CS 5450: Networked and Distributed Computing
Lab 3
Peer to Peer networking
Spring 2017
Instructor: Professor Deborah Estrin, Vitaly Shmatikov
TA: Eugene Bagdasaryan

Due: 11:59PM Wed, Apr 12 2017

Note

Your slip day count is now 2 days for Lab 3 and Lab 4. You can't reuse days left from Lab 1 and Lab 2.

Introduction

In this lab, you will build a small peer-to-peer application in C++. We will specify the functionality and protocol your application needs to implement, along with some implementation hints and pointers to relevant information, however in this lab you can come up with your own design and gather necessary information yourself. As in previous labs we expect you to test it extensively and write a good report with analysis of vulnerabilities. Moreover, as you develop a communication application that is supposed to speak the same protocol, 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, however write code on your own. We don't require P2Papp to run on different hosts and will test it by running multiple instances on one machine

This lab focuses on putting together the basic elements of a very simple peer-to-peer application providing text-based multicast chat, similar to Internet Relay Chat (IRC). The lab contains three main components:

- Constructing a simple user interface for viewing and entering messages.
- Creating a UDP (datagram) socket with which to communicate with other nodes.
- Implementing a simple gossip algorithm to distribute messages among directly and indirectly connected nodes.

In this lab we would like you to use C++ and QT framework v. 4.8. Template code can be compiled in Ubuntu (or any Linux with QT configured). Development should be done in Ubuntu and containers can not be used for this assignment.

```
$ qmake-qt4
$ make
$ ./p2papp
```

Basic GUI Programming with Qt

This first part of the lab will introduce you to the basics of graphical user interface (GUI) programming in C++ using Qt. Although GUI programming isn't a primary topic of this course, your P2P application will need a user interface of some kind anyway, and Qt (like any decent GUI framework) makes this easy.

First look through the source files `main.cc` and corresponding header file `main.hh` in the template code we provided. To help you understand it thoroughly, look through the following online document to get an overview of the Qt model for widgets, layouts, and event notification via signals and slots: Documentation.

If you make and run the template code, it should display a bare-bones chat window. You can enter a line of text into the lower text entry box, and when you press Enter it should appear in the upper (chat log) box. Since there's no real networking yet, your messages not yet going anywhere except back to you in this window.

Study the constructor for the `ChatDialog` class and the documentation for `QTextEdit`, `QLineEdit`, and `QVBoxLayout` and make sure you understand how this window is being constructed.

Then, reviewing the signal/slot documentation above as necessary, make sure you understand how the ChatDialog class is arranging to get notified via a callback when the user presses Enter in the textline widget. This is Qt's primary event notification mechanism, which we will use in both GUI and network programming.

Part 2: Event-Driven Network Programming in Qt

This lab will be based on UDP for communication, here are some links:

- QUdpSocket Class Reference
- Linux UDP man page
- RFC 768

The `NetSocket` class in the provided template code simply extends Qt's `QUdpSocket` class with some code that automatically binds the socket to one of four particular UDP port numbers, computed based on the Unix user ID of the user running the program. This way each user on your system will by default use a different and non-overlapping range of UDP ports.

Serializing and Deserializing Messages

As you can see we haven't provided any actual code to send or receive messages, though; you'll have to do that yourself. In applications implementing Internet protocols, a lot of code is often devoted solely to the task of producing and parsing messages in a variety of text-based formats, or serializing and deserializing messages in binary formats. In this course we will eliminate a lot of this work by relying on Qt's serialization mechanism, which automatically supports a wide variety of simple and structured types.

To make our network message format simple but extensible, every network message P2Papp sends or receives will be a serialized instance of a `QVariantMap` or its synonym `QMap`, which is a

simple key/value dictionary - similar to dictionaries in Python and many other high-level languages - in which the keys are `QStrings` and the values are `QVariants`, a wrapper class that can hold many other types of objects such as strings, numbers, dates, lists, etc. To serialize a message you'll need to construct a `QVariantMap` describing the message (we'll specify what key/value pairs are needed as we go along), then serialize it into a `QByteArray` using a `QDataStream` object, and finally send the message via `QUdpSocket::writeDatagram()`. For example, Alice sends "Hello world!" to Bob. The message should be <`"ChatText", "Hello world!"`>.

You'll need to specify a destination host and UDP port to `writeDatagram()`. For now just re-send a copy of each message to each of the ports in the range `myPortMin` to `myPortMax` at the local host, which you can name implicitly via `QHostAddress(QHostAddress::LocalHost)`. We'll improve on this later.

To receive messages, you'll need to connect the `readyRead()` signal in `QUdpSocket` to a new slot you define in some class: perhaps in `ChatDialog`, or in `NetSocket`, or in some other class; it's your design choice. Then in the C++ method implementing that slot, you'll need to deserialize the message back into a `QVariant` again using `QDataStream`, and handle the message as appropriate.

Implement the message receive path, so that whenever your application receives a message that successfully deserializes to a `QVariant`, containing a `ChatText` key with a value of type `QString`, you add that message to the chat-log in the ChatDialog.

Part 3: A Simple Gossip Protocol

There are two key problems with the trivial protocol we implemented so far:

- The Internet in general, and UDP in particular, are unreliable and offer best-effort communication, which means messages may get lost or duplicated in the network for a wide variety of reasons (which we'll leave to a networking course to explore in detail). This may be unlikely to happen when you're sending a message from one UDP socket to another on the same host, but becomes much more likely once you start sending messages between hosts: UDP datagram loss or duplication in the network may cause one user's chat message to be lost, or to appear multiple times, in another user's chat log.

- Although in the Internet's original architecture the Network Layer (IP) underlying UDP 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, UDP-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 UDP. Thus, 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; you should familiarize yourself with them in some of the many resources available online or in any distributed systems textbook. A few pointers to start with:

- 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 our lab

To implement a gossip protocol, we will basically need to do two things:

- Since these messages will be propagated via unreliable UDP datagrams rather than on reliable TCP streams, hosts will need to acknowledge messages they have received, and the sender must be able to resend messages whose UDP datagrams may have been dropped by the network (i.e., for which the sender did not receive an acknowledgment).
- Since all peers 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, we will need to give messages unique IDs with which P2Papp hosts can keep track of and refer to user chat messages. 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 identifying 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 be a `QVariantMap` containing three keys: a `ChatText` key as above whose value is a `QString` containing user-entered text; a new `Origin` key identifying the message's original sender as a `QString` value; and a new `SeqNo` key containing the monotonically increasing sequence number assigned by the original sender, as a `quint32` value. Let's make its format more clear. For example, if Alice is using her machine to send "Hi" to Bob (we assume the sequence number is 23), the rumor message should be: <"ChatText","Hi"> <"Origin","tiger"> <"SeqNo",23>. From the above example, we could observe all the involved should be `QVariantMap`.
- **Status message**: Summarizes the set of messages the sending peer has seen so far. The message `QVariantMap` contains one key, `Want`, whose value is of type `QVariantMap`. Note that this is a second `QVariantMap` nested within the `QVariantMap` describing the whole message. The keys of this nested `QVariantMap` are the origin IDs (`QString`) the peer knows about, and its associated values (`quint32`) represents the lowest sequence number for which the peer has not yet seen a message from the corresponding origin. That is, if A sends a status message to B containing the pair <C,4> in its `Want` map, 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). Anyway, for a regular status message, it should be `<"Want",<"tiger",4>>`.

Rumormongering

We will now 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 (e.g., qrand()), 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).

An important question is how an application finds its neighbors. For now, your P2Papp should consider its neighbors to be whatever application, if any, owns the UDP ports with numbers im-

mediately above or immediately below itself on the local host, while staying within the myPortMin through myPortMax range. That is, the P2Papp instance that obtains port myPortMin (most likely the first instance you start) has only one neighbor, at port myPortMin+1; this second instance has as its neighbors the instances at myPortMin and myPortMin+2, and so on. This way, you can set up a simple linear topology on one server machine within your range of four UDP port numbers. In this lab we will not deal with topologies across multiple nodes.

Another important question is how to get origin identifier. Ultimately how you come up with an origin identifier is up to you, as long as you have reasonably good reason to believe it will be unique. `QHostInfo` might provide some useful way to handle it, but what you get back from that might not be really guaranteed to be globally unique (especially for "nameless" hosts behind NATs); a better approach might be to pick a random number when the program starts (e.g., qrandom()) and include that with the string. Even better might be to use a long, cryptographically strong random number or cryptographic has, but we will get to do that in later labs.

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 `QTimer` class is your friend. 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 is not guaranteed to ensure 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.

The lab is due: 11:59PM Wed, Apr 12 2017