MP2.2: A Highly Available and Scalable Tiny SNS

175 points

1 Overview

The objective of this assignment is to incorporate additional features of fault tolerance and high availability into the SNS service built in MP2.1. Any failures in the system must be handled transparently to the user.

The architecture for the TinySNS is shown in Figure 1, with the following specification changes:

1. In MP2.1, each cluster X_i had a single server S_i serving client requests. Now, this server is duplicated to form two processes M_i and S_i which act as Master-Slave pair processes.
2. When the client c_i contacts the Coordinator C for an active server, it calculates the clusterID using the mod 3 previously and returns the Master server's IP and port.
3. Both Master and Slave have their own directories to persist the user data.
4. For fault tolerance and high availability, the operations of Master M_i are mirrored by Slave S_i. In this MP, the Master and Slave are on the same machine. (Note: In the real world they will be on different machines.) Thus, the interface for communication between M_i and S_i must be based on gRPC.
5. The updates to the timelines that need to be made because of the "Following" relationship (i.e., client c_1 following client c_2) are only performed by F_i Follower Synchronization processes. An F_i process checks every 30 seconds which timelines on cluster X_i were updated in the last 30 seconds. E.g., if t_2 was changed, then F_2 informs F_1 (because c_1 follows c_2) to update the timeline of c_1. Since F_i and F_j are on different clusters, the inter-process communication must use gRPC.

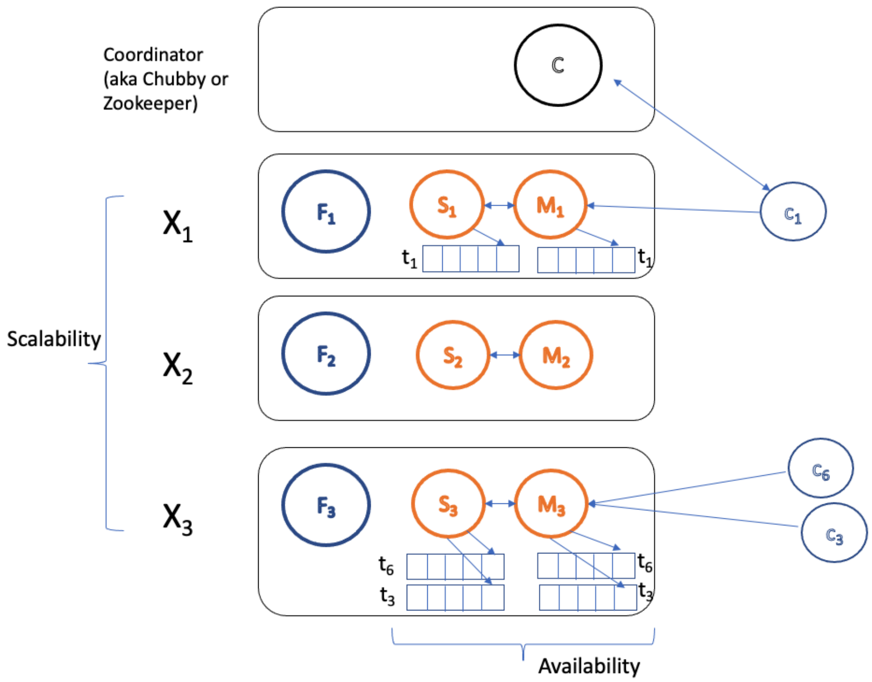

Figure 1: Architecture for a fault tolerant and highly scalable and available Tiny Social Network Service

6. **Failure Model:** The F_i processes and the Coordinator process C never fail. (Note: In the real world (cloud environments), F_i processes run as batch processes and can be restarted any time.) The only processes that can fail in this MP are the Master M_i processes. When the Master M_i fails, the Slave S_i takes over as Master.

2 Development Process

2.1 Client-Coordinator Interaction

Develop the Coordinator C process which returns to a client the IP and port number on which its Master runs. In the example above, the Coordinator return the client c_1 the IP/port for M_1.

2.2 Client-Master Interaction

The Client-Master Server Interaction remains the same as before. In addition, the Master also forwards the request to Slave.

2.3 Master-Slave/Coordinator Interaction

The Master and Slave processes are identical. The only difference is that the Master process interacts with the clients and informs the Slave process about the updates from the clients.

The Master and Slave processes also send periodic (every 10 seconds) heartbeats to the Coordinator. The absence of 2 heartbeats from a Master M_i is deemed by the Coordinator as failure, and, thus, Slave S_i is becoming the new master M_i.

When the failed server is restarted, it is its responsibility to sync user data with the master server. This can be done through a call to the Master server or since the follower synchonizer already has the logic to check for updates, it can also sync the data from master to slave. Only after the server is in sync with the lost updates, the coordinator can set the status of the server to be Active.

2.4 Follower Synchronizer F_i/F_j Interaction

When a client c_i enters "FOLLOW" command for c_j, an entry into the file containing follower / following information is appended by the Server, indicating that c_i follows c_j.

All F_i processes check periodically (every 30seconds) the following:

- New entries or updates in the follower / following information file. If F_i detects a change in this file where c_i follows c_j, then F_i informs F_j about the new FOLLOWING request. To find out which F_j is responsible for c_j, a request to the Coordinator must be made.
- Changes to the timelines file for the clients assigned to their cluster.

A change to the timeline t_i for client c_i must be propagated by F_i to those F_j's responsible for followers of c_i.

3 Implementation Details

3.1 Master/Slave Servers

The role of the servers (master vs slave) will be decided by the coordinator. When the servers start, they register themselves with the coordinator. The first server to contact the coordinator can be considered as Master.

Apart from this, you should also build a heartbeat mechanism from the servers to the coordinator every 10 seconds to monitor the status of the servers. The invocation command remains the same as before:

```
$./server -c <clusterId> -s <serverId>
-h <coordinatorIP> -k <coordinatorPort> -p <portNum>

Master : $./server -c 1 -s 1 -h localhost -k 9000 -p 10000
Slave : $./server -c 1 -s 2 -h localhost -k 9000 -p 10001
```

3.2 Client

The client code should incorporate changes to automatically reconnect to the new master on failures. Below is a sample invocation:

```
$./client -h <coordinatorIP> -k <coordinatorPort> -u <userId>
$./client -h localhost -k 9000 -u 1
```

The client should call the provided function “displayReConnectionMessage” so that we know to where the client is connected.

```
void displayReConnectionMessage(const std::string& host,
```

```
                                    const std::string & port) {
    std::cout << "Reconnecting to " << host << ":" << port
    << "..." << std::endl;
}
```

3.3 Coordinator

The Coordinator's job is to manage incoming clients, be alert to changes associated with the server to keep track of who are active and who are not, to switch to the slave server once the master server is down.

Example,

Assume (M1, S1), (M2, S2), (M3, S3) forms 3 (Master, Slave) pairs. At a time, only one among the Master-Slave pair is active. Then,

Routing Table

Cluster ID	Server ID	Port Num	Status
1	1	9190	Active
1	2	9191	Active
2	1	9290	Inactive
2	2	9291	Active
3	1	9390	Active
3	2	9391	Active

Follower Synchronizer routing tables

Server ID	Port Num	Status
1	9790	Active
2	9890	Active
3	9990	Active

```
getServer(client_id):
    clusterId = ((client_id - 1) % 3) + 1
    for serverId in routing_table[clusterId]:
        if routing_table[clusterId][serverId] is ‘Active’:
            return routing_table[clusterId][serverId] #Note that ID starts
```

```
getFollowerSyncer(client_id):
    serverId = ((client_id - 1) % 3) + 1
    return followerSyncer[serverId][1] #Note that ID starts from 1
```

Below is a sample invocation:

```
$./coordinator -p <portNum>
$./coordinator -p 9090
```

3.4 Follower Synchronizer

This process deals with updating follower information and timeline information between all the clusters. The Follower synchronizer DOES NOT directly communicate with the Master or Slave servers. Any update that the synchronizer makes is reflected only on the context files read by the server.

Below is a sample invocation:

```
$./synchronizer -h <coordinatorIP> -k <coordinatorPort>
            -p <portNum> -i <synchronizerId>
$./synchronizer -h localhost -k 9000 -p 9090 -i 1
```

3.5 Logging

All output/logging on Servers, Coordinator, and Synchronizer must be logged using the glog logging library as described previously.

3.6 StartUp Script

Make changes to your startUp script to add Master-Slave pairs.

4 What to Hand In

4.1 Design

Start with your design document first. The result should be a system level design document, which you hand in along with the source code. Do not get carried away with it (2-3 pages of detailed description is necessary), but make sure it convinces the reader that you know how to attack the problem. List and describe the components of the system. Ensure that this **PDF** document is submitted via Canvas.

4.2 Source code

Hand in the source code, comprising of: a makefile; source code files; and startup scripts for starting your system via your GitHub repository. The code should be easy to read (read: **well-commented!**). The instructors reserve the right to deduct points for code that they consider undecipherable.

4.3 Grading criteria

The 175pts for this assignment are given as follows: 5% for complete design document, 5% for compilation, and 90% for test cases (the test cases have different weights). Refer to provided test cases which cover most scenarios but these are slightly different with the test cases for grading.