CSCI-551 Project B
Analyzing BitDrip

March 27, 2006

This project is due noon, May 3, 2006. Late submissions cause a 25% per day penalty for that part of the project. See the class web page for details.

1 Introduction

The purpose of this project is to explore analyzing network traffic. You will be modifying and using your code from Project A, or if you prefer you can use the code provided by the TA.

Your final turned in code must compile and run on **nunki.usc.edu**, but for testing and development we highly recommend you use on-campus workstations when ever possible.

(Just a reminder: ISD has asked that students *not* run their programs on aludra.usc.edu, since that server is used by many people. They also encourage students doing development on campus to use workstations rather than nunki. Students working remotely should use nunki.)

Your modifications to the BitDrip P2P simulation code will help you do a simple analysis of the throughput between peers. Additionally you will be using libpcap tools to analyze pcap network traffic traces of your BitDrip peer to peer simulation.

2 Overview

To complete project B, you must modify the BitDrip code to do two things:

1. Detect *freeloaders*. Freeloaders discussed in more detail in section 2.1. You will design an approach that lets each peer detect peers that don't "pull their weight".
2. Analyze *tcpdump traces* of your peer interaction, both to verify your performance, and to relate TCP behavior to your application behavior.

2.1 Freeloaders

Peer-to-Peer systems work because everyone shares data with each other, thus the system benefits from everyone's bandwidth rather than just the server's bandwidth or the

seeder's bandwidth. (Recall, a *seeder* is a peer that has a complete copy of the desired file.)

A **freeloader** is a peer that only takes data from the system but does not upload any data to others, or that takes much more than it sends.

To study freeloaders, in Project B, when you ask for class peers, one (or more) of them may be a freeloader. It is your job to minimize the impact of the freeloader on the peer group. To do this, you must do two things:

1. Identify if there is a freeloader, and which peer is freeloading.

2. Reduce the amount of sharing done with a freeloader.

It is OK if your simulation quits and the Class Freeloader does not have the full file—in fact, this is desirable.

Be careful though in determining which peer is a freeloader—remember that peers just starting out in downloading a file may not have any data to upload and share—and so these new peers may appear to be freeloaders at first. It is important to give new peers a chance to download some of the file before you determine if they are a freeloader.

Note that we expect each peer to determine for itself which other peers may or may not be freeloaders. (We do not expect you to create a distributed algorithm where your peers communicate out-of-band for this purpose!)

3 Phases

Like Project A, we've divided Project B into phases to help you work through the project step by step. In Project B, however, a phase number corresponds to an actual test—most phases do not involve any more coding, they only require you to test your code for that phase. In other words, phases numbers are used to provide an easy way to select from different inputs.

Also, a significant focus of Project B is not just your code, but your writeup about what the code does. Please consider that when writing up your answers for the project.

3.1 Phase 10–19

The purpose of these phases is to add and test the ability to detect freeloaders (on the fly).

First, you need to document your plan to detect freeloaders. Create a text file called "freeloader-plan.txt". In this file, describe specifically a) how you define a freeloader (quantitatively), b) what statistics each peer collects to identify this, c) when you expect to be able to make a decision, and why you cannot make a decision earlier or later.

Starting with your Phase 6 code from Project A (or the TA's code for project A), modify the code so that when a freeloading peer is detected your peers write out this information to a file called "xx-freeloader.out", where xx is the peer ID of the peer that detected the freeloader. The contents of the file should contain three numbers—the peer ID of the detected freeloader, the time in peer xx determined that this peer ID

was a freeloader (measured in seconds since the freeloader contacted peer xx), and the number of bytes peer xx sent to the detected freeloader before it was detected. If no freeloader is detected by the time a peer is ready to terminate, the peer should output "no freeloaders" to the xx-freeloader.out file.

(For this question, "bytes sent" mean all application bytes sent over the TCP connection. This count doesn't include TCP or IP headers, but does include application protocol headers and control messages.)

For example, if peer 3 first contacts the tracker at 5 seconds after your experiment was started (i.e., when you started your manager), and then peer 3 determined that peer 28 was a freeloader one second later after sending it 4,148 bytes, peer 3's "3-freeloader.txt" should contain:

```
28  1 4148
```

When testing with the class peers, the class peer spawner will spawn 2–4 class peers for you. At most one of those peers will be a freeloader. In some cases none will be a freeloader, in some cases there will be freeloaders who share nothing, and in others cases freeloaders who share just a bit (a "semi-freeloader").

We will provide three known test cases:

- Phase 10 will have no freeloaders.
- Phase 11 will always have a known complete freeloader running on one class peer.
- Phase 12 will always have a semi-freeloader running on one class peer.
- Phase 13 will have a random class peer be either a freeloader or a semi-freeloader. In some cases no freeloader will be started for phase 13.

We will use phases 14–19 for our use to evaluate your code (they will not work before the due date, but you should treat them like phases 10–13 for freeloader detection).

3.1.1 Sample Input

```
# Your peers to spawn.
# Peer ID    File Name       Start Time
  1          file09.jpg       0
  2          file09.jpg       1
  3          file09.jpg       1
  -1         -----           -1
# Class Peers
# Number to Spawn   Phase Number
  3                 11
```

3.2 Phases 20–29

For these phases, the emphasis is on *analysis* of the network performance of your protocol.

After modifying your code to output received throughput and related statistics, you will run phases 20–25 and summarize the results in a file called "analysis.txt" as discussed below.

We will use phases 26–29 for our use to evaluate your code (they will not work before the due date, but you should treat them like phases 20–25).

Starting with your code for the previous phases, modify it so that your peer can determine the following:

1. The start and end time your peer communicates with each other peer.
2. The number of bytes your peer receives from each other peer.
3. The mean received throughput to retrieve your file (for the whole duration of the exchange, from when you get your first byte to the last byte).

Each peer should output two files: one called "xx-stats.out" and one called "xx-total.out".

The file "xx-stats.out" should have the following format:

```
peer_id start_time end_time bytes_from_peer bytes_to_peer mean_throughput
```

There should be one line in this file for each peer your peer communicates with.

The "xx-total.out" file should contain the average (across all peers your peer communicated with) received throughput obtained.

All units should be in seconds, bytes or bytes/s.

When you're code is ready, run one experiment for each of the six configuration files for phase 20-26 listed below. Calculate the average throughput (the average of the results from all the "xx-total.out" files from that run) across all your peers for each experiment and put the results in a file called "analysis.txt".

The file "analysis.txt" should contains one line per phase number:

```
phase_num averg_recved_throuput
```

We will look at several possible cases: phases 20–22 will vary parameters on the class peer with a single student peer. Phases 23–25 will vary the number of student peers. Phases 26–29 will be used in testing.

When you have completed these experiments, create a file called "analysis-discussion.txt" and in it answer the following questions:

1. What trends, if any, did you see in phases 20–22? What was the cause of these trends (if any)?
2. What trends, if any, did you see in phases 23–25? What was the cause of these trends (if any)?
3. How does the overall downloading rate change when you have more non-freeloading peers? What about your experiments support that result? Why does the downloading result change this way?

3.2.1 Sample Input

Sample input manager.conf files:

1. Phase 20 "manager.conf":

```
# Your peers to spawn.
# Peer ID    File Name       Start Time
  1          file09.jpg       0
  -1         -----           -1
# Class Peers
# Number to Spawn   Phase Number
  1                 20
```

2. The other phase 21–22 will be the same as phase 20 except for the phase number. (We will change parameters on the class peer.)
3. Phase 23 "manager.conf":

```
# Your peers to spawn.
# Peer ID    File Name       Start Time
  1          file09.jpg       0
  2          file09.jpg       0
  -1         -----           -1
# Class Peers
# Number to Spawn   Phase Number
  1                 23
```

4. Phase 24 "manager.conf":

```
# Your peers to spawn.
# Peer ID    File Name       Start Time
  1          file09.jpg       0
  2          file09.jpg       0
  3          file09.jpg       0
  -1         -----           -1
# Class Peers
# Number to Spawn   Phase Number
  1                 24
```

5. Phase 25 "manager.conf":

```
# Your peers to spawn.
# Peer ID    File Name       Start Time
  1          file09.jpg       0
  2          file09.jpg       0
  3          file09.jpg       0
  4          file09.jpg       0
  -1         -----           -1
# Class Peers
# Number to Spawn   Phase Number
  1                 25
```

3.3 Phases 30–39

These final phases explore analysis using tcpdump.

For this phase you need to relate the TCP packets you see in a tcpdump output to the application-level behavior you know about from bit drip.

Each student will do *one* of phases 30–39. To pick which one you do, use the last digit of your student ID and put a '3' in front of it.

First, modify your code to output your tracker's listening port and the listening port of the class peer. (It's probably easiest to have your tracker output this information).

Run a one-on-one experiment with the class peer spawner using the phase number that corresponds to your student ID. A text output of a tcpdump packet capture of your experiment can then be found at:
`http://sea.usc.edu/csci551/dumps/USERNAME-CLASSPEERPORT-TRACKERPORT.txt` where USERNAME is replaced by the username you gave to the class peer spawner (your

class wiki username), CLASSPEERPORT is replaced by the listening port the class peer reported to your tracker and TRACKERPORT is the listening port of your tracker.

Copy this tcpdump file into the file "phase-xx-tcpdump.txt" (where xx is the phase you chose). Make a second copy of that file as phase-xx-annotated.txt, and then go through and add the following annotations:

- "start of new connection for zz" any time your peer makes a new connection to the seeder to send zz type of request. If you're sending multiple requests per connection, say what is requested during this connection.
- "connection established" after a connection has been established.
- "communication with tracker" any time the class peer contacts your tracker.
- "segment update request" every time your peer asks for a segment update request
- "segment update reply" every time the class peer responds with its segment update
- "segment xxx request" every time your peer requests a segment from the class peer
- "segment xxx reply" every time the class peer responds with a segment
- "end message" after every time a BitDrip message has been completely sent

In addition, answer the following questions in the file "phase-xx-discussion.txt":

1. How did you determine which TCP segments correspond to which application-level events?
2. Did you observe anything unusual observed in the protocol? If so, what was it? (For example, did it follow the protocol.)
3. What is the delay between requesting a piece and the class peer's response? Is this delay consistent?

3.3.1 Sample Annotated Tcpdump File

Below is a snippet of what we expect your annotated tcpdump file to look like. Lines are truncated to just show part of the tcpdump output.

```
>>>>>> start of new connection for segment 1 request
1143444533.571610 128.125.5.168.34906 > 204.57.0.97.36978: S [tcp sum ok]
1143444533.571627 204.57.0.97.36978 > 128.125.5.168.34906: S [tcp sum ok]
1143444533.572198 128.125.5.168.34906 > 204.57.0.97.36978: . [tcp sum ok]
>>>>>> connection established

>>>>>> segment 1 request
1143444533.572343 128.125.5.168.34906 > 204.57.0.97.36978: P [tcp sum ok]
1143444533.572355 204.57.0.97.36978 > 128.125.5.168.34906: . [tcp sum ok]
1143444533.573081 128.125.5.168.34906 > 204.57.0.97.36978: P [tcp sum ok]
1143444533.573221 204.57.0.97.36978 > 128.125.5.168.34906: . [tcp sum ok]
1143444533.573956 128.125.5.168.34906 > 204.57.0.97.36978: P [tcp sum ok]
1143444533.574093 204.57.0.97.36978 > 128.125.5.168.34906: . [tcp sum ok]
[ .... ]
```

4 File Layout, Turn In and Writeup

Your program *must run on nunki.usc.edu*, and be written in C or C++. It must compile using */usr/usc/bin/gcc* (if using C), or */usr/usc/bin/g++* (if using C++).

Your project must have the following for turn-in:

1. A **Makefile** for compiling all source files.
   The Makefile should have the following targets:

   all : builds all executables, including an executable file called "projb"

   clean : removes all old .o files (*.o) and all executables.

   The make file must also use /usr/ccs/bin/make - so be careful not to use extensions that will not work with SunOS's make.
2. All C/C++ files needed to run your simulation. The whole project should be broken up into at least two C/C++ files (modularize!!). If you have a good file hierarchy in mind, break it up into more files, but the divisions should be logical and not just spreading functions into many files. Indicate in a comment at the front of each file what functions that file contains.
3. Header files (.h files) used to define all data structures you use, any **#define**s you use and **#include**s.
4. **freeloader-plan.txt**, as described in 3.1.
5. **analysis.txt**, as described in 3.2.
6. **analysis-discussion.txt**, as described in 3.2.
7. **phase-xx-tcpdump.txt**, as described in 3.3.
8. **phase-xx-annotated.txt**, as described in 3.3.
9. A "README" file. This file describes your project and must include the following sections:

   Re-used Code : Did you use code from another source in your project? If not, say so. If you did, say what functions you borrowed, and where they came from. (Also comment this in the source code.) If you use the class timer code, you must say this here and describe any changes you made to it. If you used the TA code for Project A, please note that here.

   Idiosyncrasies : Is there anything in your code that does not work? Is there a phase you didn't implement or test?

Take time to write a good "README" file. It should not be just a few sentences. You need to take some time to describe what you did and especially anything you didn't do. Expect the grader to take off more points for things they have to figure out are broken (rather than learning about the problems through your "README" file).

5 Submission

On nunki, create a directory of all the files you want to turn in and call this directory "projb-xxx", where "xxx" is your username on nunki.

TEST your code before turning it in. Sanity check what you're turning in. Run a few tests to make sure you're turning in your most up-to-date code.

Do not turn in any binaries. Remove all output text files your project produces (xx-file.dat,xx.config,xx.out etc.).

Then create a tar file of this directory by running the following command from the directory containing your "projb-xxx" directory:

```
% tar cvf projb.tar projb-xxx
```

If you are turning in your project on or before the deadline, turn in your tar file by running the following on nunki:

```
% submit -user csci551 tag projb projb.tar
```

If you are turning in your project late, but would like to use your slip-day please email the TA that you will be using your slip day and turn in your tar file by running the following on nunki:

```
% submit -user csci551 tag projbSlip projb.tar
```

If you are turning in your project late and will not be using your slip-day turn in your tar file by running the following on nunki:

```
% submit -user csci551 tag projbLate projb.tar
```

6 Cautionary Words

In view of what is a recurring complaint near the end of the project, we want to make it clear that the target platform on which the project is suppose to run is SunOS. Although students are encourage to develop their programs on their personal machines, the final project must run on nunki.usc.edu under SunOS. If you choose to do initial development on other machines, make sure you include only the libraries in your code that are available on nunki.

All students are expected to write ALL their code on their own. Copying any code from friends is plagiarism and any copying of code will **result in an F** for the entire course. Any libraries or other code that you did not write must be listed in your "README" file. All programs will be compared using automated tools and by the grader to detect any similarities between turned in code from students this year as well as code turned in in previous years. *Any demonstration of code copying will result in an F for the entire course.*

IF YOU HAVE ANY QUESTIONS ABOUT WHAT IS OR ISN'T ALLOWED, TALK TO THE TA OR PROFESSOR. "I didn't know" is not an excuse.

You should expect to spend at least 5-15 hours or more on this assignment. Please plan accordingly. If you leave all the work until the week before it is due, you are unlikely to have a successful outcome.