RAFT Cluster Reconfiguration Managed by Trusted Registry

Zixi An, Danny Lee, Jingchi Ma

Stanford University

anzixi@stanford.edu
dglee@stanford.edu
jingchi@stanford.edu

***Abstract*— In a RAFT cluster operating at high capacity, a cluster configuration change could affect latency and throughput of client requests made to the system. This is because all configuration change requests are processed by the leader, and propagated to follower nodes in the same fashion as appending log entries. As such, extra complexities are added to the system to ensure the safety of reconfigurations, preventing multiple leaders in a single term. Furthermore, the original RAFT paper provides no method to automatically detect node failures. To reduce the extra overhead of handling configuration changes, and to automatically detect and remove failed nodes from the cluster, we proposed and implemented a new method to manage RAFT cluster membership, using a trusted registry service.**

***Keywords*— RAFT, 2 Phase Commit, Cluster Failure Detection, Trusted Registry**

I. Overview

The RAFT consensus algorithm [2] is gradually overtaking Paxos as the preferred option both in the classroom and on deployment servers because of its relative simplicity, safety, and availability. By design, RAFT offers high availability despite node failures, internal leader change, temporary network partition, and even cluster configuration changes. Certain systems go offline when adding/removing active nodes [1], but RAFT remains operational by entering a special transitional state during reconfigurations [2], while continuing to respond to AppendEntries client requests.

By default, when active nodes change from cluster C1 to cluster C2, the leader appends a special "C1/C2" log entry and broadcasts to followers through AppendEntries RPC. Upon receipt, nodes enter a "joint consensus" state where committing new log entries or electing new leaders require majorities from both C1 and C2. Once the special log entry is committed, the leader will broadcast a second special log entry, "C2" to signal the end of the joint consensus state. This is analogous to the 2 phase commit protocol, where 2 special change-log entries represent the "prepare" and "commit" phases. However, when the cluster is under heavy load, each node has many concurrent AppendEntries RPCs to process, and the special change-log AppendEntries requests might experience lock starvation or other delays due to constraints on nodes' processing power. Furthermore, if C1 and C2 are disjoint, log entries added during joint consensus requires twice as many RPCs to commit, leading to heavier network and CPU load on the leader.

In his dissertation, Ongaro proposed a simplified method for configuration change, by adding the **addServer** and **removeServer** RPCs to the cluster [3]. This allows nodes to use the new configuration as soon as it is received, but imposes the constraint that only a single node may be changed at a time. It writes a change-log record, and relies on **AppendEntries** RPC to propagate to the cluster.

With either method, the cluster configuration change in RAFT is always triggered by an administrator, and there's no built-in automatic failure detection. As a result, a majority of node failures undetected by administrators could indefinitely stall the system.

To simplify and separate the config change process from the log replication process, and to automatically detect node failures, we've implemented a registry service with a mechanism similar to two-phase commit [4] that coordinates reconfiguration without adding change-log entries and leader broadcasts, and to detect failed nodes using a simple heartbeat mechanism. This removes the need for joint consensus state, reduces reconfiguration overhead, and prevents undetected failures from stalling the cluster.

II. Registry model

The registry is modeled as a service providing two functions: coordinating RAFT cluster

reconfiguration, and sending periodic heartbeats to nodes for failure detection. It is separate from the RAFT cluster and is run on a separate server. System administrators interact with the registry service through 3 public APIs: **updateGroup, addSingleMember, removeSingleMember.**

updateGroup(newMembers):

This API takes in the new cluster configuration, updates the current cluster members with a 2 phase commit mechanism, and terminates once the entire cluster acknowledges the new configuration.

addSingleMember(nodeId, url):

This serves as the registry wrapper for the cluster's **addServer** method.

removeSingleMember(nodeId):

This serves as the registry wrapper for the cluster's **removeServer** method.

The **updateGroup** is designed to provide a similar safety guarantee as the original joint consensus mechanism, without interfering with RAFT's log replication process. However, if reconfiguration only involves a single node change, using RAFT's **addServer/removeServer** is preferable, so they are included as registry wrappers.

III. Registry coordinated reconfiguration

When **updateGroup** is invoked, the registry serves as the coordinator of the 2 phase commit process to communicate the new configuration to all cluster nodes. The following changes need to be made to RAFT nodes:

prepareCommit RPC

args
newGroup: cluster nodes in the new configuration
timestamp: the unix epoch the configuration request is made

result
commit: whether the node can commit the new configuration

Receiver implementation
1. If the timestamp argument is less than the **membershipChangeTimestamp** of the node, returns false.
2. If the timestamp argument of the request is greater than the **membershipChangeTimestamp** of the node, update the **membershipChangeTimestamp** and the **pendingMembers** with the newGroup argument.
3. If the current node is not included in **pendingMembers**, become a non-voting member that never participates in elections. If any nodes in the current cluster is not included in **pendingMembers,** make them non-voting members.

Figure 1: prepareCommit RPC

commit RPC

args
timestamp: the unix epoch associated with the configuration change

result
ack: acknowledges the new configuration

Receiver implementation
1. If the timestamp argument is different from the **membershipChangeTimestamp** of the node, return false.
2. Updates the node's active cluster members with the **pendingMembers**, and reset **pendingMembers**.
3. If current node is leader, bring all newly added nodes' log up to date.

Figure 2: commit RPC

Additional State

membershipChangeTimestamp: the timestamp of the latest uncommitted configuration change request.
pendingMembers: the new cluster members of the latest uncommitted configuration change request.

Figure 3: additional states

Additional Rule

membershipChangeTimestamp will be included in **AppendEntreis** and **RequestVote** RPCs , if it doesn't match with the receiver's **membershipChangeTimestamp**, fail the request and returns the current cluster configuration for sender to update itself.

Figure 4: additional rules

During the prepare phase, the registry sends **prepareCommit** to every node in the old, and new configuration. When every node replies with a **yes** vote, the registry sends the **commit** and marks the config change as complete, responding to the original

updateGroup request immediately. The **prepareCommit** and **commit** RPCs ensure the atomicity of configuration changes by providing the following guarantees once an **updateGroup** request comes back successful:

1. Every node either has their **pendingMembers,** or active members set to the new configuration.
2. Only nodes in the new configuration may participate in elections and voting.

Let C1 be the set of nodes in the old configuration, C2 be the set of nodes in the new configuration. The figure below describes the key states of a updateGroup request:

t0: registry sends prepareCommit to all nodes

t1: registry receives first prepareCommit vote from RAFT node

t2: registry receives all prepareCommit vote, and sends commit to all nodes, returns updateGroup request as true

t3: registry receives first ack from commit

t4: registry receives acks from all nodes.

Because of the **prepareCommit** and **commit** implementation, C2 - C1 won't participate in voting or election before t2 (1), and C1 - C2 won't participate in voting or election after t2 (2). The "-" notation represents a set difference. For duplicate leaders to be elected, there must exist a time where node_i sees C1' as cluster configuration, node_j sees C2' as cluster configuration, such that |C1'- C2'| > |C1'| / 2 (3) and |C2'- C1'| > |C2'| / 2 (4) hold true. Only then could node_i and node_j both receive majority vote in an election, from their respective clusters. Given observations (1) and (2), we know that once a server receives the prepareCommit message, it will not vote for a stale node (present in the old config, not the new config), and any stale node will not participate in an election. As a result, (3) and (4) will never hold, thus our registry coordinated **updateGroup** safely performs configuration change without taking the cluster offline.

If a node fails after sending the **prepareCommit** response, but before receiving **commit** from the registry, and recovers after every other node receives the commit, its cluster configuration may become out of sync with the other active nodes. To eliminate this issue, the additional rule in Figure 4 guarantees that nodes with stale cluster information are immediately updated once they come in contact with the current cluster.

Aside from the safety guarantee, **updateGroup** adds very little overhead to the leader, does not require a complex joint consensus mechanism, and is not constrained to add/remove one node at a time. Admittedly, with joint consensus, the config update commits after a majority of cluster members respond, whereas **updateGroup** requires **all** cluster members to respond, we believe the modularity and simplicity are fair trade-offs to the potentially higher tail latencies.

Finally, although we modeled the registry service to be always available and partition resistant, it can, and will fail. When the registry is unavailable, administrators can easily switch to RAFT's own **addServer/removeServer** RPCs to update cluster configurations.

IV. Automatic failure detection

The registry also monitors the health of active RAFT nodes, and automatically removes nodes from the cluster after a period of inactivity. We've implemented a simple heartbeat mechanism as follows:

```
pingCluster
every T_heartbeat do:
for nodeId in active_nodes:
   ping nodeId
   if ping failed:
      consequtive_failure_count[nodeId]++
      if consequtive_failure_count[nodeId] > failure_threshold:
        mark nodeId inactive, remove from cluster
   if ping succeeded:
      consequtive_failure_count[nodeId]=0
```

Figure 5: pingCluster protocol

Each RAFT receiver node responds to the cluster ping with a simple ACK, without checking any internal state, or acquiring any locks. The **T_heartbeat** parameter determines the duration between each heartbeat ping, and **failure_threshold** determines the number of consecutive pings a node can miss before being marked inactive by the registry. In our implementation, we used fixed values

for these parameters. We've set **T_heartbeat** to the average reboot time of a cluster node (30s) and **failure_threshold = 2.**

We need to avoid doing reconfigurations too often because they are costly and can potentially delay normal operations. Setting the threshold to be too small will cause noises like high load and server reboot to trigger reconfigurations. As a result, the threshold needs to be greater than the average reboot time to accommodate normal server maintenance, and much greater than the normal network round trip time between servers and registries. In general, it is difficult to figure out the best value for every network environment; we decided that we would rather detect server failures with a higher latency than have to do the reconfiguration for the whole cluster over and over again.

To summarize, the heartbeat mechanism is a convenient and cheap method for us to detect server failures and start cluster-wide reconfigurations, although it does incur some overhead in raft servers and is sometimes vulnerable to failure noises. We decide that the benefits outweigh those potential drawbacks.

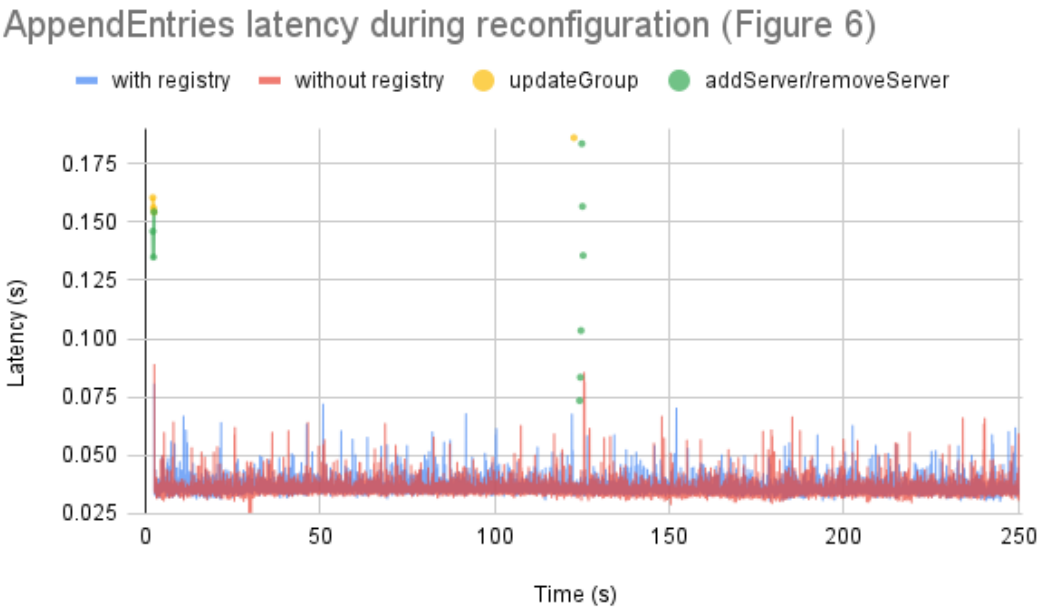
V. Evaluation

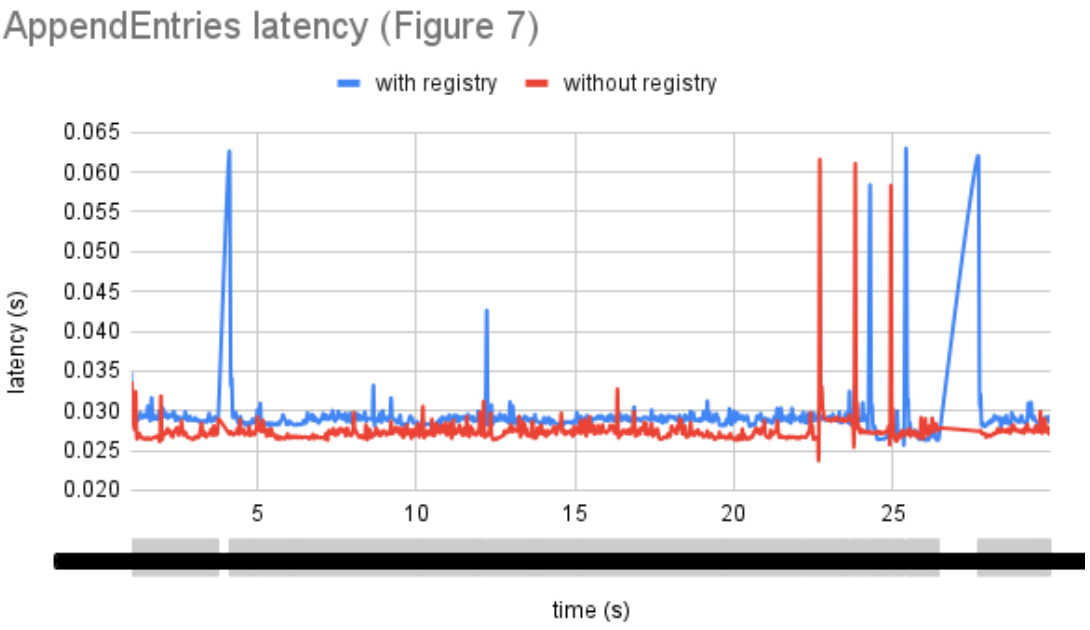
We implemented the RAFT algorithm with the **addServer/removeServer** RPCs, and implemented the registry service described in this paper. We picked Golang for its popularity with distributed systems and ease of use with multithreading. For our experiments, we deployed RAFT nodes to 7 AWS EC2 instances with 4 CPUs and 16 G memory. The registry service was deployed to a single EC2 instance with 1 CPU and 1 G memory. In our testing and evaluations, we focused on 2 aspects of the registry service:

1. How registry based reconfiguration (**updateGroup**) affects normal **AppendEntries** processing, compared to RAFT's **addServer/removeServer** config change method.
2. How much overhead does the heartbeat mechanism add during normal RAFT operations.

To measure how **ApendEntries** is affected under different cluster configuration change protocols, we ran a "complete reconfiguration" scenario, where we initialized the cluster with nodes 1, 2, and 3, then reconfigured the cluster to include nodes 4, 5, and 6. The change occurred after 2 minutes of sustained **AppendEntries** client requests, and the requests continued for 2 minutes after reconfiguration. We logged the latency of each **AppendEntries** request, as well as the latencies of configuration change requests, and present them in Figure 6. The blue line represents the **AppendEntries** latency when registry is turned on, and **updateGroup** is used to perform the configuration change. The red line represents the latency when only **addServer/removeServer** are used to perform the config change. As we can observe, a single **updateGroup** request has higher latency (0.179s) than **addServer/removeServer** requests (avg 0.11s), ranging from 75ms to 178ms. However, in this "complete reconfiguration" scenario, 6 consecutive **addServer/removerServer** requests are needed, compared to the one **updateGroup** request to complete the config change. Without the registry, the transitional period from old to new configuration spans over 0.7 seconds, significantly higher than the 0.18 seconds with the registry. After the configuration change is complete, we can observe a higher **AppendEntries** latency spike in the "no registry" case. This behavior could be attributed to extra leader elections and cluster instability after successively adding and removing single nodes, and the extra load placed on the leader for committing change-logs.

Overall, the **updateGroup** request is slower than a single **addServer/removeServer** request, but it removes the need to issue multiple **addServer/removeServer** requests for total configuration change, reducing the total transitional duration. Coupled with the fact that it removes the complex joint consensus mechanism, and the need to keep change-logs, we believe the registry's **updateGroup** method to be valuable.





The average **AppendEntries** latencies with the registry (36.4ms) is slightly higher than that without the registry (35.7ms). This 2% difference accounts for the extra overhead of the registry's heartbeat mechanism on the RAFT cluster, and is further analyzed.

To measure registry's overhead on normal operations, we simply initialized a cluster with all 7 nodes without the registry, continuously issued concurrent client **AppendEntreis** requests to simulate heavy load. We repeated the run with the registry turned on, and recorded the request latencies in both cases. The latencies in both test cases are presented in Figure 7. Note that the average latencies in Figure 7 are consistently lower than those of Figure 6, this is because the second experiment is run on a client instance with better network connectivity and closer to the cluster's physical location (us-west-2).

From the data, we observe a small, but clear difference in client request latencie. When the heartbeat mechanism is turned on, the average latency of 28.9ms is 5.1% higher than when registry is turned off (27.5ms). In addition, we observe slightly more latency spikes with the heartbeat overhead, potentially triggered by extra leader elections and cluster instability.

The evidence suggests that there is a non-trivial overhead associated with the heartbeat mechanism, because its periodic pings add extra load to nodes' network IO, competing with client requests and internal RPCs. Under heavy load, this network overhead could delay the leader from sending **AppendEntries** to followers long enough to trigger unwanted elections. However, the registry heartbeat's overhead can be reduced by tuning the **T_heartbeat** parameter, and optimizing the transport protocols to use for RPCs. In our implementation, we used HTTP over TCP for RPCs because of its simplicity, but it has a much higher cost to open/close connections compared to UDP. The overhead would also be less noticeable when the cluster is under lighter load, when the heartbeat pings aren't competing for network IO resources with other concurrent client requests.

Despite the added overhead, we still believe the heartbeat mechanism to be a meaningful feature of the registry service, for it provides a valuable automatic failure detection function to the RAFT cluster at relatively low cost.

VI. Future works

In this project, we only used a simplified version of the registry service, deployed to a single node. During registry failure, configuration change requests will be served by the RAFT cluster directly as backup. In reality, we can scale the registry service to multiple distributed nodes for higher availability and redundancy, and transform it to a configuration management service for multiple client RAFT clusters. We can provide standardized RAFT server code compatible with the registry service and distribute to clients' clusters. When the clusters with registry compatible RAFT code comes online, administrators can use a streamlined interface to safely update configurations for multiple RAFT clusters.

Furthermore, our current heartbeat implementation is rudimentary and inflexible. It requires a fair amount of parameter tuning and specific knowledge of the RAFT cluster's network topology to minimize false positives. In an improved version, the registry could keep a rolling average latency for each node's ping request, and adjust the heartbeat timeout for each node. With a heartbeat mechanism that adapts to individual clusters, false positives will be significantly reduced.

Finally, the registry's **updateGroup** performance can be further evaluated in a more diverse set of use cases. In our experiment, we did not introduce arbitrary network delay, random node failures, out of order RPC deliveries. Nor did we experiment with log compaction and snapshotting techniques when adding new members after a long period of active cluster activity. We are certain that new opportunities for improvement for the registry model will arise when tested in a more production-like environment.

References

[1] LISKOV, B., AND COWLING, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

[2] ONGARO, D., AND OUSTERHAUT, J. In Search of and Understandable Consensus Algorithm. Stanford University. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference. USENIX Association, USA, 305-320.

[3] ONGARO, D. Consensus: Bridging Theory and Practice. PhD thesis, Stanford University,2014. http://ramcloud.stanford.edu/~ongaro/thesis.pdf.

[4] Lampson, B., and Lomet, D., "A New Presumed Commit Optimization for Two Phase Commit," Technical report, Digital Equipment Corporation, February 1993.