Multicore Programming - Final Projcet

Kevin Chun-Hao Yang, Wei-Kai Pan

1. Introduction - Wei-Kai Pan

The consensus algorithm aims to help replicated systems agree on system logs and values. This property is important for distributed system because servers in the system are prone to failure. The reasons behind the failures include hardware crashes, network error, and software bugs. A reliable and fault-tolerant distributed system has to be able to survive and recover from machine failures in a collection of servers. To recover from failure and keep the service available, the system saves command logs as replications across multiple servers. Once a server crashed, other server takes over and uses its replication to restore system states and continue the service.

The replicated logs are all committed in the same execution order across all servers during the system is running. In order to achieve the consistency and coherency of replicated logs during the system is running, consensus algorithm plays a role of synchronizing replicated logs on machines. Raft is one of the consensus algorithm that allows servers agree on the state value such that the system is active as long as the majority servers are up and ensure systems correctness. It also is designed to be more understandable than another consensus algorithm, for example, Paxos.

In this scenario, a system consists of multiple servers and there are clients who issue commands to the system. Each server has one state machine which is responsible for executing commands based on the logs. In order to keep all servers with the same replications and minimize the loss of state, servers need a protocol to synchronize logs and maintain states with each other. Raft consensus algorithm is a protocol that allows the system to survive from failures of systems members.

Raft elects the leader to manage the replicated logs. This leader replicates log entries from clients and sends them across servers. Logs are guaranteed being sent from the leader to followers using RPCs and this is the only flow of logs. This policy is called strong leader.

Raft algorithm decomposes the consensus problem into three components: leader election, log replication, and safety. Before going to these details, we introduce some terminologies here.

- Heartbeat: Heartbeat is a mechanism for checking the leader of system is alive. The leader sends RPCs with empty log entries periodically to followers indicating that it is alive.
- Term: Time is divided into terms. For each term, it has election time, then normal operation. Note that if the leader couldn't be elected, it is called split vote and the system will re-elect again.

2. Leader Election - Wei-Kai Pan

Raft, unlike other consensus algorithm, takes the leader-based approach to achieve consensus. In the Raft algorithm, servers have three states: Leader, Follower and Candidate. The leader handles all clients requests and sends heartbeat to notify followers that the leader is still alive. The algorithm also guarantees there is only one leader in each term. Followers only accept incoming RPCs. When a server enters the Candidate state, it is eligible to be voted as the leader of next term.

During the beginning of each term, the servers vote for a leader by holding an election. Each server votes for itself first. After a server is elected, the system proceeds normal operations under the leadership of it. During the operation, severs may encounter network failure or machine crashes. These result in an election timeout, meaning followers do not receive heartbeat from the leader. In this case, followers assume that no viable leader presents in the system. If a follower is down, the system operates normally. But if the leader is down, following things happen:

- The leader stops sending heartbeat to followers.
- Followers detects the election timeout, it transits to candidate state.
- The term counter is incremented representing a new term.
- Candidates request votes from others using RPCs. The follow things happen.

- Once a candidate receives majority of votes, it is the leader of this new term.
- The leader is elected, it transits to follower state.
- No one wins, then start a new term again.

The election must satisfy two invariants: Safety and Liveness. Safety means only one leader is elected during in a term. To ensure Safety, a server can only vote for one machine. This vote would be saved in the disk in case that the server crashes after voting, and recovers during the same term then votes to another server. Liveness means the leader must be selected during each term. It is possible that candidates cannot receive majority of votes. The system restarts the election to resolve this, but is still possible that no leader is elected indefinitely. Raft avoids this situation by randomizing election timeout for servers so that all servers will not start election then time out at the same time.

3. Log Replicate - Kevin Chun-Hao Yang

The main goal for log replication is to replicate leaders log to follows log and maintains the consistency between leaders log and followers log. To maintain such consistency, RAFT introduce a special log structure, entry, each with term number and command and identified by index. If two entries in different log have same index and term, they store the same command. Also, if those two entries are identical, all the preceding entries must be identical.

To guarantee the two consistent properties, leader creates at most one entry with a given index in a given term and leaders log is immutable. In addition, when sending an entry, leader includes term and index information so that followers can use such information to determine whether they want to accept the entry.

In beginning of the replication process, a client sends a leader a request(command) and the leader appends such request as an entry to its log. After that, the leader starts to replicate(send) such entry to the followers through remote procedure call (RPC). When a follower receives the entry, it compares the term and index information in the entry with that in its current log. If the information from the receiving entry is not identical to that in the followers log, the follower rejects the entry, or the follower accepts the it.

Once the majority of the followers receive and accept the entry, the leader commits the entry and apply it to the state machine (execute the entry). A committed entry is guaranteed to be durable and will be executed. Also, the preceding entries are all committed.

During the replication procedure, the leader handles the inconsistence by forcing the followers log to be identical to its own. If a follower keeps accepting entries sent, the leader and the follow will eventually have same the log. If a follower rejects an entry, the leader retries by sending the entry preceding the rejected one until the follower accepts it. After that, leader deletes all the inconsistent entries in the followers log and replaces the log with its own log. Note that leader might delete all the entries in a followers log if the follower does not accept any entry.

4. Safety - Kevin Chun-Hao Yang

4.1. Safety requirement

RAFT has the following safety requirement: if an entry is committed by a leader, such entry will be presented in the log of all future leader. This requirement guarantees that leaders never overwrite in their log (only appends), only entries in leaders log can be committed, and entries must be committed before applying to state machine.

4.2. Leader change

To fulfill the requirement above, RAFT puts restriction on leader election, that only a candidate with all committed entries can become a leader, and RAFT also delays committing an entry until it is safe to do so. To be specific, during the election a candidate is considered the most ideal leader if it has the entry with the latest term and latest index. Thus, candidate who wins the election should have the most complete log among the electing majority. On the other hand, after a new leader takes over, it is considered safe to commit the first entry only if one new entry with the leaders latest term is stored in the majority of the followers.

4.3. Deal with old leader

Under the scenario that old leader disconnects with the cluster and reconnects back, the old leader is likely to remain acting as a leader, responding request and sending heartbeat, since it is not aware of the existence of the new leader. To solve this issue, term number is contained in every RPC sent

by a leader. If term number in RPC is older than the current one in receivers log, such RPC is rejected and the sender reverts to follower and update its term number. If current term number of the receiver is older than RPCs, the receiver reverts to follower (from, perhaps candidate) and update its term number. Such solution guarantees that a disposed server (follower or leader) cannot commit a new entry unless it updates its term number, since majority of servers have up-to-date term information and a disposed server needs to at least communicate with one of the servers to commit a new entry.

4.4. Client protocol

Normally, clients send command to a leader, or to a follower and then redirect to a leader. The leader then responds such clients after the command has logged, committed, and executed by the leaders state machine. However, such leader might be dead before it responds to the client but after it executes the command. In this case, the request would reach the timeout, clients would resend the same request, and such command would be executed twice. To solve this issue, RAFT adds a unique ID to every command. Before a leader executes a command, it checks whether its log contains such ID. If so, the leader just returns the response rather than execute such command again, avoiding duplicate execution.

4.5. Configuration change

RAFT uses the system configuration to define crucial information such as number of majority. However, during the migration from old configuration to a new one, RAFT could form two majority group within one cluster, leading to fetal issue. For example, if cluster C1 contains Server1(S1), S2 and S3, and S1 and S2 form a majority group at a given time. However, in this moment the configuration of C1 changes, S4 and S5 join the cluster, and S3, S4, and S5 also form majority group in C1. To resolve the problem, RAFT leverages two phase migration. In phase one, RAFT can reach the consensus only if such consensus is agreed by both the majority of the old configuration and that of the new one. In our example, it should be agreed by the majority among S1, S2, and S3, and that among S1, S2, S3, S4, and S5. In phase two, however, it can reach the consensus just with new configuration. In addition, if the configuration of the current leader remains old after the new configuration commits, such leader would step down.

5. Insights - Kevin Chun-Hao Yang

Leveraging the power of multithreads, developers use Mutex, CAS, Conditional Variable, and Semaphore to overcome data race. Some programming languages even provide more sophisticated parallel programming model such as Monitor, Transactional Memory, and Actor to not only resolve data race but also deadlock and starvation.

RAFT leads to another interesting idea of multithreads paradigm that each thread has its own memory space and there is no shared space between threads. The benefit not to have shared space is to resolve data race. However, we still want threads to communicate with each other. That is where RAFT come in to picture. Unlike Actor, RAFT is not susceptible to deadlock according to its state-transition design. Also, it does not need further information about memory as Transactional Memory model does. Therefore, unlike Transactional Memory, the overhead for overcoming data race does not increase as number of the contentions and variables increases.

Also, leveraging RAFT in multithreads programming has advantage that each thread can execute a given job in detached fashion so that if one thread exits abnormally, it will not drag the entire program to hell. During the computation, if threads exit unexpectedly, the system can just produce a new one to replace it. These characteristics are particularly good for task that requires substantial computation time and fault tolerance.

A real-world use case is a simulator built to test the capacity of an access points controller. During my internship, our team builds the simulator that can produce 5000 thousand fake access points, each generating traffics to attack the access points controller which is the main product of the company. Specification given by Quality Assurance Team is that the simulation should have 5000 functional access points and should last for two weeks. In this case, the simulator needs to maintain 5000 thousand access points(threads) in detached fashion so that the entire simulation can still sustain when access points crash. Also, the simulator should be able to collect data and produce analytic report as the result of simulation. This scenario perfectly matches the RAFT model mentioned above since it can provide fault tolerance to support the need of relatively long-term simulations.

6. Insights - Wei-Kai Pan

The problem that Raft solved is when a server is down, then the service continues running correctly. Because servers have independent hardware,

they can only communicate with each other through channels and RPCs. In this way, all machines agree on their committed replicated logs through this communication protocol. Now, leveling down servers from machines to processes, the system is as below:

- A set of processes (act like servers)
- Communicate with each other through message queue
- Each process hold its own resources in their local memory
- Processes are running some codes that may easily crash them
- All processes hold its own replica and command log entries.

To achieve consensus, we can use message passing with Raft algorithm. The advantage of this method is the state is kept in each process. There is no synchronization overhead and risks of data races, deadlock (Raft guarantees liveness), and potential synchronization errors. Another advantage is we can avoid network error by putting message queue in shared memory. Therefore the communication between processes is actually through shared memory.

The above approach needs to keep replications of log entries for each process. This is memory costly and since the processes are on the same machine, we can keep log entries in a shared memory section. This way, there is no message passing overhead and memory access is actually faster than message queue. But in this way, programmers have to design synchronization mechanisms and facing the risk of synchronization errors such as deadlock.

My conclusion is putting services into a single machine and applying Raft to achieve consensus can avoid the risk of network failure by replacing network channels with in-memory message queue and still keeps the strength of Raft. Though keeping one shared log entries saves much memory space, programmers need to design synchronization mechanisms for this complicated system which could be very difficult to guarantee the correctness of the system.