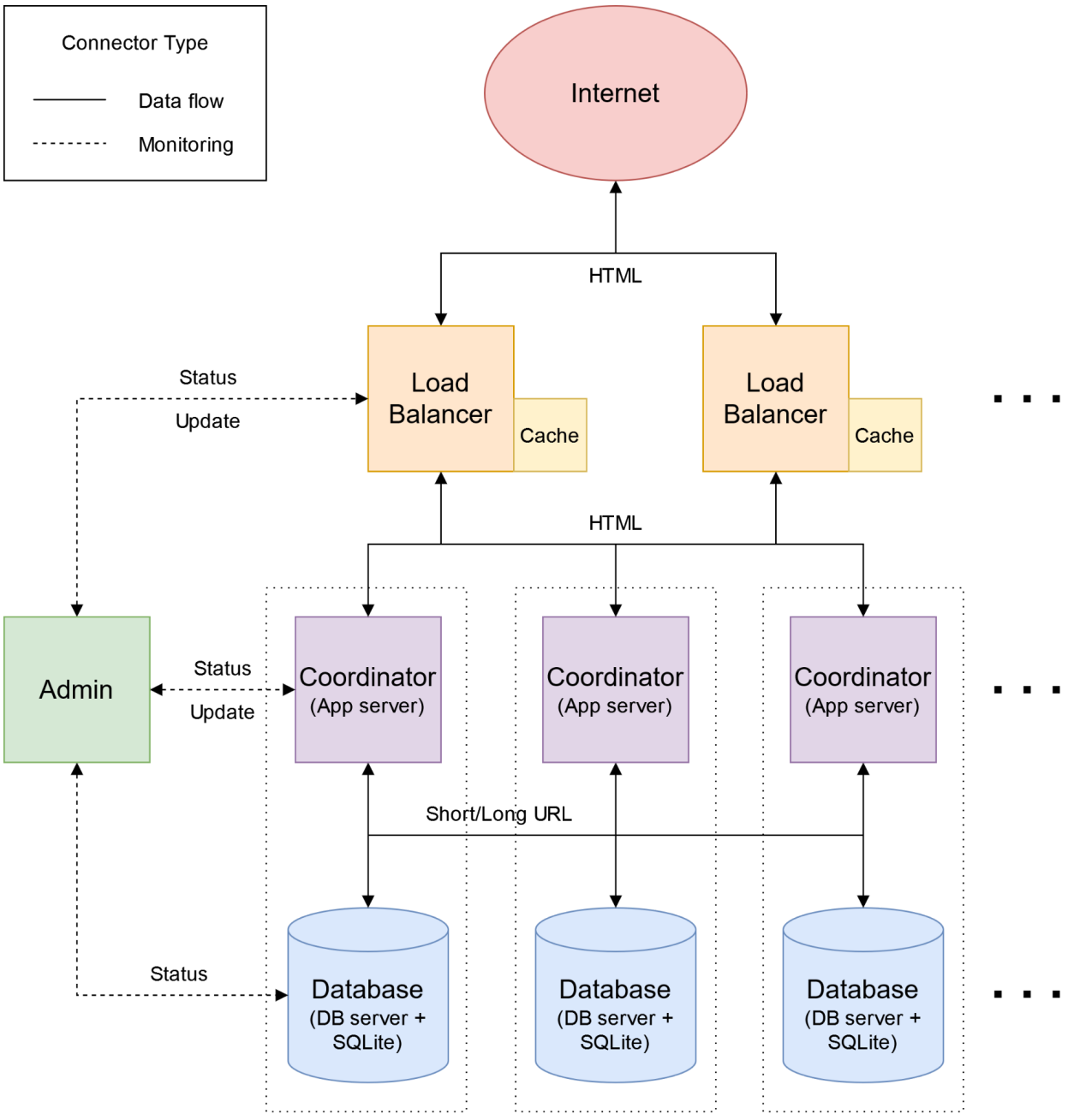
Assignment 1 Report

CSC409

Youan Cong
Wentao Zhou
Takafumi Murase

System Architecture

Diagram



Note: Admin communicates with every server

Consistency [Partial]

Our system does not have full strong consistency and only partial local consistency. The database system has a ring structure where each registered data has a responsible database node and is backed up in two neighbouring databases. This means that if a data point should be written to host X, we will also save it to hosts X+1 and X-1. If we cannot read host X at a later read time because it goes down, we will search for the data in hosts X+1 and X-1. This implies that for only the range consisting of a database node and its two neighbours, they are consistent for most of the runtime. Therefore, due to the tradeoff between availability and consistency, our system is only partially and locally consistent.

Availability [Yes]

Our system has good availability because users can at least get a response in almost all cases, except for a failure of the load balancer/reverse proxy. In the case of a node software failure, the system is still available since we have implemented an automatic recovery system such that the software that goes down will come back up within a few seconds. In the meantime, users can still perform write and read operations because other nodes can handle the requests. In the case of a dedicated database hardware failure for a specific data partition, that is, a host physically goes down, our system is still available and users can still perform write and read operations because we have data replicas in two adjacent database nodes. This only becomes an issue when the host of the database alongside the adjacent two hosts goes down at the same time since we only have a replication factor of 3 for each data partition. At this point, users can still get an appropriate responding webpage. In the case that all database hosts are down, similarly, the coordinator can still return the appropriate webpage. Further, in the case that one or more coordinators go down, the system can still handle requests as long as there is at least one coordinator up, and the user will get a static page even if all coordinators are down. Finally, in the case that all load balancers/reverse proxies go down, our system will not work, however, as a single point of user input, at least one of the load balancers/reverse proxies are often assumed to be working all the time.

Partition tolerance [Yes]

Our system has good partition tolerance since we have database replication for each partition, which means that even if a database node for a partition goes down, we can still find replicas from the two adjacent database nodes while the system will still be operational. Even in the worst case, assuming all of our database nodes and all coordinators are down, the system can

still continue to operate despite data loss while returning appropriate responses, such as a static page or a URL from the load balancer's cache.

Data partitioning

We partition the data by dynamically assigning a responsible database node to each data point using a function based on the hashing algorithm and the number of active database nodes. Specifically, suppose the number of database nodes currently active is N, we utilized Java's hashcode() function, which hashes a string to a data space of -2^{31} to $2^{31}-1$, along with a floormod N calculation to further map an input URL to database nodes with values 0 to N-1.

Data Replication

In addition to partitioning, our system utilizes a ring structure where each database node will be assigned a partition according to the total number of database nodes in the system. Since our replication factor is 3, all data will be replicated in at most two adjacent database nodes. More specifically, let N represent the number of active database nodes, when N>2, each data in partition X will have replicas in the partitions X-1 and X+1. However, when N=2, the data in a database node will only have replicas at the other database node. Our system will use the data replication strategy discussed above when writing to the database. For reads to the database, the system will try to read from the target node first, followed by reading from its two neighbours if the target node is not available (X+1 first, then X-1).

Load balancing

We have decided to implement the load balancer in Java using a multi-threaded reverse proxy server. A reverse proxy server can redirect requests from clients to the appropriate coordinator and allows easy management of shared data between threads. We separated the load balancer into two parts: a shared data class and a logic class. The shared data class uses two synchronization locks to keep an internal server list and cache consistent without losing too much performance from threads waiting for the locks. The logic class has a thread that listens for client requests and spawns additional threads to handle them. It also has another thread that cleans the cache periodically (more in the caching section). The load balancer will attempt to redirect client requests up to 5 times before giving up to prevent occasional socket errors from affecting performance and availability. Lastly, we used the round-robin method for distributing requests. Round-robin is an excellent choice because it can efficiently balance the workload between coordinators since the tasks performed by each coordinator are small and approximately even.

Caching

The load balancer caches short and long URL pairs upon successful GET and PUT requests. We cache upon PUT requests because recently created short URLs are likely to be accessed immediately. The cache only exists on the load balancer's memory, so any cached data will be lost when it goes down. However, this is not a big problem because the data in the cache has an expiration time of 60 seconds to improve consistency. Without cache clearing, some load balancers could have outdated data whenever there are multiple load balancers since only one will get the PUT request to update a short URL. We have also decided to only store long URLs instead of entire HTML documents to save space on memory. The redirect response will always be static, so the load balancer can quickly generate the HTML document without increasing the workload.

Process disaster recovery

Our admin service manages the system's process disaster recovery. After performing a health check, the admin will attempt to launch unresponsive threads using the Runtime API from Java's standard library. The admin will be able to recover a process within roughly 2-4 seconds after detecting that it is unresponsive. Since the admin is solely responsible for recovering processes, it is somewhat of a single point of failure. Even if the admin is down, the system can still function if there is a link between a load balancer, coordinator, and database (or if the load balancer is up and the data is cached). A script to act as disaster recovery for the admin would not help much because it only shifts the SPOF to another process.

Data disaster recovery

Due to our implementation of data replication to the two adjacent databases, our system will be able to recover data upon a data disaster happens. Once the coordinator didn't receive a correct response from the target database, it could be a signal of a database failure. In this case, the coordinator will try to reach out to one of the two adjacent databases in the order of right first and then left. If any one of the databases responds, it will return the exact same value as from the target database.

Orchestration

We use three bash scripts to orchestrate our service. These scripts rely on the "config" and "hosts" files to get the appropriate data. The "config" file contains process type headers (e.g.

"LoadBalancers:") followed by host and port pairs, and the "hosts" file contains all hosts in the URL shortener system. The first script, "setup.sh", creates the SQLite database. The second script, "launch.sh", reads the "config" file to start processes on the appropriate hosts. The last script, "shutdown.sh", reads the "hosts" file to terminate all running service processes.

Healthcheck

Our admin server takes responsibility to perform health checks on each of the existing services in our system. For every two seconds, the admin will iterate through a full node list that contains all the node information we have in the system and sends out a status check code simultaneously. If a node is alive, it will return a response with its node type for example, "DBALIVE" which shows that this node is a database node. Once the admin receives the responses, the nodes will be recorded to dedicated lists of active nodes for each node type. On the other hand, if the nodes don't respond while being presented in the "active" lists, or in case there is a new node added but not appeared in the "active" list, such changes will be then notified to all the nodes in charge.

Horizontal Scalability

The method to scale our system horizontally is by using the scaling feature in the admin console, where the user can input the IP and node type. Once a new node was added to our system, the performance will be increased from two different perspectives. The first perspective is speed. More hosts mean more processing power will be added to handle the requests and improve efficiency. The second perspective is workload. By adding a new database node, we increase the number of partitions such that each database is in charge of fewer data points, reducing the traffic between nodes by a significant amount. However, for coordinator nodes and load balancers, there is a bottleneck of time they need to take to decide where to send requests.

Vertical Scalability

The way our system performs multi-threading is by using the thread pool from the java ExecutorService with a fixed number of threads. The argument passed to the FixedThreadPool we are using indicates the maximum number of threads that can be run simultaneously, and it is set to 8 in accordance with the CPU threads number in the lab machines. For further vertical scaling, one can change the number of threads in the thread pool in case there is a better CPU with more than 8 physical threads, which improves the performance of our system.

System Performance

Load Test 1

Tools: Apache Bench + GNUPlot + ab-graph
Requests: 1000
Users: 10
Requests/s: 2725.00

Analysis:

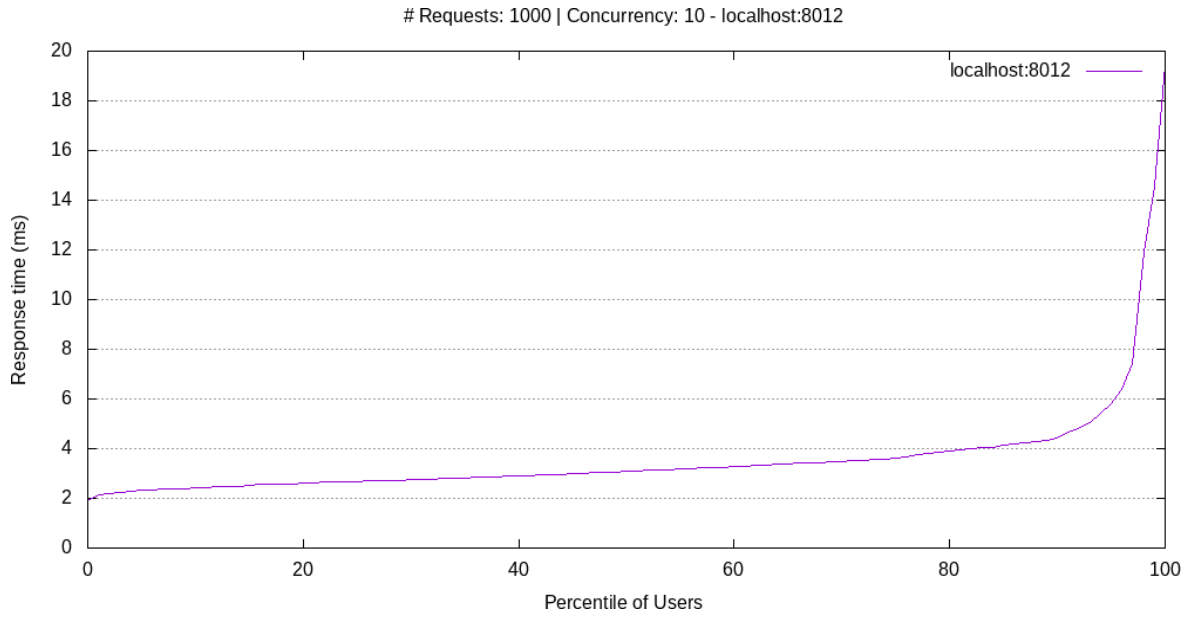
From the percentage graph, it can be concluded that 90% of requests can be responded to within 5ms when the system is accessed by 10 users concurrently, while some of the numbers clearly spike on the right-hand side, which represents a small portion of the outliers. It represents that the highest peak response time is around 19ms at such a setting. Also, according to the values graph, we can see that most of the requests respond in around 3ms.

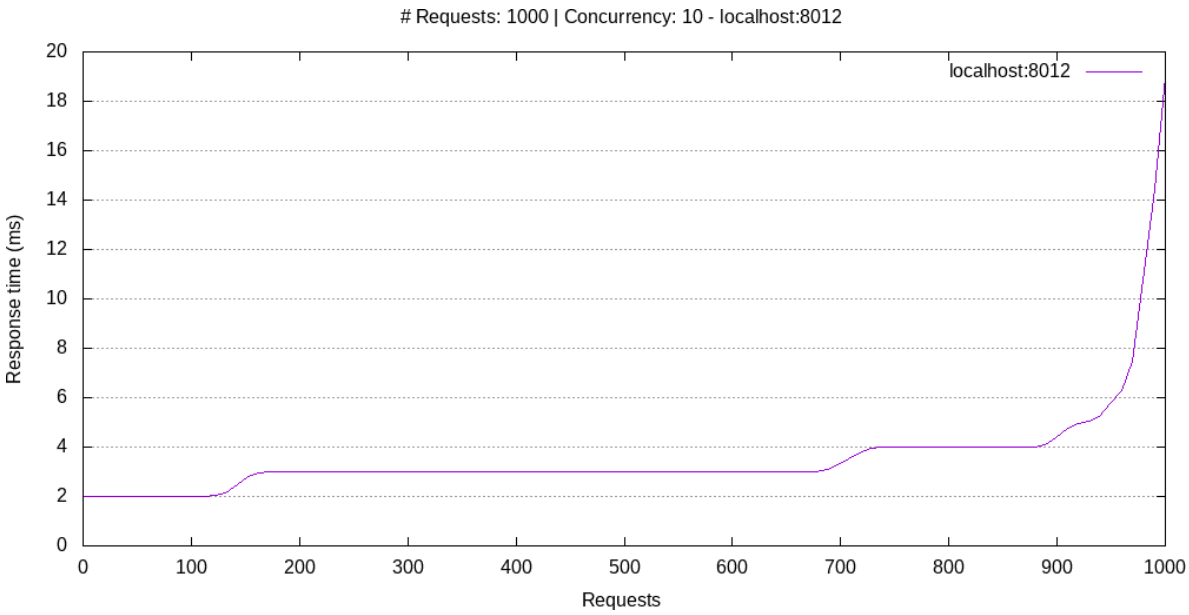
How we used the testing tools:

Apache Bench was used to send HTTP GET requests to our URL shortener system. We used the command line concurrency flag to simulate multiple users sending requests at the same time.

GNUPlot was used to generate the two graphs.

ab-graph was used to help automatically transform Apache Bench's output into GNUPlot graphs.


Requests: 1000 | Concurrency: 10 - localhost:8012
localhost:8012
Response time (ms)
Percentile of Users
0
2
4
6
8
10
12
14
16
18
20
0
20
40
60
80
100


Requests: 1000 | Concurrency: 10 - localhost:8012
localhost:8012
Response time (ms)
Requests
0
2
4
6
8
10
12
14
16
18
20
0
100
200
300
400
500
600
700
800
900
1000

Load Test 2

Tools: Apache Bench + GNUPlot + ab-graph
Requests: 10000
Users: 50
Requests/s: 2968.51

Analysis:

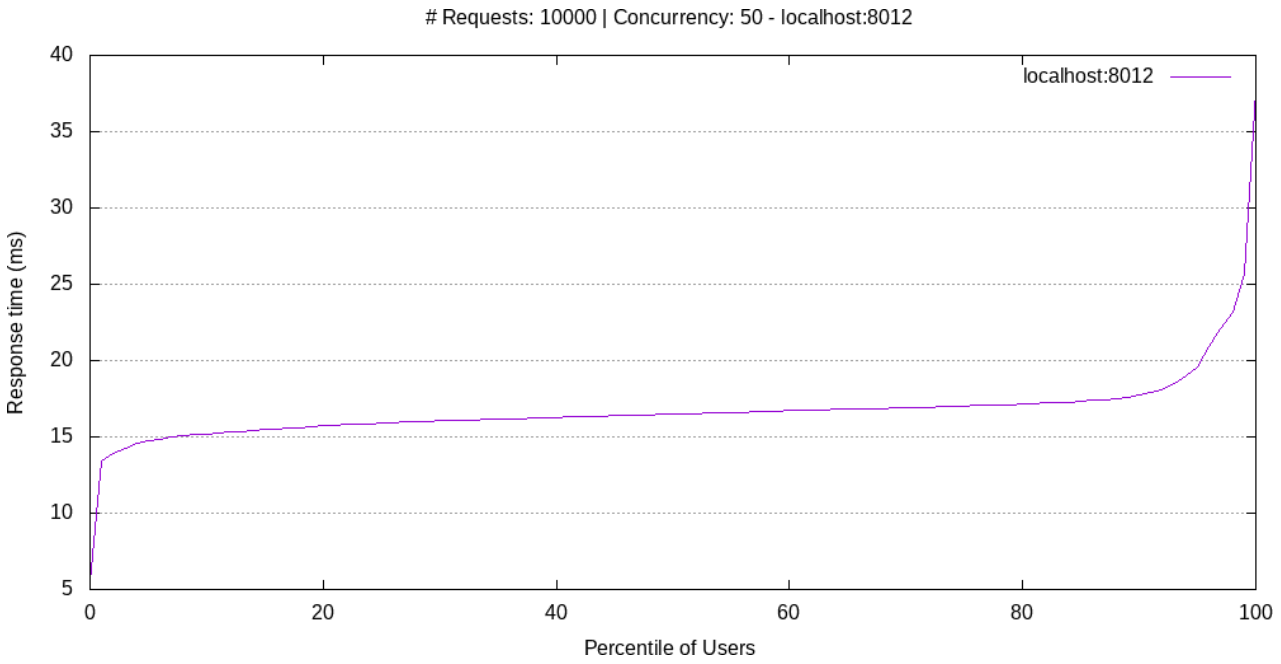
Compared to the previous load test, 90% of requests can be responded to within 17ms when accessed by 50 users in parallel at the same time, according to the percentage graph. At this point, the worst response time spikes to around 38ms, which means that with the current settings, users will wait up to 38ms to get a response from the server. In addition, according to the value graph, we can see that most of the response times to user requests are within the 15-20ms range.

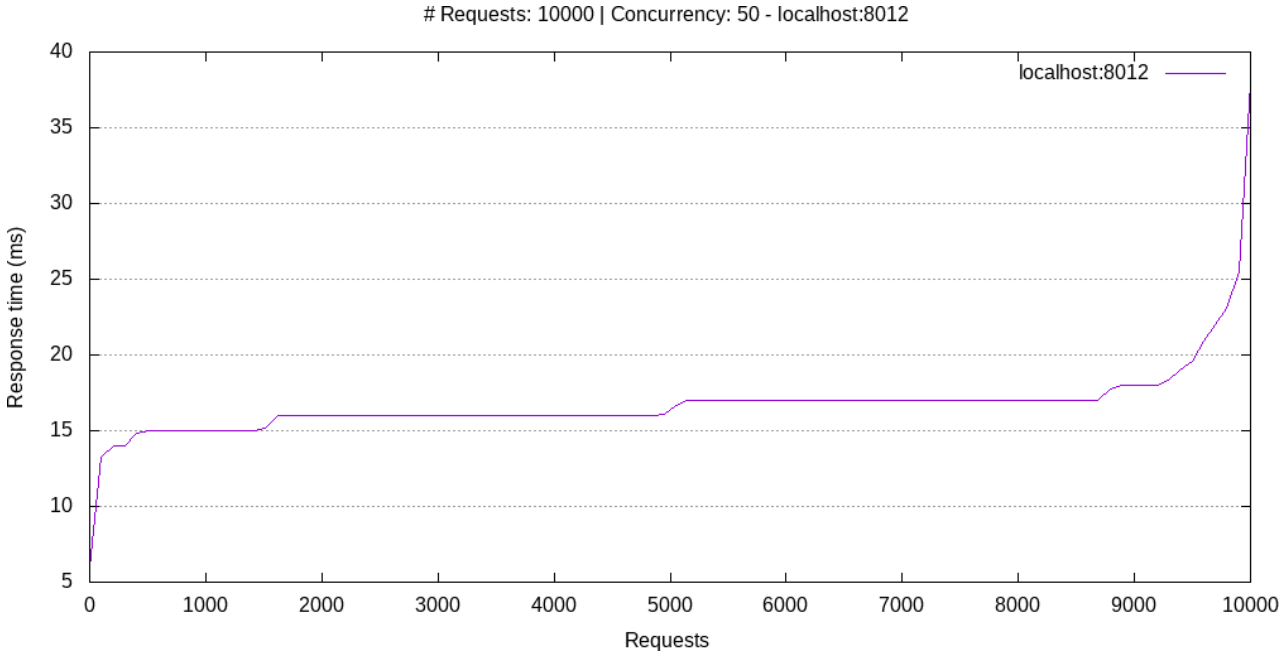
How we used the testing tools:

Apache Bench was used to send HTTP GET requests to our URL shortener system. We used the command line concurrency flag to simulate multiple users sending requests at the same time.

GNUPlot was used to generate the two graphs.

ab-graph was used to help automatically transform Apache Bench's output into GNUPlot graphs.


Requests: 10000 | Concurrency: 50 - localhost:8012
localhost:8012
Response time (ms)
Percentile of Users
40
35
30
25
20
15
10
5
0
20
40
60
80
100


Requests: 10000 | Concurrency: 50 - localhost:8012
localhost:8012
Response time (ms)
Requests
40
35
30
25
20
15
10
5
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000

System Scalability

Number of hosts vs read requests per second

The number of hosts and read requests per second are positively correlated, and it has been tested that each increase increases the number of read requests per second by roughly 50-60 in a setting of a single user sending a small number of requests. However, there is a marginal effect that starts to emerge when the number increases to 5 hosts and above, because of the cost of interconnection as well as the time load balancer and coordinators need to take for deciding which node it will allocate for data partitioning.

Number of hosts vs write requests per second

The number of hosts and write requests per second are also positively correlated but compared to the read speed, the write speed will be more dependent on the connection between the database nodes and the coordinator nodes. Each input data point will have to be replicated by 2 neighbouring database nodes, which represents that for our system, adding more hosts will have less improvement on the overall write speed, and the node distribution overhead will be higher because a larger list in the ring structure leads to more time to determine the neighbouring nodes. Therefore improvement to the write speed by adding hosts will be lower than the read speed, only about 1/2 to 1/3.

Number of hosts vs data stored

Since we utilized hash-based data partitioning, adding more hosts means more distributed data partitioning, so the amount of data to be stored per host should be reduced evenly. In the ideal case, assuming we have N database nodes and when N is large enough, each database should be responsible for keeping 3/N of the total data storage, where 3 is the replication factor we used.

Tool Used

1. Apache Bench
   Apache Bench is a tool for load stress testing HTTP servers, especially to test how many requests a server can handle in a second as well as the response time. We use scripts to simulate multiple requests from multiple users concurrently, the scripts are included in the A1 files.
2. GNUPlot+ab-graph
   GNUplot is a built-in portable CLI graphical tool. In addition, we used a third-party drawing template to draw the request statistics generated by the ab test. The detailed scripts are included in the A1 files.
3. Bash scripts
   Bash scripts are terminal scripts that come with Linux and can be run directly. We use bash scripts to kill some processes to test the system's fault tolerance and stability of the system as well as the monitoring system.
4. Python scripts
   Python is a very versatile and portable programming language, in this case, we use python scripts to simulate user requests because we need to remotely deploy the system on multiple lab machines, but we cannot use the browser remotely via ssh. So we need scripts to simulate access to assist in testing. The detailed script is included in the A1 file.