Raft Consensus Algorithm for Data Center Network

Aman Mangal, Flavio Castro, Kapil Agarwal

Data centers are becoming increasingly popular for their flexibility and processing capabilities. They are managed by a single entity and used by thousands of users. Centralized administration makes them extensible, predictable and more reliable, making them a good environment for innovative distributed applications. The flexibility and stronger guarantees provided by the data center network allow us to co-design distributed systems with their network layer and optimize their overall performance. In this course project, we have analyzed the performance of the Raft consensus algorithm in presence of heavy network load. We show that the degraded system performance can be significantly improved by co-designing the system with the network layer. We evaluate our idea on a 5 node cluster of etcd key-value store, setup on banana pi boards connected in a star topology. We also discuss how we can expose the network layer improvements to the application in a transparent manner in order to simplify application development.

1 Introduction

Most distributed systems are designed independently from the underlying network. For example, Chandy-Lamport distributed snapshot algorithm [1] assumes a model of distributed system in which messages experience arbitrary delays, nodes receive messages sent by different senders in random order, and the system consists of arbitrary connections (channels) without a structure among the nodes etc. Fast Paxos [2] is designed with similar assumptions for an asynchronous system assuming unbounded network delays. These assumptions are necessary to achieve safety properties in a distributed system where message paths are unpredictable, nodes are placed in an arbitrary manner and network congestion is unknown and unexpected.

However, today's applications are deployed in a data center environment running tens of thousands of physical machines managed by single service provider. These data centers are well planned, highly structured unlike the internet so that they are easy to manage while providing cloud services to several users. These properties of data center network can be utilized in co-designing distributed systems with the network layer providing stronger guarantees to the higher layers. This will not only reduce the complexity of the system but improve the overall performance as well.

In this course project, we plan to analyze Raft [3] consensus algorithm in a data center like environment. We have chosen to play with *etcd* [4] key value store which uses Raft in order to manage highly available replicated logs. From the perspective of the network,

we decided to provide different Quality of Service (QoS) guarantees which can be used by *etcd* to achieve higher performance. The rest of the report is structured as follows. In Section 2, we define a few concepts necessary to understand the project. Section 3 describes ways to provide QoS priorities to the application in a transparent manner. We discuss implementation details in Section 4. Section 5 details the experimental setup, evaluation methodology and the results. Finally we conclude and discuss future work in Section 6 and 7 respectively.

2 Background

In this section, we summarize the Raft consensus protocol. We, then briefly, describe *etcd* key value store and its use of Golang http library.

2.1 *Raft*

Raft is a consensus algorithm based on Paxos for managing replicated logs. It uses a stronger form of leadership than other consensus algorithms in order to have fewer states and a simpler, more understandable algorithm. It first elects a distinguished leader, then gives 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. Having a leader simplifies the management of the replicated log, however, it becomes the centralized point of failure. It can fail or become disconnected from other servers, in which case a new leader is selected. Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.

2.2 *etcd*

etcd is a distributed, consistent key-value store for shared configuration and service discovery, with a focus on being simple, secure, fast and reliable. It is written in Golang [5] and uses Raft consensus algorithm to manage highly available replicated logs. It provides simple APIs to create, modify, read and delete key value pairs. It provides strong consistency guarantees using Raft. *etcd* API provides the following operations -

- **Set(key, value):** This is a write operation which requires *etcd* node to send the request to the Raft leader. The leader then, constructs a proposal and gets a consensus from majority of nodes before applying it to logs.
- **Get(key):** This is a read operation which doesn't require a consensus among *etcd* nodes. A node can simply return the value available locally. Though, client can force a consensus by explicitly stating quorum to be true in order to get the most recent written value.
- **Delete(key):** This is a write operation performed similar to *Set* operation.
- **Watch(key):** This operation allows an *etcd* client to watch changes in a value of key. As soon as a change occurs, the client gets a reliable notification.

2.3 Golang & its HTTP library

Go is an open source programming language built by engineers at Google. It makes it easy to build simple, reliable, and efficient software. It is designed to write highly concurrent programs and provides the concept of light weight threads known as *goroutine*. Go routines are cheap and expected to be created in large number in a Go program. Golang provides a *http* library which allows to create HTTP server, send HTTP requests etc. The *http* library makes full use of Go routines. The client library creates a pool of TCP connections with server and avoids a TCP handshake every time a new HTTP request is sent. Similarly, server library allows concurrent processing of multiple HTTP requests and makes use of all the available CPU cores. *etcd* uses Golang *http* library to communicate with *etcd* client as well as peer Raft nodes (also referred as *etcd* nodes in this report).

3 Improvements in *etcd*

In this section, we discuss what changes we made in *etcd* code base in order to improve its performance. We detail what features of a network and a distributed system helped us in building an overall more efficient distributed application.

3.1 Networking Support

To co-design *etcd* with the underlined network, we chose QoS priority levels available in the network as a feature to expose directly to the application. Usually the network is modeled as a best-effort channel. It has unpredictable delays, delivery is not always guaranteed and messages are mostly delivered in a first-in-first-out (FIFO) order with no stronger guarantees. The basic premise change of our system is the FIFO delivery order. In order to expose different QoS levels to an application, we propose to use network priorities to reorder the messages in the channel. For example, if a channel is full of low priority messages waiting to be processed, and a high priority message arrives in the channel, the network will deliver the high-priority message first.

In congested networks, instead of dropping packets, each switch momentarily buffers messages that surpass its output rate before forwarding them to a next hop. As a consequence, small queuing delays are incurred in the message transmission. A Differentiated Services architecture uses the priority field of a packet to define its class of service (CoS). Different CoS can have different forwarding behaviors. A high priority CoS can have an strict expedited forwarding behavior in order to meet its QoS requirements.

3.2 Expose Priorities to Application

In order to provide the concept of priorities to the application in a transparent manner, we need to achieve following goals-

- Set a priority of a message (HTTP request) while sending it
- Set a priority of a message while sending a response to the incoming HTTP request

Golang *http* library puts a challenge to provide QoS priorities directly to the application due to its use of pool of TCP connections and its concurrent nature. We need to ensure that the priorities are set to the socket connection while sending data instead of while setting up the TCP connection due to use of pool of TCP connections. The functions exposed to the application making use of *http* library, then, can decide the priority while sending a message or responding to an incoming HTTP request.

3.3 Using Priorities in *etcd*

We have various messages sent/received by an *etcd* node which could be a leader or a follower. We divide thsese messages into 2 groups-

- **Messages from Client:** These messages are sent from *etcd* client to a peer node. It may be a request for creating, deleting, modifying or watching a key or a query message sent to request information about health of a node, version of the code base, peer nodes in the cluster. We ignore these messages for setting priorities for the course of this project.
- **Messages from Peers:** These messages are sent from one *etcd* node to another *etcd* peer node. These messages consist of heartbeats, probing for health check and latency measurements, raft proposals and other book-keeping messages.

We can make use of the modified interface provided by Golang *http* library in order to set different priorities to each message. Setting different priorities will affect the delivery of the message to other nodes. A message sent with higher priority will reach the other node with lower latency. We can also ensure quicker delivery of critical messages in presence of high workload.

4 Implementation Details

In this section, we describe how we implemented priorities in the network. We also discuss how we provided a transparent interface to the *etcd* application to set different priorities to different messages while sending a new request or responding to an existing one.

4.1 Setup

We built a 5 node *etcd* cluster deployed over a set of 5 ARM computers in a star topology as shown in figure 1 connected through Gigabit Ethernet interfaces. The switching latency is of the order of $100u$ seconds. Each ARM computer runs 1GHz Dual-core CPU with 1GB RAM attached to a 1Gbps Ethernet interface. The switch used is a 8x1G Ethernet Switch with IEEE 802.1p enabled. IEEE 802.1p is a standard for VLAN priority, it prioritizes the transmission with higher VLAN priority fields.

We used a debian based operating system called Bananian 15.08. Bananian kernel supports all necessary features such as VLAN tagging required to setup message priorities. The disk in these machines, were simple SD cards. Hence, to avoid slow disk operations which could jeopardize the experiment, we mounted the application folder used by *etcd*, in memory using in memory disk called *ramdisks* provided by Linux kernel.

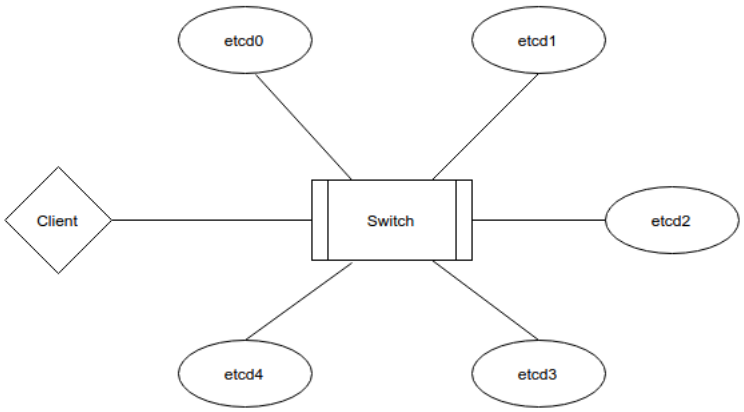


Figure 1: Banana Pi Topology

4.2 Changes in the Network

To maximize the impact of congestion, we rate-limited the output rate of each network interface to 10Mbps using a feature provided by the switch. In the absence of congestion, we observed an average round-trip time of 200us. In presence of congestion, we observed a maximum round-trip time of 25ms. We also verified a high priority packet would indeed have its round-trip time untouched if the congestion had a lower network priority.

In each node, we configured a VLAN interface and mapped the VLAN priority field according to a socket level priority field set by the application as discussed in next section. For example, in the code excerpt below, *vconfig* stands for VLAN configuration, we add a VLAN tag id 11 to interface *eth0* and further, map a socket priority level 6 to VLAN priority 6. We discuss how to set priority in application in the next section.

```
vconfig add eth0 11
vconfig set_egress_map eth0.11 6 6
```

4.3 Modifications in Golang *http* library

To set priority levels in the application, we need to modify the Golang *http* library to provide an interface to applications as well as set the priority provided by the application. We achieved it as follows. We added an overloaded *http Get* function with another argument of priority as follows-

```
func (c *Client) GetWithPriority(url string, priority int) (resp *Response,
    err error) {
      req, err := NewRequest("GET", url, nil)
      if err != nil {
            return nil, err
      }
      req.Priority = priority
      return c.doFollowingRedirects(req, shouldRedirectGet)
}
```

We added *Priority* as a data member to the *Request* class (struct in Golang) so that when we are about to send the request on a TCP connection, we can set the priority of the TCP connection before sending data on it. We set the *IP_TOS* field of the IP header using *SetSocketOpt* function of Golang *syscall* library. This options allows us to achieve differentiated service as explained in Section 3. Similarly, we added a *Priority* data member to the *Response* class (struct in Golang) as well in order to set the priority of responses for an incoming HTTP request.

4.4 Modifications in *etcd*

Now that we have the interface to set the priority, we need to be able to set message specific priority in *etcd*. We defined a generic function *GetPriority* which is called to figure out priority of a specific message by *etcd*. This allows us to make least modifications to *etcd* code and we can simply call this function to find out the priority of a message and pass it along to the Golang *http* library.

```
func GetPriority(m pb.Message) int {
      switch m.Type {
      case pb.MsgHup:
            return QJ_MSGHUP_PRIORITY
      case pb.MsgProp:
            return QJ_MSGPROP_PRIORITY
      case pb.MsgApp:
      case pb.MsgAppResp:
            return QJ_MSGAPP_PRIORITY
      case pb.MsgVote:
      case pb.MsgVoteResp:
            return QJ_MSGAPPVOTE_PRIORITY
      case pb.MsgSnap:
      case pb.MsgSnapStatus:
            return QJ_MSGSNAP_PRIORITY
      case pb.MsgHeartbeat:
      case pb.MsgHeartbeatResp:
            return QJ_MSGHEARTBEAT_PRIORITY
      case pb.MsgUnreachable:
            return QJ_MSGUNRECHABLE_PRIORITY
      default:
      }
}
```

All the variables having prefix *QJ*, are defined in a configuration file where we can set the priority for any message we would like, in *etcd*.

5 Evaluation

In this section, we describe our setup, evaluation methodology and the results of various experiments that we performed on a cluster of Banana Pi boards.

5.1 Setup

As we have described before, we performed our experiments on a cluster of banana pi boards. We chose to do so as we need access to the switch to modify/setup priorities in the network. We also decided to use *etcd* and *iperf* instead of performing experiments with only *etcd* due to low throughput requirements of *etcd* application. We decided not to change the *etcd* as an example application at this stage of our project.

5.2 Methodology

To evaluate if network congestion can degrade Raft performance we looked at the number of write requests processed per second by *etcd* under different workloads. Note that read requests do not require a consensus and therefore, are not limited by network congestion. If interference can degrade the write performance of Raft, we will observe a reduced write throughput in presence of congestion.

To achieve network congestion, we simultaneously ran a throughput intensive and a latency sensitive application. We used *etcd* as the test latency sensitive application which is a distributed key-value store whereas we used *iperf* as the test throughput intensive application that establishes a TCP connection and sends as much data as possible in order to congest the network. We also performed a baseline measurement with an unmodified *etcd* application. We performed 3 different set of experiments in order to quantify the impact of interference on Raft described as follows-

- Write throughput of unmodified *etcd* and no other network traffic
- Write throughput of unmodified *etcd* with background network traffic using *iperf*
- Write throughput of modified *etcd* with background network traffic. In this case, we modify *etcd* application as discussed in previous sections, to set higher priorities for the messages sent. Note that *iperf* is still running with default (lower) priority.

We used a 2.0 Ghz dual core machine with a Gigabit Ethernet interface as the client (a laptop) which connects to the *etcd* nodes and sends write requests to them as shown in figure 1. A set of experiment consisted of following runs-

1. Send requests to the leader
2. Send requests to one follower at a time (total of 4 followers)
3. Send requests to all servers simultaneously

We used *boom* [6] benchmark library for performing our experiments. *boom* is a program written in Go programming language which can be used to send HTTP requests to web applications. *boom* supports custom headers, request body and basic authentication. It executes provided number of requests at given concurrency level by creating Go routines equal to concurrency level and prints statistics at the end of the experiments. We decided to measure the performance of *etcd* on the basis of its operation latency i.e. the time to perform each operation from initiation to getting back the response. We also varied the size of the *value* of a key to measure the effect of request size on the

performance when messages are prioritized. The following code snippet shows how we used *boom* to send write requests to a server. We launch one *boom* process per *etcd* server which is responsible for sending write requests to that server in the final case when we are sending requests to all the server simultaneously.

```
# server address
server=http://192.168.10.2:2379
# generate 64 bit key
key=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 64 | head -n 1)
# generate 64 bit value
val=$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 64 | head -n 1)
# number of concurrent requests
curr=16
# total number of requests to send
total=4096
# run boom to issue write requests
./boom -m PUT -n $total -d value=$val -c $curr -T
    application/x-www-form-urlencoded $server/v2/keys/$key
```

We measured CPU, received and sent network usage, memory usage in order to identify the bottleneck in *etcd* application while running the experiments. We collect metrics provided by linux *proc* file system and store them in a text file locally for later analysis.

5.3 Results

We first present some initial experiments that we performed with *etcd* to understand its behavior and resource requirements.

5.3.1 Concurrency

In this experiment, we vary the concurrency level and measure the throughput and latency of *etcd*. Concurrency refers to the number of requests that are sent concurrently from the client to one server. *boom* creates as many worker threads (Go routines) as the concurrency level, each of which creates a new synchronous HTTP request.

We fix the value size to 64 bytes and vary the concurrency from 1 to 256 in steps of factor of 4. We did the same experiment for both leader and followers. We present the average throughput/latency for followers. Note that, in a cluster of 5 *etcd* nodes, one node will act as a leader whereas rest of the 4 nodes will behave as followers. As we can see in Figure 2, as we increase concurrency, throughput increases and converges to a maximum value. Throughput doesn't increase further due to CPU bottleneck. We confirmed our findings by measuring system CPU usage and observed that CPU usage of leader in fact reaches close to 200% (2 core). Note that, leader is involved in each requests sent to any node which makes it bottleneck in the system.

On the other hand, latency increases linearly as we increase concurrency. While the throughput is increasing, latency also increases because multiple requests are getting processed in parallel by *etcd* nodes. *Latency-95* is the 95^{th} percentile of latency measured for all the requests. We can also see that leader has lower throughput and higher latency compared to followers except for concurrency value of 1. This can be explained based on

CPU usage of the leader. We observed that for concurrency value of 1, CPU usage of leader are well below the maximum usage (200%). Whereas, for larger concurrency values, leader has already exhausted its available CPU. We observe high performance when interacting with followers, as the total work is now divided among multiple nodes and leader's involvements is minimum compared to the case when requests are sent directly to the leader who has already exhausted its available CPU.

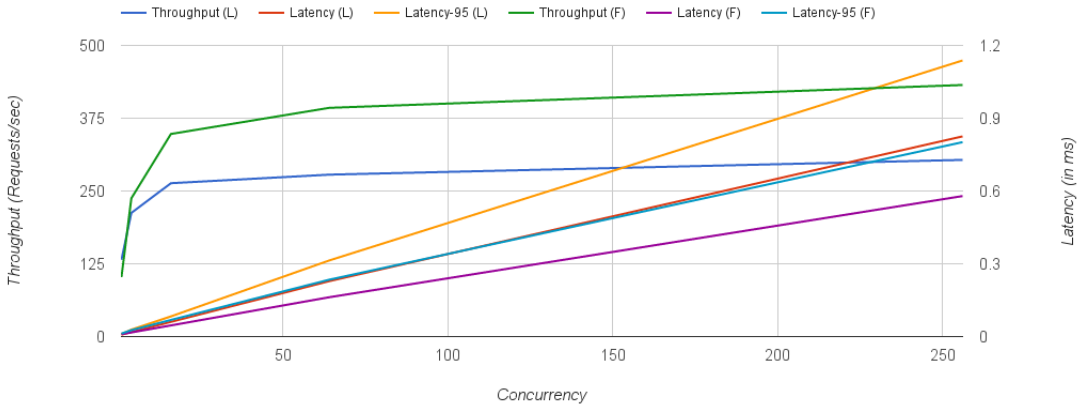


Figure 2: Performance vs Concurrency (L:Leader, F:Followers), Requests are sent to one node at a time

We observe a similar pattern when we send requests to all the servers simultaneously as shown in Figure 3. It is clear from the two graphs (Figure 2 and 3) that we observe low performance in this case which is due to leader being the system bottleneck.

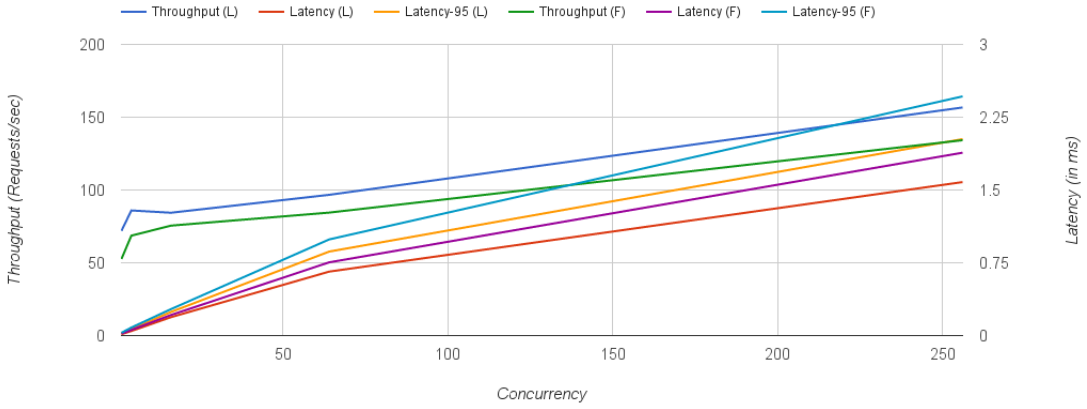


Figure 3: Performance vs Concurrency (L:Leader, F:Followers), Requests are sent to all the nodes at the same time

5.3.2 Value Size

In this experiment, we var value size and observe throughput and latency metrics. We keep the concurrency value equal to its highest value as 256 so that we can see maximum

difference due to change in value size, if at all. The results are shown in Figure 4 and 5. As we can see, the throughput and latency values do not change significantly as we change the key size. This is because when we increase the key size, we use more network bandwidth. Though, the marginal increase in bandwidth usage doesn't affect the performance due to no congestion in the network. We confirmed our findings of network usage by measuring RX/TX bandwidth usage by *etcd* nodes.

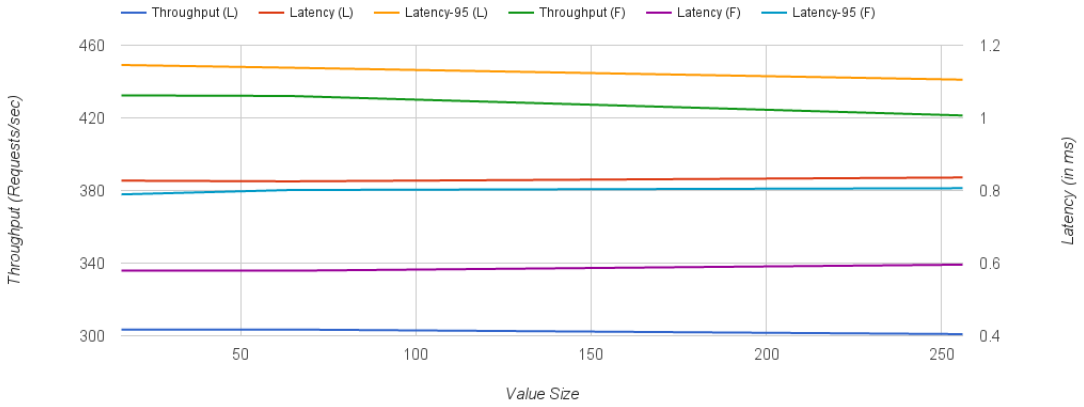


Figure 4: Performance vs Value Size (L:Leader, F:Followers), Requests are sent to one node at a time

Similar to the previous section, we see performance degradation when we send requests to all the nodes at the same time as shown in figure 5 compared to when we send request to each node one by one.

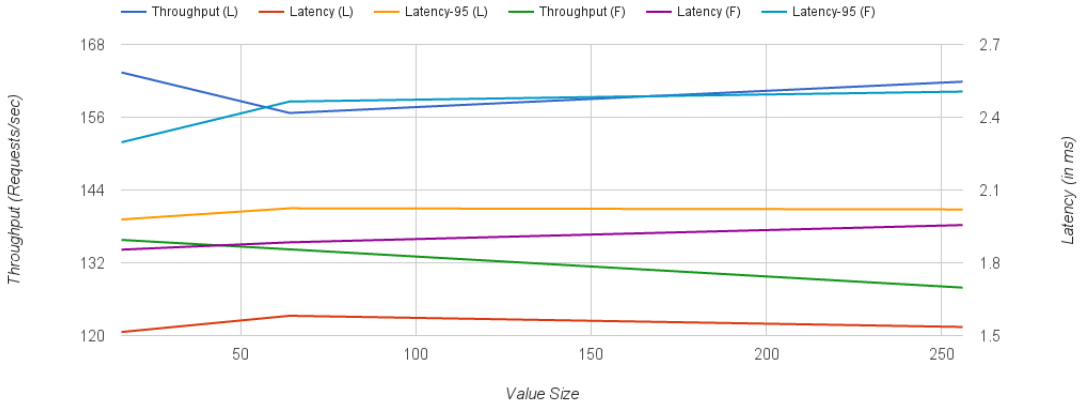


Figure 5: Performance vs Value Size (L:Leader, F:Followers), Requests are sent to all the nodes at the same time

5.3.3 Concurrency Variation in Presence of Network Congestion

Now we show the performance of *etcd* in presence of high congestion network workload. In this experiment, we keep the value size to be fixed as 64 bytes and increase the load

concurrency. We run *iperf* server on each node and connect *iperf* client to every other *iperf* server including the local *iperf* server. We did so to achieve symmetry in the system. We also pin all the *iperf* instance running on the node to CPU0. This constraints *iperf* to consume only 1 core available on a banana pi board. The goal of running *iperf* is to consume bandwidth and not CPU.

We can see in Figure 6, as we increase the load concurrency, throughput and latency both increases due to requests being processed in parallel by *etcd* nodes. Though, unlike Figure 2, we see that leader show higher throughput and lower latency values compared to followers. This is due to the fact that CPU is not a bottleneck anymore. The presence of network congestion and high latency leads to limited requests arrival at the *etcd* nodes from client.

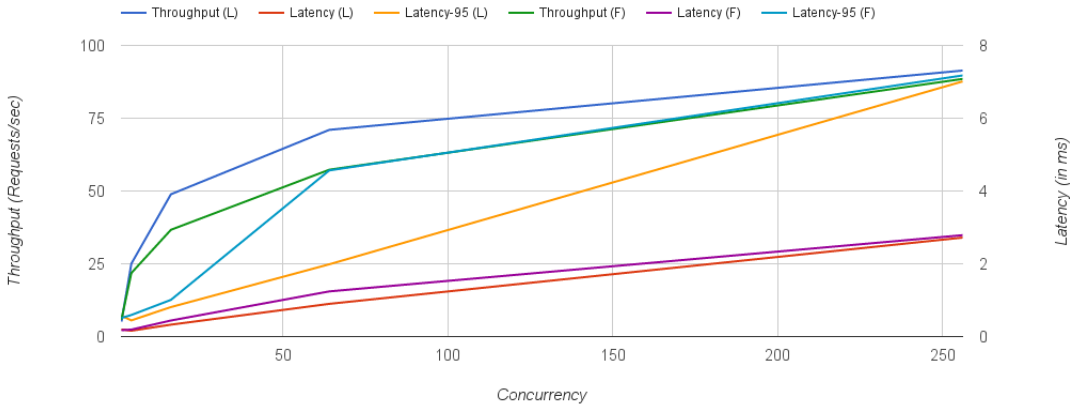


Figure 6: Performance (with *iperf* running) vs Concurrency (L:Leader, F:Followers), Requests are sent to one node at a time

We see a similar pattern in Figure 7 as above. Though, throughput is lower and latency is higher as compared to Figure 6.

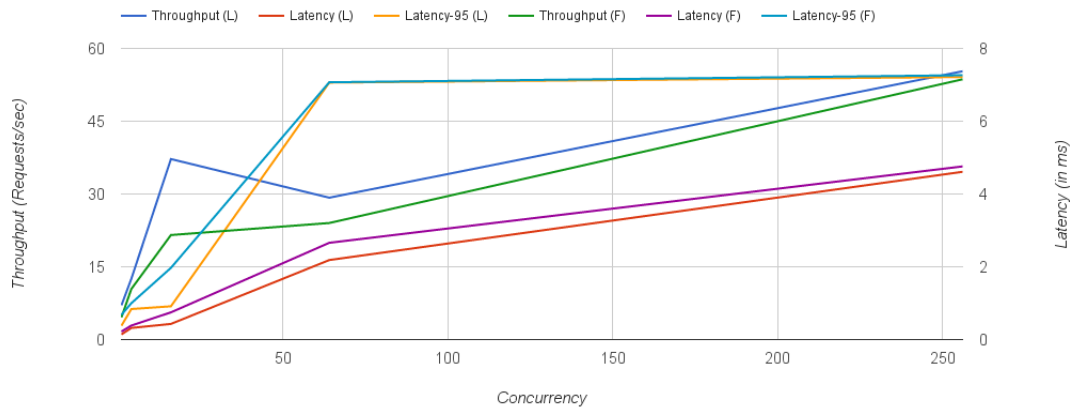


Figure 7: Performance (with *iperf* running) vs Concurrency (L:Leader, F:Followers), Requests are sent to all the nodes at the same time

5.3.4 Value Size Variation in Presence of Network Congestion

In these set of experiments, we keep the load concurrency to be highest as 256 and vary the value size. Unlike previous experiments, we see that in this case the throughput in fact, drops down as we increase the value size as shown in Figure 8. Due to the congestion in the network, CPU is not the bottleneck anymore and as we increase the *etcd* network traffic by increasing value size, it affects performance of the system due to congested network.

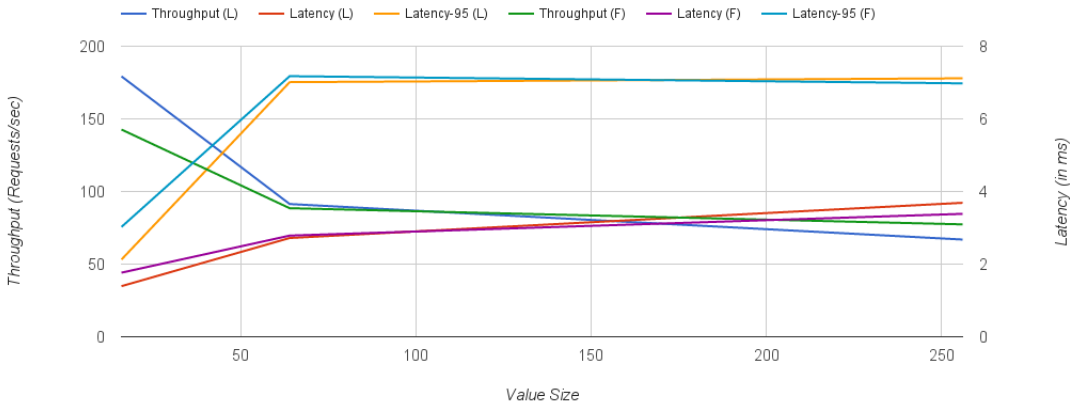


Figure 8: Performance (with *iperf* running) vs Value Size (L:Leader, F:Followers), Requests are sent to one node at a time

We also observe performance of *etcd* while sending requests to all the nodes while varying the value size in presence of network congestion. We see similar drop down as above in performance as shown in Figure 9. We can see that throughput decreases and latency decreases as we increase the value size from 16 to 256.

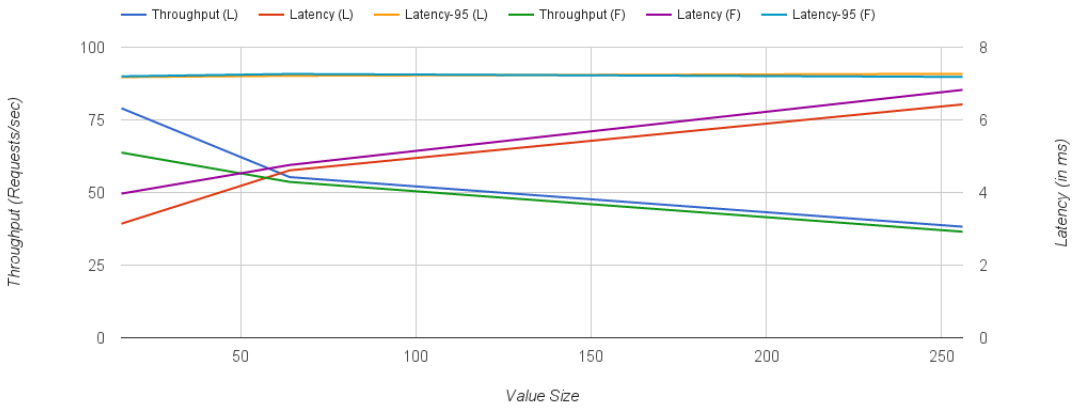


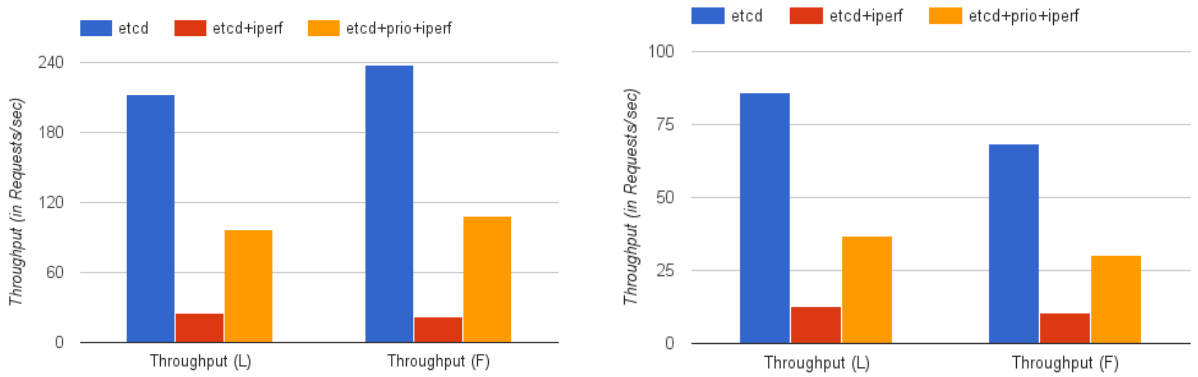
Figure 9: Performance (with *iperf* running) vs Value Size (L:Leader, F:Followers), Requests are sent to all the nodes at the same time

5.4 Priorities Enabled *etcd*

Now that we understand the behavior of *etcd*, we compare its performance in the aforementioned 3 scenarios- *etcd*, *etcd* +*iperf*, priority enabled *etcd* + *iperf*. The goal of these experiments is as follows-

- Quantify the effect of throughput intensive flows on latency sensitive flows
- Effect of priorities on latency and throughput in presence of heavy congestion

We fix the values size to 64 bytes and load concurrency to 4. Figure 10a and 10b shows that as we add congestion to the network using *iperf*, throughput significantly goes down. We then, increase the priority of etcd messages, and throughput increases. We do not see throughput increase as much as the base case because available CPU in case of priority enabled *etcd* experiment is only 100% (1 core) due to the fact that *iperf* consumes nearly other 100% (1 core) of CPU.



(a) One node at a time

(b) All nodes at the same time

Figure 10: Throughput Comparison

We see similar increase in latency as we add congestion to the network but latency decreases again when *etcd* messages are given higher priorities as shown in Figure 11a and 11b.

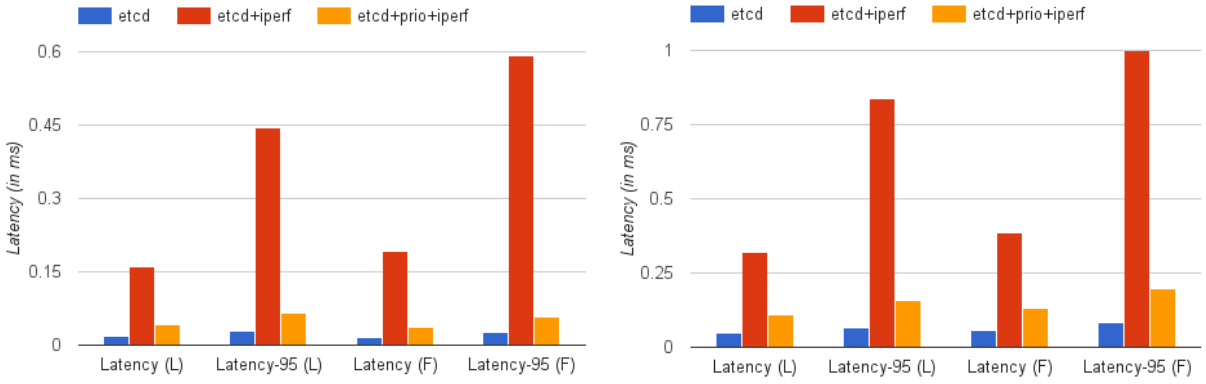


Figure 11: Latency Comparison

6 Conclusion

In this project, we showed that network congestion can affect performance of a system significantly. If the distributed system is co-designed with the underlined network, we can improve the efficiency of the system. We chose network priorities as one example of network feature which can be used in designing a distributed system optimally. We implemented priorities in the network using a combination of VLAN tagging and 802.1p priorities. We exposed an interface to the application by modifying Golang *http* library as well as *etcd* code base so that we can assign different priority to messages within the application. We then showed that network congestion can become a bottleneck and affect the performance of the distributed system which can be improved greatly by using network enabled message priorities. We believe that, as the underlined network becomes more reliable and predictable, it can help us in designing efficient distributed applications. We should, though, remember that such a design decision doesn't help us in achieving safety properties of a distributed system.

7 Future Work

In future, we would like to perform more experiments and compare performance of the system by setting different priorities to messages within an application. While such a design helps us implementing an efficient distributed system, it does come with some issues. There are applications where reordering of messages affects the safety properties of the system. Also, exposing network level features directly to the application imposes risk of security vulnerabilities in the data center which can make such a mechanisms almost infeasible.

References

[1] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[2] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annual Technical Conference*, pp. 305–320, 2014.

[4] "A distributed Consistent Key-Value Store for Shared Configuration and Service Discovery." `https://github.com/coreos/etcd`.

[5] "The Go Programming Language." `https://golang.org/`.

[6] "Boom, HTTP(S) load generator." `https://github.com/rakyll/boom`.