Group

- Nathan Wong
- Billy Zhou
- Shivam Bhatt

Description and Goals

The goal of this project was to implement a level 7 HTTP load balancer with a focus on core functionality such as static and dynamic routing, sticky session handling, and failure prevention/handling.

The specific goals were as follows:

- Implementation of a reverse proxy that could handle concurrent connections from scratch (using python socket API)
- Implementation of static and dynamic load balancing algorithms:
    - Round Robin (non-weighted, weighted)
    - IP Hash
    - Least Connections
    - Least Response time
- Load shedding to prevent failures
- Server health checks
- Server failover (on failure)
- Create a test environment in mininet to properly test distributed applications

Contributions

Shivam

- Least connections, least response time, and weighted round robin implementations
- Testing performance of those three algorithms
- Performance plots for RTT and throughput over time

Billy

- Initial setup
- Consistent hashing strategy
- Sticky session

Nathan

- Test helpers (sending requests, request result wrapper, plots)
- Failover test
- Load Shedding service and associated tests
- Socket reverse proxy implementation
- (non-weighted) Round Robin strategy
- Health Check service
- README/documentation

Setup, Run, and Tests

NOTE: These instructions with more detail can also be found in the top level README of the project.

Environment Setup

The project should be copied to and run on a Mininet VM (setup as in Lab 3/SR project). This VM image is essential as it allows us to use CPULimitedHosts to properly test the distributed system later on. Mininet CPULimitedHosts are broken in newer versions of Ubuntu. Once the project is copied or cloned into the VM, run the `sudo setup.sh` script in the load_balancer folder to download dependencies. It may be necessary to do `sudo chmod +x setup.sh` if there are permission errors.

It's also recommended to increase the processor count and memory that is assigned to the VM as some tests are computationally expensive. While it is possible to run the VM on a lab machine, we find that resources are too limited to test at scale. In our tests, we ran on a VM with an allocation of 12Gb of memory and 3 CPU cores.

The VSCode Remote Development Extension Pack is useful if you want to see the resulting graphs from test runs and run in a nicer development environment.

Running the Load Balancer and Topo Setup

To make the load balancer as customizable as possible, we've defined a JSON format for load balancer configs:

```
{
  "load_balancer_ip": "10.0.0.254", /* IP of the LB host */
  "load_balancer_port": 80, /* What port the LB is served on */

  /* How often health checks trigger (seconds) */
  "health_check_interval": 3,

  /* How long a health check should wait before marking the server unhealthy */
  "health_check_timeout": 2,

  /* For each server, dedicated health check endpoint path */
  "health_check_path": "/health",

  /* Each usable server name, IP, and port */
  "servers": [
    { "name": "s1", "ip": "10.0.1.1", "port": 80 },
    { "name": "s2", "ip": "10.0.1.2", "port": 80 },
    { "name": "s3", "ip": "10.0.1.3", "port": 80 }
  ],

  "strategy": "round_robin", /* round_robin, hash, ... */
  "sticky_sessions": false, /* enable sticky sessions? */
  "debug_mode": true, /* print to lb.log? */
  "load_shedding_enabled": true, /* load shedding enabled? */
  "load_shed_params": {
    "sim_conn_threshold": 5, /*how many simultaneous connections before shed*/

    /*hard - shed all above thresh, exponential - probability based*/
    "strategy": "hard"
  }
}
```

In mininet, the load balancer, clients, and servers are individual hosts. The load balancer is connected by switches and links to each client and server (IE the only connection for a client or server is to the load balancer). Commands should be executed from the /load_balancer directory.

For each server host, we can start up an application server instance. In our tests, we use the command `python3 test/setup/test_server.py 80 "hello from {server_name}"` to start a simple server with a healthcheck endpoint served on port 80 that responds with “hello from s{1,2,3}” to a GET request.

The load balancer can be started on the dedicated load balancer host with the command `python3 -u run_load_balancer.py {`*`config.json path`*`}` to start the load balancer with options as configured in the JSON file.

From the client’s perspective the load balancer *is* the application, so when clients want to send a request to the application, they address it to the load balancer’s IP. The load balancer then forwards the request and responses to the appropriate server and back to the client. In testing, we use curl to send GET requests to the application.

Running Tests

For convenience, and to test our implementation, we've included a test suite under `load_balancer/tests`. From the root directory, you can run these tests using the `command sudo python3 -m test.tests.test_{test-name}` which will output test results to the command line, relevant plots in `load_balancer/test/results` and debug logs can be observed in `lb.log`.

Implementation Details

Tech Choices

We chose to implement everything in Python since we were all familiar with the language. Although Python has many packages that can be used to simulate the functionalities of a LB, we deliberately avoided them in order to make use of materials learned in class and challenge ourselves to implement them from scratch.

Reverse Proxy Implementation

The core task of the load balancer is to act as a reverse proxy. We decided to self-implement this using python's socket API rather than rely on pre-existing proxy implementations such as nginx. All of the implementation for this area sits in `load_balancer.py`.

We bind a socket to the IP and port as specified in the configuration JSON file then spin in a loop, waiting for a client connection. When a client connection is ready, a server is chosen for that connection (more on this later) and a thread for the function `handle_connection()` is created to handle the communication between client and server. In this function, we set up a socket connection between the load balancer and the chosen server and forward the client's request to the server. The server will then send its response back to the load balancer, which forwards it back to the client socket connection. This is implemented using the `select` function which blocks until one or more sockets (either client or server) are ready for I/O. Data that is received from the client is immediately sent to the server (and vice versa). Once no more data is received, both connections are closed and the thread is closed.

Server Selection

Finding which server to forward is also a core load balancer task. We encapsulate data about each server (name, ip, port, healthy, auxiliary data) in a Server class (serv_obj.py) and store each Server in an array on the load balancer as a global variable. A global lock for the server array was also necessary to avoid a server being marked unhealthy by the health service (running in a separate daemon thread) in the middle of server selection and to avoid race conditions when updating auxiliary information (e.g. number of global connections).

For extensibility and modularity, we used a Strategy pattern for each server selection strategy. We implemented various algorithms including round robin, hashing, least connections, and least response time. Our algorithms also include a weighted option if the servers have weights associated with them.

Round robin has a simple implementation and maintains a current index which is incremented to the next (healthy) server whenever a new server needs to be picked. Weighted round robin extends this by giving each server as many turns in a row as its weight, with the weight defaulting to 1 if it is not set for a server. The least response times algorithm selects a healthy server with the minimum average RTT response time divided by the weight of the server defaulting to 1. The least connections algorithm selects a healthy server with the fewest active connections divided by the weight for that server defaulting to 1. Our implementations of the least response time and least connections algorithms support both network topologies with server weights and without. If the servers have been assigned weights, then those weights will be used while determining the destination server. Otherwise, the weights will default to 1 and effectively be ignored.

Hashing matches the IP of the client to the nearest (in one direction) healthy server node on the hash ring to get the server responsible for the client. On initialization of the strategy, each server in the topology is assigned 100 nodes/points on the hash ring; the hashing algorithm ensures they are spread out evenly. When LB receives a client request, the source IP gets hashed to some point on the ring, and the algorithm returns the server of the first node that has a greater hash than source IP. In our implementation, the hash ring is split into two parts, a dictionary that maps nodes to their server and a sorted list of the nodes, to make searching easier. We chose to use a hash ring instead of modulo to reduce the number of redirections when the status of servers changes.

Health Checks

A dedicated health check endpoint has to be implemented on the application server. In our test server (test_server.py), this is a simple endpoint at /health that responds with 200 OK on a GET request. The health check service is implemented in health_check.py and is responsible for keeping track of auxiliary information and the current health of each server. When the service is started, it runs in a thread and sends health check GET requests to the health check endpoint of each server using the specified health check path, health check interval, and timeout values provided in the config JSON. For each check, it measures the RTT and adds that to a weighted moving average for each server stored in the auxiliary info of the Server object. Additionally, on a timeout or a failure response from the server, the corresponding Server object is marked unhealthy (obviously obtaining the server lock first to avoid invalidating a server while server selection is happening).

Load Shedding

Load shedding in our implementation is based on a maximum threshold of simultaneous connections to the load balancer defined as `sim_conn_threshold` in the config JSON. Connections are incremented/decremented whenever a connection thread is created or finished (either due to client/server closing the connection or due to an error). Whenever a client connects to the load balancer, if the load shedding feature is turned on, the load shed service decides whether to shed the connection based on the current number of current simultaneous connections. If the load shed strategy is set to “hard” then any connection that exceeds the threshold will be shed, while the “exponential” strategy returns an exponentially increasing probability of dropping a connection based on how far above the threshold the current number of connections is (following the formula $1 - e^{-K(simConn - thresh)}$) where K is some scaling factor heuristically set to 0.3. When a connection is dropped, an HTTP 503 error is returned to the client from the load balancer without need for any server to process a request.

The initial intent for load shedding was to make a resource-based load shedder that would prevent any single server from being overloaded by considering its CPU and memory usage. However, due to technical limitations in mininet, this proved infeasible (more on this in discussion).

Sticky Sessions

If sticky sessions are enabled in the setup file, it takes priority over the strategies. Ideally, the client would generate a unique identifier and pass it to the LB in the header of their request. This ID would then be used to remember which server they were assigned by one the strategies and any subsequent request with the same ID will go to the same server bypassing the use of strategies. If no ID is given, their IP is used instead. In our tests, all clients have unique IPs so ID is not necessary, but should multiple clients be under the same IP like in school networks, ID can be used to better distribute load. The sticky mapping also has an expiry time, set by the constant STICKY_TIMEOUT, and refreshed by subsequent requests. Once expired, mapping is voided, and a new server is assigned using one of the strategies. A simple dictionary of IDs to a tuple of the server and last request time is used.

Testing/Test Base

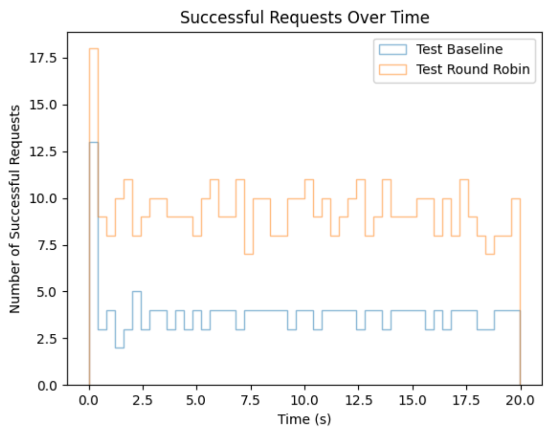
In `load_balancer/test/setup` we set up modular mininet topology classes that take in parameters for number of clients and number of servers then place a load balancer in the middle to connect clients to the application. Additional parameters include the fraction of CPU time given to all clients, fraction of CPU time given to each server instance, and the json configuration file to set up the load balancer.

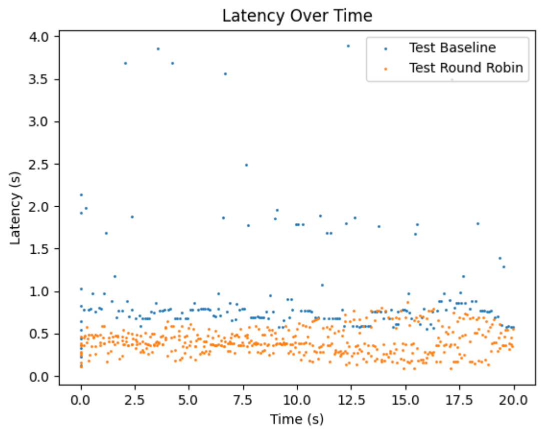
We use these topologies as the base environment of each test (e.g. 10 client, 3 server in test_round_robin). Every test also relies on clients sending simultaneous requests to the application (load balancer) to test functionality and performance. To assist with this, We define a helper object and function to assist with execution of requests and keep track of request results.

The file `req_result_obj.py` defines `RequestResult` to keep track of a single request result and data such as the time the request was sent and received. It also contains convenience methods to determine the status of the request. Additional helper functions in the file take in a collection of request results and output summaries or plots (using matplotlib) to aid with visualization. In `send_request_helper.py` we define a helper function to send requests to the application from a single client for a set result of time and append results to a results list.

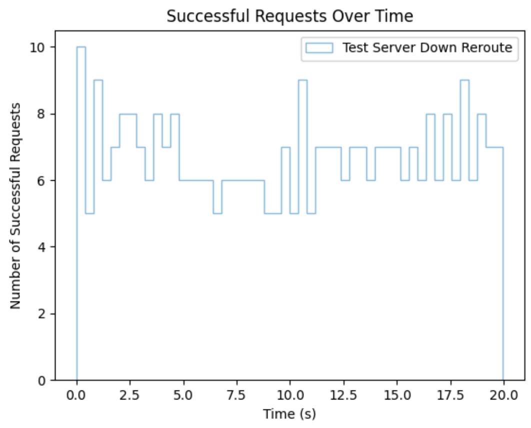
The typical test picks a mininet topo as a base and tests a specific feature (e.g. load shed) by pointing to its corresponding pre-defined configuration file under `load_balancer/test/setup`. We then create separate threads for each client using the `send_requests` helper described earlier to simulate sending simultaneous requests to the load balancer for a specific amount of time (e.g. 20 seconds). The collected results are then shown using the result summary and plotting functions described above. Due to differences in host environments (e.g. fraction of CPU time each host receives, VM resources, etc…), we pick a conservative sleep time (6-12 seconds) to allow servers and load balancers to stabilize before starting each test.

The load balancer successfully allows us to successfully horizontally scale a backend application. This can be shown through the round robin test in which 10 clients simultaneously send requests to an application with 1 server (baseline) then to an application with 3 equally proportioned servers (round robin). In the test results, we see approximately a 50% drop in latency and subsequent doubling in throughput (measured as the number of successful requests) when comparing the application with 3 servers behind a load balancer vs the single server setup.



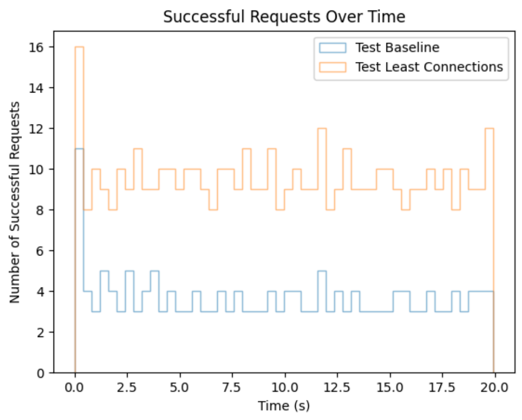


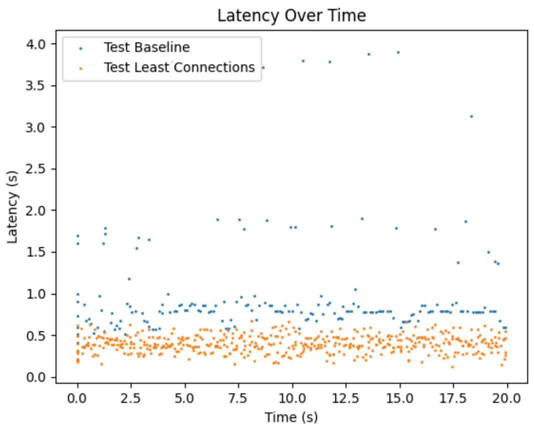
In failover tests we also see that the load balancer successfully reroutes traffic to healthy servers when a server goes down. In this case, 5 seconds in and the server went down for 10 seconds.

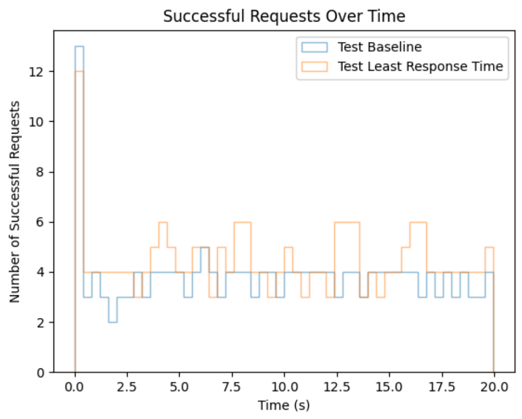


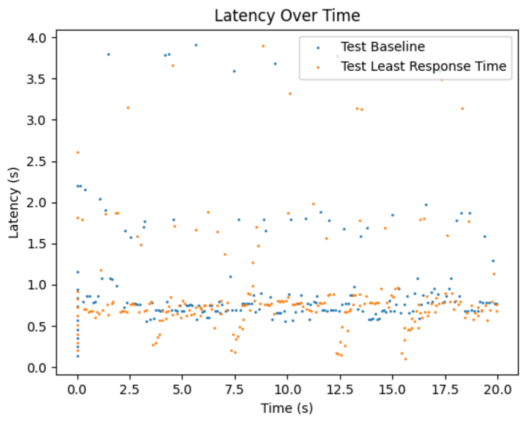
The test was run with 5 clients and 5 servers using the hashing algorithm. But due to the way the client IPs were generated, none of them is hashed to server 2 so technically only 4 servers. Looking at the graph, there is a slight drop in successful requests from the health check interval being fairly high. While a shorter interval may improve failover performance, it also comes at the cost of more health check traffic being sent over the network. If we decrease the number of servers, successful requests will drop significantly as redirects will overload a server. Future iterations to the load balancer should also include a fast-fail mechanism where servers are marked unhealthy when a (configurable) threshold of failures/timeouts is reached.

We also tested and compared the performance of the load balancing algorithms we implemented: round robin (both the weighted and unweighted variants), least connections, and least response time. The tests were made to be as consistent as possible by running them all on one VM for an equal 20 second duration and using the same network topology containing three servers. We compared the results of those three algorithms with each other and also with a baseline test for just one server. As expected, in all cases, our algorithms outperform the baseline results in terms of having lower latency and higher throughput. However, we found that the least connections algorithm (as shown in the first two plots below) performed significantly better than least response (which can be seen in the bottom two plots).

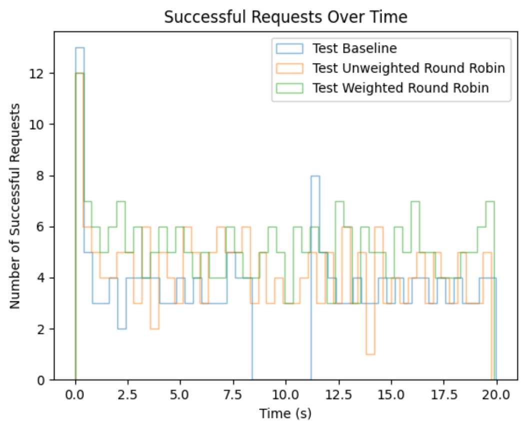


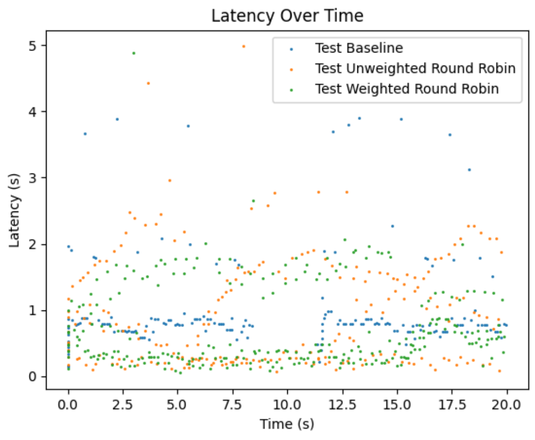




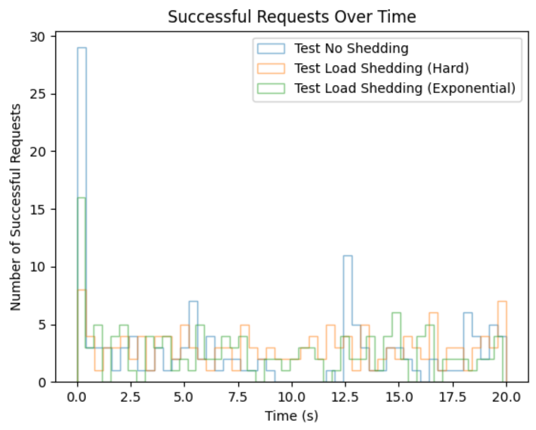


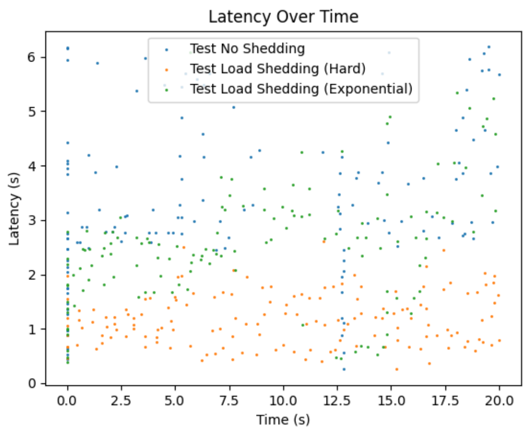
We also compared the weighted and unweighted round robin strategies for a topology consisting of three servers with increasing fractions of CPU time and configured with weights of 1, 2, and 3 along with the baseline test. We found that the weighted version performed better with 264 successful responses in 20 seconds and an average latency of 0.729 seconds compared to 217 responses and 0.934 seconds for standard round robin. This emphasizes the importance of choosing the correct parameters/weights to improve load balancer performance.

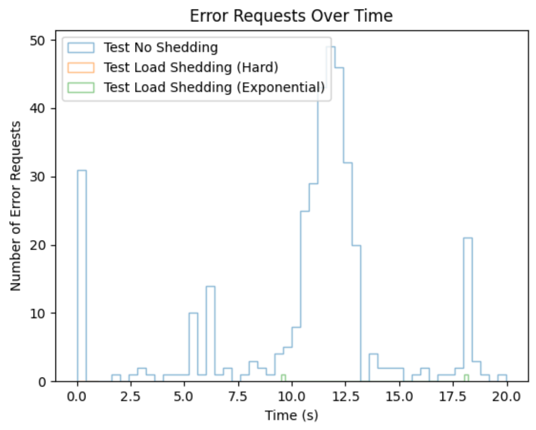




Through our load shedding tests, we find that load shedding significantly decreases the average RTT for successful requests while maintaining the same or increasing the total amount of useful work done (IE number successful requests) on an application with 3 equally-proportioned server topology using round-robin selection. We also see that a hard threshold on the maximum number of simultaneous connections leads to better results compared to exponential which suffers from inconsistent performance and higher overall latency. This emphasizes the sensitivity of load shedding with respect to its parameter setup and availability/compute power of servers, as the exponential strategy could behave closer to a hard cap with a lower choice of max simultaneous connections or a different choice of K.

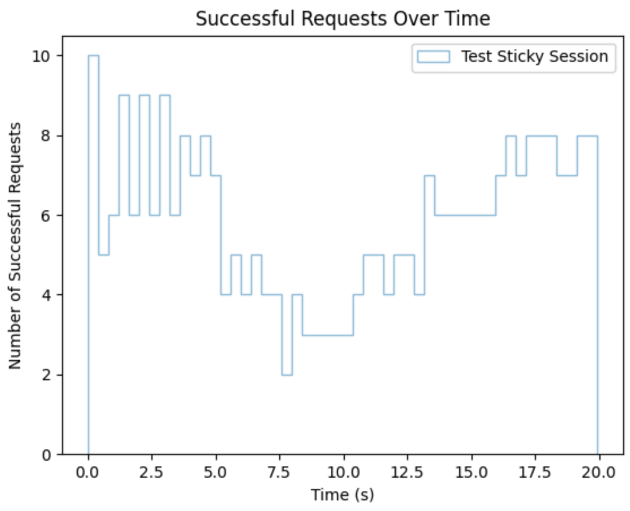






Looking at the difference between server errors returned to clients, both load shedding strategies also successfully prevent cascading failures from occurring, particularly when one server enters an unhealthy state. While a greater increase in useful work done could reasonably be expected, one factor could be that the health check marking servers as unhealthy allows for an "informal load shed" where overloaded servers are taken out of consideration for future requests. Once the overload cascades and takes down all servers, the load balancer rejects any requests until the servers recover (see the spikes in error occurring at 0s, 10s, 18s). Therefore, without the health check, it is reasonable to expect that the overloaded servers would be further overwhelmed by requests without a chance of recovery, leading to even worse performance than seen in our tests.

We also find that test results are highly dependent on environmental factors such as VM resource allocation and the fraction of CPU time assigned to CPULimitedhosts in the mininet simulation. For instance, RTT consistently increases when servers are provisioned with less CPU time or when the host VM operates under resource constraints.



The sticky sessions test checks for two things, whether sticky sessions would forward traffic to unhealthy servers and whether sessions expire. Around the five seconds mark, two servers go down. Around seven seconds, two clients go to sleep and wake up after their sticky session has expired. This is shown in the graph as drops in successful requests around the five mark due to health check interval and at seven second mark due to less clients sending requests. This test also showed a disadvantage of using sticky sessions as after the servers went back up, since clients kept sending data, none of the requests will be redirected back. Although it is the purpose of sticky sessions, this overwhelmed our servers when there were too many clients and led to no successful requests.

Many modern load balancers will have server-side agents that report server resource usage. This can be used to inform resource based load shedding, health checks, and dynamic server selection. While these features were part of the original plan, implementation was complicated by the limitations of simulating a distributed system on a single physical device. Most importantly, resource (CPU, memory, etc) usage in one simulated host on mininet would affect the readings across all hosts. The `psutil` library, used to monitor system resources on individual virtual hosts, appeared to take the usage of the host system rather than that of the simulated host. This limitation obscured the actual resource usage of the individual simulated mininet hosts. Pinning hosts on individual cpu cores to simulate truly independent distributed systems was considered, however, hardware limits the scale at which these tests could operate and additional complications such as the host OS' scheduler introduced further test flakiness.

Concluding Remarks and Lessons Learned

Our project demonstrates that a customizable load balancer built in Python, even if it is not the fastest language, can effectively support horizontal scaling at the application layer. Furthermore, we demonstrate that it is possible to test the correctness and effectiveness of load balancers and distributed systems in a simulated environment.

Experiments during testing also revealed that simulated distributed systems are not perfect substitutes for testing on real distributed environments. Mininet's limitations with respect to CPU isolation and host resource reporting highly restricted our options for more sophisticated dynamic load balancing algorithms/strategies which rely on accurate host resource reporting. Running mininet in a virtual machine further complicates the testing environment as variance/constraints in CPU/memory resources affect test results.

Through testing we also learned that the configuration of the load balancer and the server setup (number, fractional cpu allocation) highly affect the performance of the load balancer. Poor load balancer configuration with respect to its environment can actually decrease throughput, and increase latency/errors.

Overall, this project exhibits the fact that reliability in distributed applications rarely depends on a single algorithm, but rather emerges from multiple mechanisms such as health checks, routing, failover, and load shedding working together to ensure the health and stability of the backend system.