi-threading and single thread

6.1.4.1 mutilthread

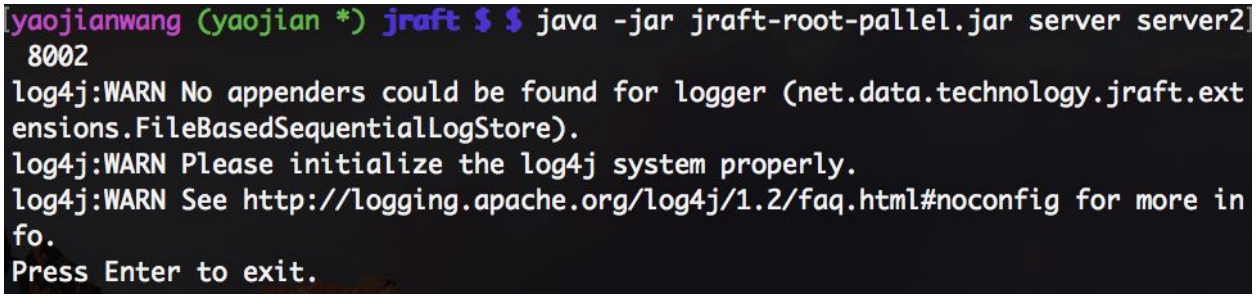
Start client

```
[yaojianwang (yaojian *) jraft $ java -jar jraft-root-parallel.jar client client ]
 log4j:WARN No appenders could be found for logger (net.data.technology.jraft.ext
 ensions.FileBasedSequentialLogStore).
 log4j:WARN Please initialize the log4j system properly.
 log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
 fo.
 Message:fmt:test for parallel
 How many?
```

Start server1

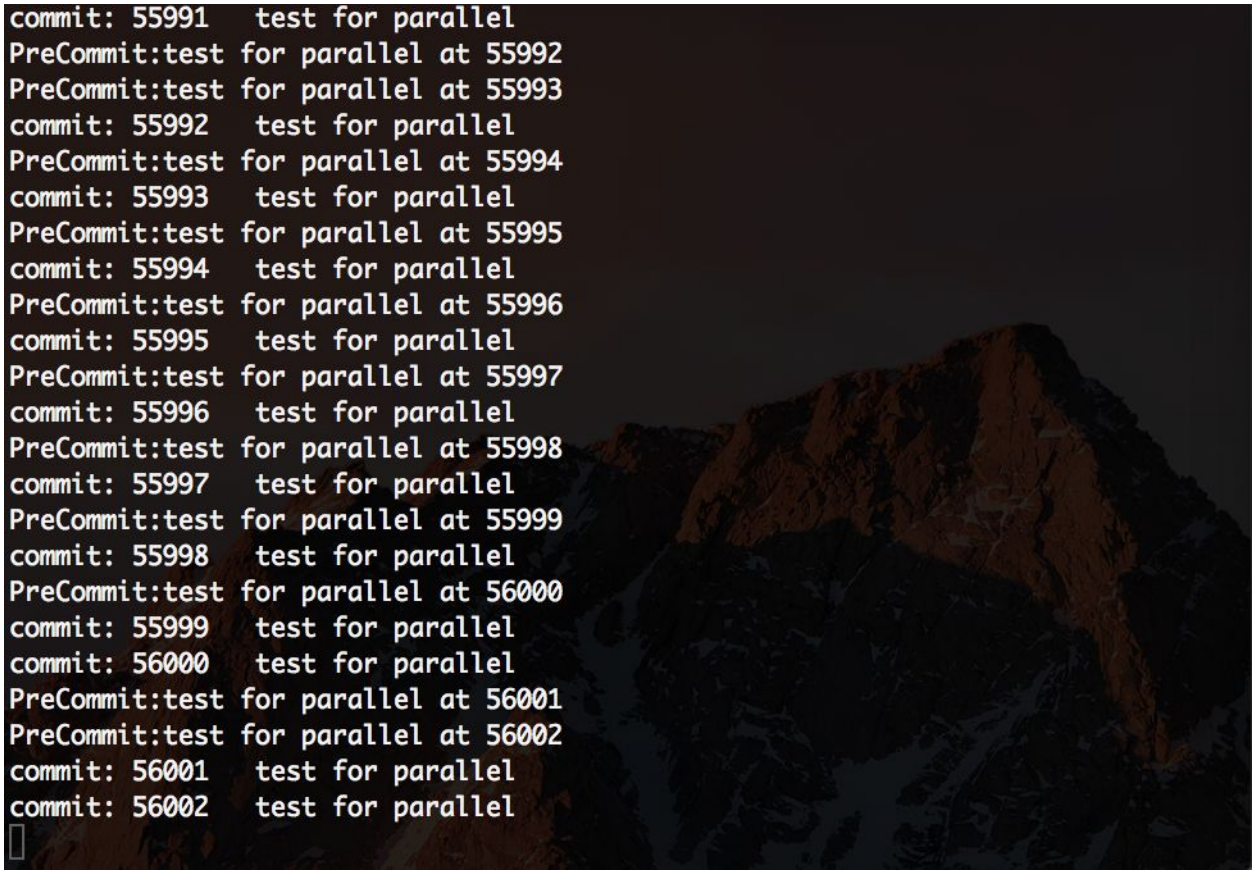
```
[yaojianwang (yaojian *) jraft $ java -jar jraft-root-parallel.jar server server1]
  8001
log4j:WARN No appenders could be found for logger (net.data.technology.jraft.ext
ensions.FileBasedSequentialLogStore).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more in
fo.
Press Enter to exit.
```

Start server2



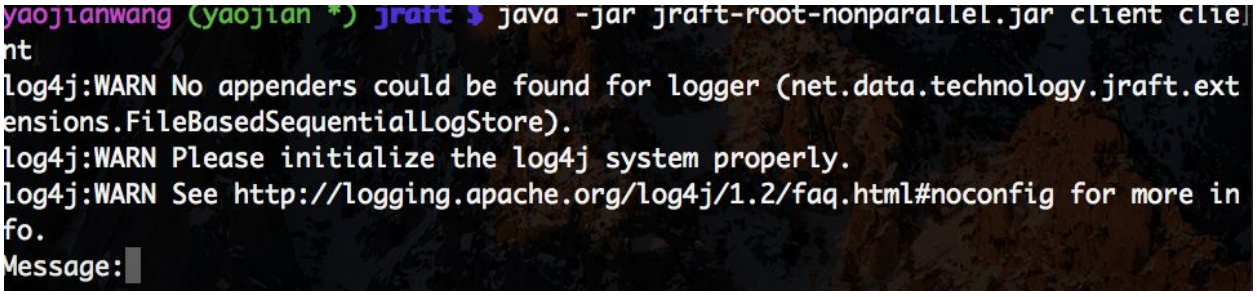
Start server 3

Result for client sending 10000 request

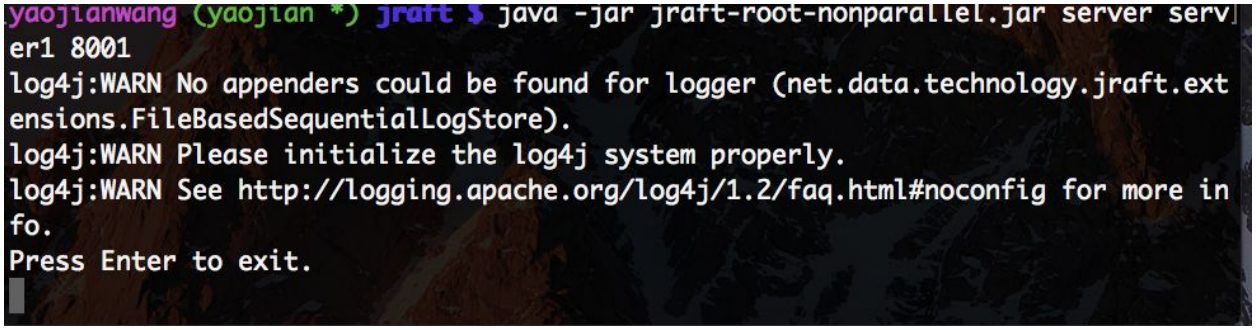


6.1.4.2 single thread

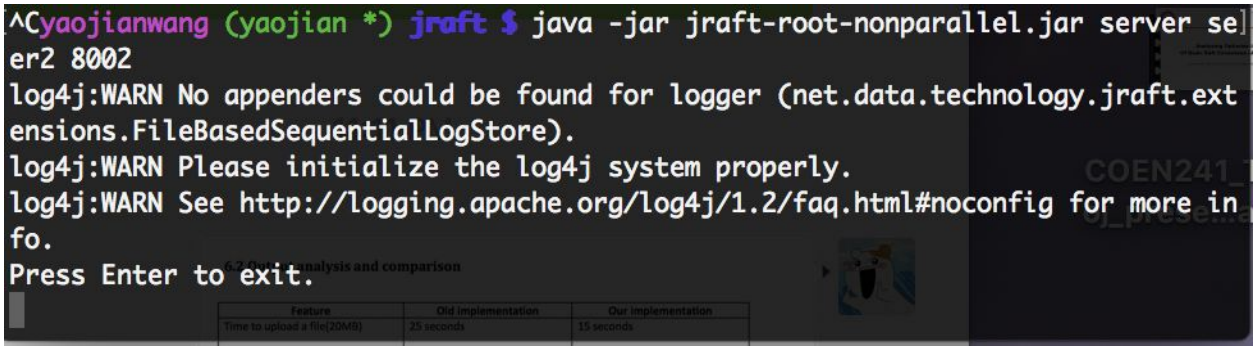
Start client



Start server 1

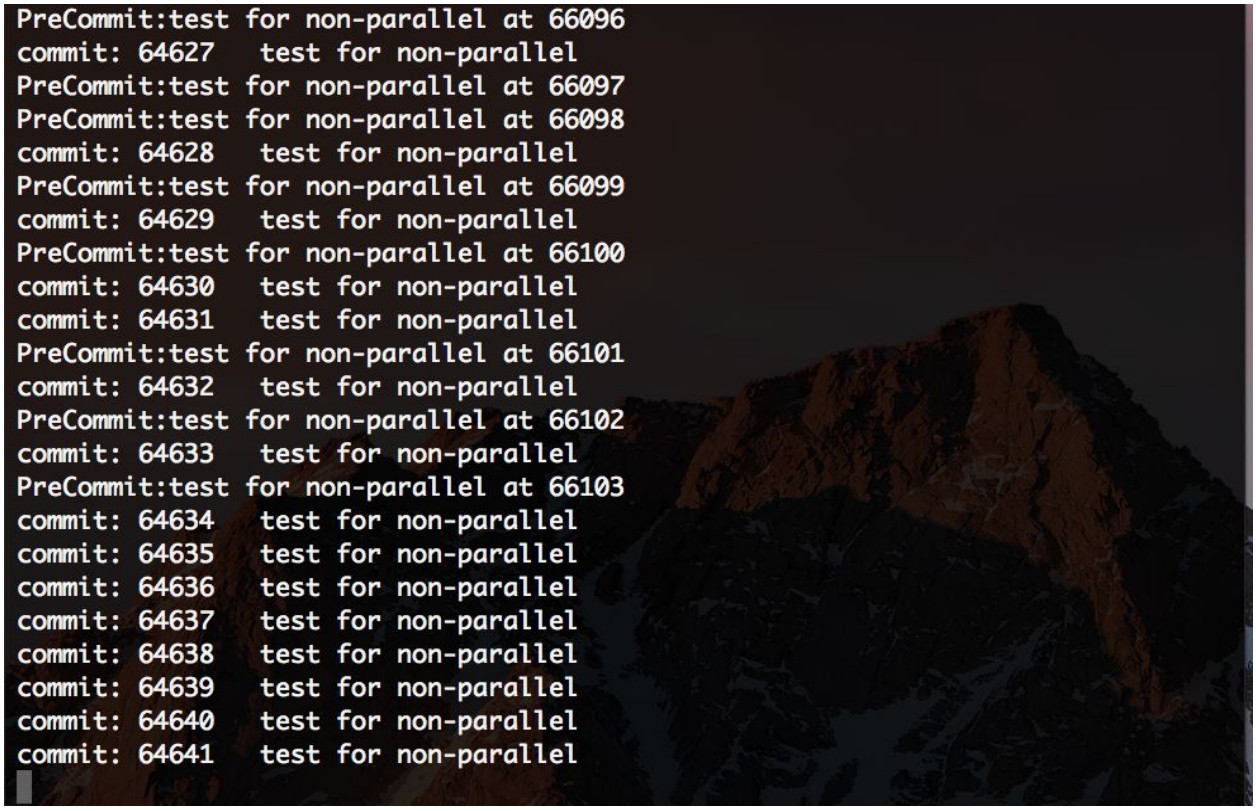


Start server2



Start server3

Result for sending 10000 requests



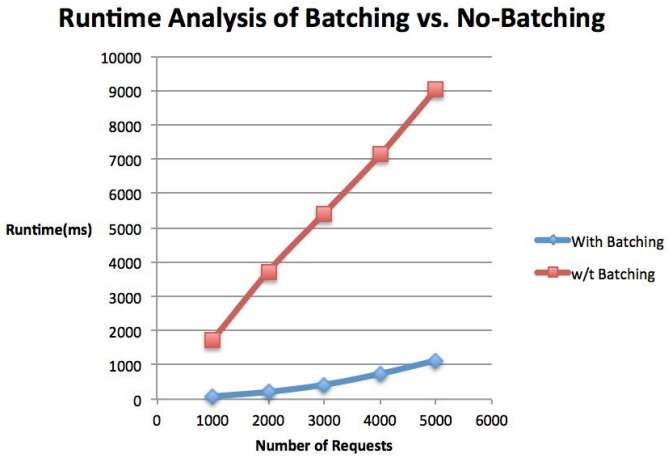
6.2 Output analysis

Runtime analysis of with_batching VS without_batching (Table 1 and Graph 1):

The table one and graph one show the benefit of batching. The X-axis of graph one represents the number of requests and the Y-axis of graph one represents the running time. As we can see in the graph, with the number of requests increasing, the running time of with batching and without batching is also increasing. And the running time of with batching is much smaller than without batching. What's more, the increasing of rate of running time of with-batching is much slower than without batching. For the test, we first set the batching size to be 128;

		Batching Analysis		
With Batching			w/t Batching	
Num of Req	Runtime(ms)		Num of Req	Runtime(ms)
1000	83.182		1000	1705.791
2000	214.785		2000	3727.118
3000	423.419		3000	5425.242
4000	741.838		4000	7128.524
5000	1131.435		5000	9035.188

Table 1. Runtime data of with batching VS without batching tests



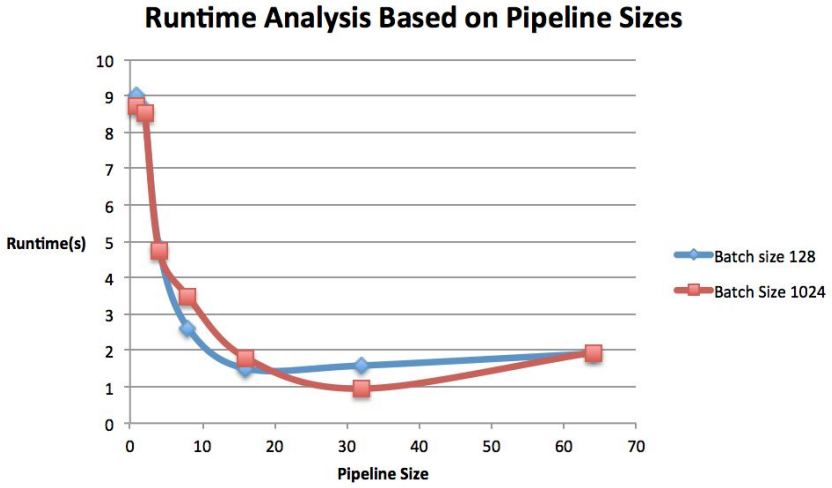
Graph 1. Runtime Analysis of with batching VS without batching tests

Runtime Analysis of Pipeline based on its different size (Table 2 and Graph 2):

The table two and graph two show the comparison of with and without pipeline. When the pipeline window size is 1, which means no pipeline at all. And then we start pipeline size from 2 to 64. From this graph, we can see the running time drops dramatically after using pipeline. Because when the next component finish the current batch, the next batch is already transmitted to server. But with the growth of pipeline size, we don't see a huge improvement in running time, that's because the bottleneck is on the server side;

Pipeline Analysis			
Batch Size 128		Batch Size 1024	
Pipeline Size	Runtime(s)	Pipeline Size	Runtime(s)
1	9.02324	1	8.71864
2	8.59863	2	8.54675
4	4.84509	4	4.75321
8	2.62373	8	3.48294
16	1.50313	16	1.79849
32	1.58418	32	0.95233
64	1.91497	64	1.94147

Table 2. Runtime data with Pipeline and without Pipeline



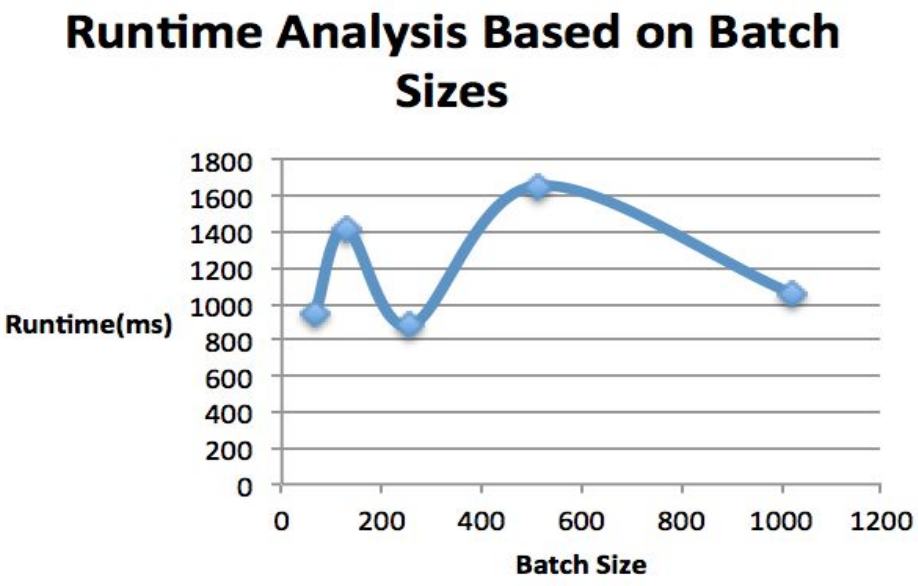
Graph 2. Runtime Analysis with Pipeline and without Pipeline

Runtime Analysis of Pipeline based on batching size (Table 3 and Graph 3):

The Table three and Graph three show that the running time changes randomly based on different batching size. The X-axis of graph three represents the size of batch and The Y-axis of graph three represents the running time. And at first we fix the batch size to 5000.

Batch Size Runtime Analysis	
Batch Size	Runtime(ms)
64	952.48
128	1415.68
256	878.61
512	1654.49
1024	1052.94

Table 3. Runtime Analysis based on batch size



Graph 3. Runtime Analysis based on batch size

Runtime Analysis based on parallelism with different requests size (Table 4 and Graph 4):

According to the table, we tried to compare the time complexity of different requests. We increase the number of requests sequentially with 1000 requests each time.

When Non-Parallelism is applied, when the number is increasing, the runtime is increasing which approximately increase polynomially. When the number of request reaches 5000, it takes

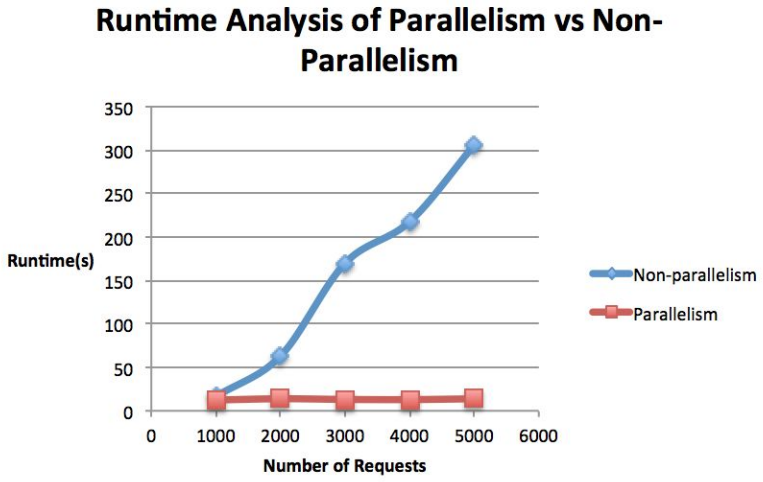
more than 306 seconds, which is more than 5 minutes. The clients definitely feel inefficient about the system.

However, parallelism is applied, the runtime nearly remain unchanged. Because the server has multiple CPU, which can handle the requests parallelly. So, multiple requests can be calculated concurrently. When the leader commit the requests, a parallel thread in server can handle the commitment concurrently, in the meanwhile, the leader handle handle other requests.

Because the physical machine of the leader has multiple CPUs, commitment and handling request can be handled concurrently. This is the main cause that parallelism can improve the performance of leader.

Non-Parallelism		Parallelism	
Num of Req.	Runtime(s)	Num of Req.	Runtime(s)
1000	16.192	1000	12.374
2000	63.17	2000	13.915
3000	169.984	3000	13.117
4000	217.972	4000	12.933
5000	306.233	5000	13.859

Table 4. Runtime Analysis of Non-parallelism vs Parallelism



Graph 4. Runtime Analysis of Non-parallelism vs Parallelism

6.3 Compare output against hypothesis

From our test result, we can tell that runtime will be higher with more requests based on the same batch size. Generally, runtime will be smaller with the bigger batch size. It will save much time when we use multi-thread to implement the commit method.

6.4 Abnormal case explanation (the most important task)

For the same number of request, we tested different batch size. But it seems that there is not a pattern about the relation between batch size and runtime. We think it caused by the heartbeat of raft server. It will generate a random time to send the request to its follower. So, sometimes the runtime will be higher with a big batch size. Otherwise, it's also related to the parameter time_out. For example, batch size is 1000, but it only get 500 requests. The server will wait for another 500 requests. Or it will stop waiting until time is out. So, it's also an important factor about the runtime.

7. Conclusions and recommendations

7.1 Summary and conclusions

Batching is capable of significantly improving the throughputs of the system when average request latency is not critical;

Building pipeline allows a system to serialize different components and to get resources ready for following components;

Asynchronization is the best way to leverage multi-core computers;

A system is able to eliminate synchronizations by utilizing parallelism in execution path.

7.2 Recommendations for future studies

The limited computing ability of a server is the main cause of Pipeline's bottleneck, therefore, the resolutions could be scaling up the server either vertically or horizontally.

8. Bibliography

[1] In search of an understandable consensus algorithm. Ongaro, Diego, & Ousterhout (2014) USENIX Annual Technical Conference (USENIX ATC 14).

[2] ARC: analysis of Raft consensus. Howard (2014). Technical Report UCAM-CL-TR-857.

[3] Consensus in the Cloud: Paxos Systems Demystified Ailijiang, Charapko, & Demirbas (2016). The 25th International Conference on Computer Communication and Networks (ICCCN).

[4] Paxos made live: an engineering perspective. Chandra, Griesemer, & Redstone (2007). Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing.

[5] Ongaro, Diego. *Consensus: Bridging theory and practice*. Diss. Stanford University, 2014.

[6] https://github.com/coreos/etcd/tree/master/raft

[7] https://github.com/datatechnology/jraft