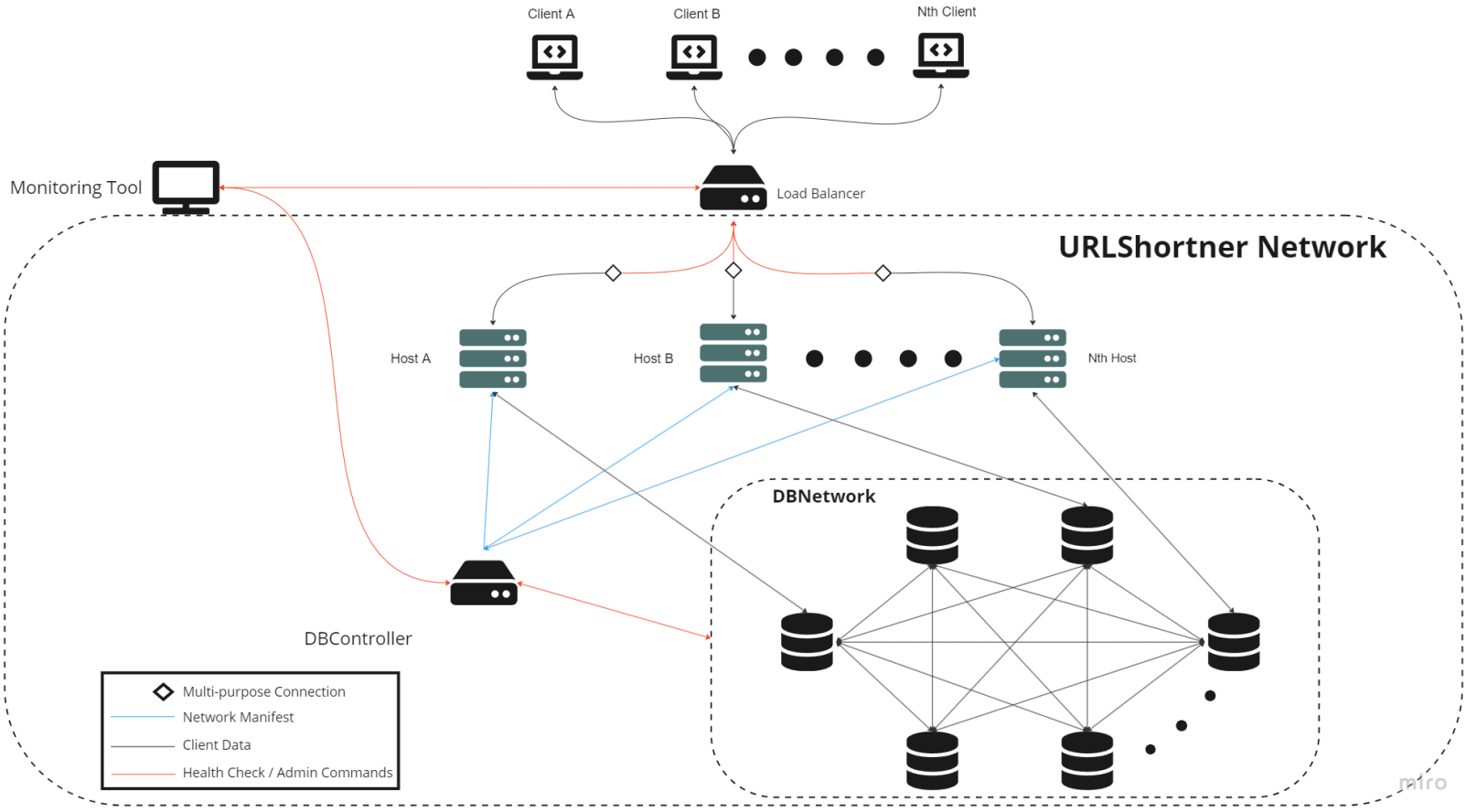
CSC409 Report A1

Name: Lakshya Gupta utorid: guptalak
Name: Ibrahim Fazili utorid: faziliib
Name: Jayvin Chang utorid: changj87



As you can see, we have a three-tier architecture where clients send requests to the reverse proxy and then the reverse proxy distributes those requests between the hosts who execute the requests. The hosts can then fetch needed data from one of the database nodes. There is also a monitoring tool which can send admin requests to the reverse proxy and database controller, such as add/kill hosts and database nodes, respectively.

Discussion:

- Consistency
    - With 2 layers of caching and a Cassandra-like distributed database, our system offers eventual consistency to the clients. Depending on the load, changes to the data can take time to propagate to the entire system, leading to temporarily inaccurate responses. For e.g., under heavy load, if a PUT request is made,

there can be a lag between when the PUT request is made vs when the URL pair is actually saved in the database by all the nodes responsible for that pair. For GET requests, since each DBNode can have different delays (depending on how many writes they're doing at any given time), the database queries each node that should be responsible for that data until one of the nodes replies with a good response. This ensures that any GET request can be responded to correctly as long as one of the nodes responsible for the URL pair has caught up.
    - We also have caching on the DBNodes and the hosts (URLShortner), this allows us to respond to the most frequent GET requests fast, without the need to go to disk to look in the database. This leads to a consistency problem for an existing URL pair that is updated frequently. Depending on how a GET request travels through the network, each GET request could have a different response. This is only a short-term problem, however, as eventually, cache entries expire, which fixes the inconsistency.
- Availability
    - Our application offers availability. For a GET request, the hosts (as long as it is not dead) will always either return a 200 or a 404 error and if it's a PUT that is successful then it will return 200. The only time it will fail and return a 500 would be if every DBNode in the database system were to crash at the same time which is highly unlikely.
    - Due to the way we handle requests on the reverse proxy in the round-robin method, we will only connect the client to a host to execute the request if it passes the continuous health check we have in the background. If a host is failing the health check, then the reverse proxy finds the next host that is not failing to handle the request. In this way, this ensures that the client will always get some response from a healthy host.
- Partition Tolerance
    - The database system has a Cassandra-like architecture. It has 2 components,
        - DBController
            - The DBController is a supervisor that watches over the entire database network (DBNetwork). Its job is to health-check DBNodes, restart crashed nodes, start new nodes, coordinate network level changes, and provide a manifest of DBNetwork to other systems within the entire URLShortner network
        - DBNode
            - It's a process that handles write/reads for the database, and talks to other DBNodes to coordinate data distribution and fetching. The data distribution/replication is controlled by a replication factor set during system startup. The replication factor can be changed at runtime through DBController's API.
            - Whenever one of the nodes does go down, DBController restarts the crashed node, or a node can be added by the admin using the DBController API. When a node restarts, it asks other nodes to send the data it might be responsible for. When a new node is

added, the entire network does a rehash of the data to re-distribute the data evenly across the entire network. The replication factor can be changed on the fly to increase or decrease the chance of data loss.

- Data partitioning/replication
    - We've already discussed the architecture of our database system. The architecture is based on partitioning and replication.
    - Partitioning
        - Every node in the network is responsible for a chunk of URL pairs.
        - When the client sends a PUT request, the URLShortner can submit the URL pair to any of the DBNodes. The DBNode that receives the URL pair hashes (using a 64-bit FNV-1 hash) the short URL to figure out where this pair should be stored. Then it coordinates with the appropriate nodes to save this URL pair.
    - Replication
        - Every URL pair is replicated across a few nodes (based on the hash of the short URL). The amount of replication is controlled by the replication factor (already discussed above). Data replication across multiple nodes ensures that data loss is extremely rare. If the replication factor is set to an integer 'n', for data loss to occur, 'n' specific DBNodes need to go down. For e.g., if there are 5 nodes up with a replication factor of 2, the probability of data loss is 1 / 10, i.e., 10% in the event of the 2 required nodes going down. The data loss probability is inversely related to the number of nodes and the replication factor. The higher these 2 numbers are, the lower the chance of data loss
- Data disaster recovery
    - We've already discussed some of this above in the last 2 sections. The way we do disaster recovery is:
        - When a DBNode goes down, the DBController detects this and restarts the node.
        - If the restart fails, new nodes can be added on the fly
            - Note: adding a replacement node for a crashed node on a different system is done through the admin dashboard. The reason for not automating this was the fact that every new node triggers a rehash on the entire network. This is a very expensive operation. Repeated rehashes can crash nodes when the data set is really big. So, we wanted to control when these rehashes are triggered
        - Whenever a node restarts, it asks other nodes for data to make sure it's caught up to the changes that happened while it was down. This ensures that the data this node was responsible for is not at risk of loss
        - The underlying SQLite database file is still on the disk, so the replacement node still has the old data. It uses the above process to catch up to the changes, but the old data is still on disk and not lost

        - When a new node starts, the data is re-distributed to ensure all data is replicated properly to reduce the chance of data loss
- Load Balancing
    - The reverse proxy has a single server socket connection to which clients will connect. Since we start an n number of hosts before accepting connections, there will always be a host that can execute the request. Each time a server socket gets a client connection, a new thread is spawned that connects the client to the host, so this ensures that each request is executed independently of one other. Under heavy loads, we could have multiple requests from a client to a particular host and they would execute independently. With a round-robin architecture, we assign a client connection to the host and the next client connection to the next host in the list. This ensures that we try to balance the number of requests we serve between all the hosts.
- Caching
    - The caching policy that we decided on is the LRU architecture. This is because if a particular short is being fetched/updated frequently, we would want to have quick access to it. Because we want to have a sufficiently large cache with quick runtime, we implemented it with two data structures, hashmap and doubly-linked list (dll). Each node in the dll has short and long values, as well as references to the previous and next node. Each key in the hashmap is the short value with the value being a node in the dll. In this way, whenever there's a GET/PUT request for a short and long, we can manipulate its position in the linked list in constant time. Thus any add, get, update, and remove node happens in constant time, O(1).
    - As mentioned earlier, we are using cache on two levels, the DBNodes and the hosts. Each of the hosts has a cache size of 1000 and DBNode has a cache size of 500. Every time we make a GET/PUT request, we add it to the cache in case it gets reused soon. In particular, we have a cache timeout for the DBNode. This ensures that any stale key pairs in the cache get evicted
- Process disaster recovery
    - The reverse proxy has a health check that runs continuously in the background. All it does is loop through the current hosts that are active and execute a simple GET request. If a host does not respond, we try to ping it two more times and if it does not send a response, we mark it as dead and remove it from the list of active hosts. We constantly check if the list of active hosts is less than the minimum number of hosts and if it is, then the reverse proxy spins up another host to take the dead one's place. This ensures that the number of active hosts meets the minimum number of hosts that must be active.
- Orchestration
    - Custom build script for Java files. The database uses pre-build binaries for DBNode and DBController
    - Deploy Node bash scripts that deploy host (URLShortner) & DBNode on target machines

    - Reverse proxy and DBController have a hosts file which they use as the initial pool of machines they can use for the deployment of hosts & DBNodes respectively
    - A main run.py python script that builds java files, deploys database network, starts reverse proxy, and launches the monitoring dashboard/client UI.
        - Accepts command line args for system config: number of hosts, number of DBNodes, replication factor
        - Can take down the entire system & clean up as well when done
- Health checks
    - We have two health checks that are implemented, one on the reverse proxy and one on the DBController. The reverse proxy checks on the hosts while the DBController checks on the DBNodes
    - Covered in detail in previous points
- Horizontal Scalability
    - Reverse proxy
        - As mentioned earlier, during the health checks, if the reverse proxy finds a current active host to be dead, it will remove it from the list and check if the length of active hosts is below the minimum number of hosts required. If so it will spin up more hosts on the fly
        - We have developed API routes that when called can add/kill specific hosts. The appropriate errors are thrown 400 if it is an invalid name, 500 if is unable to start the host (because the machine could be off) or kill the host (because it is not an active host). Through the UI the admin would just need to specify which host and submit.
        - An additional guarantee is that the admin can dynamically set the number of minimum hosts that should be up, again through the UI.
            - If the reverse proxy has 4 current active hosts and the admin sets a minimum host of 2, then the reverse proxy will only scale automatically if the current active hosts fall below 2.
            - If the reverse proxy has 4 current active hosts and the admin sets a minimum host of 6, then the reverse proxy will scale and automatically add 2 hosts so that it will reach the new minimum
    - Database
        - We have developed API routes that the admin can use to add or remove nodes. The DBController would be responsible for this
        - The appropriate errors are thrown 400 if it is an invalid name, 500 if it is unable to start the DBNode(because the machine could be off) or kill the DBNode (because it is not an active DBNode).
- Vertical scalability
    - Since we are multithreading the proxy server and the url shortener, a better processor with more threads means we can spawn more threads and instances which will improve runtime.
    - With a bigger RAM, this means we would be able to have a larger cache size than right now, which can speed up operations.

- Document:
    - Please refer to the README file

Discussion 2:

- We'll be comparing our system under 2 types of loads, 1 is read-only, and 1 is write-only. The tests will be on 2 configs.
- **Config 1:** 4 Hosts, 4 DBNodes, Replication factor: 2
    - Refer to figures in Discussion 3 - Config 2

Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/get	248715	0	7	1	189	119	3905.2	0.0
	Aggregated	**248715**	**0**	**7**	**1**	**189**	**119**	**3905.2**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/get	7	8	8	9	10	10	12	190
	Aggregated	**7**	**8**	**8**	**9**	**10**	**10**	**12**	**190**

Charts

- **Config 2 - Read Test:** 6 Hosts, 6 DBNodes, Replication factor: 2

- The read RPS for the 6-6 config is similar to the 4-4 config. This is because the bottleneck is not the hosts or database. The single load balancer is the bottleneck, which is limiting the RPS to ~4200

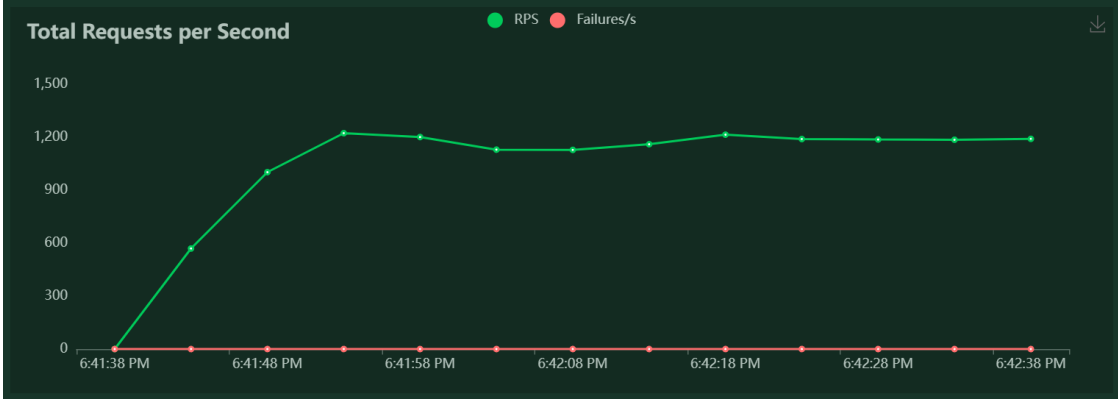
Request Statistics

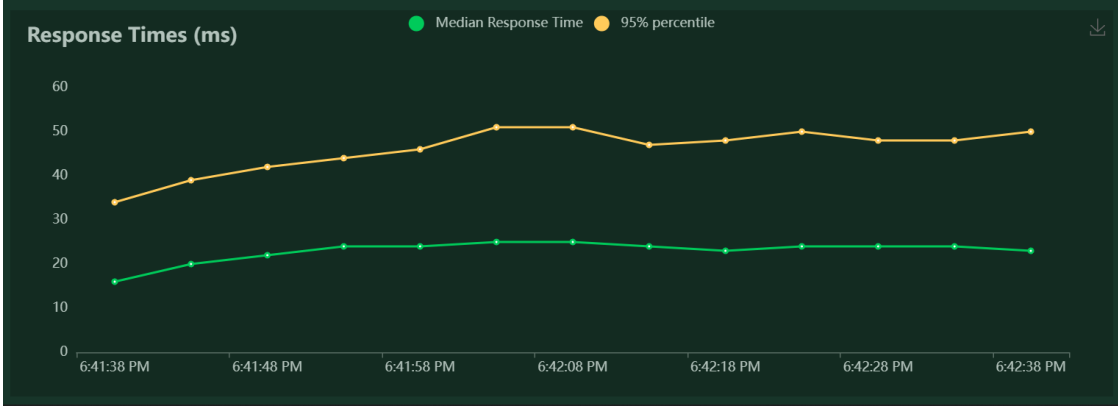
Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
PUT	/put	73727	0	26	5	158	61	560.0	0.0
	Aggregated	**73727**	**0**	**26**	**5**	**158**	**61**	**560.0**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
PUT	/put	24	26	29	34	41	48	66	160
	Aggregated	**24**	**26**	**29**	**34**	**41**	**48**	**66**	**160**

Charts





- **Config 2 - Write Test:** 6 Hosts, 6 DBNodes, Replication factor: 2
  - Here, we can see that the write RPS is about 1200, which is much higher than the 4-4 config (~ 950). This means that the number of hosts is where the bottle-neck is, and adding more hosts should further increase the throughput

- **Misc - Database Write Test:** 4 DBNodes, Replication factor: 2

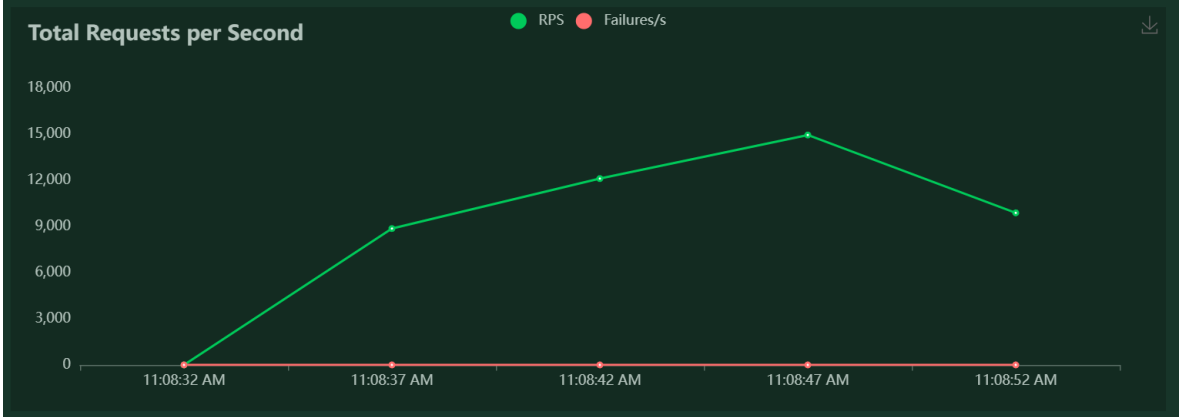
Request Statistics

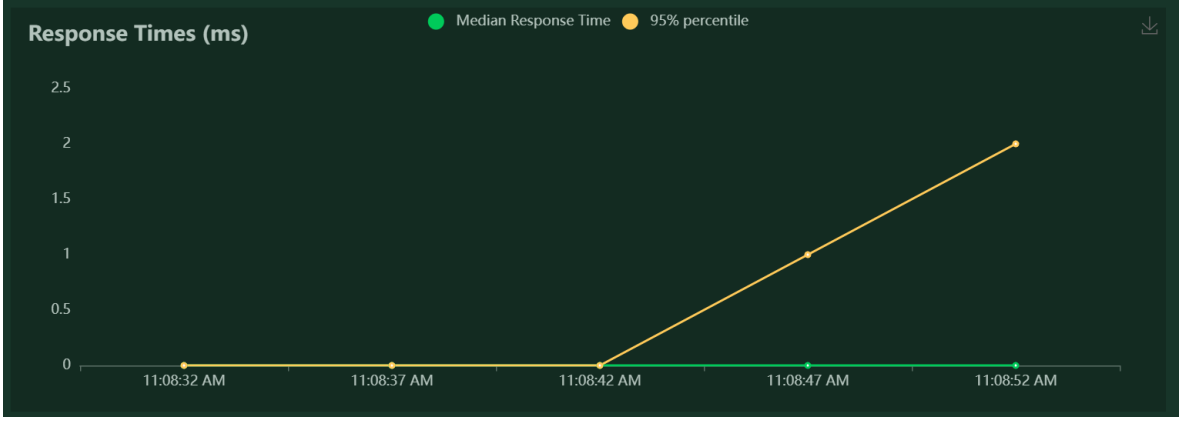
Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/set	237881	0	0	0	597	7	9687.7	0.0
	Aggregated	**237881**	**0**	**0**	**0**	**597**	**7**	**9687.7**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/set	0	0	0	0	0	1	6	600
	Aggregated	**0**	**0**	**0**	**0**	**0**	**1**	**6**	**600**

Charts

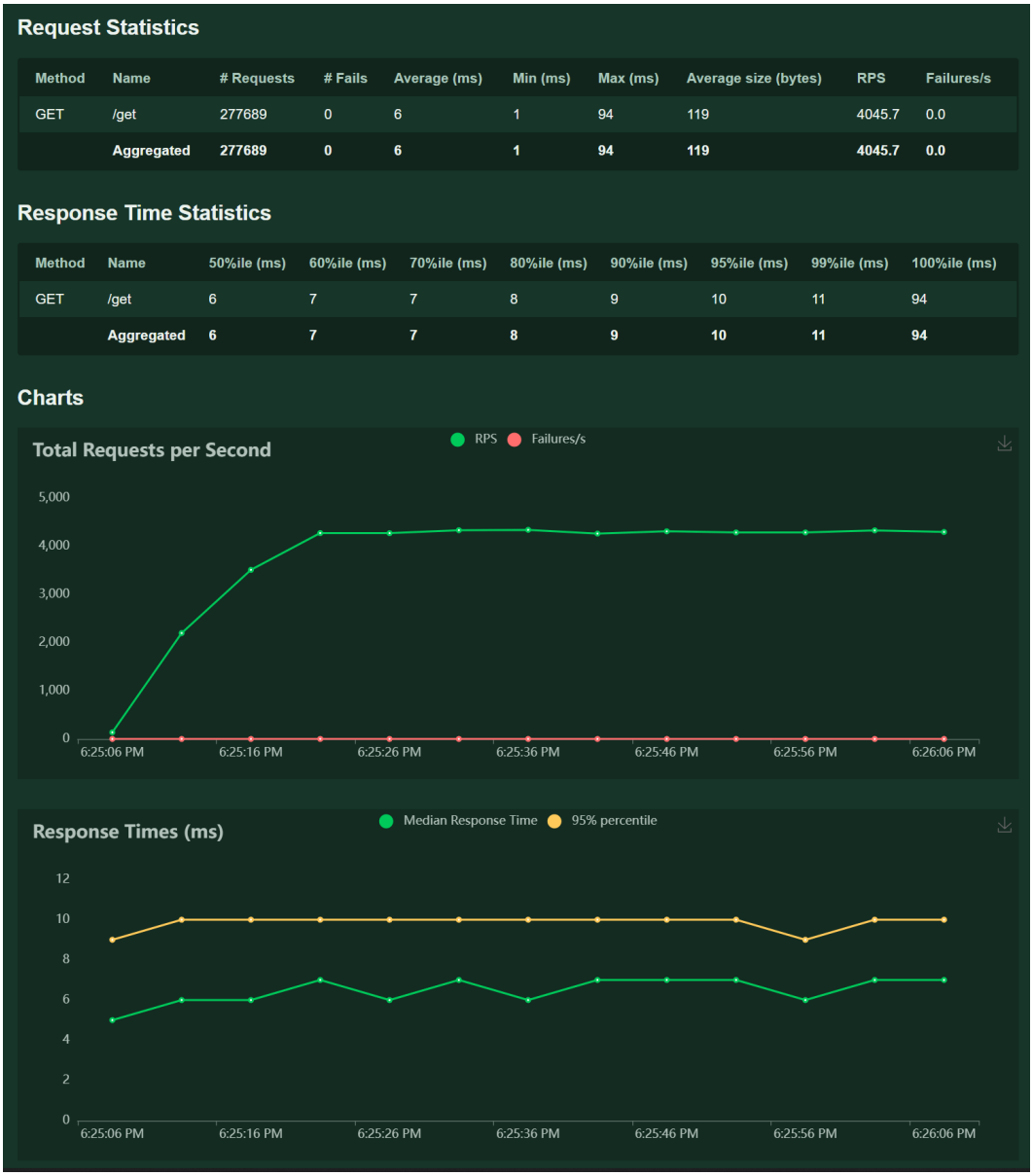




  -
  - This is a write test for the database with 4 nodes that were hit randomly by the users. This shows that our database can handle much higher throughput (~15,000 RPS at peak) if we had more gateways than just a single proxy.

Discussion 3:

- **Config 1 - Test 1**: 1 Host, 1 DBNode, Replication factor: 1, ran load test with 32 simulated users from 8 machines doing **reads** only



Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/get	277689	0	6	1	94	119	4045.7	0.0
	Aggregated	**277689**	**0**	**6**	**1**	**94**	**119**	**4045.7**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/get	6	7	7	8	9	10	11	94
	Aggregated	**6**	**7**	**7**	**8**	**9**	**10**	**11**	**94**

Charts

  - We were able to get an average of 4000 GET requests per second when the database had about 500 entries in it. Since reads are a very lightweight operation and can use the cache as well, it is expected to perform much better than writes (PUT)

- **Config 1 - Test 2:** 1 Host, 1 DBNode, Replication factor: 1, ran load test with 32 simulated users from 8 machines doing **writes** only

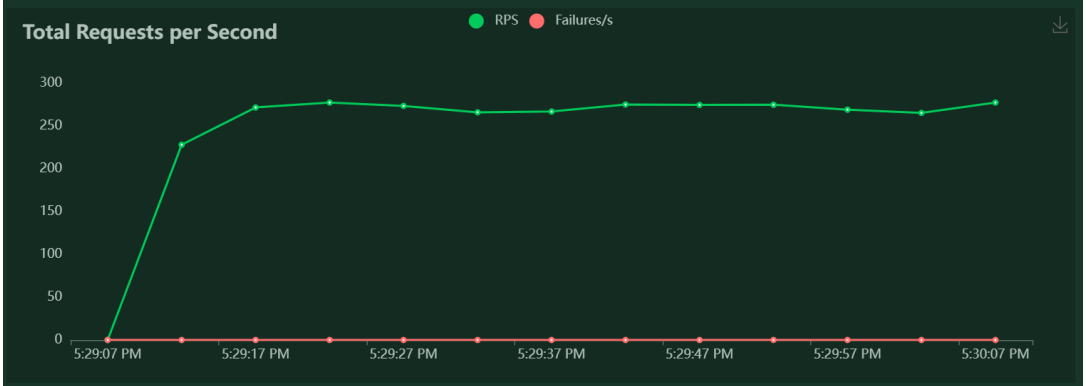
Request Statistics

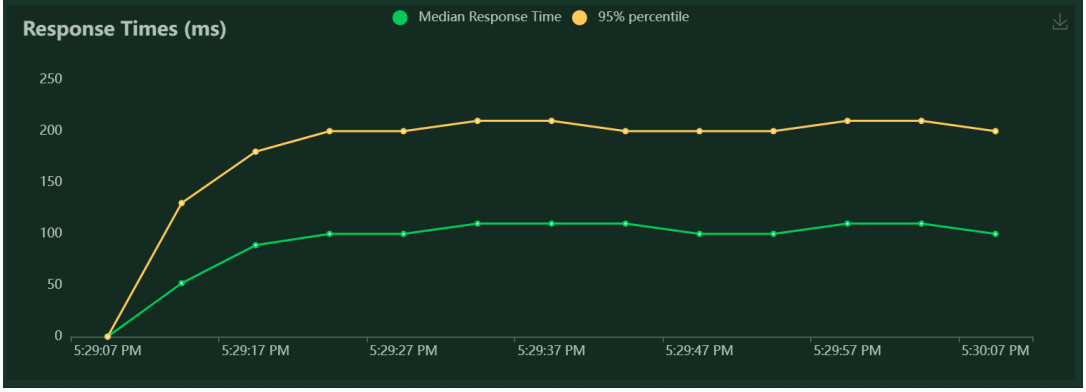
Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
PUT	/put	17141	0	113	8	367	61	271.5	0.0
	Aggregated	**17141**	**0**	**113**	**8**	**367**	**61**	**271.5**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
PUT	/put	100	110	130	150	180	200	250	370
	Aggregated	**100**	**110**	**130**	**150**	**180**	**200**	**250**	**370**

Charts





-
- We were able to get about 271 RPS with writes only. The response times are also significantly higher. This is expected as writes are significantly slower than reads, and they can't leverage cache which means every request has to travel through the entire network (from proxy to database) before it's responded to. Adding more nodes will significantly speed up this process, as writes can be distributed over different DBNodes and hosts

- **Config 2 - Test 1**: 4 Hosts, 4 DBNode, Replication factor: 2, ran load test with 32 simulated users from 8 machines doing **reads** only

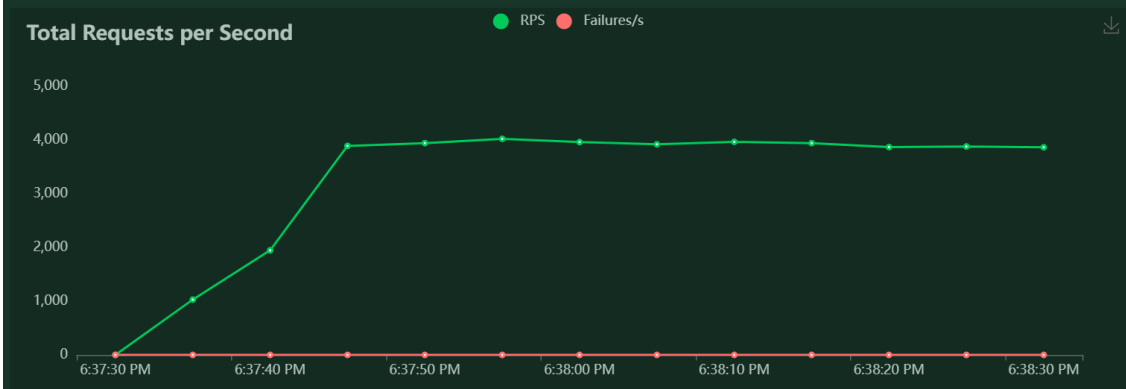
Request Statistics

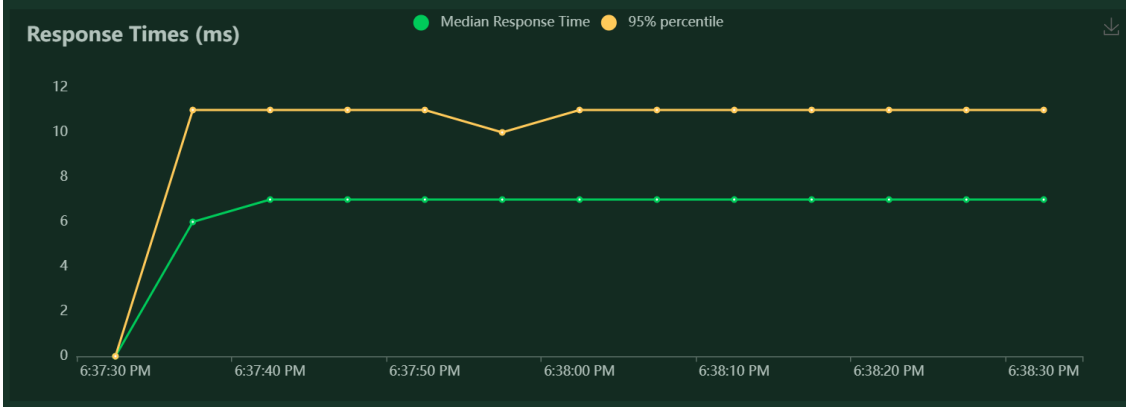
Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/get	246629	0	7	1	97	119	2250.7	0.0
	Aggregated	**246629**	**0**	**7**	**1**	**97**	**119**	**2250.7**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/get	7	8	8	9	10	11	13	97
	Aggregated	**7**	**8**	**8**	**9**	**10**	**11**	**13**	**97**

Charts





  -
  - With 4 hosts & DBNodes, the avg RPS actually goes down for reads, while the peak RPS stays about the same. This seems surprising at first, but the result makes sense when we actually understand what's going on. Since we are doing reads only with 500 entries in the database, in the case of a single node, after the first few hundred requests, most of the entries are in the hosts' cache. This

means the initial ramp-up time before reaching peak RPS is less, leading to increased avg RPS. But, in both cases, since there's only a single load balancer, which ends up being a bottleneck, constraining us to about 4200 RPS max. We predict that having more gateways, i.e., more load balancers that clients can connect to would significantly boost the peak RPS in this scenario, and we will see much higher throughput compared to the 1 host, 1 DBNode scenario

- **Config 2 - Test 2**: 4 Host, 4 DBNode, Replication factor: 2, ran load test with 32 simulated users from 8 machines doing **writes** only
    -

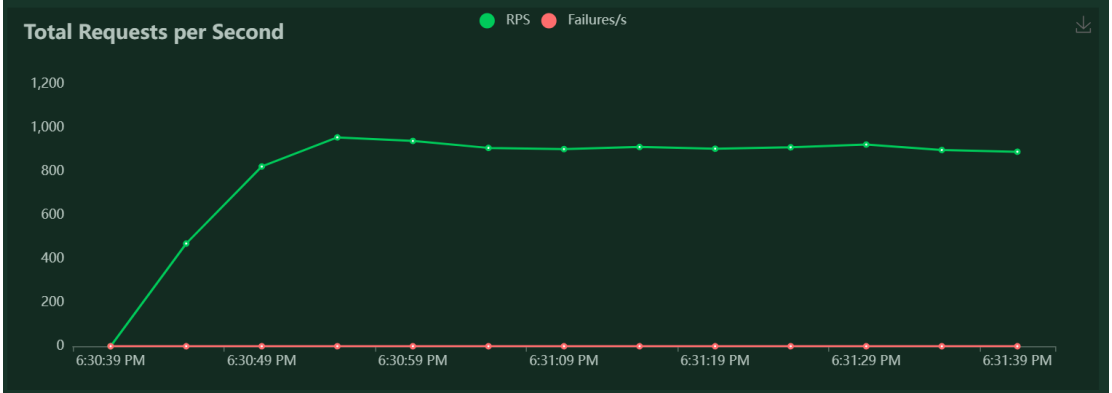
Request Statistics

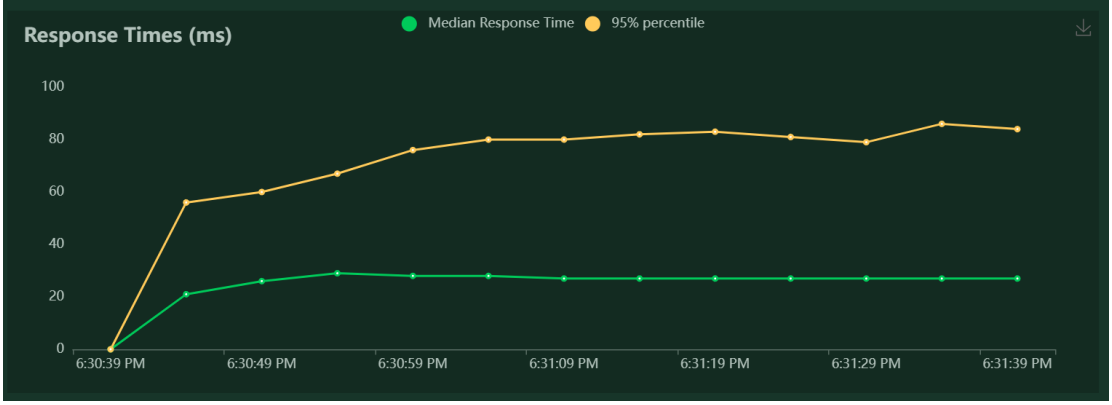
Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
PUT	/put	55472	0	33	6	207	54	905.3	0.0
	Aggregated	**55472**	**0**	**33**	**6**	**207**	**54**	**905.3**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
PUT	/put	27	32	38	47	61	78	110	210
	Aggregated	**27**	**32**	**38**	**47**	**61**	**78**	**110**	**210**

Charts





    - With 4 hosts & 4 DBNodes, we see about a 4x increase in write RPS when compared to the 1 host & 1 DBNode test. This clearly shows that for writes, the number of hosts/DBNodes was the bottleneck. The 95th percentile response times for this case are also much more reasonable than the first test case

Discussion 4:

- Curl/Postman: we needed to use curl/postman because the application relies on HTTP requests to send requests from the client to the proxy server to the hosts and back
- Locust: To load test our system and generate performance metrics, we use locust - a python load testing framework. We used locust to do distributed load testing. The graphs shown above are generated from locust