FSD 19/20 – Report

Alberto Faria (A79077), João Marques (A81826), Nuno Rei (A81918)

3 January 2020

1 INTRODUCTION

This document consists of the report for the practical assignment of the course entitled *Fundamentos de Sistemas Distribuídos*, part of the Integrated Master's Degree in Informatics Engineering at the University of Minho, lectured in the 2019/2020 school year. With this assignment, students were prompted to develop a Twitter-like distributed messaging system with global ordering guarantees and persistence.

This report is organized as follows. We begin in Section 2 by describing the system's requirements. Section 3 then details the architecture of the developed system and justifies its major design decisions. Section 4 describes several aspects of the system's implementation, and Section 5 concludes the report.

2 PROBLEM STATEMENT

The objective of the assignment to which this report pertains is to develop a distributed messaging system with global message ordering and persistence. The system is composed by an arbitrary number of servers, and clients are allowed to connect to any server by choice. A client, when connected to a server performs operations impacting the overall state of the system.

Requirements. These are the properties proposed in the assignment that the system implementation should abide:

- Servers can *restart* at any given point, but once they are *all back up* and running the whole system should be operational again.
- Clients can connect to *any* server, and perform any of these three operations:

(1) Publish a message tagged by one or more topics.
(2) Subscribe to a list of topics.
(3) Get the last ten messages containing any of the topics he's subscribed to.

- Messages obtained by each client should reflect a *coherent order* of all operations performed on the system by that same client or any other client.

3 DESIGN

In this section, we describe the architecture and functioning of the developed system, identifying and justifying major design decisions.

System overview. The system is composed of *client* and *server* nodes. The set of server nodes is assumed to be known to every other server, and each server has a unique numerical identifier. Each client is, at any single time, connected to exactly one server, and every server may have zero or more clients connected to it. In contrast, every server is connected to all other servers, forming a fully connected peer-to-peer network. Clients perform operations on the system by submitting requests to the server to which they are connected, and communication between each client and its server is sequential — a client may only have one outstanding request at any instant in time.

Each request may be of one of two types: `publish` and `get`. A `publish` request submits a new chirp to the system, while a `get` request has the objective of obtaining the most recently published chirps for one or more given topics. The previously described global ordering constraints apply,

and all clients must observe chirps in the same order regardless of the server to which they are connected.

We first note that each server acts solely on behalf of its clients. As such, for simplicity of presentation, we omit clients from the discussion that follows and regard each server as autonomously generating its own requests. This simplification has no design implications. For a given server, we may also refer to all other servers as "its peers."

Communication channels between any pair of servers are taken to follow a FIFO policy. Further, message latency is considered to be unbounded, and as such an asynchronous system model is assumed. However, as a practical compromise, timeouts may be used to detect server or communication link failures.

Workload assumptions. The most fundamental design decisions regarding what messages are exchanged between servers and when depend on expected properties of the workloads to be experienced by the system. The ratio between number of `get` and `publish` requests submitted by clients is pariticularly decisive. If, for instance, it is expected that the amount and frequency of `get` request is much higher than for `publish` requests, one would try to avoid the necessity for inter-server communication when serving the former type of requests. On the other extreme, if the ratio between the amoount of the two types of requests is inverted, it would be optimal to be able to serve `publish` requests without inter-server communication.

Due to the nature and intended purpose of the system being developed, we argue that it is reasonable to assume a workload under which `get` requests are significantly more frequent than `publish` requests. This leads to a push-based architectural design, wherein servers immediately propagate chirp publications to their peers, as discussed below.

Dissemination and ordering of published chirps. In order to serve a `publish` request, the server publishing the chirp communicates with all of its peers to inform them of the new chirp. By following this approach, it is then possible to serve `get` requests locally without any inter-server communication, as each server has up-to-date knowledge of all published chirps. This is a major advantage due to the assumption of `get`-intensive workloads.

However, simply broadcasting every publish request to all peer servers is not enough to guarantee a global order of published chirps. This problem is solved through the use of *logical clocks* to timestamp every chirp: the server from which a `publish` request originates broadcasts the new chirp, tagged with a timestamp sampled from the local logical clock, to all its peers, and then increments its local clock. Further, upon receiving a broadcast chirp, a server updates its local clock by setting it to the maximum value of itself and the timestamp receive with the chirp, and then increments it. Chirps are then ordered by ascending timestamp, and originating server identifiers are used to break ties.

Resilience to server and communication link failures. As previously stated, the system is only required to make progress if all servers are alive and connected. However, correctness must be ensured in the face of transient network partitions and server failures. For instance, the sets of published chirps for servers in separate partitions of a partitioned network may not diverge. Further, servers must survive transient crashes during which their volatile memory contents are lost.

In order to ensure that either all or no servers are informed of a published chirp, the Two-Phase Commit (2PC) protocol is used. When a server desires to publish a new chirp, it begins a transaction wherein it performs the role of coordinator and all servers (including itself) are participants. This effectively provides an all-or-nothing broadcasting primitive that guarantees that either all servers

or no servers receive the broadcast message, which in this case represents the publishing of a new chirp. Further, to allow crashed servers to recover, its set of known chirps is persistently stored.

4 IMPLEMENTATION

The system was implemented using the Java SE 11 language and platform. The implementation was first divided into client-specific code, server-specific code, and code common to both client and server. The implementation consists on an *IntelliJ* project named *chirper*, a pun based on the well known *Twitter* and its similarities to our system. This project contains three modules: client, server, and shared, corresponding to the aforementioned code divisions. The client and server projects provide runnable applications.

Client. The client provides a basic command line interface through which the user can subscribe to a certain set of topics, perform `get` requests, and perform `publish` requests. Communication with servers and also in between servers is accomplished using Netty. The interface provided by the client application accepts input lines with the following forms:

- `!get`, to fetch the latest messages, *chirps*, with the subscribed topics (all topics are considered if the client did not first subscribe to specific topics);
- `!subscribe [topics...]` or *!sub [topics...]*, which subscribes the client to the specified topics.
- `<chirp>`, a free-form line representing a chirp to be published, which must include one or more topics prefixed by the `#` character (*e.g.*, `My first chirp about #oranges.`).

Server. The server module is further subdivided into three main components: (1) the `ServerNetwork` class, which provides an abstraction layer on top of Netty to allow messaging with addressing based on unique server identifiers; (2) the `AllOrNothingBroadcaster` class and related classes, which provide an abstraction (implemented using the 2PC protocol) for performing broadcasts with the guarantee that either all or no peer servers receive the broadcast message, and (3) the `ChirpStore` class, which provides state persistence. The `Server` class glues these components together, and also hosts the logical clock implementation. Note that a `BasicBroadcaster` class is also included, which provides a basic broadcast primitive with no delivery guarantees.

5 CONCLUSION

In this report we described the development of a Twitter-like distributed messaging system with global ordering guarantees and persistence. We first laid out the requirements that the system should satisfy, and then detailed its design justifying its major design decisions. Finally, we described several aspects of the system's implementation.

The system currently assumes that the set of server nodes is predetermined and known *a priori*, and relies on predefined unique server identifiers. As future work, it would be interesting to extend the system to scenarios were peer servers are originally unknown, and also to support growing and shrinking the server cluster. The ability for the system to continue functioning in the face of server failures, instead of blocking if not all servers or communications links are available, is also further future work.