Design-Doc

Kamesh Balasubramanian and YenSung Chen

1. Overall System Design

This design for this lab builds on the design for the previous. The difference here is that we now have replicas of the catalog and order servers. We assume two replicas of each. We design the system so that the failure of one catalog or order server does not stop the system from processing requests. Also, when a crashed catalog server restarts, we ensure that it syncs its database state with the other catalog server. Again, the servers are served as python flask processes.

2. Changes to the Frontend

We have to deal with server replication and fault tolerance. The frontend also has an in-memory cache that is used to provide faster searches and lookups.

2.1 In Memory Cache

The frontend now has a cache that stored the result of a query to the catalog servers. This is to achieve lower latency for searches and lookups that result in cache hits. The cache supports sets, gets and deletes and is implemented using the python multiprocessing module's dictionary object. This is to ensure that race conditions do not occur when the cache receives multiple sets and deletes. The cache receives an item upon a successful search/lookup in the frontend. To maintain cache consistency, the catalog servers send an invalidate request to the frontend to remove a cache entry right before an update is made in the database.

2.2 Fault Tolerance

Since servers can crash at any time, the frontend uses a heartbeat protocol to request periodic responses from all server replicas to check if they are alive or not. The heartbeat protocol works as follows: A status request is sent to all servers asking for a response that they are alive. If they do not respond within a timeout of 3 seconds, they are determined to be offline. These status requests are sent every 3 seconds. If a server responds, then it is determined to be alive. We maintain a list of online and dead servers at any given time. This list is used to get an online server during load balancing at the frontend.

2.3 Load Balancing

Since we have multiple catalog and order servers, the frontend uses load balancing to ensure that traffic is balanced between the different servers. We use a simple round-robin load balancing scheme where consecutive requests are sent to different servers. Queries go to the catalog servers and buy requests go to the order servers. It may happen that a server that is online dies while the frontend is issuing a request to it. In this case, the heartbeat protocol might not be able to detect it before the request is sent. Requests that fail (due to servers being down) get redirected to active servers.

3. Changes to the Backend Microservices

As we now have two catalog and order servers, we need to ensure that the catalog servers are in sync after each buy request. The following changes have been made to ensure data consistency and fault tolerance.

3.1 Changes to the Catalog server

Since we have two replicas that need to be in sync, we add endpoints to the catalog server so that the replicas may synchronize with each other during updates to the database. As we assume that a replica may crash at any time, a catalog server syncs with the other server when it restarts from a crash. We assume that at least one replica is alive at any given moment to server requests. Also, we do not consider a restarting catalog server to be "ready" to serve until it has finished its sync. Therefore, our system works with under the guarantee that an online replica will not crash until the restarting replica finishes synchronizing its state.

3.2 Changes to the Order Server

The order servers make lookup and update requests to the catalog servers upon receiving a buy request. When only one catalog server existed all requests were forwarded to this server. However, we now have two catalog server replicas. At any given time, an order server replica maintains a master catalog replica, and a slave catalog replica. The updates are only made to the catalog replica and the slave replica is then requested by the order server to sync with the newly updated master replica to maintain data consensus. The order server only responds to the buy request from the frontend after data consistency across the catalog server replicas is achieved.

3.3 Data Consistency

An update request from an order server is only sent to its master catalog replica. If this request is successful, the order server then requests tbe slave catalog reeplica to sync with the master. (Note this is not an update. It is a request telling the slave to sync with the master only after the master was sucessfully updated). If the sync request fails (because the master catalog server was down), the original update request gets resent to the slave as we have lost the update made in the master catalog server. A sync request to the slave can also fail either because the slave was down, or the master was down. If the slave was down, there is no need to sync with the updated target and the order server completes the buy request. If the master was down, the slave cannot sync with it. In this case, the sync request fails but the order server reissues the original update request to the slave. One thing to note here is that, whenever an update request from the order server is successful at one of the catalog replicas, the order server requests the other catalog server to sync. Consider the case where the first update request to the master replica fails (becasue it was down) and the reissued update request to the slave suceeds. The master might come back alive before this update at th slave succeeds, so we will need to request the master to sync with the slave to ensure consensus. While it is true that the master will automatically sync with the slave upon restarting, we cannot guarantee that this will happen after the update at the slave occurs. So we must request a sync from the order server. However, this is not resilient to all fault configuratinos. If the master catalog server successfully satisfies the original update request then crashes, restarts and finishes syncing with the slave before the order server requests the slave to sync, we would end up with no updates in the databases. However, we proceed with this design as we assume that the master catalog server cannot crash, restart and sync its state with the slave before the order server issues a sync request.

3.4 Distributed Mutex

Since we have two order servers, we want to ensure that the order servers do no make concurrent updates to two different master catalog replicas. This would make obtaining consensus very challening. We launch another extremely lightweight flask server that controls access to the critical section in the order servers so that only one concurrent buy request is being processed at any given time. Since, we ensure consensus after every buy, this would avoid race conditions. There is one caveat. An order server might crash while holding the distributed mutex. In this case, the lock server would never receive a release signal leading to a deadlock. So we use a timeout of 1 second on the distributed lock to ensure that the other order server can continue executing in this situation.

Algorithm 1: How an order server implements a buy request to ensure consensus

```
while True do
    try:
        Make Update Request to MASTER
        *If Update Request was Successful*
        try:
            Request SLAVE to Sync with MASTER
            if Successful Sync then
                break
            else
                *Sync failed because MASTER was down. Resend update request to SLAVE by swapping
                 MASTER and SLAVE*
                swap(MASTER, SLAVE)
                continue
            end
        catch:
            *Sync failed because slave was down. Nothing to sync in this case*
            break
        end
    catch:
        *Update request to Master failed because it was down. Swap MASTER and SLAVE and continue*
        swap(MASTER, SLAVE)
    end
end
```

4. How the System Works Overall

When a client issues a query to the frontend, the frontend first looks at the in-memory cache. If the item exists in the cache the request is satisfied immediately and no network calls are made. If there is a cache miss, the frontend uses load balancing to get a catalog server that is online and forwards the request to that server. Should the catalog server die before the frontend receives a response, the frontend resends the request to the other catalog server. Meanwhile, the heartbeat daemon will recognize the crash and update the status of that catalog server to be dead. If the request is a buy request, the frontend behaves the same way to send the request to an online order server or resends to an online server. Once the request reaches the order server, the order server gets exclusive access to its critical section and carries out the buy as outlined in the algorithm above. If the item is out of stock, the order server responds to the frontend with a message to that effect and the buy fails. Otherwise, the buy is successful. On the completion of a successful buy, the frontend invalidated that entry in the in-memory cache to maitain cachwe consistency. It does not invalidate the cache if the buy fails.

5. Drawbacks and Improvements

There are a few drawbacks to our approach. We use a distributed mutex that is launched as a seperate server to ensure race conditions do not occur with the two order server replicas. The system becomes compromised if this lock server fails. Instead of using a mutex, we could employ a a fault tolerant consensus algorithm like RAFT to avoid race conditions on writes. The distributed update procedure in the order servers is also not resilient to all configurations as mentioned. We design our system based on the assumption that requests between an order server and catalog server happen faster than a catalog server can crash and restart. Of course, some external agent could make the order server sleep thereby giving a catalog server more time to restart but this would count as a byzantine fault.