CPSC 426 Lab 2: Point-to-Point Messaging

Handed out Monday September 16, 2013
Bakeoff version due Friday September 20, 2013 11:59PM
Final version due Friday September 27, 2013 11:59PM

Introduction

In this lab you will enhance Peerster with the following features:

1. A simple routing protocol enabling Peerster instances to send each other unicast, point-to-point messages, rather than having to broadcast everything via gossip.
2. A NAT hole-punching implementation enabling Peerster instances to connect with each other directly even from behind NAT-firewalls.

Development and Hand-in

For this and subsequent labs you should keep working in, and committing changes to, the same Git repository you checked out for lab 1. Since Git records all history in an "append-only " log, you will always be able to roll back to previous versions for debugging or other reasons any time you need to.

As before, hand in your completed bakeoff version and final version, by the respective deadlines above, by using `git commit` and `git push` to push your changes to your master repository in the SUBMIT directory. Don't forget to `git pull` your handed-in version to your `peerster-test` repository or another clone of the repository to make sure the submitted version is complete and actually works! Since shopping period is over, points will be deducted for late or incomplete assignments as explained in the course overview

Part 1: Routing

To give Peerster nodes point-to-point communication ability, you will first implement a simple *destination-sequenced distance vector* (DSDV) routing scheme. This scheme has proven popular for routing in ad hoc mobile networks, due to its combination of simplicity and robustness against routing loops. You should first familiarize yourself with the general scheme via appropriate background reading sources, such as:

- Wikipedia (very brief summary)
- Original DSDV paper by Perkins and Bhagwat
- Background: Tanenbaum "Computer Networks" 5.2 Routing Algorithms; Coulouris 3.3.5 Routing.

In this scheme, each node maintains a table of destinations and, for each destination, a "next hop" to reach that destination. Peerster will piggyback its routing scheme on its gossip protocol: each rumor message will double as a route announcement message, and the sequence numbering scheme you already implemented for gossip purposes will act as the "destination sequence numbers" that the DSDV routing scheme requires.

> **Exercise 1.** Change your Peerster implementation to build and maintain a next-hop routing table: a key/value dictionary (e.g., `QHash`) whose keys are `Origin` identifiers

and whose values are (IP address, port number) pairs (e.g., `QPair<QHostAddress,quint16>`). Whenever your node receives a Rumor message with a new sequence number, record in your next-hop table, at the key corresponding to the message's `Origin`, the IP address and port number from which that rumor message arrived. This will be the next-hop on your route to the given node, which will remain in effect until you receive the next rumor message (with a higher sequence number) from the same node. For example, at the beginning, Dave sends a Rumor message to Bob.

As an example, assume that Dave's IP is "1.2.3.4", his port is "43433" and Sequence # is 23. Thus, Bob updates his DSDV routing table with item <Dave<"1.2.3.4", 43433>>. Then, Bob forwards that Rumor message to Alice. After receiving the message, Alice also updates her DSDV routing table with item <Dave<"5.6.7.8", 33333>>. Here, "5.6.7.8" is Bob's IP address and "33333" is the port of Bob. It is easy to know if Alice wants to send message to Dave, she just needs to send message to "5.6.7.8" (with port 33333), although she does not know the concrete IP address of Dave. After several hours, if Eve sends another Rumor message to Alice through Bob, both Alice and Bob will add new items in their routing tables. Specifically, a new item <Eve<"5.6.7.8", 33333>> will be added into Alice's routing table. (Of course, Bob will also update his routing table.) Please note that if Dave sends a new Rumor message with sequence number 25 to Alice through Jack, Alice will have a new item <Dave<"3.3.3.3", 55555>> instead of the old one (i.e., <Dave<"1.2.3.4", 43433>>). We assume "3.3.3.3" is Jack's IP and his port is "55555".

Unfortunately, if Peerster nodes only ever "announce" themselves to the network when the local user actually types in a message, nodes whose users are inactive for extended periods will never be announced in the network, and thus other nodes won't be able to find or route to them. We'll fix this by ensuring that nodes always send Rumor messages occasionally, merely for the purpose of updating other nodes' routing tables, even when the local user is idle.

Exercise 2. Add a periodic timer, set to fire about once per minute, that generates a *route rumor* message. A route rumor message is just like the "chat rumor" messages you already generate, except it contains only the `Origin` and `SeqNo` fields with no `ChatText` field. Modify your Peerster implementation's receive path to accept and forward route rumors as well as chat rumors: the processing should be essentially the same, except you display a message to the user only when the rumor message contains a `ChatText`. Besides the once-per-minute announcement, a Peerster node should also generate a single route rumor message when it first starts up (either sent to a random neighbor or broadcast to all neighbors; your choice) to "prime the pump" and get it known to other nodes quickly. Test your code by ensuring that a routing table entry appears "quickly" in other Peerster instances after startup without having to manually type any chat message.

Finally, you will need to add a mechanism for sending point-to-point messages that *uses* the next-hop routing tables built above.

Exercise 3. Add a GUI mechanism to display a list of node `Origin` identifiers known to this node—i.e., for whom a "recent" next-hop route is available. This node list might be located right beside the main chat text-display box, for example. The user should be able to select a node from this list (e.g., double-click, or select an item then press a separate button) to open a "private message" dialog box, into which the user enters a private message to send to that node.

A private message should be formatted as a `QVariantMap` containing three keys: a "`Dest`" key whose value is the destination node identifier (corresponding to the `Origin` of a prior rumor message); a "`ChatText` containing the text of the private message (a `QString`); and a "`HopLimit`" key whose value is a `quint32`, which you initialize to some constant value, e.g., 10, when sending a private message, and every node on the forwarding path will decrement along the way, discarding the message if the value reaches 0 before the message reaches the destination. Although routing loops should not *normally* arise in a DSDV protocol such as this, bugs or malicious behavior could still create loops, and the `HopLimit` protects against this risk.

Finally, update your Peerster implementation to accept private messages, forward them (after decrementing the `HopLimit`) if the `Dest` field refers to another node, and display them to the local user if the `Dest` field indicates that the message is for the local node. Test your code by creating indirectly-connected chains of Peerster instances, using multiple Zoo machines if necessary: e.g., instances A and B on one machine, C and D on another, connect B to C cross-machine, and verify that A can send private messages to D and vice versa.

Part 2: Middlebox Traversal

In this part of the lab you will implement a simple NAT/firwall traversal facility, so that even Peerster nodes behind NATs can (at least sometimes) establish direct connections to each other. For background and detailed information on NAT traversal techniques see:

- Wikipedia
- P2P Communication Across NATs, a paper summarizing basic NAT traversal techniques.

Setup: Using VMs to get NATted on the Zoo

To test your NAT traversal, however, you'll need Peerster instances running behind NATs, and none of the Zoo machines are behind a NAT. Of course one way to solve this problem is to run Peerster instances on one or more of your own machines on a home network, open wifi at a coffeeshop, etc. You are welcome to do this; however, we also want to make sure you can do everything you need to under "controlled conditions" on the Zoo. We can use virtual machines (VMs) for this purpose, which typically include built-in NAT functionality, so that the virtual node running inside the VM is actually behind a NAT even if the VM is hosted on a publicly addressed Zoo machine.

The QEMU virtual machine monitor is already set up on the Zoo, and for convenience we have built a master disk image, `/c/cs426/ubuntu-master.img` for an Ubuntu Linux system that is fully installed and has the Qt development tools you'll need to build and run Peerster. You should probably use this and *not*

host your own complete virtual machine disk images from scratch at least on the Zoo, as doing so will likely eat up your Zoo quota very quickly. For the same reason, you should *not* make your own copy of the `ubuntu-master.img` disk image; instead, you can use Copy-On-Write (COW) mode to create your own local virtual disk images that only store changes with respect to the master image. We've provided a script, `mkimg.sh`, to do this for you:

```
$ /c/cs426/mkimg.sh vm1.img
$ /c/cs426/mkimg.sh vm2.img
```

You'll probably want to make two separate VM images, as shown in the example above, so that you can run two Peerster instances concurrently in two VMs implementing two separate NATs, and make sure the two instances can talk to each other. We've also provided a script that starts QEMU with appropriate options to use these virtual images:

```
$ /c/cs426/runqemu.sh vm1.img &
$ /c/cs426/runqemu.sh vm2.img &
```

Implementing NAT Traversal in Peerster

To implement NAT traversal in Peerster, we will assume we have at least three Peerster nodes, A, B, and S, where S has a public IP address and acts like a "rendezvous server" allowing the two NATted nodes A and B to communicate. For testing you will start S on a Zoo machine directly (not in a VM), and start A and B inside two separate VMs, connecting both A and B to S. Peerster already knows how to forward messages indirectly via gossip, of course, so if you do this setup (with A and B each connected to S but not to each other) you should already see that A and B can talk to each other indirectly—but they're making S do all the work. To make sure we can actually verify that NAT traversal is working correctly once we implement it, let's deliberately "break" node S just enough to make it refuse to forward chat messages between A and B.

> **Exercise 4.** Add a feature to your Peerster to recognize a command-line option, "`-noforward`", which you will use when starting your rendezvous server S (but not the NATted regular nodes A and B). When in this mode, Peerster will *never* forward either private point-to-point messages or chat rumor messages to other nodes, only route rumor messages (containing no `ChatText`). Receive processing remains the same for all types of messages: that is, once you receive either a chat rumor or a route rumor, you update your received sequence number vector accordingly; the only difference is that when you get a message containing a `ChatText` key, you never forward it to another node (even if you received a status message from another node indicating that they don't have that chat message, which would normally trigger you to gossip it to them). You *must* still gossip route rumor messages (messages without a `ChatText` key) normally, however, as that is how NATted nodes such as A and B will learn about each other. To be clear, this behavior is definitely "broken" and not the way we would normally want Peerster nodes to behave, but we need it for testing purposes.
>
> Test your new `-noforward` feature with the A-S-B topology described above, and verify that messages typed into A never reach B and vice versa, but make sure (via your debugging output) that the route rumor messages are still propagating so that A and B obtain next-hop routing table entries for each other. These next-hop routing

tables will be useless for sending private messages, of course, because the intermediate node S is refusing to forward them.

Now it's time to implement the actual NAT traversal functionality. To do this, A and B need to learn each others' *public* IP addresses and port numbers, but only a node outside their NAT boundaries (such as S) can see this information. We will therefore modify Peerster's route-learning algorithm so that intermediary nodes like S can inform NATted nodes like A and B of possible NAT traversal opportunities and supply them with the needed information.

Exercise 5. Modify your Peerster so that whenever a node receives either a route rumor or a chat rumor message, it modifies the message to insert (or overwrite) two new keys: `LastIP` and `LastPort`. Your node should copy the source IP address that the message came from (as reported by `QUdpSocket::readDatagram()`) into the values associated with these keys, as values of type `quint32` and `quint16` respectively. This way, when S or any intermediary forwards a rumor message, it will give the receiver of the rumor the information needed to try to "short-circuit" past the intermediary if possible. For consistency you should implement this behavior for *all* rumor messages, even if due to the `-noforward` option you're not actually forwarding chat rumors.

Now modify your Peerster to look for valid `LastIP` and `LastPort` pairs in messages received, and if present, simply add those IP/port pairs automatically to the dynamic list of possible peers. In the A-S-B test topology, once A and B have each received a route rumor indirectly from each other, they should then have each others' public IP/port pairs in their neighbor peer tables, and should attempt to send each other route rumor messages directly via the normal periodic route rumor announcement process. If all goes well, once one such direct announcement has been sent and "punched holes" through the NATs in each direction, a working direct A-B communication path should exist, and chat rumors should subsequently get passed between A and B even when S is in the broken `-noforward` mode.

Now you have basic NAT traversal implemented, which should be sufficient for forwarding broadcast-style messages directly between the NATted nodes A and B. Unfortunately, you might find that private messaging between A and B is still unreliable—it might work or might not, depending on the circumstances—because the NATted nodes A and B will have *two* possible routes to each other: one via S (which won't work because S is intentionally broken), and one direct path (which should work). If we wanted to be smart and resistant to bad behavior in the network, such as nodes like S that selectively refuse to forward traffic, we might want to design a protocol by which A and B can interactively test several possible routes and use one that works. You're welcome to design such a mechanism if you like, as an extension to the basic protocol. To keep things simple for now, however, we'll use a simple policy of preferring direct routes over indirect routes. In the current Peerster protocol as it stands, an easy way to spot the difference is to check whether or not there's a `LastIP`/`LastPort` pair in a message.

Exercise 6. Change your Peerster routing algorithm to prioritize direct routes. Whenever you receive a (route or chat) rumor message, consider the path represented

in the message to be a direct route if the message contains no `LastIP`/`LastPort` pair, and an indirect route if it does. As before, routes with new (higher) sequence numbers unconditionally override older routes from messages with lower sequence numbers; we need to maintain this invariant to ensure that routes get updated properly on a dynamic basis no matter how the system changes. However, when you get a rumor message multiple times with the *same* sequence number (i.e., due to being forwarded over multiple different paths), if the routing table entry you have for that sequence number represents an indirect route, but a newly arriving copy of the same message represents a direct route, then you should override the older indirect route with the newer direct route.

One more change you might have to make this work reliably is to modify the gossip behavior so that new route rumor messages always get forwarded to *all* peers immediately on receipt, rather than forwarding the rumor only to one randomly-chosen peer and letting the rumormongering process complete the propagation. Although this obviously increases network load and there are smarter ways to fix the problem, as long as you're sending route rumor messages only occasionally (e.g., once a minute), the increased load should be negligible for our purposes.

In the A-S-B topology with A and B behind NATs and S refusing to forward either chat rumors or private messages, once the NAT traversal and route learning process has set up “preferred” direct forwarding routes between A and B, you should now reliably be able to send private messages directly between A and B.

This completes the lab.