CSC409A2 Report

Application overview

- List of tools used:
    - Docker Engine - Community Version: 20.10.7, API version: 1.41 (minimum version 1.12)
        - Redis official image
            - For caching requests, very fast response time
        - Redis sentinel official image
            - For high availability, fault tolerance
        - Cassandra official image
            - For database, persistence to disk
        - URL Shortener custom image
            - Python flask server, main application
        - Cassandra writer
            - For performance improvements on writes by caching Cassandra writes
    - Ubuntu 18.04.6 LTS
    - python 3.8
        - gunicorn
        - Flask

Application flow: GET request

- GET requests are managed by the url shortners, which run as docker containers, and thus requests are automatically load balanced by docker.
- Docker maps ports 1:1 on port 80, which means port 80 inside the container will be mapped to port 80 on the outside world. Even though URL shorteners live in containers while
  exposing port 80, docker port mapping effectively creates services listening on port 80 outside.
- URL shorteners will acquire addresses of Redis sentinels, where they can receive information about other Redis servers including replicas and masters. GET requests will prioritize replicas,
  since they are read-only, this acts as a load balancer between redis servers.
- If all redis servers are not available, or the request is not cached, it will try to retrieve the data from the Cassandra cluster. If successful, the application will cache the result in redis servers, making it easier to access for later requests.
- Picture: (https://i.imgur.com/DFw1KCK.png)
  (https://imgur.com/gallery/y5sCP9y)

Application flow: PUT request

- Similarly for PUT requests, they are load balanced by docker. Since Redis master is the only Redis server that can write, requests will only go to Redis master and never to replicas.
- The redis master responds immediately to the PUT requests, caching it on its redis server. This will soon be automatically replicated to other Redis replicas.
- Cassandra writes are queued in a blocking list on Redis servers. A writer program wakes up periodically, or when master is idle, to perform inserts into Cassandra databases.
- Caching writes is advantageous because Cassandra writes to disk, which takes longer to complete compared to Redis caching in memory.
- Picture: (https://i.imgur.com/JaCpdJr.png) (https://imgur.com/gallery/SDp1gVl)

Application structure: Redis

- The Redis server is made up of masters and replicas, and sentinels will watch over masters for failovers.
- Each master will persist data to disk and can have an arbitrary number of replicas. The master
  is aware of its replicas ip and port.
- Replicas will not have persistence and will automatically replicate from its master.
- Both Redis master and replicas will evict data using allkeys-random as memory is getting used up.
- Sentinels can watch over masters for failovers. They can auto-discover masters' replicas and other
  sentinels watching over the same master. It can also determine the state of replicas.
- Picture (https://i.imgur.com/W2VrNE4.png) (https://imgur.com/gallery/HOnKBQ4)

Application fault tolorence: redis master re-election

- In the event of a Redis master being out of reach of a sentinel, it will subjectively mark the master as down (s_down).
- If the number of sentinels on the same master and marking the master as s_down is equal to or
  greater than the set quorum (default to 2), the master will be considered as objectively down (o_down).
- At this point the sentinels start to vote for a new master. When a master is elected, all existing replicas will be alerted by the sentinels and replace its old master with the new master.
- The old master will be alerted by the sentinels and become a replica of the new master.
- Picture: (https://i.imgur.com/og4Pszk.png) (https://imgur.com/gallery/TwFJPvf)

Application analysis: strength

- Load balance requests for URL shorteners automatically handled by docker.
- Fault Tolerance: docker restarts any failing URL shortener, Redis, and Cassandra.
- High availability: Redis sentinel re-electing masters ensures writes
- High consistency: Redis automatically replicates data from master servers.
- Caching: Redis is very fast for read requests because of caching.

- Improved writing: Queuing writes to databases improves write throughput.

Application analysis: weakness

- The new redis master elected by sentinels will not have persistance to disk, since it was originally a replica service without disk mounting.
- If a redis master goes down before the writer gets to perform writes or the queue of writes gets replicated, data could be lost.
- Redis relys on memory and will have to evict keys using allkeys-random if memory is getting used up.

Application analysis: testing

- Testing scripts: writeTest.py

Discussion of each of the following with respect to your system.

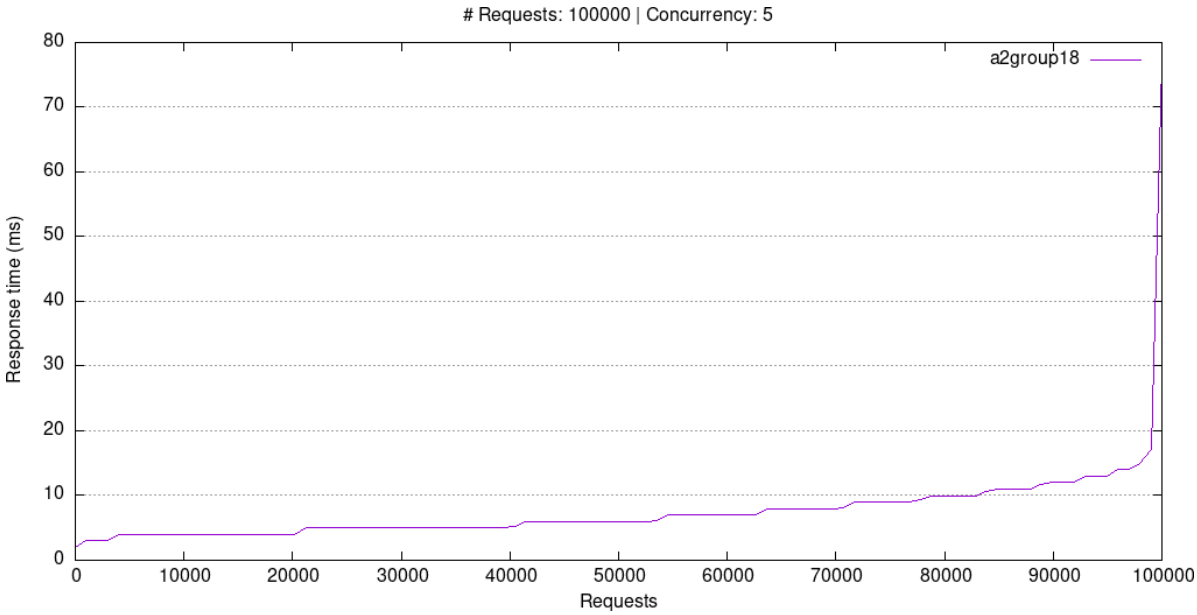
- For each point, as appropriate, show an appropriate diagram, list performance guarantees, discuss code/architecture choices.
- Example:
    - Availability:
        - the availability guarantees your system provides
        - the architectural choices made to implement this
    - Data Partitioning:
        - diagram explaining how data is partitioned
        - outline how your code implements this, for example, if you hash, then which hash algorithm
- [0] 1 Consistency
    - GET Requests are cached by redis and replicated across all replicas. It is also written to cassandra cluster
    - PUT requests are queued in redis master, and writers periodically wake up to perform queued
      writes to cassandra. In the meanwhile PUT requests will be cached and replicated in redis.
- [0] 1 Availability
    - When a hosts fails, requests will be redirected by docker as a part of load balancing.
    - If a redis master dies, redis sentinel will re-elect master as explained above to ensure write requests can be processed.
    - Docker will also restart any failing containers.
- [0] 1 Data replication
    - Data replication is automatically handled by redis replicas and cassandra cluster.
    - Redis uses complete replication and evict keys using allkeys-random as memory is used up.
- [0] 1 Load balancing
    - Requests will be automatically load balanced by docker among the url shortners.
    - Redis sentinels will automatically load balance redis requests from url shortners.

- [0] 1 Caching
  - Redis is known for fast reads with caching.
  - In this application writes to cassandra are queued and a writer wakes up periodically to perform writes to cassandra. This improves write throughput at a cost of consistancy.
- [0] 1 Process disaster recovery
  - Docker will automatically restart any failing container, ensuring availability.
  - In case of a redis master going down, redis sentinels will re-elect master to ensure that write requests can still be performed.
- [0] 1 Data disaster recovery
  - Both redis and cassandra servers are data persistant by storing data on a mounted disk in container.
    This means if the program crashes, we can still recover from disk.
    Cassandra is in a cassandra cluster, which means data will be replicated across all cassandra servers.
    If one cassandra nodes lost its data, data could still be recovered from other nodes in the cluster.
- [0] 1 Orchestration
  - We have a script that automatically starts, stops, and manages all services, including cassandra cluster, docker swarm and services on stack.
- [0] 1 Healthcheck
  - The health checker will periodically pings the services to check if they are down, by translating docker
    built in health check feature to user-friendly web interface.
- [0] 1 Horizontal scalability
  - The system takes advantage of additional hosts and adds them to the cluster.
    - Additional cassandra nodes means more storage for data. We can keep more data in our bank.
      This also means cassandra is more available; even if some nodes become unavailable, it is more likely that
      we still have some nodes online with the additional nodes.
    - Redis also benefit from additional hosts. As we get more nodes, we can start to hash requests and
      direct them to different masters that is responsible for that hash, each master having its own replicas.
      This means we have more memory space available, and redis servers are less likely to have to evict data
      as each master is responsible for less data. Generally improves throughput as more data will be cached.
    - Replicas will also be able to more quickly replicate new data as requests are hashed, because
      their master is responsible for less data, and replicas replicate less data. This leads to increased
      consistency as replicas will be more up-to-date.
    - More Redis sentinels means higher availability, as more sentinel watch over masters and failures can be
      more accurately detected. Re-electing new masters promptly will lead to increased availability for writes
      as the time of having no master is decreased.

        - More writers to cassandra result in increased consistency, because queued write requests will be
          written to Cassandra more promptly. So the time between writes being queued in redis and write is
          performed in Cassandra is shortened, and requests are more likely to get consistent response.
        - Docker automatically load balance for URL shorteners, and thus the system is able to handle more
          requests simutaniously, as each system is less loaded.
- [0] 1 Vertical scalability
    - CPU benefit: Overall performance boost to all parts of system. More CPU cycles means the service
      gets more attention and is able to execute more instructions per second.
        - Docker can load balance for a larger amount of requests, because CPU is used to redirect requests
          to available hosts, and a better CPU means a higher capacity.
        - URL shorteners will be able to query Redis and Cassandra faster, as more clock cycles help the
          program advance in the process of parsing the request and fetching static data.
        - Redis will be able to fetch data faster. Although Redis is known for caching in memory, additional
          CPU cycles can help with finding the key value in memory faster.
        - Redis replicas can sync with its master faster, as data is transferred and processed faster.
        - Redis sentinels can also re-elect masters faster, because it can evaluate replicas for master
          eligibility faster.
    - RAM benefit:
        - More throughput: redis having larger memory means more data can be cached. Less requests need to
          contact cassandra,, and redis can answer for more requests, where data is fetched faster
          than in cassandra because of fast memory reads/writes compared to disk IO. Redis can also fetch from and write to memory faster, if memory hardware is upgraded.
        - More services: we can run more services on each machine, as each service takes up some memory,
          more memory means more service capacity for each individual machine.
    - Disk benefit
        - Cassandra benefits from this by reading and writing to disk faster. As a result the application
          will experience increased throughput because requests that need to be read or written to cassandra
          takes less time to complete.
        - Additional disk space means we can store more data on disks for cassandra. Also Redis will have a
          large persistent capacity.

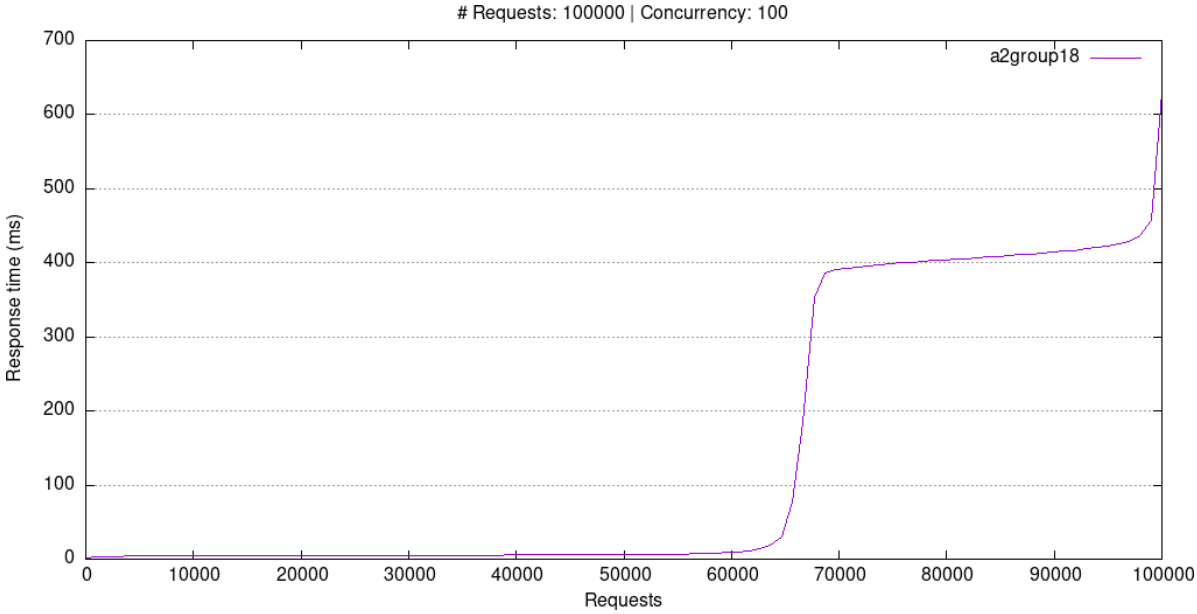
Discussing the system's performance

- [0] 1 Graph showing performance of system under load1: **100k requests with 5 concurrent requests**



- Our URL shortener was able to finish most (~84%) of the requests within 10ms and ~99% of the requests within 20ms as the system load is low.
- As we have multithreading for the shortener and 3replicas across swarm, we can deal with the load even with 100k requests sent, if they are not coming at the same time.

- [0] 1 Graph showing performance of system under load2: **100k requests with 100 concurrent requests**



- With the same amount of the requests but now with 100 concurrent requests, out system requires more time for each of the requests and peak at around 400ms.
- At around 65000 requests, the system can't really finish requests instantly as old requests were still processing.

0/4 Discussing the system's scalability

- [0] 1 Number of hosts vs read requests per second
    - Read requests scale proportionally with number of hosts as docker swarm distributes works across all replicas.
    - All read requests will be filtered by Redis before hitting Cassandra on disk, thus no bottleneck around disk IO.
    - However, there will be some overhead in managing the swarm (and also TCP costs) and result will not be perfectly proportional.
    - Our implementation gives around 730 reads per second with 3 hosts while having 560 reads with 2 hosts.
- [0] 1 Number of hosts vs write requests per second
    - Similar to read requests, write requests also scale proportionally with number of hosts as docker swarm distributes works across all replicas.
    - All write requests will be sent directly to Redis without any operation on Cassandra and dick, and all Redis methods we used should be in O(1) time.
    - However, there will be some overhead in managing the swarm (and also TCP costs) and result will not be perfectly proportional.
    - Our implementation gives around 500writes per second with 3 hosts while having 330 writes per second with 2 hosts.
- [0] 1 Number of hosts vs data stored
    - Our Cassandra cluster uses a replication factor of 2, which means each node will store around $\frac{2}{n}$ of the data where n is the number of nodes in the cluster.
    - Thus, data stored = 2 × data regardless of the number of nodes.

0/2 Discussion of the tools used during testing. Listing them is not enough.

You must define each tool used, and how you used it

- Apache Bench
    - ab -n 100000 -c 100 -g 100k-100.tsv http://127.0.0.1:80/dsh
        - -n to define number of requests.
        - -c to define number of concurrent requests.
        - -g output to load.tsv
        - url to test
    - Output file
        - Start time: human readable time at the time of request
        - seconds: unix seconds since epoch
        - ctime: connection time in ms, time taken to fully establish connection
        - dtime: process time in ms, time taken to finish processing request
        - ttime: total time in ms, ctime+dtime
        - wait: waited time, time process waited to receive data after connection
    - Results:
        - Concurrency Level: 100
          Time taken for tests: 141.504 seconds
          Complete requests: 100000
          Failed requests: 0

Non-2xx responses: 100000
Total transferred: 42000000 bytes
HTML transferred: 22100000 bytes
Requests per second: 706.69 [#/sec] (mean)
Time per request: 141.504 [ms] (mean)
Time per request: 1.415 [ms] (mean, across all concurrent requests)
Transfer rate: 289.85 [Kbytes/sec] received

For PUT requests:

- ab -n 10000 -c 10 -u /dev/null -T application/json 'http://127.0.0.1/?short=arnold&long=https://www.cs.toronto.edu/~arnold/'
- Results
  - Document Path: /?short=arnold&long=https://www.cs.toronto.edu/~arnold/
    Document Length: 7 bytes

    Concurrency Level: 10
    Time taken for tests: 21.620 seconds
    Complete requests: 10000
    Failed requests: 0
    Total transferred: 1640000 bytes
    Total body sent: 1820000
    HTML transferred: 70000 bytes
    Requests per second: 462.54 [#/sec] (mean)
    Time per request: 21.620 [ms] (mean)
    Time per request: 2.162 [ms] (mean, across all concurrent requests)
    Transfer rate: 74.08 [Kbytes/sec] received
    82.21 kb/s sent
    156.29 kb/s total

    Connection Times (ms)
    min mean[+/-sd] median max
    Connect: 0 0 0.2 0 4
    Processing: 2 21 24.7 4 106
    Waiting: 2 21 24.3 4 106
    Total: 3 22 24.7 4 106

    Percentage of the requests served within a certain time (ms)
    50% 4
    66% 19
    75% 53
    80% 55
    90% 58
    95% 61
    98% 64
    99% 69
    100% 106 (longest request)