MP2.1: A Scalable Tiny SNS

75 points
Due: Please check canvas

1 Overview

The objective of this assignment is to develop the next generation Tiny SNS that is scalable to a large number of users. This is built upon MP1. The Tiny SNS functionality that was required in MP1 is still required for this assignment.

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 S_i are the server processes that respond to client commands. The Follower Synchronizer process will be developed in MP2.2.
3. In the figure, there is a Coordinator server C and several clients with ClientIDs c_i. The Coordinator server serves the same role as Chubby or Zookeeper. It provides a similar API to Zookeeper. A draft interface file (.proto) for the Coordinator is provided.
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 - 1) mod(3) + 1) formula; and b) it returns the IP address and port number of the server S_i to connect to. The client interacts with the TinySNS only through this server S_i.
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.

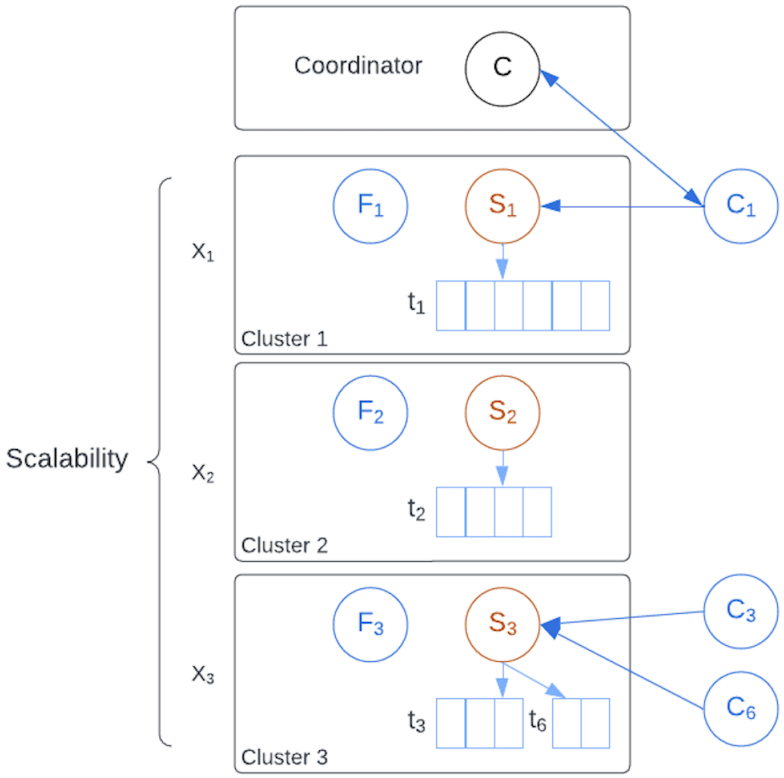

Figure 1: Architecture for a highly scalable Tiny Social Network Service

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

7. The Servers S_i register with the Coordinator and use keep-alive messages to indicate that they are available.

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

Develop the Coordinator C process which returns to a client the IP and port number on which its Server runs. In the example above, the Coordinator returns the client c_1 the IP/port for S_1.

The coordinator implements a Centralized Algorithm for keeping track of the IP/port all the servers 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-Server Interaction

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

A client c_i interacts ONLY with its Server. Even updates from clients that c_i follows, come to c_i through its Server. Effectively, the Server 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 Server re-sends latest 20 posts 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 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 (to be developed in MP2.2). This means that even within a cluster, updates are not propagated immediately (like in MP1) 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 (in MP2.2).

2.3 Follower Synchronizer F_i/F_j Interaction

This will be implemented in MP2.2.

2.4 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 Servers

Each server holds context information (timelines, follower or following information) using 2 files saved within a directory titled server_clusterId_serverId

e.g., server_1_1. The context files can be named as per your liking, it should be in the same format as before:

```
T 2009-06-01 00:00:00
U http://twitter.com/testuser
W Post content
Empty line
```

Below is a sample invocation:

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

$./server -c 1 -s 1 -h localhost -k 9000 -p 10000
```

3.2 Client

The client is expected to function exactly like in MP1, completely oblivious to the design of the server architecture. The only differences with respect to MP1 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 -h <coordinatorIP> -k <coordinatorPort> -u <userId>
$./client -h localhost -k 9000 -u 1
```

3.3 Coordinator

The Coordinator's job for this MP is to manage incoming clients and forward the requests to respective server based on its routing table. The coordinator is also interacting with the Servers and keeps track of their availability (i.e., through keep-alive messages). Additional features will be added to the coordinator in MP2.2.

Example,

Assume S1, S2 and S3 are the three servers from the three clusters respectively. The routing table is organized as below. The Status column can be set to 'Active' for this MP.

Routing Table

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

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

A draft interface for the coordinator is shown below:

```
service Coordinator {
    //Keep-alive message from Server to Coordinator
   rpc Heartbeat (ServerInfo) returns (Confirmation) {}

   // Invoked by Clients to obtain the IP/port of their servers
   rpc GetServer (ID) returns (ServerInfo) {}

   // Zookeeper-like API
   // Create a path and place data in the znode
   rpc create (PathAndData) returns (Status) {}

   // Check if a path exists (checking if a Master is elected
   rpc exists (Path) returns (Status)
}
```

Below is a sample invocation:

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

3.4 Follower Synchronizer

This will be developed in MP2.2.

3.5 Logging

All output/logging on Servers, Coordinator, Synchronizer and Clients must be logged using the glog logging library. Please make sure you followed the instructions for installing the glog library. Also, 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). 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.

3.6 StartUp Script

There are 4 processes to be run on the server side before the clients can start interacting. Hence, a startUp shell script which would run all the processes in one go can help in quicker testing would be helpfu.

4 What to Hand In

4.1 Design

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 of detailed description is expected), 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 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).