Describe the design of your system

Our system implements Raft for log duplication. In our case, a log entry is a key-value pair, where the value is the original URL and the key is the shortened key. It is implemented in Java, and makes use of the RMI API for RPC.

The shortened URL is determined by finding the next available key, and then encoding with Base64.

When a new log entry is added, the leader send an appendEntries request immediately, to get the new log entry to all peers as soon as possible. Also, at every heartbeat the leader also checks if the answer was false (in which case, the peer log needs to be healed), and the leader starts working on healing the faulty peer log.

Raft mostly considers the cluster membership to not change. The original paper does consider the possibility of using a two-phase configuration migration to guarantee safety. However, we considered this algorithm to be too complex for the timespan available to implement it.

So instead, we implemented a simple membership algorithm with a gossip protocol to heal the members list. This protocol works as follows: 1. A new peer may only join through the leader. The new peer sends a join request to the leader. The leader adds the new peer to its members list, and informs all the old members (that were already part of the cluster) that a new peer has joined. Then the leader replies to the new peer with the list of all members the leader knows. 2. Every 1000ms, each peer picks a random peer, and they exchange membership information through a gossip protocol. This guarantees that member lists are healed if step 1 somehow fails to cover some edge-case. 3. Every time a peer fails to connect to another, it immediately assumes it is down, and unilaterally removes it from the members list. This procedure is not used to detect leader failures; it is only used to update the members list.

The external interface of each node is an HTTP interface with two endpoints: 1. `PUT /`: which requests a new URL to be shortened. The URL to be shortened is in the body of the request. The reply body contains the shortened key (not the whole shortened URL; because this makes it easier to use the same key to ask different nodes). 2. `GET /{shortenedKey}`: asks to redirect to the full version of the shortened key `shortenedKey`. If successful, the response has code `301 Moved Permanently` and the `Location` is set to the proper full URL. If it fails, the server replies with `404 Not Found`

All requests can be made to all nodes; `GET` requests are served locally by each node, but `PUT` requests are redirected to the leader in a transparent way.

A few notable improvements we implemented are: **- When commitIndex is modified, an extraordinary heartbeat is sent to all peers**, so that the may commit more log entries. This does not make Raft linearizable, because Raft does not wait for a log entry to be committed in all peers to return to a

`PUT` request (although it waits until a writing quorum of half the peers plus one has gotten the new log entry before replying to the `PUT`). It does however help expedite commits in non-leader peers, since the leader does not wait until the next scheduled periodic heartbeat to inform peers of commitIndex; the leader informs them immediately.

How does your cluster handle leader crashes?

The heartbeat period is 500ms. Each peer picks a random election timeout T between $T_{min} = 1000$ms and $T_{max} = 2000$ms. When a peer detects a leader crash by noticing that the last appendEntries (or requestVote) were more than T time ago, it becomes a candidate and starts an election. The election process follows the Raft protocol.

How long does it take to elect a new leader?

Communication between peers in local deployment is quite fast (around 1ms-10ms). We also only tested clusters with up to 6 nodes. Therefore, elections rarely fail, and they often take 500ms-700ms between the instant the leader dies and the instant a new leader is elected (this is because the leader may fail at any random moment between time instant 0 and T_{min}, with average $T/2 = 500$ms). This makes it so that a cluster with higher number nodes usually takes less time to elect a new leader, because it is more likely for a node to choose a lower election timeout and leader failure detection happens more frequently. This however, did not happen in our tests, which was surprising.

<> avg election time = 40ms + 37ms + 61ms + 43ms = 181ms / 4 = 45.25ms
In practice, the average election time since a node detects a leader failure for a network with 2 followers and 1 leader failing is 45ms.

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

Measurements show that a timeout between $T_{min} = 100$ms and $T_{max} = 4000$ms is has an average election time of 160ms. A timeout between $T_{min} = 50$ms and $T_{max} = 200$ms has an average election time of 63ms. And a timeout between $T_{min} = 3000$ms and $T_{max} = 4000$ms has an average election time of 3070ms. The group believes that the reason for these values to be so low and close to their respective minimums is because the azure deployment does not support a high number of nodes, and leaders are elected very quickly.

Analyze the load of your nodes:

How much resources do your nodes use?

Peers use a high amount of CPU, and an extremely high amount of memory.

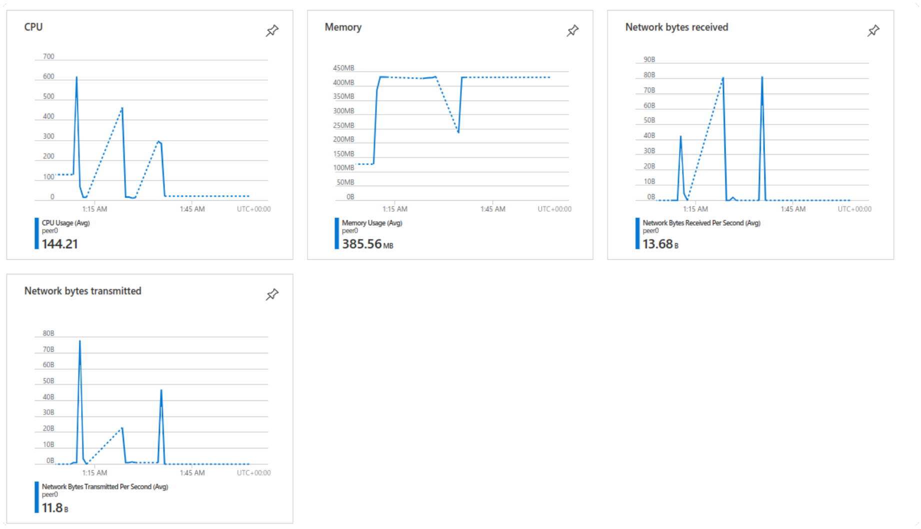

Figure 1: Peer usage

Where do inserts create most load?

Inserts create most load in the leader, since the leader has to communicate with each peer and process all requests. In addition, the leader has to enforce an order in the requests it receives, and it has to lock the log while it is being modified, which leads to a lot of contention.

Do lookups distribute the load evenly among replicas?

Since the state machine is replicated among the nodes, the peers have the ability to be self-reliant and don't depend on the leader when answering requests. Therefore, if the load balancing strategy is reasonable, then the load should be distributed evenly among the nodes.

How many nodes should you use for this system? What are their roles?

We should use at least three nodes at every time. This is because, if we were using only two nodes, there was a chance one of the node fails, so only one node is left alive and alone in the network. Thus, it is a good idea to make sure there are at least three nodes, so that if one fails, there are still two nodes in the network.

As mentioned previously, since the azure deployment doesn't allow for a high number of nodes (maximum 5), we were not able to verify this claim in practice.

Measure the latency to generate a new short URL

The same url was used for all requests. The average latency of each individual request is shown in the table below.

Experiment no	Average PUT latency (ms)
1	0.010046958923339844
2	0.008085966110229492
3	0.008549213409423828
4	0.0072901248931884766
5	0.007707357406616211
6	0.007663726806640625
7	0.007801532745361328
8	0.007688999176025391
9	0.007318258285522461
10	0.007418155670166016
Average	0.007957029342651366

Analyze where your system spends time during this operation

From looking at the timestamps of the logs of all pears, we conclude that the leader is the bottleneck of PUT requests, as all of these requests were processed by the leader.

Measure the lookup latency to get the URL from a short id

Experiment no	Average GET latency (ms)
1	0.004769325256347656
2	0.0033721923828125
3	0.003630399703979492
4	0.003074169158935547
5	0.0033195018768310547
6	0.003111600875854492
7	0.003014087677001953
8	0.0030066967010498047
9	0.0033333301544189453
10	0.002803564071655273 4
Average	0.003343486785888672

How does your system scale?

In theory, as mentioned previously, our system scales well with the number of nodes for handling GET requests. In contrast, since the leader is the bottleneck for PUT requests, the system scales poorly with the number of nodes for handling these requests, as the load of the leader is increased. Furthermore, choosing Raft as the consensus algorithm has two major downsides: 1. In order to achieve state machine replication, reasonably high traffic is required. 2. Being reliant on the leader, thus being prone to congestion if the leader is overloaded.

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

We inserted 10000 URLs, in increments of 1000 URLs. The average latency of each individual request for each increment is shown in the table below.

Number of URLs	Average PUT latency (ms)	Average GET latency (ms)
1000	0.01636631826075112	0.003454508199924376
2000	0.011002745512055188	0.003236748067344107
3000	0.009551903794451458	0.0029785200444663444
4000	0.010175535853316145	0.002687017626878692
5000	0.009258325855906416	0.002827512694568169
6000	0.008764778928058903	0.0031023625629704172
7000	0.010433147244337129	0.002518173078211342
8000	0.00938984545265756	0.0035987970305652152
9000	0.008153121994762886	0.0023620440320270817
10000	0.007965704987688763	0.0024641360306158298
Average	0.010106142788398557	0.0029093130000000003

Measure the system performance with more nodes

Due to Azure limitations, we could only test with 5 nodes. Results have shown that the system performance is not affected by the number of nodes. The same experiment as above was repeated with 3 and 5 nodes, and both results were roughly the same.