MP3: A Highly Scalable and Available Social Network Service (SNS)

250 points
Due: April 20th, 2023, at 11:59pm

1 Overview

The objective of this assignment is to develop the next generation Tiny SNS that is scalable to a large number of users, is fault tolerant and is highly available (i.e., failures in the system are handled transparently to the user).

The Tiny SNS functionality that was required in MP2 is still required for this assignment, EXCEPT the "UNFOLLOW" command. Only the following commands need to be supported: "FOLLOW", "LIST" and "TIMELINE". Thus, it might be wise to start with the provided solution for MP2.

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

1. In this assignment we will use three (3) server clusters. In the figure, X_1 through X_3 are the three server clusters. Every server cluster is identified by an IP address (i.e., a single computer/server will run the processes shown inside every server cluster).

2. In each server cluster X_i three processes are running: F_i is a Follower Synchronization process, while M_i and S_i are Master-Slave pair processes.

3. In the figure, there is a Coordinator server C and several clients with ClientIDs c_i.

4. When a client c_i wants to connect to the TinySNS, it contacts the Coordinator C which: a) assigns the client to a cluster X_i using the ClientID mod(3) formula; and b) it returns the IP address and port number for the Master M_i of the cluster X_i. The client interacts with the TinySNS only through the Master M_i.

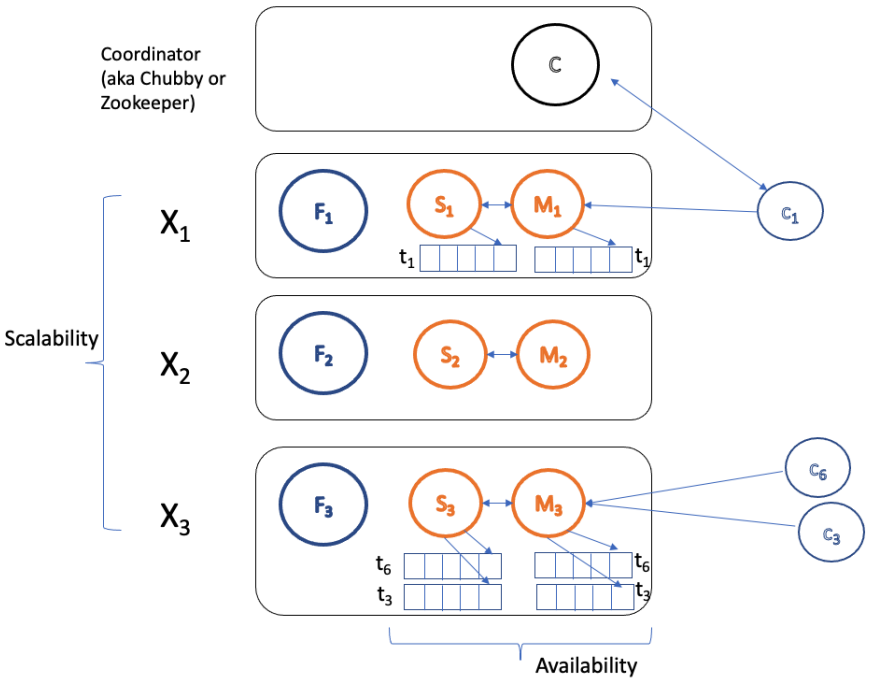

Figure 1: Architecture for a fault tolerant and highly scalable and available Tiny Social Network Service

5. A client's c_i timeline, denoted by t_i is persisted as a file in the file system. A timeline t_i contains client's c_i updates, as well as the updates from clients c_j which c_i follows.

6. A Master M_i performs updates to timelines of clients that are assigned to it. When a client c_i posts a message, its Master M_i updates c_i's timeline file.

7. 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.

8. 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 the F_i and F_j are on different clusters, the inter-process communication must use gRPC.

9. **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 process that can fail in this MP is the Master M_i. When the Master M_i fails, the Slave S_i takes over as Master. At most 1 failure can occur in a cluster.

2 Development Process

Take an incremental approach for completing this MP. First, before you write any code, make sure you COMPLETELY understand the requirements.

2.1 Client-Coordinator Interaction

Start with the provided client code.

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.

The coordinator implements a Centralized Algorithm for keeping track of the IP/port for the Master server in each cluster. More implementation details are below.

The Coordinator also keeps track of the Follower Synchronizer F_i IP/port number in each cluster. More implementation details are below.

2.2 Client-Master Interaction

After a client retrieves from the Coordinator the IP and port number for its Master, it connects with the Master.

A client c_i interacts ONLY with its Master. Even updates from clients that c_i follows, come to c_i through its Master. Effectively, the Master monitors the Last Update Time for a file (you may use the **stat()** system call), and if a timeline file was changed in the last 30 seconds, then the Master re-sends all entries in the timeline to the corresponding client. On the client side this will appear as: previous tweets scroll up and the new set of tweets is displayed. This is similar with the functionality when the client first starts up and the Master server sends the entire timeline for the first time.

If c_i follows c_j and they are both assigned to the same Cluster, the updates to c_i's timeline because of c_j posts, can ONLY occur after the F_i process runs. This means that even within a cluster, updates are not propagated immediately (like in MP2) to the followers. Moreover, the output for the LIST command which shows all who are following a given user will be updated by the same F_i process.

2.3 Master-Slave 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 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.

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 Master, 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 the Master 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.

2.5 Single Instance vs Multiple Instance Development

Please observe that for this MP you do not necessarily need multiple VMs. A single one is sufficient. You can start all the required processes on a single machine and everything should work well.

If you decide to use a single instance for the development, please make sure you still use inter-process communication (IPC) primitives that extend beyond a single machine (e.g., gRPC). Do not directly access files that are

supposed to be on a different cluster, simply because you can (when you run all services on the same machine, all services have access to all files in the file system).

3 Implementation Details

3.1 Master/Slave Servers

Each server holds context information (timelines, follower or following information) using 2 files saved within a directory titled serverType_serverId e.g., master_1. The context files can be named as per your liking.

Below is a sample invocation:

```
$./server -cip <coordinatorIP> -cp <coordinatorPort> -p <portNum>
       -id <idNum> -t <master/slave>
$./server -cip localhost -cp 9000 -p 10000 -id 1 -t master
```

3.2 Client

The client is expected to function exactly like in MP2, completely oblivious to the design of the server architecture. The only differences with respect to MP2 being that the client initially contacts the coordinator to get the endpoint that it should connect to, which will be then be used to create the server stub for further interaction.

Below is a sample invocation:

```
$./client -cip <coordinatorIP> -cp <coordinatorPort> -id <clientId>
$./client -cip localhost -cp 9000 -p 10000 -id 1
```

The client should call the provided function "displayReConnectionMessage"(in client.h) 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,

Master routing tables

Server ID	Port Num	Status
1	9190	Active
2	9290	Inactive
3	9390	Active

Slave routing tables

Server ID	Port Num	Status
1	9490	Active
2	9590	Active
3	9690	Active

Follower Synchronizer routing tables

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

```
getServer(client_id):
    serverId = -1
    if serverId = (client_id % 3) + 1:
```

```
     serverId = (client_id % 3) + 1
    if master[serverId] is 'Active':
     return master[serverId]
    else:
     return slave[serverId]

getFollowerSyncer(client_id):
    serverId = -1
    if serverId = (client_id % 3) + 1:
     serverId = (client_id % 3) + 1
    return followerSyncer[serverId]
```

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 -cip <coordinatorIP> -cp <coordinatorPort>
            -p <portNum> -id <synchronizerId>
$./synchronizer -cip localhost -cp 9000 -p 9090 -id 1
```

3.5 Logging

All output/logging on Servers, Coordinator, Synchronizer and Clients must be logged using the glog log- ging library. The library is installed for you on

the provided appliance. Please make sure you understand the different logging levels that glog provides (e.g., DEBUG, INFO, ERROR, etc) and make use of them appropriately. A good reference is: https://github.com/google/glog. For this assignment, it is **mandatory** that you log all the communication between any two entities(server-coordinator, synchronizer-synchronizer, synchronizer-coordinator and server-client interactions etc). Events like slave node becoming the master should also be logged. For this assignment, a macro has been defined to avoid buffering of log files:

```
#define log(severity, msg) LOG(severity) << msg; \
google::FlushLogFiles(google::severity);
```

Therefore, the logging can be done as:

```
log(INFO, "Server starting...");
```

All output/logging from the client, other than the I/O used for the User Interface, must also be done through the glog library. Note: Logs by default reside in /tmp directory.

4 What to Hand In

4.1 Design

Start with the provided code for MP2. Based on your design, you may find that significant portions of the provided code are no longer needed. Feel free to alter any part of the MP2 solution as you see fit.

Before you start hacking away, write a design document. 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 would be sufficient), 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; the compiled outfiles; 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 250pts 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.