Group 1 Report for URL Shortener Project in CBDP

Maxim Solodukhin   Valery Tsarou   Vasilisa Poliarus

February 8, 2024

System Design

Cloud Architecture

The cloud design is depicted in Figure 1.

Azure Load Balancer distributes Http Requests from the client between the nodes in the pool. If a follower node receives the request to shorten the URL, i.e., to write, the node forwards it to the leader. Read requests, on the other hand, can be processed by all nodes. LB doesn't send requests to the nodes that are not listening on port 80. This way, if a node crashes, it won't receive traffic.

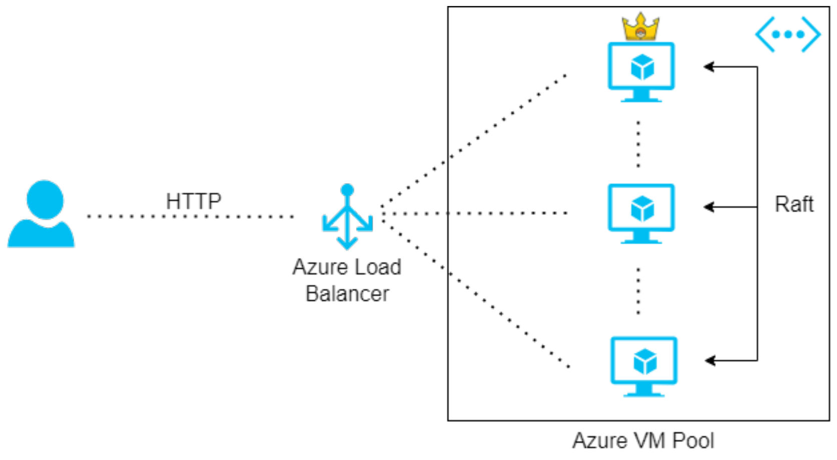

Figure 1: Cloud design.

Deployment to Azure

After cloning the project, update variables in `tf/variables.tf`. Just resource_group_name and resource_group_id should be enough. Then:

```
az login
cd tf
terraform init
terraform apply
```

After successful deployment, terraform outputs ID address of the load balancer, where you can send HTTP requests. It supports 2 requests:

```
GET: http://10.11.12.13/cut?u=<url_to_shorten>
GET: http://10.11.12.13/exp?u=<url_to_expand>
```

Node Architecture

The inter-node communication logic for write propagation as well as leader election precisely follows the Raft consensus algorithm and is implemented on the node level.
The internal design of a node is depicted in Figure 2.

Raft is conceptually the central part of the node, responsible for creating and processing RPCs.

Node Networking for RPCs is implemented with TCP sockets.

Persistent Storage (VM's filesystem) is used to store committed logs as well as the current state.

Hash Map for fast reads is reconstructed from logs and loaded from Persistent Storage on startup. Persistent Storage is used as a backup if the node process crashes. Raft and Http Server read the entries from the Hash Map only.

Http Server runs in a separate thread, and resolves read requests on its own using Hash Map. If the node is a leader, once it receives a write request, the server thread overtakes the Raft logic and propagates the write to the followers. If the node is a follower, the server forwards the request to the leader instead.

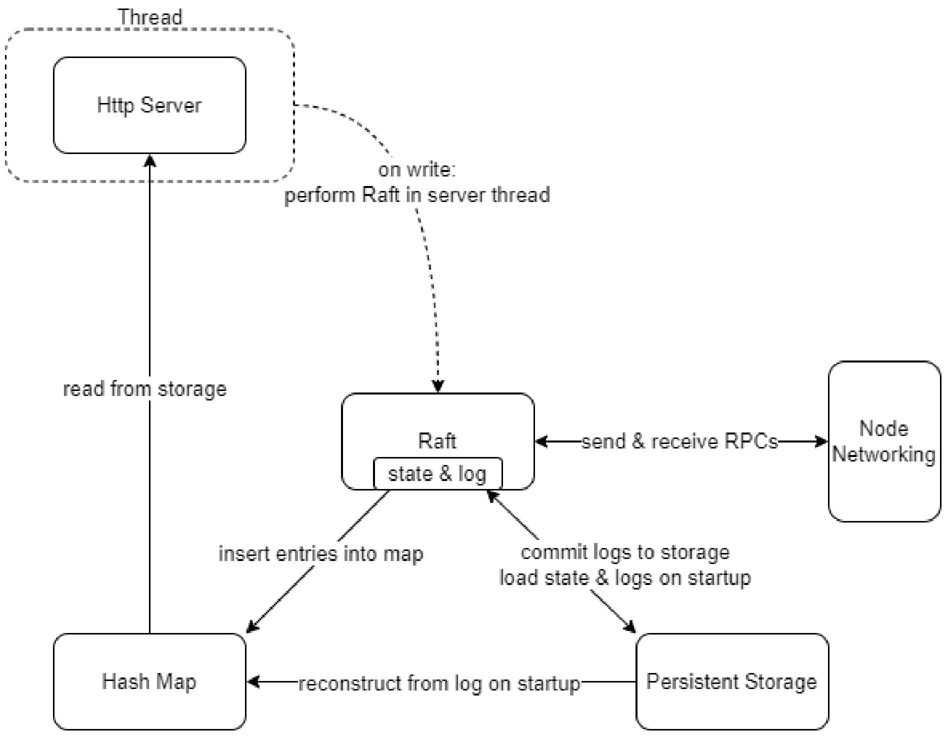

Figure 2: Internal design of a node.

Main Loop

Although the application has 2 threads - the Http Server thread and the Raft thread - the one worth elaborating on is the Raft thread, as the Http Server is active only on incoming requests and still leverages functionality of the Raft class to execute insert.
Therefore, to understand the core of the system, look at the main loop of the Raft thread in Algorithm 1.
Every node is always listening to all incoming events and reacts depending on its current type. Additionally, Leader sends regular heartbeats.

Leader Crash Handling

- *How long does it take to elect a new leader?*

  In general, everything related to latencies and durations depends on 2 timeouts: **listenTimeout** for Follower and Candidate (lines 3 and 14 in Algorithm 1), and **heartbeatTimeout** for Leader.

  The values that we found are working quite well are:
  listenTimeout = 100ms
  heartbeatTimeout = 20ms

  We could probably squeeze some milliseconds and reduce timeouts even more, but we find the latencies with the current setup already pretty low and we didn't want to overload the network with empty heartbeats.

  After a leader crashes and until all nodes agree on a new leader, the following events should happen:

  1. Follower waits maximum one **listenTimeout** to become a candidate.
  2. Candidate starts election and sends RequestVote RPCs.
  3. Candidate waits maximum one more **listenTimeout** to collect the votes and then becomes Leader.
  4. Leader sends the first heartbeat and notifies every node about its election.

  Assuming all the nodes except for the crashed leader work stable, one round of election is enough. As steps 2 and 4 are extremely fast, two timeouts in steps 1 and 3 take most of the time. Therefore, the expected election time would be around 200ms.

  Based on 10 observations, the election time was varying between 17 ms and 312 ms. The average was 134 ms.

- *Measure the impact of election timeouts. Investigate what happens when it gets too short / too long.*

  The problem of a too long timeout: the system stays without a Leader for a long time after a potential crash. During this time, write requests won't be processed.

  The problem of a too short timeout: unstable leader. We tried setting both **listenTimeout** and **heartbeatTimeout** to very low values, e.g., 20ms and 10ms accordingly. This way, the Leader is not able to send heartbeats and process responses fast enough, plus networking delay becomes visible. Therefore, **heartbeatTimeout** on a Follower would expire and it will start the election process, although the leader is healthy. Popping up of random election processes made the system unstable.

Node Workload

- *How many resources do the nodes use?*

  We execute our nodes on Azure VMs, each with one CPU of Intel(R) Xeon(R) Platinum 8272CL @ 2.60GHz and 3.32GiB of available RAM. The resource utilization of the Docker container executing the node is presented in Table 1.

  The difference in resource utilization between the node being idle or performing read and write requests is very insignificant. The node type itself has a bigger influence on average usage.

Node Type	CPU Usage	RAM Usage
Follower	3 – 4%	11%
Leader	4 – 6%	12%

Table 1: Resource utilization of one node container.

- *Where do inserts create the most load?*

  As mentioned above, the inserts do not cause any substantial additional workload. Although inserts involve writing to the persistent storage on log commits, this does not influence either real-time performance or resource utilization due to write buffering. Most of the workload on inserts is caused by the networking component and sending/receiving RPCs, which is also the case for idle state, elections or read requests.

- *How many nodes should one use for this system? What are their roles?*

  Azure student plan limits us to 3 VMs at a time. Since we use Azure Load Balancer and do not need a dedicated node for that purpose, we use all 3 VMs as nodes of our URL Shortening Service.

Latency Analysis

- *Generating a new short URL.*

  From a client side, response time after sending an HTTP request ranged between 49 ms and 91 ms, with 61 ms on average.

  Inside the node that was receiving the request, the average total processing time was 31 ms. Therefore, the remaining 30 ms were spent on networking. The processing inside of the node was distributed in the following way:

  1. 2%: generate short URL.
  2. 33%: send AppendEntryRequests to all followers.
  3. 64%: await responses.
  4. 1%: write to Persistent Storage.

- *Lookup to get the URL from a short id.*

  From a client side, response time after sending an HTTP request ranged between 26 ms and 28 ms, with 27 ms on average.

  Inside the node that was receiving the request, processing took around 1 ms. As it is just a simple lookup in the hashtable, little variation is expected with a low amount of data.

Scalability Analysis

- *Measure the latency with increased data inserted, e.g., in 10% increments of inserted short URLs.*

  We use hashmaps so the storage latency isn't much influenced by the total amount of data ($O(1)$ complexity for reads and writes), at least until the number of entries is not in the 6-digit range, when cache misses become apparent. For the Raft algorithm itself, the latency also doesn't depend on the size of the data.

The limitation of such design is that our "cache" on the node (i.e. hashmap) holds all entries we have ever processed, and we are limited by the max size of the hashmap. On the other hand, it allows us to scale without performance loss.

It's important to mention that the increasing amount of data would increase the node startup time since the hashmap is reconstructed from logs that are loaded from persistent storage.

- *Measure the system performance with more nodes.*

  We technically couldn't test the system with more than 3 nodes due to the limitation mentioned previously. Between the 2-node and 3-node setup, there was no significant performance difference spotted.

  While increasing the number of nodes would improve the system's robustness and allow handling more read requests, the single leader remains a bottleneck for write requests. Moreover, one would need to adapt the timeouts, since a larger number of successful responses would be needed for consensus.

Algorithm 1 Raft::run()

```
while true do

    if $nodeType == Follower$ then
        $events \leftarrow listenToRPCs(listenTimeout)$
        for each $event$ do
            if $event == EventType :: timeout$ then
                $nodeType = Candidate$
            else if $event == EventType :: message$ then
                $handleFollowerRPC(event)$
            end if
        end for
    end if

    if $nodeType == Candidate$ then
        $requestVotes()$
        $events \leftarrow listenToRPCs(listenTimeout)$
        for each $event$ do
            if $event == EventType :: timeout$ then
                continue
            else if $event == EventType :: message$ then
                $handleCandidateRPC(event)$
            end if
        end for
    end if

    if $nodeType == Leader$ then
        $elapsed \leftarrow elapsedFromLastHeartbeat()$
        if $elapsed \geq heartbeatTimeout$ then
            for each $node$ do
                $sendAppendEntriesRPC(node)$
            end for
        end if
        $events \leftarrow listenToRPCs(heartbeatTimeout)$
        for each $event$ do
            if $event == EventType :: timeout$ then
                continue
            else if $event == EventType :: message$ then
                $handleLeaderRPC(event)$
            end if
        end for
    end if

end while
```