Lab 4

Handin Procedure

When you are finished, edit the `answers.txt` file and add your name(s) and email address(es), student ID(s), any challenge problems you may have done.

ALSO REMEMBER TO WRITE THE INFORMATION REQUIRED BY TASKS 2 AND 3, INCLUDING WHICH ROBUSTNESS PROBLEMS YOU FIXED, AND HOW YOU MADE YOUR PEER EVIL!

Also include any other information you'd like us to have. Then run `make tarball`, which will generate a file `lab4-`*`yourusername`*`.tar.gz` inside the lab4 directory. Upload this file to CourseWeb using a web browser to turn in the project.

Overview

This lab concerns distributed computing and security via defensive programming.

Distributed computing uses operating system interfaces to build complex systems from *many* interacting computers. Your Lab 4 code participates in a *peer-to-peer (P2P) network*, formed by connecting nodes that can perform *both* the roles of clients and servers. You're probably familiar with peer-to-peer networks used for downloading files, such as BitTorrent; we have designed a peer-to-peer system somewhat like BitTorrent, from which you can download some lolcats [http://www.icanhascheezburger.com/].

Our peer-to-peer network consists of two kinds of nodes: **trackers** and **peers**. Trackers keep track of which peers are connected to the network; peers actually download files from each other. We have written a tracker and a functional peer. When you download `lab4.tar.gz`, our peer code is in `osppeer.c`. We also have five trackers running on our own server. They are:

- 131.179.80.139:11111 -- the *normal tracker*. Students' peers can connect to this tracker and serve files to each other. This tracker is "seeded" with a mixture of good and slow peers running on our servers.
- 131.179.80.139:11112 -- the *good tracker*. Our own peers are connected to this tracker.
- 131.179.80.139:11113 -- the *slow tracker*. Slow peers, which serve files slowly, are running on this tracker. You can use this tracker to verify that your peer downloads files in parallel.
- 131.179.80.139:11114 -- the *bad tracker*. Bad peers, which attack other peers that attempt to connect, are running on this tracker.
- 131.179.80.139:11115 -- the *popular tracker*. Our own peers are connected to this tracker, plus a bunch of fake peers, making the tracker look very popular.

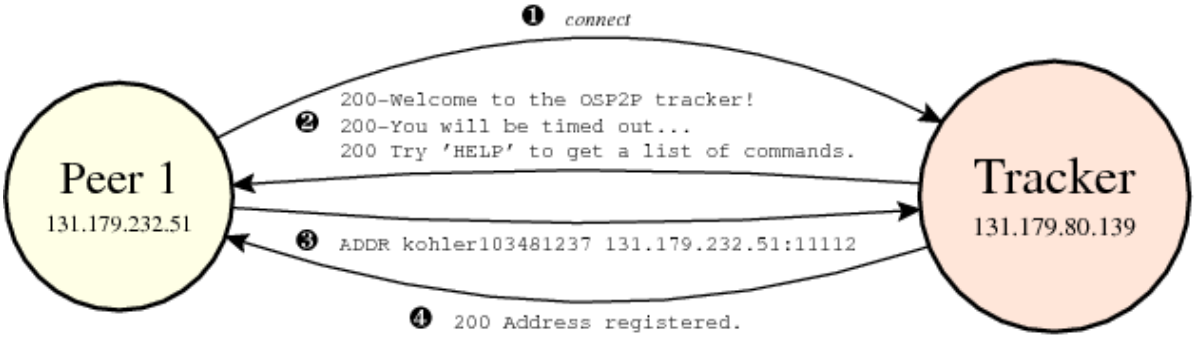
Peer-to-peer communication in the OSP2P system is built from a series of remote procedure calls (RPCs). These RPCs are formatted as normal text, formatted somewhat like other Internet protocols (HTTP, FTP, SMTP, and so forth). So you can use the `telnet` program to try out the RPCs yourself!

The OSP2P Protocol

A peer logs in to a tracker as follows.

1. The peer connects to the tracker machine.
2. The tracker responds with a greeting message.
3. The peer registers its IP address and port with an "ADDR" RPC.
4. The tracker responds with a message indicating success or failure. Now the tracker will report this peer to other peers who want to download files.

Here's a picture of this protocol.

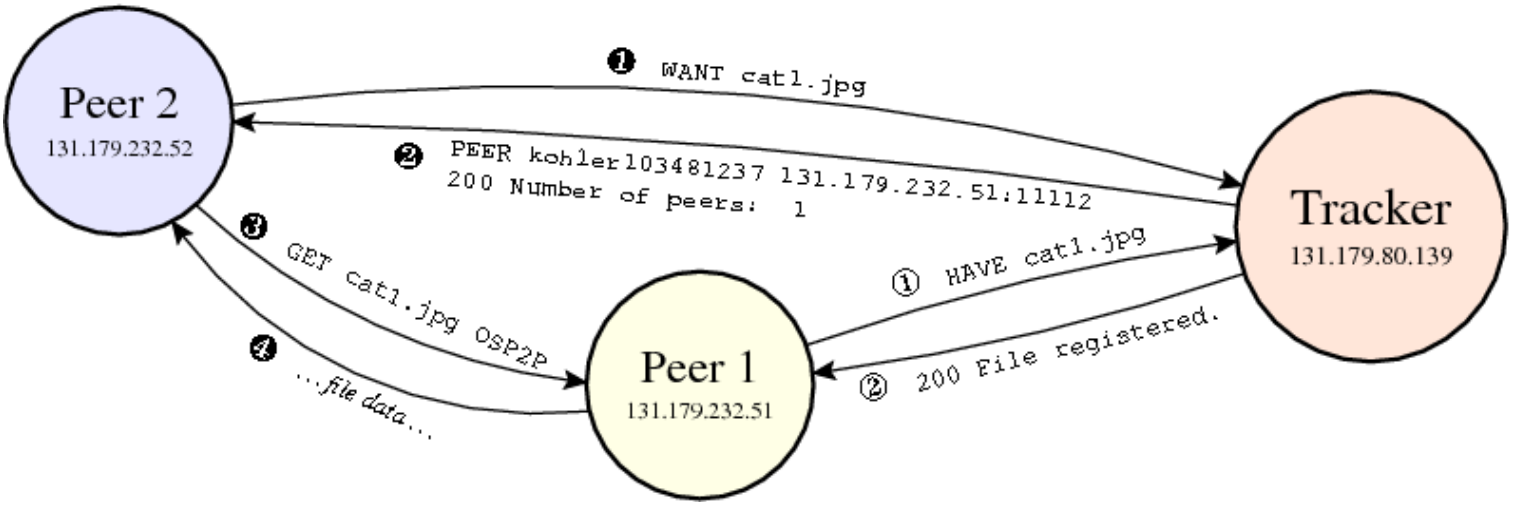


The peer then informs the tracker of the files it is willing to upload to others.

1. The peer registers each file with a "`HAVE` *`filename`*" RPC.
2. The tracker responds with a message indicating success or failure.

When a peer wants to download a file from the network, it communicates with *both* the tracker *and* other peers.

1. The downloading peer asks the tracker which peers have the file with a "`WANT` *`filename`*" RPC.
2. The tracker responds with a message listing the available peers willing to serve that file.
3. The downloading peer picks an available peer from this set, and connects to that peer, sending it a "`GET` *`filename`* `OSP2P`" RPC.
4. The uploading peer responds to this RPC by sending the entire file.



Once the downloading peer has downloaded the entire file, it informs the tracker that it, too, has the file.

1. The downloading peer registers its newly downloaded file with a "`HAVE` *`filename`*" RPC.
2. The tracker responds with a message indicating success or failure.

What You Must Do

At a high level, there are three parts to this lab.

1. The peer code that we hand out is *sequential*: it downloads from peers, and uploads to peers, one at a time. Since the network to any one peer is likely a bottleneck, sequential downloading has poor utilization. You are to make the peer behave *in parallel*: it should be able to download from peers, and upload to peers, more than one at a time.
2. The peer code that we hand out is *not robust*: communicating with an evil peer can cause your peer code to crash or hang forever. You are to fix the peer code to make it robust. This requires defensive programming.

3. The peer code that we hand out is *good*, i.e., not evil. You are to make your peer code optionally *evil*: if you start the peer with the `-b` option (b is for "bad"), it will try to confuse or crash any peers that attempt to download files from it.

Lab 4 does not have explicitly marked exercises; you have more freedom to change the code however you think is necessary.

This lab can be completed on any Unix-compatible machine, including the SEASnet Linux machines, the Linux lab machines, and Macintosh machines.

Solaris notes: To run the lab on SEASnet Solaris machines, you will need to edit the `GNUmakefile` and remove the `#` sign from the `#LIBS` line near the top of the file. You will also need to use the `gmake` program, not `make`; for example, `gmake run-good`.

Investigating the OSP2P Protocol

We recommend that you first play around with the OSP2P protocol by running our peer, and by using the `telnet` program to talk directly to our trackers and peers.

First, run `make run-good`. This will build our peer, start it and connect to the good tracker, and download three files into the `test/` directory, `cat1.jpg`, `cat2.jpg`, and `cat3.jpg`. Here's what we see.

```
+ ./osppeer -dtest -t11112 cat1.jpg cat2.jpg cat3.jpg
* Tracker's greeting:
200-Welcome to the OSP2P tracker!
200-You will be timed out after 600 seconds of inactivity.
200 Try 'HELP' to get a list of commands.
* Listening on port 11112
* Tracker's response to our IP address registration:
220-This tracker is set up to ignore external peers,
220-so that you don't have to worry about problems caused by other students.
220 The address registration has been ignored.
* Registering our files with tracker
* Finding peers for 'cat1.jpg'
* Connecting to 131.179.80.160:11114 to download 'cat1.jpg'
* Saving result to 'cat1.jpg'
* Downloaded 'cat1.jpg' was 44400 bytes long
* Finding peers for 'cat2.jpg'
* Connecting to 131.179.80.160:11114 to download 'cat2.jpg'
* Saving result to 'cat2.jpg'
* Downloaded 'cat2.jpg' was 41259 bytes long
* Finding peers for 'cat3.jpg'
* Connecting to 131.179.80.31:11114 to download 'cat3.jpg'
* Saving result to 'cat3.jpg'
* Downloaded 'cat3.jpg' was 34812 bytes long
```

`osppeer` will then pause, waiting for other peers to upload files from it. If you wait long enough, you will see some additional messages, like this:

```
* Got connection from 131.179.80.31:33916
* Transferring file cat3.jpg
* Upload of cat3.jpg complete
* Got connection from 131.179.80.31:33917
* Odd request CHECKUPLOAD cat3.jpg successful!
```

The "good" tracker has a special additional feature, where our peers connect back to your peer and attempt to download files from it. If the download succeeds, and the file contents match the expected values, our peers inform you by connecting with a message like "`CHECKUPLOAD cat3.jpg successful!`". If there is a problem, you will see a message like "`CHECKUPLOAD cat3.jpg FAILED: empty file`", or something else. *This only works on the good tracker.*

Press Control-C to quit `osppeer`. To run against different trackers, try `make run` (normal tracker), `make run-slow`, and `make run-bad`.

You can also run `./osppeer` yourself from the command line. Here's what its arguments mean.

- `-dtest` : Run in the `test` directory. The peer will register all files located in the `test` directory with the tracker, and download files into the `test` directory. You should generally run with the `-dtest` argument.

- `-t11112` : The `-t` argument specifies the tracker IP address and/or port. You will generally just specify ports; 11111 is the normal tracker, 11112 is the good tracker, 11113 is the slow tracker, and 11114 is the evil tracker. The default tracker IP address is 131.179.80.139.
- `cat1.jpg cat2.jpg cat3.jpg` : These are the names of files that `./osppeer` should try to download. If you give no filename arguments, `./osppeer` will not download any files.

If you run `make run-slow`, you will see what we mean by "slow peers". And if you run `make run-bad`, our peers will do their worst to your peer, attempting to crash it or monopolize its resources.

Now, try connecting to the tracker and to a peer *yourself* with the `telnet` program. All the OSP2P RPCs use a text-based format which you can type yourself. Run "`telnet read.cs.ucla.edu 11111`" to connect to a tracker. Type `HELP`, then play around with some of the other commands. Type `QUIT` to quit the connection. Run "`telnet read.cs.ucla.edu 11116`" to connect to one of our (normal) peers. Type "`GET about.txt OSP2P`" to request a file. The peer should shut down the connection automatically after sending the file.

Task 1: Parallel Uploads and Downloads

The skeleton code that we hand out is a fully functioning OSP2P peer client, but all of its actions are performed *serially*. In particular, if multiple desired files are given on the command line, they are processed one at a time. Only after the client successfully downloads the first file (or fails for all peers sharing the file) does it attempt to download the second. Similarly, once the client begins accepting connections from peers, it processes them one at a time.

Your first task is to change the code so that it can process multiple downloads in parallel and multiple uploads in parallel. This is a fairly open-ended problem. You may choose to implement the concurrency by forking processes (like your shell in lab 1), or by using threads. You can also implement parallel downloads and uploads using *non-blocking I/O* and an *event-driven* style. Event-driven programming is the most challenging programming style for this lab, but frequently event-driven programs both run faster and use fewer resources than other programming styles. If you are a real performance junkie try events. (We have a description of event-driven programming here. You might also want to try using an event-driven programming library, such as libevent [http://www.monkey.org/~provos/libevent/] or Tamer [http://www.read.cs.ucla.edu/tamer/].)

Your client may process downloads in parallel with uploads, or it may processes downloads in parallel and then, once all downloads are done, processes uploads in parallel. Either approach is acceptable.

No matter which approach you choose, you must ensure that your code does not spin (busy-wait -- wait with poor utilization). In particular, when your client is not downloading or uploading, it should be using very little CPU time. Even when it is uploading or downloading multiple files at once, it should *not* take 90-100% of the CPU. You can use the `top` program to see how much CPU time is being used by `osppeer`.

Task 2: Security and Defensive Programming

Writing networked programs is difficult not just because of the need for speed, but also because it is very important to avoid **security holes**. These are programming mistakes that might allow an attacker to crash the networked program, cause it to misbehave, trick it into doing something it shouldn't, or even gain access to the machine.

Your job is to fix our peer's security holes and make it robust against as many attacks and network problems as you can imagine.

Here's what the peer *should* do: its intended specification.

- The peer should serve files located in its current directory to requesting peers.
- The peer **should not** serve files located in any other directory.
- Other peers should not be able to monopolize the peer's resources. For example, another peer should not be able to fill up this peer's disk, or fill up its file descriptor table with non-functioning connections.

Furthermore, there is at least one buffer overflow bug in the code that could crash your client with a segmentation fault -- or, worse, give an attacker control of your computer! While a buffer overflow bug would not happen in a safe language like Java, other security bugs can happen in **any** language. It is important for you to understand the need for careful programming in any network environment like this.

Your job is to solve the following problems:

- The peer should not serve files outside the current directory.
- Find and fix any buffer overrun bugs. **List conditions that would have triggered the bugs in your `answers.txt` file.**
- Otherwise improve the peer's robustness. Use `make run-bad` to test your client; our "bad" tracker tries to abuse connecting peers in a variety of interesting ways. (It chooses random attack strategies each time.) You should be able to run `make run-bad` indefinitely, without eventually running out of disk space or memory or file descriptors. **Tell us what robustness problems you fixed in your `answers.txt` file.**

If you find other problems, feel free to solve them too, and tell us about them.

In general, you may assume that only peers will attack you; the tracker is *trusted*. However, there is one bug in our handout code's tracker communication: if many, many peers are logged in at once, then our peer will be unable to download anything, because communications with the tracker will get confused. You should find and fix this bug as well. To test, run your `osppeer` program against the popular tracker. A message like `tracker connection closed prematurely!` indicates that you haven't fixed the bug.

Task 3: Attack!!!

Your final task is to be the bad guy. Find **at least two** ways that a malicious peer client could attack the client implemented by the skeleton code (and potentially other students' clients). The attack may cause the client to crash or may simply cause it to fail to respond to legitimate requests or run out of resources (denial of service). Your peer should go into "malicious mode" when the `evil_mode` global variable is nonzero. It may attack only those peers that actively try to download its files ("uploader attacks"), or, even better, **seek out new peers to attack** by connecting to them ("downloader attacks")!

Add these security exploits to your client code. These attacks should only be active when the `evil_mode` flag is set. Otherwise, your code should behave nicely. **Also add a description of your attacks in the `answers.txt` file. Why are your attacks actually attacks?**

Good luck!!!

Extra Credit Task: File Integrity

Many of the evil peers' attacks are pretty easy to detect, but some are not so easy. In particular, how can your peer detect that an evil peer has sent a corrupted version of the file?

The OSP2P system actually already supports a strategy for detecting corruption, called *cryptographic hash functions* or *checksums*. When our clients connect to the tracker, they not only register interest in the file, they also tell the tracker the MD5 checksum [http://en.wikipedia.org/wiki/MD5] of the file's data (calculated with `md5_init`, `md5_append`, and `md5_finish_text(...,1)`; these functions are defined in `md5.h`). Another peer can compare this checksum with the checksum of the file they download.

Your job in this extra credit task is to detect corruption (intentional or not) using the trackers' checksum support. We aren't saying much more about *how* to do this; poke around on the tracker using `telnet` to figure out how checksum support works.

Design Problems

Design problem guidelines

Downloading From Multiple Peers

Currently, our client downloads a file from one peer at a time. Although we benefit from concurrency by allowing multiple files to be downloaded at the same time, we are not taking advantage of the fact that multiple peers are serving the same file. If we are downloading a large file, we can speed up the process by downloading it in parts from multiple peers at the same time. Downloading and sharing files in parts is the fundamental premise behind peer-to-peer networks such as Bittorrent. For this design problem consider how you might download a file from multiple peers at the same time. Our protocol currently only supports getting a file in its entirety, so you will need to suggest how we should augment the

protocol to support this new behavior. You will also need to consider other factors such as how might we decide to break a file into parts?

Another main consideration is file integrity. Given the number of potentially adversarial peers in ours system, there's a good chance we might receive a corrupted part. Our checksums are valid for the entire file, so we would have to wait until the last piece to check! Even worse, we would have no idea who sent us the bad piece, meaning we might connect to them again. What changes could we use to protect ourselves against this?

As an additional part of this design project, you could also explore performance issues. What sort of downloading strategies might result in faster download times? Should some peers be favored over others? Design an algorithm that implements a strategy you think should would and explain why.

When working on this lab, if you modify your client to support additional protocol commands, you should be aware that other clients will not likely not support these new commands. As such, you will need to test any implementation using a network of clients running your code.

Access Control

The peer is happy to serve other peers any data contained in its current directory. Fancier programs, such as real web servers, allow users to specify *which* files in a directory can be served. For example, this syntax tells Apache to refuse to serve the `osppeer.c` file:

```
<Files "osppeer.c">
Order allow,deny
Deny from all
</Files>
```

This syntax would usually go in a file called `.htaccess`, in the directory containing `osppeer.c`. (Note that our current peer actually already supports primitive access control! How??)

Design an access control syntax for our peers. Will you support Apache-style `.htaccess` files, or something else? What type of syntax will you support? For full credit, you should design a very flexible access control syntax. Consider such issues as limiting access for some files to *limited* sets of peers, defined based on (say) network address; symbolic links; and so forth.

Transmitting Encrypted Files

Extend our current design to allow peers to send **encrypted** data. You should consider three types of encryption. In increasing order of safety:

1. Hiding file contents from network snoopers.
2. Hiding file contents from unauthorized peers. I.e., if a peer does not know the right key(s), then the peer will not be able to understand a download file.
3. Hiding the *existence* of a file from unauthorized peers. I.e., if peer 1 does not know the right key(s), then peer 1 cannot tell which files peer 2 has made available. (Perhaps peer 1 will be able to tell that peer 2 has registered 5 files with weird gobbledegook names, but peer 1 cannot tell what those files' true names are, and peer 1 cannot download their data -- encrypted or not -- from peer 2.)

Minimizing Bandwidth When Sharing

(**Note**: this is the most difficult design problem and requires *significant* implementation)

Bandwidth is an important consideration for peers that pay per byte when uploading. Our current implementation ignores how much it uploads which could be costly when running on such a service. One way to address this problem is by minimizing the number of bytes sent when transmitting groups of files, while still guaranteeing that all files have been sent. BitTorrent-like networks can minimize any *individual* peer's upload bandwidth by attempting to *spread* upload requests over the whole network.

For example, say that peer 1 has files "cat1.jpg", "cat2.jpg", and "cat3.jpg", whereupon peers 2-5 log in and attempt to download these 3 files. In our current system, all of peers 2-5 might attempt to download all 3 files from peer 1! Peer 1 wastes a lot of upload bandwidth, uploading each file 4 times. Alternately, what if the peers **spread** their uploads? Here's an

example of how this might work.

1. Peer 1 uploads "cat1.jpg" to Peer 2.
2. Peer 1 uploads "cat2.jpg" to Peer 3; Peer 2 uploads "cat1.jpg" to Peer 4.
3. Peer 1 uploads "cat3.jpg" to Peer 4; Peer 2 uploads "cat1.jpg" to Peer 5; Peer 3 uploads "cat2.jpg" to Peer 2.
4. Peer 1 uploads "cat3.jpg" to Peer 5; Peer 2 uploads "cat1.jpg" to Peer 3; Peer 3 uploads "cat2.jpg" to Peer 4; Peer 4 uploads "cat3.jpg" to Peer 2.
5. Peer 2 uploads "cat3.jpg" to Peer 3; Peer 4 uploads "cat2.jpg" to Peer 5.

The download has completed pretty quickly, with all sorts of parallel downloads across peers -- but even better, no peer has uploaded more than 4 files!! This is 3 times better for Peer 1 than the naive approach.

Your job in this design project is to design concrete algorithms by which a peer network can reduce any *single* peer's contribution to the network by spreading uploads across multiple peers. You should consider issues like misbehaving peers as well. Again, note that this requires a several (potentially significant) changes to the current design before you can begin investigating your design choice.