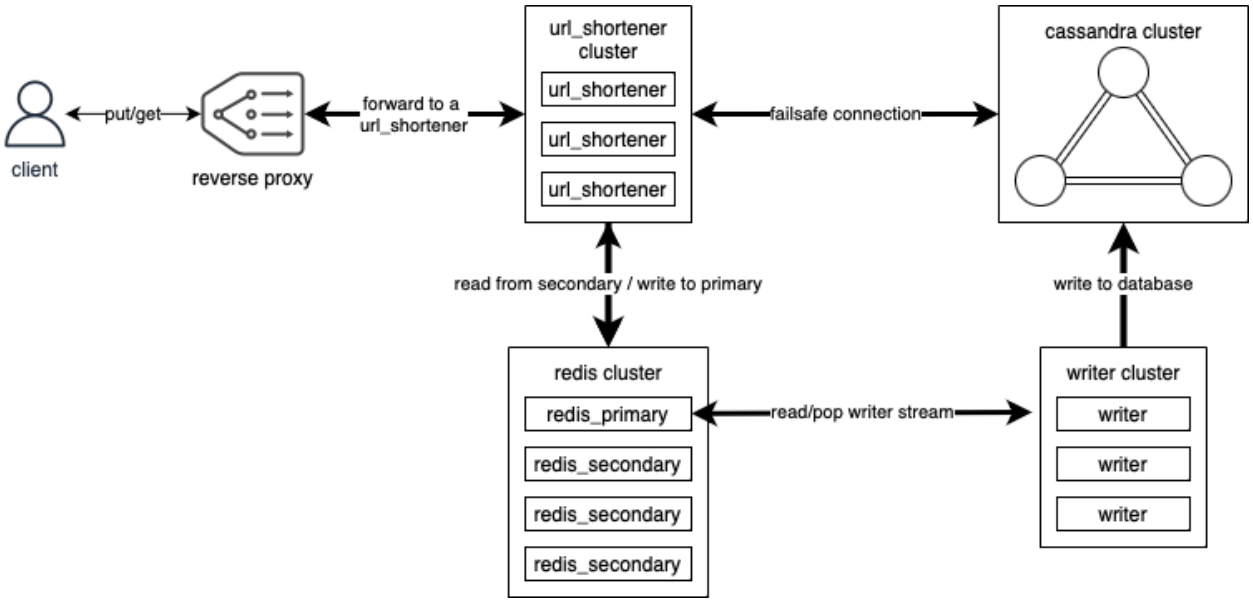

System Architecture



reverse_proxy:

- forwards the clients request to one of the url_shorteners

url_shortener:

- connects to redis_primary, one redis_secondary and the cassandra cluster
- parses the clients input
- reads from redis_secondary
- if redis_secondary is down, reads from redis_primary
- if redis_primary is down or does not contain required data, reads from cassandra
- writes to redis_primary
- if redis_primary is down, writes to cassandra

redis cluster:

- single primary and multiple secondaries
- primary contains a list for use by writers

writers:

- pop from redis_primary list
- attempt writing to cassandra database until successful

cassandra_cluster:

- persists data

dashboard:

- monitors all swarm services through docker

System Discussion

Cache

- Redis is used as a cache for short:long pairs (redis is also used for the writer stream)
- Uses LRU policy to evict keys
- Each short/long pair in the cache has a finite TTL to ensure consistency and fault tolerance

Consistency

- Every request is timestamped to ensure that the latest PUT request is stored in the database.
- The redis cache (excluding the streaming list) has a TTL timer for each key value pair to ensure it becomes consistent with the cassandra database if redis malfunctions but is not restarted (therefore some requests are sent directly to cassandra but don't exist within redis)
- In the event that redis goes down, the redis cache (excluding streaming list) is purged as data in the cache may be outdated.

Availability / Fault tolerance

- Multiple url_shorteners so if one goes down, another is available to hand user requests.
- If redis_secondaries go down, url_shorteners will redirect all requests to redis_primary
- If redis cluster goes down, url_shorteners will redirect all requests to cassandra
- If cassandra is down, PUT requests will still succeed and be stored in a redis queue, writers will then send data from the queue to cassandra when it restarts. GET requests will succeed only if requested data is in the redis cache
- redis_primary stream is persisted in case redis goes down
- Cassandra uses replication factor of 2 to ensure data recoverability if a node goes down

Orchestration

- Single script starts the cassandra cluster and the system stack
- Everything other than cassandra is deployed from a single docker-compose file
- Adding more url_shorteners, redis_secondaries and writers is possible by changing the 'replicas' number in the docker-compose file
- Adding / Removing cassandra nodes can be done through the add/removeCassNode script

Healing

- Every task in the swarm has a health check, if the health check fails or the process exits unexpectedly, docker will automatically restart the task.
- Cassandra uses replication factor of 2 to ensure data recoverability if a node goes down

Load Balancing

- each url_shortener attempts to connect to a different redis_secondary through dockers load balancing

- traefik is used as reverse proxy to forward user requests to a different url_shortener using round-robin

Scalability
- Storage capacity can be horizontally scaled by adding additional cassandra nodes using script
- Throughput can be scaled by increasing the number url_shortener replicas and in the docker-compose file
- Redis storage size can be vertically scaled by increasing max size in the docker-compose file
- Redis read throughput can be horizontally scaled by increasing redis_secondary replicas in the docker-compose file

Monitoring
- A dashboard UI accessed at URL: localhost:4001 shows the state of every service within the swarm including the dashboard itself
- Health checks ensure service states are up do date
- node status is used to monitor cassandra cluster status

Logging
- All services output logs which are accessible through the Monitor dashboard
- Logs are persisted in each machine by docker

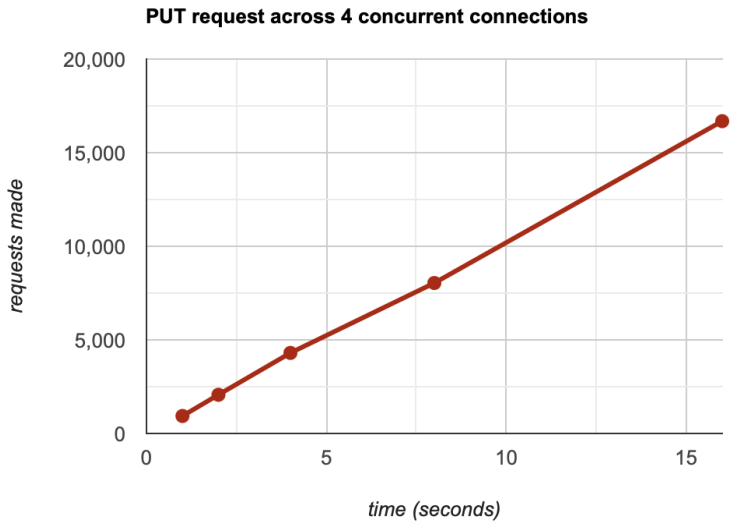
Strengths
- high availability, some url_shortereners, redis_secondaries, redis_primary and writers can go down and the system will remain functional
- easy to scale by changing the number of replicas in the docker-compose file

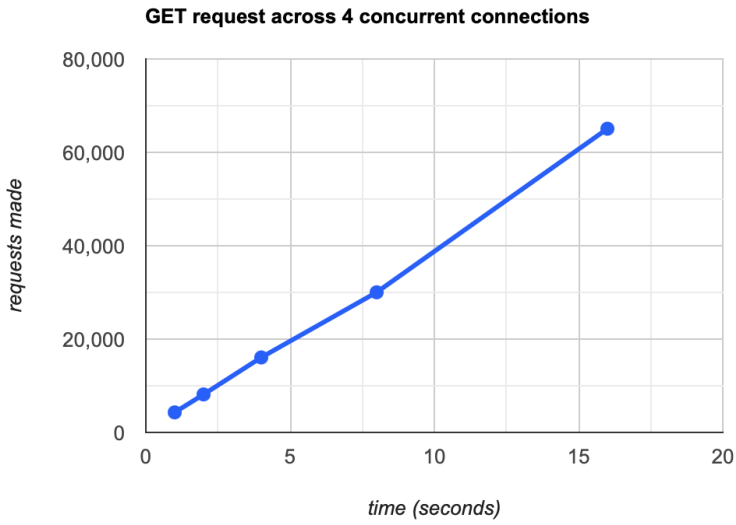
Weaknesses
- To have additional reverse proxies, they must each publish different ports, therefore they must be configured manually.
- url_shortener uses flask which is single threaded, therefore it does not utilize system resources efficiently, docker swarm does mitigate this by stacking services on the same system

System Performance

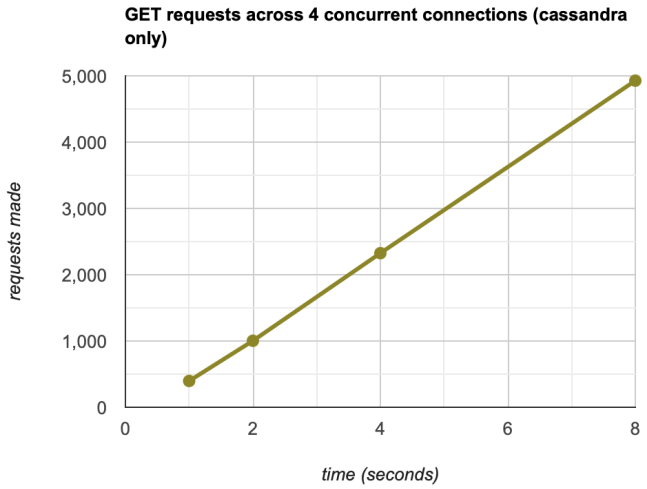
Setup

- 4 virtual machines each having 1 core and 2GB ram
- testing done on host machine using remaining 4 threads
- tests done using wrk '[GitHub - wg/wrk: Modern HTTP benchmarking tool]'
- architecture contains 1 writer, 2 url_shorteners, 2 redis secondaries, 3 cassandra nodes, 1 redis primary.





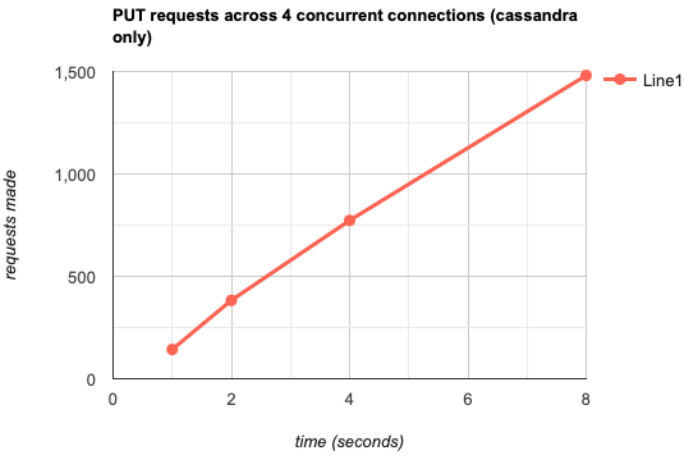
- both put and get requests scale linearly
- puts are significantly slower as all connections are directed to a single redis_primary node
- average latency was below 4ms for all requests



- get requests made directly to the cassandra were significantly slower, only completing 5000 in 8 seconds which is 6 times slower than get requests using redis with 2 secondaries

PUT Performance without writers

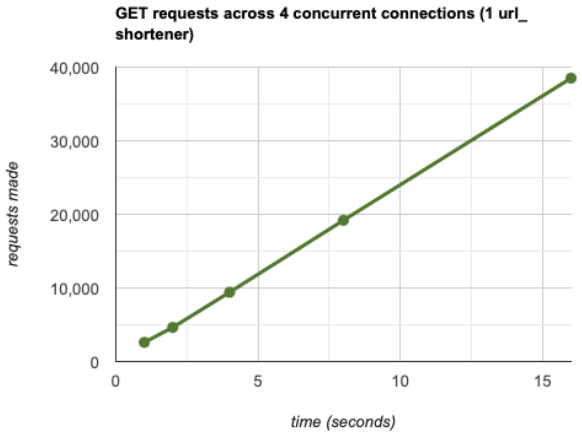
(same configuration as previous tests, PUT requests sent directly to cassandra)



Without the writer architecture, PUT requests suffer considerably, only managing to complete ~1500 requests in 8 seconds which is over 5 times slower than using redis with writers.

Notes:

- Increasing the number of writers does not improve PUT performance as the bottleneck is the single redis_primary node. More writers can however improve the reliability and consistency of the service.
- Having more redis_secondaries than url_shorteners is inefficient as the url_shorteners use flask which is single threaded therefore each url_shortener can only utilize the performance of a single redis_secondary at most. Therefore the number of redis_secondaries should be <= number of url_shorteners
- only 2 url_shorteners were used as the VMs only have a single core each therefore adding more url_shorteners would result in more context switching resulting in worse performance. Using more VM nodes or having access to more cores would significantly improve performance as the main bottleneck for get requests (which exist in redis) is the url_shorteners.



Using just 1 url_shortener, performance dropped by between 30% and 60% as opposed to using 2 url_shorteners.