MP2.2: A Highly Available and Scalable Tiny SNS

150 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 in MP2.2, 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, the Coordinator calculates the clusterID using an equation (mod 3) described in detail in Section 3.3, and returns the Master server's IP and port.
3. For fault tolerance and high availability, the operations of Master M_i are mirrored by Slave S_i. In this MP, for the sake of convenience, the Master and Slave are on the same VM. However, in the real world, each server runs on its own machine, meaning that the master server cannot possibly modify the file system that is used by the slave server. This means that you MUST use gRPC to communicate between the master servers and their corresponding slave servers. Thus, the interface for communication between M_i and S_i must be based on gRPC.
4. Both Master and Slave have their own directories to persist the user data. As shown in Figure 1, the Master and Slave servers can be on separate machines with their own directories.
5. As shown in Figure 1, 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

on the Master machine. Follower Synchronization processes on the Slave machines do not send updates to other Follower Synchronization processes. The Follower Synchronization process on the Master machine and the Follower Synchronization process on the Slave machine from the same cluster never talk with each other. An F_{Mi} process checks every 30 seconds which timelines on cluster X_i were updated in the last 30 seconds. E.g., if t_1 was changed, then F_{M1} informs F_{M3} and F_{S3} (because c_3 follows c_1) to update the timeline of c_1 on cluster 3. Since F_{M1} and F_{M3} are on different clusters, the inter-process communication between Follower synchronizer processes must use RabbitMQ message queues, same for F_{M1} and F_{S3}.

6. Each Synchronizer synchronizes 3 kinds of information: 1) user existence, 2) follow/following, and 3) timeline. This means that each user from cluster 1 will initially and immediately see other users in the cluster 1, and eventually (due to synchronization) will have the info of other users from other clusters.

7. Since two processes (M_i and F_{Mi}) update a timeline, a file-write synchronization primitive needs to be used. You must use named semaphores for synchronization of write operations to the same file by multiple processes. Named semaphores, in contrast to unnamed semaphores, can be used for file-write synchronization among different processes.

8. **Failure Model:** The Synchronizer processes and the Coordinator process C never fail. (Note: In the real world, e.g., cloud environments, the Synchronizer processes run as batch processes and can be restarted at 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. The clients that are already connected with the old Master must be manually killed (e.g., Ctrl+C). To connect with the new Master, the clients need to be manually relaunched using the same login commands. In MP2.2, please make sure a duplicate client is able to log in after disconnection for 2 rounds of heartbeat (2 heartbeats missing), e.g., user c_1 should be able to log in to your system after we kill c_1 and wait another 60 seconds. Once a server is killed, we never restart it again in this MP2.2.

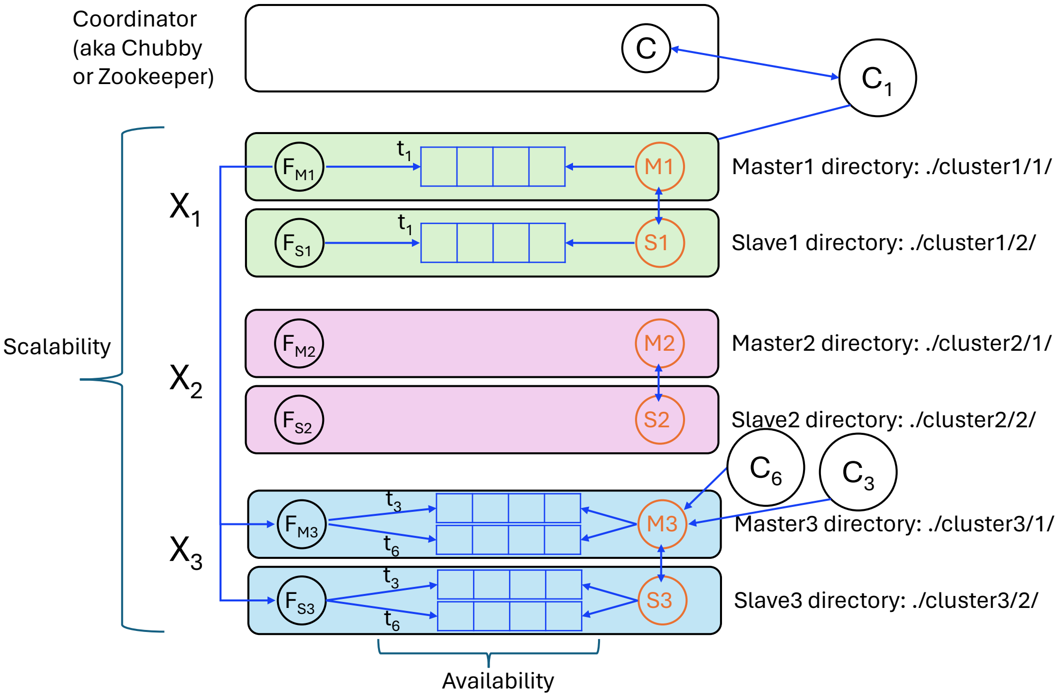
Coordinator
(aka Chubby
or Zookeeper)
C
C_1
Scalability
X_1
X_2
X_3
F_{M1}
t_1
M1
Master1 directory: ./cluster1/1/
F_{S1}
t_1
S1
Slave1 directory: ./cluster1/2/
F_{M2}
M2
Master2 directory: ./cluster2/1/
F_{S2}
S2
Slave2 directory: ./cluster2/2/
C_6
C_3
F_{M3}
t_3
t_6
M3
Master3 directory: ./cluster3/1/
F_{S3}
t_3
t_6
S3
Slave3 directory: ./cluster3/2/
Availability

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

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 returns 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 (i.e., mirroring).

2.3 Master-Slave/Coordinator Interaction

The Master and Slave processes are identical, and they read/write to/from their local directory on the same cluster, respectively, e.g., "./cluster1/1/" and "./cluster1/2/". The format of the data within the directories is up to you. If you need to write additional information to the file system, please feel free to do so. The only difference between the Master and Slave processes 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 a failure, and, thus, Slave S_i becomes the new master M_i.

2.4 Follower Synchronizer Interaction

There is a key difference between the communication between follower synchronizers on different clusters and the communication between any other types of services. All communications between any two follower synchronizer processes MUST use RabbitMQ message queues. All other types of communication (e.g., synchronizer-coordinator, coordinator-tsd processes) will continue to use gRPC. Detailed instructions on how to install a RabbitMQ cluster on your VM and verify its correctness is provided in the starter code, along with a script to automatically install all the needed dependencies to get started with using RabbitMQ with C++.

For this reason, the communication between follower synchronizers is a little unusual as compared to other types of communication in the TinySNS. When you use gRPC, all communication is synchronous, which means processes need to respond to a gRPC request and wait before proceeding with what they want to do next. With RabbitMQ message queues however, communication is asynchronous, meaning that a process can just put a message (the information it wants to convey) onto the appropriate queue and another process can consume this message from the queue whenever it wants to (or even never!). The time when a process consumes the message does not affect the functioning of the process that publishes the message onto the queue because asynchronous communication is non-blocking.

Now, here are the details of what the follower synchronizer processes need to do:

When a client c_i first logs in, F_{Mi} (the synchronizer responsible for c_i) will append c_i to its all_users.txt file. Then, periodically, all synchronizers will read their all_users.txt file and broadcast the contents into the message queues of all other follower synchronizers so that all the synchronizers know the list of all users across all clusters.

When a client c_i enters "FOLLOW" command for c_j, an entry into the file containing follower / following information is appended by the Master/Slave Server if c_i and c_j are on the same cluster, indicating that c_i follows c_j.

If c_i and c_j are on different clusters, this update should be appended by the Synchronizer processes on c_i's and c_j's clusters. More specifically, if c_i and c_j are on different clusters, F_{M_i} will write c_j to the c_i_following.txt file. Then, when each follower synchronizer periodically sends out the follower lists to other relevant synchronizers, F_{Mj} will find out about c_i following c_j and will write c_i to c_j_followers.txt.

Now, since c_i follows c_j, if c_j posts something on its timeline, c_i will find out eventually as F_{M_j} will periodically send all its users' timeline files to all the synchronizers responsible for managing the followers of its users. For example, since c_i is in c_j_followers.txt, F_{M_j} will send c_j's timeline to F_{M_i}, which will populate c_i's timeline file. Now c_i knows what c_j posted.

All Follower Synchronizer processes periodically (every 30 seconds) check and update the following by reading their respective message queues and publishing messages on the queues belonging to other synchronizers:

- The list of all users across all clusters.
- The followers/following files for all its users so as to update follower relations between all clients in all the clusters.
- New entries or updates in the timeline files among all users in the cluster. If F_{Mj} detects a change in c_j's timeline file, it will send c_j's new timeline to all its followers, which it learns by reading c_j_followers.txt. Now, since c_i follows c_j, F_{Mj} then informs F_{Mi} and F_{Si} about the new timeline posts. To find out which F_i is responsible for c_i, a request to the Coordinator can be made.

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 of a cluster to contact the coordinator can be considered as the 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:

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

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

3.2 Client

Below is a sample invocation:

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

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 is active and who is not, and finally, 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	10000	Active
1	2	10001	Active
2	1	20000	Inactive
2	2	20001	Active
3	1	30000	Active
3	2	30001	Active

Follower Synchronizer routing tables

Cluster ID	Synchronizer ID	Port Num	Status
1	1	9001	Active
1	4	9004	Active
2	2	9002	Active
2	5	9005	Active
3	3	9003	Active
3	6	9006	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 9000
```

3.4 Follower Synchronizer

Once again, make sure you read through section 2.4 thoroughly, as it contains information on how the communications between follower synchronizers is different as it uses asynchronous communication via RabbitMQ message queues, rather than synchronous communication using gRPC.

This Follower Synchronizer process deals with updating follower information and timeline information among all the clusters. The Follower synchronizer DOES NOT directly communicate with the Master or Slave servers.

There is one Follower Synchronizer process per machine, i.e., one Follower Synchronizer process on the Master server machine and one Follower Synchronizer process on the Slave server machine. One Follower Synchronizer process does not talk with other Follower Synchronizer processes on the same cluster. The Follower Synchronizer on a Master machine will talk to all Follower Synchronizers on other clusters. The Follower Synchronizer directly modifies the existing files (e.g., timeline files shown in Figure 1) in the Master/Slave directories.

When a Follower Synchronizer is launched, it contacts the coordinator for 2 kinds of information: 1) whether the server on the same machine is a Master server (the Follower Synchronizer uses this information to decide if it needs to send file updates to all Follower Synchronizer on other clusters), 2) the information regarding all Follower Synchronizer on other clusters (e.g., IP, port). It's important to note the Follower Synchronizer on the Slave machine does not actively talk with any other processes, however, when the Master machine is killed, this Follower Synchronizer on the Slave machine becomes responsible for sending file updates to all Follower Synchronizer on other clusters because its machine becomes a Master machine.

The Follower Synchronizer gets other users' information from other clusters and writes those information to existing files in both the Master/Slave directories. Any update that the synchronizer writes is reflected only on the files read by the Master server. More specifically, the Synchronizers will synchronize 3 kinds of information: 1) users' existence information, 2) following and follower relationship among all users, 3) the timeline posts from all users. We will not test the "UNFOLLOW" command in this MP2.2.

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 Heartbeat

The 30-second heartbeat frequency is mentioned above as an example. You can use a heartbeat frequency of less than 30 seconds.

3.6 Logging

All output/logging on Servers, the Coordinator, and Synchronizers must be logged using the glog logging library as described previously. If your glog is not working and you want to cout, it's fine but please let us know in your design document.

3.7 StartUp Script

In the starter code, there are two scripts, one to start all the required services, and one to stop all of them. These scripts will make your testing and development more efficient.

If you need to make changes to any parts of the skeleton C++ text files other than "YOUR CODE HERE", please feel free to do so. Please clarify your changes if you did so. Make changes to your startup script to add Master-Slave pairs and the follower synchronizers.

3.8 RabbitMQ Installation Script

In the starter code provided on Canvas, there is a subdirectory called install_rabbitmq. In this folder, there is a README.md that has clear instructions on how to install and verify that your RabbitMQ cluster installation and setup on your VM was successful. Please read through all of it and follow all the steps to ensure that the setup goes smoothly.

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

On Canvas, hand in all the source code, comprising of a makefile, source code files, and startup scripts for starting your system. 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 150 points for this assignment are given as follows: 5% for a complete design document, 5% for compilation, and 90% for test cases (the test cases have different weights). Refer to the provided test cases which cover most scenarios but these are slightly different from the test cases for grading.