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 Synchronizer process (**not** a Follower process), and S_i is a server process that responds 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 formula: *(ClientID - 1) mod(3) + 1*; and b) it returns the IP address and port number of the server S_i to connect to. After that, the client c_i 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

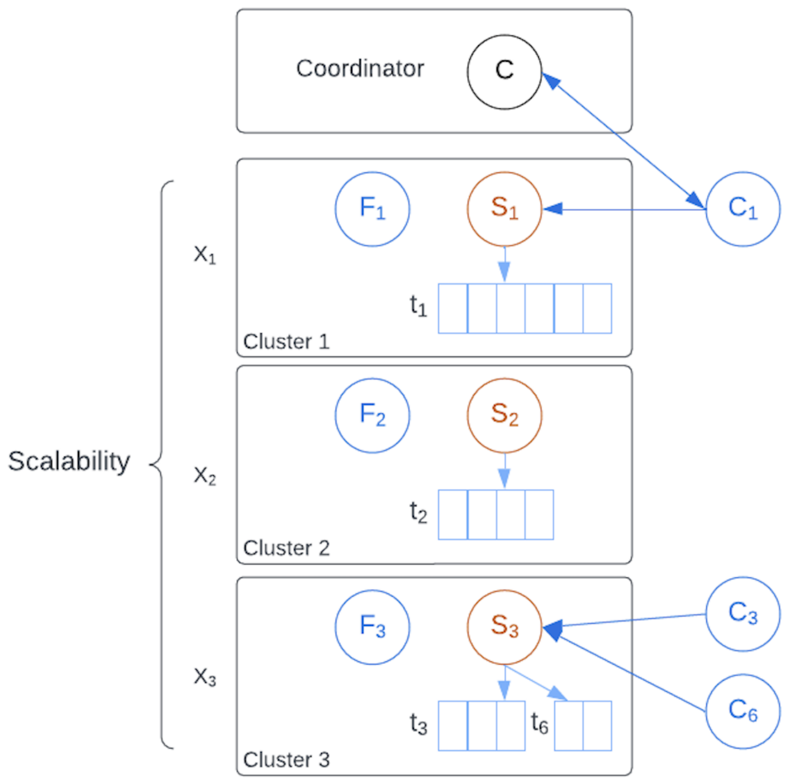

Figure 1: Architecture for a highly scalable Tiny Social Network Service

from clients c_j which c_i follows. Here the file system refers to the disk of your VM.

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 heartbeat 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. Concretely, this MP2.1 focuses on implmenting 1) the coordinator's capability of forwarding clients' connection requests and 2) heartbeat interactions between the coordinator and server.

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's 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). It's good to note that these follower and following functionalities are not tested in this MP2.1.

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, even though you can (when you run all services on the same machine, all processes 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:

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

$./tsd -c 1 -s 1 -h localhost -k 9090 -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 difference with respect to MP1 is that the client initially contacts the coordinator to get the server information it should connect to, which will then be used to create the server stub for further interaction. In implementations, you may want to create 2 gRPC stubs for client-coordinator and client-server communication, separately.

Below is a sample invocation. Please pay attention to the “-u 1” here. It’s different from “-u u1” in MP1. This is because you want to use the number “1” after the “-u” option in the coordinator to do the mod formula operation to get the server id.

```
$./tsc -h <coordinatorIP> -k <coordinatorPort> -u <userId>
$./tsc -h localhost -k 9090 -u 1
```

3.3 Coordinator

The Coordinator's job for this MP is to manage incoming clients and forward the connection requests to the respective server based on its routing table. The coordinator also interacts with the servers and keeps track of their availability (i.e., through periodic keep-alive heartbeat gRPC messages from servers to the coordinator every 5 seconds). Additional features will be added to the coordinator in MP2.2.

Example,

Assume there is only one server process in each cluster, and S1, S2 and S3 are the three server processes 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	10000	Active
2	1	10001	Active
3	1	10002	Active

```
getServer(client_id):
    clusterId = ((client_id - 1) % 3) + 1
    serverId = 0
    return routing_table[clusterId][serverId]
```

A draft interface for the coordinator is shown below. In the provided coordinator.proto file, you don't need to worry about the *create* and *exists* calls for MP 2.1.

```
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 Server registration with the Coordinator

How exactly does a server register itself with the coordinator process?

This involves implementation details. Here we provide one implementation option: on the server side, a server uses its first heartbeat (i.e., registration heartbeat) to register with the coordinator process. As we have specified each server's cluster-ID in Section 3.1 using the '-c' argument, each server can add its cluster ID information in the registration heartbeat sent to the coordinator (e.g., using 'context.AddMetadata("clusterid", clusterId)'). On the coordinator side, the coordinator checks if this is a registration heartbeat or a normal heartbeat. If this is a registration heartbeat, the coordinator takes out the clusterID metadata and server id information. If everything looks good, the coordinator replies OK and finish the registration process.

3.5 Follower Synchronizer

This will be developed in MP2.2.

3.6 Logging

All output/logging on Servers, Coordinator, Synchronizer and Clients need to be logged using the glog logging 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.

4 What to Hand In

The running system will consist of the coordinator (coordinator.cc), two instances of the tiny SNS server (tsd.cc) and the tiny SNS client (tsc.cc). As for the platform, you should use the provided virtual machine where Google Protocol Buffers v3 and gRPC are installed, and develop the program in C++.

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, but make sure it convinces the reader that you know how to attack the problem. List and describe the components of the system: Client/Server Program, and their interaction.

4.2 Source code

Hand in a zip file including a design document and source code (comprising of a Makefile, coordinator.cc, tsd.cc, tsc.cc, your startup script for convenience, and any auxiliary files, for example, .proto files). The design document must precisely indicate the commands used for each test case and the terminal output screen shot for each command. You can have multiple commands in 1 single screenshot. If the TA/grader cannot compile/run your code on TA/grader's virtual machine, your score will be either deducted or you have to come to TA's office hours to do an in-person demo.

The code should be easy to read (read: well-commented!). The instructor reserves the right to deduct points for code that he/she considers undecipherable.

The zip file mentioned above must be submitted through Canvas.

4.3 Grading criteria

The 75pts for this assignment are given as follows: 20% for complete design document and submission, 10% for compilation, and 70% for test cases (the test cases have different weights).