DESIGN DOCUMENT FOR LAB 3

Description

1. Assumption and Guarantees

Our system makes the following assumptions:

- We have at most 3 replicas and at most 1 replica can crash
- No failure happens during starting up and recovery period
- Our communication channel is stable and synchronous, and no message is lost
- No replica is significantly slower than others
- Replica can reliably detect who the crash process is

Our system guarantees the followings:

- All client requests are served
- Consistency: all replicas see the same data
- Availability: System replies to clients in presence of faults

2. Front-end Service

This service exposes all the API endpoints that the users will be able to interact with. It is also responsible for the processing and abstracting back-end servers from the users. Based on the API the front-end service decides to call the catalog service or the order service.

Also this service includes features such as Caching and Load Balancing depending on the services that are running.

Caching and Invalidation of Caching

A simple in memory dictionary acts as a cache at the front end server. Here, when the front end receives the request it first checks if the data is present in cache. If it is present it returns the cached result to the client, else it forwards the request to the appropriate server and caches the response and returns it to the client.

Also, the front end server has an API endpoint to invalidate the cache. Everytime a request makes an update to the database, before writing the changes to the database, the catalog server calls this endpoint to invalidate the cache if it exists on the front end and then writes the changes to the database.

Load Balancer at Frontend Server

A simple round robin algorithm is used to implement the load balancer. The round robin algorithm guarantees that all the servers have the same workload. To guarantee this, alternate requests are sent to the different replicas.

Also, before calling the actual request, the frontend would check if the service is actually running by calling the simple HealthCheck endpoint. On receiving the response from the healthcheck the front-end does the actual call to the server.

Handling Fault Tolerance at the Frontend Server

The Frontend server handles faults by resending the request if the response is not received. Also, it ensures that the write requests are executed only once by associating every request with a **request_id.** Given a scenario, where for a buy request if the order service crashes after calling the buy endpoint of the catalog server - the frontend would receive an error but the catalog would have executed the request. As a result, when the frontend resends the request - the catalog would check the **request_id**, and if it is present it will just return the result and not perform any update.

3. Order Service

The order server supports a single operation: buy(item_number). Buy is the only non idempotent operation we have. We want to ensure that the buy operation is not repeated in case the response is dropped or the service fails before receiving the response. We generate a request id for each buy operation and pass it along with the request. We store this request id in a log and check the log before executing the request. Upon receiving a buy request, the order server must first verify that the item is in stock by querying the catalog server and then decrement the number of items in stock by one. The buy request can fail if the item is out of stock.

The buy request takes a book id as input. If the id is not valid and the book is not present it returns an error message. If the id is present, it sends a query to the catalogue service. If the stock >0 it sends an update request with -1 to the catalogue service. Otherwise the buy request fails.

The order service exposes 3 end points order, log and healthcheck. We use health check pings to keep a track of the alive replicas. We use the log endpoint to see all the buy queries executed.

4. Catalog Service

We implemented a Primary-Backup replication protocol for our catalog service. Every replica is assigned an unique process_id. All replicas have information about each other's **process_id** and **endpoint**.

When catalog replicas start, they will begin a leader election to appoint the replica with the highest **process_id** to be the primary. For every query request, the front-end service can perform a read directly from any replica. For buy/update requests, the front-end service load-balances the request to an order service, which sends the request to the corresponding catalog replica. If that catalog replica is a backup replica, it will forward the request to the primary replica, which will propagate updates to all backup replicas.

In the presence of failure for a backup replica, the primary replica won't be able to propagate updates to that replica, and thus the primary replica just continues the operation. When that crashed replica recovers, it will first ask all currently alive replicas who is the current primary replica, and send an election message to that primary replica. The primary replica will push the current state of the database and announce itself as the primary to the recovering replica.

In the presence of failure for a primary replica, the backup replicas will detect the primary failure and begin an election to choose a new primary replica. When the crashed primary recovers, it will sync the database with the current primary replica in the system, and then announce itself as the new primary.

Design Tradeoffs

As the system is not that huge - we decided to implement a simple in-memory caching vs a separate caching server container. This is because a small database in-memory caching would be faster by avoiding another API call. Keeping the system scope in mind we also implemented a simple round robin algorithm for load balancing.

Also, to implement replication and consistency among the database - we are electing the process ID that is the highest as the leader. This leader also acts as a Primary server for the Primary-Backup protocol which is used for synchronization and consistency.

Future Improvements and Extensions

The following would be good improvements and extensions to consider:

1. Implementing a background queue worker like rq and redis to handle background processing tasks such as writing to database and syncing with the other replicas.
2. Also, scalability would be another good extension of the system. Currently, one order service communicates with only one catalog service. Implementing a hardware load balancer at the backend would also make the system more robust against faults and crashes.

Testing and Evaluation

To run the system, please follow the instructions in the readme.md.

Q1. Compute the average response time (query/buy) of your new systems as before. What is the response time with and without caching? How much does caching help?
Ans: The below is the total and average response time of 100 lookup requests and buy and lookup requests for the scenarios with and without caching respectively.

INFO:root:Total response time with cache: 1946.7551708221436 ms
INFO:root:Average response time with cache: 19.467551708221436 ms
INFO:root:Total response time with cache: 1624.7100830078125 ms
INFO:root:Average response time with cache: 16.247100830078125 ms

Here, as the system and the database and the updates are not that computationally intensive, there is approximately 3 ms difference between cached and not cached responses.

Q2. Construct a simple experiment that issues orders or catalog updates (i.e., database writes) to invalidate the cache and maintain cache consistency. What is the overhead of cache consistency operations? What is the latency of a subsequent request if it sees a cache miss?
Ans: The cache consistency operation consists of invalidating the cache on the front end before performing the write operation to the database. The approximate overhead is:
Invalidate cache overhead 0.04582953453063965 ms
Also, the response time for a cached and the latency of a subsequent request if it sees a cache miss is as follows:
INFO:root:Response time for cache lookup 8.93402099609375 ms
INFO:root:Response time for cache miss lookup 27.62007713317871 ms

Q3. Construct an experiment to show your fault tolerance does the work as you expect. You should start your system with no failures and introduce a crash fault and show that the fault is detected and the other replica can mask this fault. Also be sure to show the process of recovery and resynchronization.
Ans: The sample logs for the following failure and recovery scenarios:

1. **Order service crash**

Response from the API Endpoints:
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request POST localhost:5004/buy/1
{

```
  "message": "successfully purchased the book How to get a good grade in 677 in 20 minutes a day. from order_service_3"
}
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request GET localhost:5004/lookup/1
{
  "cost": 1000.0,
  "stock": 992
}
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request POST localhost:5004/buy/1
{
  "message": "successfully purchased the book How to get a good grade in 677 in 20 minutes a day. from order_service_2"
}
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request GET localhost:5004/lookup/1
{
  "cost": 1000.0,
  "stock": 991
}
```

For the second buy request we can observe from the FrontEnd Server logs, that it detects the order_service_1 has failed and it resends the request to another service

```
front_end_service    | front-end-service 2021-04-30 23:50:58,144 INFO     resources.py:post server not available at order_service_1
front_end_service    | front-end-service 2021-04-30 23:50:58,145 INFO     resources.py:choose_host_for_buy Trying HOST.....order_service_2 at PORT......5007
front_end_service    | front-end-service 2021-04-30 23:50:58,234 INFO     resources.py:post request data: {'id': 1}
front_end_service    | 192.168.208.8 - - [30/Apr/2021 23:50:58] "POST /invalidate-cache HTTP/1.1" 200 -
front_end_service    | front-end-service 2021-04-30 23:50:58,235 INFO     resources.py:post In id data cache: {'lookup-1': {'cost': 1000.0, 'stock': 992}}
front_end_service    | front-end-service 2021-04-30 23:50:58,235 INFO     resources.py:post target key to pop: lookup-1
front_end_service    | front-end-service 2021-04-30 23:50:58,326 INFO     resources.py:choose_host_for_buy in rerouting host order_service_2
front_end_service    | front-end-service 2021-04-30 23:50:58,327 INFO     resources.py:post execution time for buy: 943.9264061450958
front_end_service    | 192.168.208.1 - - [30/Apr/2021 23:50:58] "POST /buy/1 HTTP/1.1" 200 -
front_end_service    | front-end-service 2021-04-30 23:51:03,340 INFO     resources.py:get execution time for search: 948.9405417442322
```

front_end_service | front-end-service 2021-04-30 23:51:03,341 INFO   resources.py:get execution time for search: 948.9410190582275
front_end_service | 192.168.208.1 - - [30/Apr/2021 23:51:03] "GET /lookup/1 HTTP/1.1" 200

2. **Order service recovery**

Response from the API Endpoints:

```
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request POST localhost:5004/buy/1
{
  "message": "successfully purchased the book How to get a good grade in 677 in 20 minutes a day. from order_service_3"
}
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request GET localhost:5004/lookup/1
{
  "cost": 1000.0,
  "stock": 989
}
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request POST localhost:5004/buy/1
{
  "message": "successfully purchased the book How to get a good grade in 677 in 20 minutes a day. from order_service_1"
}
(base) Hoangs-MacBook-Pro:CompSci677_Lab3 hoangho$ curl --request GET localhost:5004/lookup/1
{
  "cost": 1000.0,
  "stock": 988
}
```

3. **Backup catalog crash**

For the Backup Catalog crash we can observe in the log that the primary node detects the crash

catalog_service_1 | Catalog-Service 2021-05-01 00:16:25,750 INFO resources.py:push_invalidate_cache Invalidate cache overhead 0.025603771209716797
catalog_service_1 | Catalog-Service 2021-05-01 00:16:25,762 INFO resources.py:propagateUpdates Propagate updates to other replicas {111: 'catalog_service_2', 114: 'catalog_service_3'}

catalog_service_1 | Catalog-Service 2021-05-01 00:16:25,823 INFO resources.py:wrapper_put_request Node not alive at: http://catalog_service_2:5002/update_database

We can also observe from the FrontEnd Server logs, that it detects the order_service_2 is not able to respond and it resends the request to another service

front_end_service | front-end-service 2021-05-01 00:13:03,725 INFO resources.py:get server not available at catalog_service_2
front_end_service | front-end-service 2021-05-01 00:13:03,727 INFO resources.py:choose_host Trying HOST.....catalog_service_1 at PORT.....5002
front_end_service | front-end-service 2021-05-01 00:13:03,781 INFO resources.py:get execution time for lookup: 2269.381381750107
front_end_service | front-end-service 2021-05-01 00:13:03,781 INFO resources.py:get execution time for search: 2269.3816974163055

4. **Backup catalog recovery**

We can observe in the Primary catalog log that it sends the data for the backup catalog to recover
catalog_service_1 | Catalog-Service 2021-04-30 23:56:38,508 INFO resources.py:push_data Sending data {'Books': [{'id': 1, 'title': 'How to get a good grade in 677 in 20 minutes a day.', 'topic': 'distributed systems', 'stock': 986, 'cost': 1000.0}, {'id': 2, 'title': 'RPCs for Dummies.', 'topic': 'distributed systems', 'stock': 1000, 'cost': 1.0}, {'id': 3, 'title': 'Xen and the Art of Surviving Graduate School.', 'topic': 'graduate school', 'stock': 1000, 'cost': 100.0}, {'id': 4, 'title': 'Cooking for the Impatient Graduate Student.', 'topic': 'graduate school', 'stock': 1000, 'cost': 1000.0}, {'id': 5, 'title': 'How to finish Project 3 on time.', 'topic': 'distributed systems', 'stock': 1000, 'cost': 1000.0}, {'id': 6, 'title': 'Why theory classes are so hard.', 'topic': 'distributed systems', 'stock': 1000, 'cost': 1.0}, {'id': 7, 'title': 'Spring in Pioneer Valley.', 'topic': 'graduate school', 'stock': 1000, 'cost': 10000.0}]}
catalog_service_1 | Catalog-Service 2021-04-30 23:56:38,559 INFO PrimaryBackup.py:announce Done notifying node 111
catalog_service_1 | Catalog-Service 2021-04-30 23:56:38,622 INFO PrimaryBackup.py:announce Done notifying node 114
catalog_service_1 | 192.168.208.6 - - [30/Apr/2021 23:56:38] "POST /election HTTP/1.1" 200 -

5. **Primary catalog crash**

Here, Primary with process_id 123 is dead, so another backup take up the primary role as can be observed in the catalog_service_3 logs:

catalog_service_3 | 192.168.208.8 - - [30/Apr/2021 23:56:38] "POST /coordinator HTTP/1.1" 200 -
catalog_service_3 | Catalog-Service 2021-04-30 23:58:03,453 INFO PrimaryBackup.py:announce Could not notify node 123
catalog_service_3 | Catalog-Service 2021-04-30 23:58:03,461 INFO PrimaryBackup.py:announce Done notifying node 111
catalog_service_3 | 192.168.208.7 - - [30/Apr/2021 23:58:05] "GET /info HTTP/1.1" 200 -

We can also observe from the FrontEnd Server logs, that it detects the order_service_1 is not able to respond because catalog_service_1 has crashed and it resends the request to another service

front_end_service | front-end-service 2021-04-30 23:50:58,144 INFO resources.py:post server not available at order_service_1
front_end_service | front-end-service 2021-04-30 23:50:58,145 INFO resources.py:choose_host_for_buy Trying HOST.....order_service_2 at PORT.....5007
front_end_service | front-end-service 2021-04-30 23:50:58,234 INFO resources.py:post request data: {'id': 1}
front_end_service | 192.168.208.8 - - [30/Apr/2021 23:50:58] "POST /invalidate-cache HTTP/1.1" 200 -
front_end_service | front-end-service 2021-04-30 23:50:58,235 INFO resources.py:post In id data cache: {'lookup-1': {'cost': 1000.0, 'stock': 992}}

6. **Primary catalog recovery**

We can observe from the logs of the existing primary node in the system, when a node with a higher ID joins the system, syncs the database and starts the election process

catalog_service_3 | 192.168.208.8 - - [01/May/2021 00:19:38] "GET /sync_database HTTP/1.1" 200 -
catalog_service_3 | 192.168.208.8 - - [01/May/2021 00:19:38] "POST /coordinator HTTP/1.1" 200 -

CRASHED PRIMARY THEN ANNOUNCE ITSELF AS THE COORDINATOR

catalog_service_1 | Catalog-Service 2021-05-01 00:19:38,819 INFO PrimaryBackup.py:announce Done notifying node 111
catalog_service_1 | Catalog-Service 2021-05-01 00:19:38,844 INFO PrimaryBackup.py:announce Done notifying node 114