**CS 5450: Networked and Distributed Computing**

Lab 2

Peer to Peer networking

Spring 2020

Instructor: Professor Vitaly Shmatikov

TA: Jialing Pei, Eugene Bagdasaryan

**Due: 11:59PM Wed, Mar 18 2020**

## Introduction

In this lab, you will build a small peer-to-peer application in C++. Below, we specify the functionality and protocol your application needs to implement, along with some implementation hints and pointers to relevant information. You can also come up with your own design and gather necessary information yourself. We expect you to thoroughly test your implementation and write a good report with analysis of vulnerabilities.

The purpose of this project is to develop a communication application that is supposed to speak a standard protocol. Therefore, **your application should interoperate with implementations built by other groups**. You are welcome to discuss the challenges and techniques with other students and run tests together, but you must write all code on your own. We don't require your implementation to run on different hosts and will test it instead by running multiple processes on a single machine using provided proxy code.

**Note 1.** We don't impose a particular message structure for service messages and suggest to have this discussion settled over Piazza altogether for interoperability (you will be given bonus points for using the common message struct across groups). Please keep using C and make sure that your program interacts with our proxy script.

**Note 2.** We don't test scalability, thus you can assume the message size to be less than 200 chars, total number of messages to be less than 1000, and total number of peers to be maximum 4.

## Overview

Your goal is to implement a distributed chat room application. The system consists of two main components: client and servers, all independent of each other. Servers are used to maintain messages in the chatroom. They are responsible for receiving messages and sending messages to other servers and clients. A server also needs to maintain a chat log to store all messages it received so far and return the chat log to the client.

To convince yourself and us that you have successfully implemented what we asked for, your code will have to pass several tests. The test cases typically require the system to process some command and to behave correctly in the presence of failures. Client commands will be typed into the command line by user and fed into a proxy program. The proxy program will then open up server processes and set up a socket connection with them. Once the connection is set up, the proxy program will send client commands to the corresponding server processes.

For instance, if the test case calls for a server to crash, it will be up to the proxy to send the appropriate crash command to the relevant server. If it calls for issuing a "get chat log" command, the proxy will send the command to the specified not-crashed server, and that server should respond. The proxy is also responsible for sending chat messages from clients to the specified server. The server that receives a chat message will first store it in local memory and then transfer it to other servers if necessary(see Gossip Protocol below). We have developed a proxy program and we are providing its source code to you. **Your objective is to implement a server program which meets the API requirement specified in the bottom.**

### A Simple Gossip Protocol

There are one key problem with the trivial socket communication so far. Although in the Internet's original architecture the Network Layer (IP) underlying TCP was intended to guarantee universal any-to-any connectivity, so that any Internet host could communicate directly with any other, this original design principle has become seriously eroded as middleboxes such as firewalls and Network Address Translators (NATs) have proliferated in the Internet for practical and security reasons. Thus, TCP-level connectivity is now often asymmetric: if A can talk to B and B can talk to C, that doesn't necessarily mean A can talk directly to C via TCP. Therefore, we will need to explore mechanisms for indirect communication, so B can forward a message from A to C if necessary.

When the main objective is merely to ensure that a number of cooperating hosts or processes each obtain copies of whatever messages any of them send, as when implementing a chat room, one of the simplest yet also fastest and most reliable known algorithms for propagating those messages is known as a gossip protocol. USENET, the Internet's original widespread and decentralized public chat room used such an algorithm. We will not describe gossip protocols here in detail. There are many resources available online, for example:

- The Wikipedia page offers a high-level summary, though probably not all the details you will need (perhaps depending on the mood of the current editors and the phase of the moon).
- The original Epidemic Algorithms research paper by Demers et al at Xerox PARC in the 1980s. Perhaps not the easiest read, but there's no more definitive source.
- RFC 1036, the standard describing the way USENET news messages were formatted and propagated gossip-style in USENET's heyday. Pay particular attention to section 5 at the end on propagation, and section 3.2 on the Ihave/Sendme protocol.

## Gossip in this lab

Your gossip implementation must satisfy the following key property:

- Since a peer may not directly know about all others, each host must be able to forward messages it has received to other hosts who might not yet have received them, while ensuring that this forwarding does not cause infinite loops (e.g., A sends a message to B, which sends it back to A, which sends it back to B, etc.).

To accomplish these goals, messages need unique IDs with which P2Papp hosts can keep track of and refer to them. Read (or re-read) sections 2.1.5, 3.2, and 5 of RFC 1036 for one classic example of how to design and use message IDs in gossip protocols.

P2Papp will identify messages via a pair of values: an origin uniquely identifying the application from which a particular user chat message originated, and a sequence number that distinguishes successive messages from a given origin. A given P2Papp node will assign sequence numbers to user messages consecutively starting with 1, a convention that will make it easy for peers to compare notes on which messages from which other peers they have or have not received. For example, if A has seen messages originating from C up to sequence number 5, and compares notes with B who has seen C's messages only up to sequence number 3, then A knows that it should propagate C's messages 4 and 5 to B. This convention essentially amounts to implementing a *vector clock*:

- Wikipedia
- Fidge, Timestamps in Message-Passing Systems That Preserve Partial Ordering
- Mattern, Virtual Time and Global States of Distributed Systems
- Background: Lamport, Time, Clocks, and the Ordering of Events in a Distributed System

Given this approach to identify user messages, we can now define more specifically the two types of messages comprising P2Papp's gossip protocol:

- **Rumor message**: Contains the actual text of a user message to be gossipped. The message must contain at least three pieces of information: `ChatText` information which stores user-entered text; `Origin` information which stores the message's original sender and `SeqNo` information containing the monotonically increasing sequence number assigned by the original sender. For example, if Alice is using her machine to send "Hi" to Bob (we assume the sequence number is 23), the rumor message could be: <ChatText:Hi,From:Alice,SeqNo:23">.
- **Status message**: Summarizes the set of messages the sending peer has seen so far. The message contains a set of key-value pairs. Each key-value pair should have original sender as the key and the lowest sequence number for which the peer has not yet seen a message from the corresponding origin as the value. That is, if A sends a status message to B containing the pair <C,4> in its status message, this means A has seen all messages originating from C having sequence numbers 1 through 3, but has not yet seen a message originating from C having sequence number 4 (and A may or may not have seen messages from C with sequence numbers higher than 4).

As specified above we recommend to agree on the common structure of each message among groups, so use Piazza for that.

## Rumormongering

You should implement a simple rumormongering protocol. Whenever a peer obtains a new chat message it did not have before - either by being locally entered by the user (in which case this peer becomes the message's origin), or by being received from another peer in a new rumor message -

the peer picks a random neighbor peer and resends a copy of the rumor to that target. The peer (re-)sending the rumor then waits for some period of time either for a response in the form of a status message from the target, or for a timeout to occur. If the sending peer receives a status message acknowledging the transmission, it compares the vector in the status message with its own status to see if it has any other new messages the remote peer has not yet seen and if so repeats the rumormongering process by sending one of those messages. If the sending peer does not have anything new but sees from the exchanged status that the remote peer has new messages, the sending peer itself sends a status message containing its status vector, which should cause the remote peer to start rumormongering and send its new messages back (one at a time). Finally, if neither peer appears to have any new messages the other has not yet seen, then the first (rumormongering) peer flips a coin , and either (heads) picks a new random neighbor to start rumormongering with, or (tails) ceases the rumormongering process.

To keep things simple, you should always send new rumor messages from a given origin in sequence number order. That is, if A is rumormongering with B and has new messages (C,3) and (C,4), then A should propagate (C,3) to B before propagating (C,4).

To implement this rumormongering process you will need to set timers to have a handler reinvoked after some period if no message has been received by then. The timeouts you use in the rumormongering process can be fairly short, say 1 or 2 seconds; tuning them is a matter for future work.

**Implement a simple rumormongering scheme as described above. Test your application by running two, three, or four P2Papp instances on the same server machine.**

## Anti-entropy

As you should know from what you've read on gossip algorithms so far, rumormongering by itself does not guarantee that all participating nodes receive all messages: the process may stop too soon. To ensure that all nodes eventually receive all messages, you will need to add an anti-entropy component. In P2Papp, we will take a particularly simple approach: just create a timer that fires periodically all the time, maybe once every 10 seconds or so, and causes the peer to send a status message to a randomly chosen neighbor. If the neighbor who receives the status message sees a difference in the sets of messages the two nodes know about, that neighbor should either start rumormongering itself or send another status message in response, so that the original node will know which message(s) it needs to send.

**Implement anti-entropy as outlined above. Test it to make sure it reliably propagates messages across multiple hops.**

## Proxy API Specification

Here we specify the API that both proxy and server should obey. **Note that it is important to follow these rules strictly otherwise the proxy program we provide will not work in a**

**correct way.**

You can test the protocol between proxy and servers using the provided proxy program. When sending a sequence of commands on a TCP channel, the protocol uses "\n" as a separator. Details of the API are listed in the following tables.

In Table 1, we use <id> to denote the process id **which is a self-defined value and thus different from pid defined by OS kernel**. Process ids range from 0 to n-1, where n is the total number of processes. Table 1 shows in the left column the commands that the proxy can receive as input, and in the mid column the corresponding commands that the proxy issues to the server with id <id>.

In the start command, n is the total possible number of servers (processes), port is the port number used by the server process to accept sockets from others. The proxy will connect to port and send requests to the server through it. For server-server communication, you can feel free to use any port between 20000 and 29999. Moreover, each server could only talk to its "neighbour servers" so we suggest that you use a self-defined root-id(e.g., 20000) + pid as the port for server-server communication as in this way it will be easier to identify neighbours' ports. For example, a server with port number 20005(20000+5) could only talk to server with port number 20004 or 20006(if they exist). Your implementation should remain correct in the event of failures. For crash commands, the id number is the valid pid that you want to fail on.

Table 1: Client Commands

| client→proxy | proxy→server | details |
|---|---|---|
| <id> msg <messagID> <message> | msg <messagID> <message> | client sends the message to proxy and then proxy transfers it to servers |
| <id> start <n> <port> | | proxy starts a process with ./process id n port |
| <id> get chatLog | get chatLog | get the local chat log of that process, each message is separated by comma ',' |
| exit | | call ./stopall to kill all and exit |
| <id> crash | crash | the receiver process crashes itself immediately |

Table 2 specifies what rules should server follow when transferring the message to proxy/client. Whenever the proxy sends a get chatlog message to a server, the server will respond to proxy with all local messages stored in its chatroom.

Table 2: Server Messages

| server→ proxy | details |
|---|---|
| chatLog <message>,<message>,... | server replies the local chat log separated by comma "," |

**The lab is due: 11:59PM Wed, Mar 18 2020**