CPSC 426 Lab 1: Gossip Messaging

Handed out Friday August 30, 2013
Bakeoff version due Friday September 6, 2013 11:59PM
Final version due Friday September 13, 2013 11:59PM

Introduction

In this lab, you will start building a small peer-to-peer application in C++ called—for lack of a more imaginative name—Peerster. We will specify the functionality and protocol your application needs to implement, along with some implementation hints and pointers to relevant information, but in this and all labs in the course you will ultimately be responsible for gathering all the necessary information and figuring out how to implement what you need to implement—just as you will need to in industry or research programming jobs. Since everyone in the class will be developing an application that is supposed to "speak" the same protocol, your application should—and will be expected to—interoperate with the implementations built by the other course participants, and there will be a "bakeoff" process to test and debug this interoperability as explained further below. Throughout this development and debugging process, you are welcome to discuss challenges and techniques with your fellow students, exchange pointers to relevant information or algorithms, debugging tips, etc., *provided you each write your own code independently*.

This first 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:

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

Version Control Using Git

In this course you will use the Git version control system to manage project source code. To learn more about Git, take a look at the Git tutorial and user's manual. Or if you are already familiar with other version control systems, you may find this CS-oriented overview of Git useful.

To obtain an initial Git repository for use with this course, login to a zoo machine, `cd` to the directory in which you want to keep your Peerster working repository, and run this script:

```
$ /c/cs426/getrepo.sh
Cloning into bare repository /c/cs426/SUBMIT/netid-digits...
done.
Cloning into peerster...
done.
$ cd peerster
$
```

You should run the above script *only once* at the beginning of this course. You will handin this and all subsequent

labs by committing your source code to your local **`peerster`** repository and then using **`git push`** to propagate those versions to your submission repository in `/c/cs426/SUBMIT/`. Because Peerster builds on the Qt framework, in order to obtain a Makefile you'll need to run:

```
$ qmake
```

If you want to clone your repository remotely onto a non-Zoo machine, for development on your own machine for example, then first run the `getrepo.sh` script on the Zoo as explained above, then clone your submission repository onto your development machine by running:

```
$ git clone netid@node.zoo.cs.yale.edu:/c/cs426/SUBMIT/reponame peerster
Cloning into peerster...
done.
```

Replace *netid* with your Yale NetID and *reponame* with the repository name (starting with your netid) that was displayed when you ran `getrepo.sh`. You can also find your submission repository name from any working repository by looking at `.git/config` in the working repository.

Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you should *commit* your changes by running:

```
$ git commit -am 'my solution for lab1 exercise2'
Created commit 60d2135: my solution for lab1 exercise2
 1 files changed, 1 insertions(+), 0 deletions(-)
$
```

You can keep track of your changes by using the **`git diff`** command. Running **`git diff`** will display the changes to your code since your last commit, and **`git diff commit`** will display the changes relative to a particular prior commit; run **`git log`** to see your repository's history and commit numbers.

Hand-In Procedure

Each lab will have *two* relevant "due dates", listed prominently at the top of the lab: for a *bakeoff version* and a *final version*, nominally one week apart. By the *bakeoff version* deadline, you are expected to have implemented a functionally complete version of the assigned functionality that you have tested individually (e.g., by running multiple instances of your own implementation of Peerster), and which you *believe* is fully functional.

Then, following this deadline, we will schedule several *bakeoff* periods during which you and your colleagues will be expected to gather in the Zoo to test and debug the interoperability of your independently-developed Peerster implementations, and make sure they reliably work *together*. Debugging subtle interoperability issues can be just as difficult or more than developing an individually-tested and nominally "working" implementation, so we typically allocate a full week for this interoperability testing and debugging before the *final version* is due.

We will be grading your solutions via a combination of testing and code inspection. We will run your application and make sure it works as required, both when communicating with other instances of itself and when communicating with our own and other students' solutions. Manual code inspection will mainly come into play in the grading process only when we notice something going wrong, e.g., so that we can judge whether it was due to a trivial bug or a larger design flaw. We will not dock points merely for stylistic deficiencies or ugly hacks—although we strongly encourage you to try to keep your code clean and maintainable, because you will have to

keep building on it throughout the semester, and design flaws that you manage to work around in one lab may well come back and bite you in the next.

For lab grading purposes, we mostly care about two things:

1. The *bakeoff version* should be functionally complete: i.e., the necessary code should "be there" and more-or-less work, but it's OK if it has bugs at this point (as it probably will). You will lose points if by the bakeoff deadline you handin code that doesn't even compile, is obviously (from looking at the code) missing major pieces of functionality that you were assigned to implement, etc.
2. The *final version* is expected to be not only functionally complete but debugged and reliably functioning both when interoperating only with itself as well as when interoperating with your colleagues' implementations. This means your implementation needs to be robust against bugs or misbehaviors your colleagues' implementations might contain, which is a constant challenge faced by any developer of decentralized Internet-based systems. If your application crashes or misbehaves only when triggered by a bug in someone else's implementation, that's still a bug in *your* app that you will be held responsible for!

To handin each lab, you will need to **`git commit`** the working (bakeoff or final) version of your code, then **`git push`** it back to your master repository in the `/c/cs426/SUBMIT/` directory. You are welcome and encouraged to commit intermediate versions of your code to your local copy of the repository during development, e.g., after you have completed and tested a particular component or phase of the lab. You may **`git push`** these intermediate versions back to the master submission repository as well if you like, provided you make clear in the commit logs which version(s) are intermediate versions as opposed to "hand-in" versions.

Note that two very common mistakes new Git users make is to forget to **`git add`** new source files you add to a repository, and/or to forget to **`git push`** committed changes from your local working repository to the master repository (in the `/c/cs426/SUBMIT/` directory in this course). To help you avoid these mistakes, the `getrepo.sh` script creates a second local working repository, named `peerster-test` by default, which is connected to the same master submission repository. Whenever you complete a lab and submit it via **`git push`** from your **`peerster`** working repository, you are **strongly** urged to do this:

```
$ cd ../peerster-test
$ git pull
$ qmake
$ make
$ run it and make sure it works!
```

If you've forgotten to **`git add`** or **`git push`** anything, whatever you've forgotten will fail to turn up in `peerster-test`; then you can go back to your **`peerster`** working repository and fix it. Forgetting to perform an "end-to-end check" of this form on an assignment you *thought* you handed in complete and on-time will *not* be considered a legitimate excuse for incomplete or late handins!

Part 1: 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 documents to get an overview of the Qt model for widgets, layouts, and event notification via signals and slots:

- Widgets and Layouts
- Layout Management
- Signals and Slots

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.

> **Exercise 1.** Change the UI code so that the text-entry box at the bottom is automatically selected when the application starts: i.e., so that can just start typing without having to click in it first. You might find the Qt documentation page on keyboard focus helpful.

> **Exercise 2.** Change the UI so that the text-entry box can hold multiple (say 2 or 3) lines of text, and if the user enters a long line of text it automatically word-wraps onto subsequent lines within the text-entry box rather than just scrolling horizontally within one long line. To do this you may have to use a Qt class other than `QLineEdit` for the text-entry box; what other classes might work?

Part 2: Event-Driven Network Programming in Qt

Now we will start doing what this course is about, which is building decentralized systems that communicate over the network. We will build on UDP, the User Datagram Protocol, for all communication. Those not yet familiar with UDP can learn about it at these or any number of other references:

- Wikipedia on UDP - like a box of chocolates, you never know what you'll see there today, but it'll probably be OK.
- RFC 768, the official (though fairly trivial) specification of UDP as an Internet transport protocol. You'll be reading a few RFCs in this course, so it's worth getting used to their ASCII-art style.
- QUdpSocket Class Reference on how to use UDP in the Qt framework.
- Linux udp man page on using UDP sockets via the raw Berkeley sockets API.

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 the Zoo running an instance of Peerster will by default use a different and non-overlapping range of UDP ports, and you can run up to four instances of Peerster on the same Zoo node before running out of ports in this range. (Since these ports are in no way reserved for use by Peerster alone, it's quite possible that other applications running on the Zoo machine could have already grabbed one or more ports in your range, so don't be too surprised if you can only run three Peerster instances.)

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 convenient serialization/deserialization mechanism, which automatically supports a wide variety of simple and structured types.

To make our network message format simple but extensible, every network message Peerster sends or receives will be a serialized instance of a QVariantMap object, which is a simple key/value dictionary—similar to dictionaries in Phython 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.

> **Exercise 3.** Change the code to build and send a message to each of the four UDP ports above at the local host whenever the user enters a text in the text-entry box and presses return. The message should be a `QVariantMap` with a single key/value pair, the key being the string "`ChatText`", and its value being a `QString` containing the message the user typed.

> **Exercise 4.** 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`. If you run multiple instances of Peerster concurrently on the same Zoo

machine, whenever you type a message into any Peerster instance, the message should now appear in *all* of them.

Part 3: A Simple Gossip Protocol

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

1. 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.
2. 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 PARK 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.
- Textbook: Tanenbaum 4.5.2 "Gossip-Based Data Dissemination"

Gossip in Peerster

To implement a gossip protocol in Peerster, 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 Peerster 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.

Peerster will identify messages via a pair of values: an *origin* uniquely identifying the Peerster application from which a particular user chat message originated, and a *sequence number* that distinguishes successive messages from a given origin. A given Peerster 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"
- Textbook: Tanenbaum 6.2 "Logical Clocks", especially 6.2.2 "Vector Clocks"
- 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 Peerster'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 a Zoo machine (e.g., tiger.zoo.cs.yale.edu) 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 Peerster app should consider its "neighbors" to be whatever application, if any, owns the UDP ports with numbers immediately above or immediately below itself on the local host, while staying within the `myPortMin` through `myPortMax` range. That is, the Peerster 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 Zoo machine within your range of four UDP port numbers. We will deal with more interesting topologies across multiple nodes later.

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.

> **Exercise 5.** Implement a simple rumormongering scheme as described above. Test your application by running two, three, or four Peerster instances on the same Zoo 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 Peerster, 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.

> **Exercise 6.** Implement anti-entropy as outlined above. Test it to make sure it reliably propagates messages across multiple hops.

Interoperation

The last piece of functionality your Peerster gossip implementation will need is a way to connect to and interoperate with other Peerster nodes outside your private "sandbox" of four UDP ports on the local machine. To do this you will need to do three things:

- Add a mechanism to keep track of a dynamically modifiable list of *peers*, each of which can have its own DNS host name (as a `QString`), an IP address (as a `QHostAddress`), and a UDP port number (a `quint16`). You might add a new `Peer` class instantiated once to represent each peer, for example. You should henceforth treat this dynamic list as the set of neighbors from whom you pick random peers to gossip with, for example. Of course you should be able to receive gossip messages from all these peers as well, but that should require no special effort since `QUdpSocket::readDatagram()` accepts a datagram from any source.
- Your peer list should by default (on startup) include the four UDP ports at the local host (use `QHostAddress::LocalHost` as the IP address) in the `myPortMin` to `myPortMax` range, except for the port for this instance's own socket. Then you should have a way to add new peers given a string of the form "*`hostname`*`:`*`port`*" "*`ipaddr`*`:`*`port`*". You'll probably find the many string parsing methods in class `QString` helpful. To determine whether the specified host is an IP address or hostname, first just try converting it directly to an IP address with the `QHostAddress(QString)` constructor, and if that fails to produce a usable IP address, assume it's a hostname and use the `QHostInfo` class to resolve it to an IP address via the Domain Name System (DNS). Note that this is an asynchronous process since it involves network communication: you'll need to start the lookup via `QHostInfo::lookupHost()` and later fill in the peer's IP address when that process completes. (You should consider the peer "not yet operational" and simply never try to send it messages until the lookup completes.)
- Finally, you should modify Peerster to offer three ways to add peers in addition to the default four localhost ports:
    1. The user should be able to specify *`host`*`:`*`port`* strings on the command line when invoking your Peerster instance: see `QCoreApplication::arguments()` for an easy way to obtain these strings.
    2. You should add to your Peerster GUI a way for the user to add a peer dynamically while the program is running: e.g., a text-entry box that takes a *`host`*`:`*`port`* string and adds the named peer, or an "Add Peer" button that opens a dialog for this purpose, or whatever you prefer.
    3. When you receive a valid chat message in a UDP datagram that did *not* come from any of the

currently registered peers, you should automatically add that IP address and UDP port as a peer, so that chat messages this node sends in the future will automatically go to this previously-unknown peer as well. This way you'll have to add a peer-to-peer link explicitly only in one direction; the other direction will get "learned" automatically.

The above functionality should give your application everything it needs, at least in theory, to operate "at large" over the real Internet.

> **Exercise 7.** Implement the dynamic peer adding functionality above. Test it by running Peerster instances on *two* zoo nodes, making sure you can connect them to each other via either IP address or DNS hostname.

This completes the lab.