CSci551 Fall 2014 Project B

Assigned: 2014-10-20. Project B Due: 6pm, Fri. 2014-11-14.

You are welcome to discuss your project with other students at the conceptual level, and to use external libraries *after getting permission*. Any cases of plagiarism will result in an F for the entire course. **If you have any questions about policies about academic integrity, please talk to the professor.**

Changes: 2014-09-21: none yet.

2014-10-23: added instruction on adding more network interfaces in Virtual Box. This change is *required* to test larger networks. Also clarified number of circuits needed (just one for Project B).

2014-10-26: clarified handling of router selection in stage 4. (The policy is the same as before, but handling pings to routers is hopefully clearer now.)

1 Overview

There are two *paths* for projects in CSci551 this year. The *Practical Path* project involves implementing a network-centric system. Everyone following this path solves the same problem, but of course in slightly different ways.

In the *Research Path*, students do new research working under the guidance of a mentor (usually a senior PhD student).

PhD students are required to do Research Path projects. Other students may choose either path.

In both cases the project is divided into three parts, Projects A, B and C. Each builds on the prior.

1.1 The Practical Path Project

The purpose of the CSci551 project is to get some hands on experience building a substantial distributed application running on a multi-process computing infrastructure, then to use it to study networking. It also has a secondary purpose of implementing a program using network programming (sockets) and processes (fork, in stage 1) and Unix development tools (make).

You may reuse algorithms from textbooks. You may possibly reuse functions from libraries, but if you're using anything other than the C or C++ standard library (libc, STL), or libraries mentioned on the TA's web page, you *must* check with the TA and the professor and identify it in your write-up. You need to check allowed libraries in Project Information on the class moodle. Otherwise, each student is expected to write *all* of his or her code independently. All programs will be subject to automated checking for similarity.

Please note: you should expect to spend a significant amount of time on the class projects this semester, probably at least 20 hours or more when they're all put together (or that much per project if you're new to C or C++ and network programming). Please plan accordingly.

If you leave all the work until the weekend before it is due, you are unlikely to have a successful outcome. That said, Project A is a "warm-up" project. Projects B and C will be much larger.

For the course, you will do three separate but related projects. First (Project A), you need to demonstrate that you can read the config file, do basic process control, and submit a complete assignment. (Project A doesn't really evaluate anything advanced, but it should confirm that everyone is on the same page and is comfortable with our software environment.) Project A has an early due date. Project A should be relatively short for students used to working on Linux; if not, it should help get you up to speed.

In Project B you will use this facility to implement *Minitor*, a simplified version of a TOR-like Onion Router as described in the paper by Dingledine et al. (see [Dingledine04a] in the class syllabus). It will probably be due just after the midterm. Project B will be *much* larger than Project A; you should plan your time accordingly. Project B will be assigned later and may overlap partially with Project A.

Project C will be assigned later. It will be smaller than Project B but bigger than Project A. It will build on Project B. We will offer a the TA's implementation of Project B for students who wish to use it instead of their own Project B implementation, but we do not promise to help you understand it. Project C will involve extending your Project B Minitor implementation with additional features, and perhaps measuring it or trying to de-anonymize it. It will probably be due the last week of class.

1.2 Research Path Projects

This year we are have an alternate *research path* for CSci551 projects. Students will conduct original research on topics related to the course material, helping to further our understanding or develop new approaches for open problems–at least, as much as one can within a semester! For these projects, you can use the language and libraries of your choice, subject to approval from your mentor (more on mentors below). As usual, you must properly credit and cite any code, text, or approaches that you borrow from others. Using and extending the research of others is a crucial part of research, but requires proper attribution.

All PhD students are *required* to do research path projects. MS students have the *option* of doing it, although MS students can cancel research path and pick up the practical project as an alternative after discussing this choice with the professor.

We will have a poster session at the end of the semester, where students can present their research path results. We encourage all students–not just research path students–to attend.

Research Projects in the Friday section are *individual* projects (not groups), but there may be some linked projects where two people work on different parts of the same problem. (Linked projects will share ideas and possibly some data or code or results, but each student will have a specific part they are responsible for and each student must do their own writeup.)

Research Projects in the Tuesday/Thursday section will allow two-person groups projects or individual projects.

2 Practical Project A: Getting Started

(We don't reproduce Project A here, but you may want to refer to it for background. You are required to use the same VM development environment from Project A in Projects B and C, and you should assume similar infrastructure, with a proxy and routers and log files.)

3 Research Project A: Getting Started

(See the original project A assignment for details about Research Project A.)

4 Practical Project B: Minitor

4.1 Stage 3: Pinging the Real World

In this stage you will connect your Minitor router to the outside world.

To send traffic through Minitor, there are two steps. First, you must tell your VM to route traffic through it. Second, you must tell your router to send traffic out.

4.1.1 Setting up traffic to your proxy and out of your VM

To send traffic through the VM, we will establish a manual route with the following commands:

Like proja, you first need to create a tunnel interface and configure it with the following commands: (If you already have tun1 up on your VM, you don't need to do it again.)

```
sudo ip tuntap add dev tun1 mode tun
sudo ifconfig tun1 10.5.51.2/24 up
```

Then we need to redirect some traffic to our tunnel interface with the following commands:

```
sudo ip rule add from $IP_OF_ETH0 table 9 priority 8
sudo ip route add table 9 to 18/8 dev tun1
sudo ip route add table 9 to 128/8 dev tun1
```

In addition, if you want to handle TCP, we have to get the kernel out of the way. By default, Linux will send RST packets when it sees your TCP packets. We don't want these RST packets to interfere, so we'll discard them.

```
sudo iptables -A OUTPUT -p tcp --tcp-flags RST RST -j DROP
```

Newer versions of Fedora use new naming conventions for network interfaces where they get strange names like `p2p1`. These names are supposed to be more stable than the old names (like `eth0`), but unfortunately, in VMs, the new names change whenever the host OS changes. (So for *class*, they are not stable at all!) To make the project work consistently across students and for grading, all students *must* use old-style interface names.

The easy way to get old-style interface names it to get the new VM image the TA posted on 2014-10-16. You can download them from the project information page on moodle.

Alternatively, if you want to update your VM, you may read the detailed instructions at `http://esuareznotes.wordpress.com/2014/07/11/`. The important parts are to edit `/etc/default/grub`, changing the line starting with `GRUB_CMDLINE_LINUX`, to add "`net.ifnames=0 biosdevname=0`" to the end of that line. Then create a new kernel boot with the command `sudo grub2-mkconfig -o /boot/grub2/grub.cfg; reboot`. (This can be a bit tricky, so we recommend you use the updated VM where these steps have already been done.)

We will test projects *only* using the updated VMs with old-style interface names (like `eth0`).

You need to find `$IP_OF_ETH0` from your VM. To do so, run `ip link show`; both the IP address and the devices are sometimes different in different VM configurations. The device is probably called `eth0`, and you can probably find it via `ip link show|grep UP|grep -v NO-CARRIER`. These commands basically say that packets coming from applications running on the VM guest OS (from `$IP_OF_ETH0`), sent to MIT (18/8 and 128.30/16), will be redirected to the tunnel interface. After these commands, `ping -I $IP_OF_ETH0 www.csail.mit.edu` will go into your proxy, although it will probably not come out yet. The `-I $IP_OF_ETH0` option forces ping to use `$IP_OF_ETH0` as the source IP address in the ICMP echo request packet; if you don't use this option in some cases the source IP can be different and so the ping will fail to be redirected.

4.1.2 Creating user-controlled routers that can talk to the Internet

A second step is required to allow traffic to go out of your routers to go out of the VM guest OS into the Internet. First, we need to create additional virtual network interfaces in your virtual machine, then we need to have your routers use them, and finally we need to allow their traffic to go out and replies to come back in.

To create additional network interfaces, edit the VMWare Player configuration.

On Virtual Box on Windows/Linux: open Virtual Box, right click on the virtual machine, then choose "setting", then click "Network", then click "Adapter 2", then check box "Enable Network Adaptor", then in attach to: choose "NAT", then click OK.

(Manditory addition 2014-10-23:) VirtualBox requires that we request 8 network interfaces. Using these interfaces is *required*. Our test cases will not require more than 8 interfaces in total.

(Paragraph addition 2014-10-23:) Default Virtual Box GUI only allows you to configure up to four network interfaces. To add more network interfaces, you *must* use command line to add up to eight interfaces:

```
VBoxManage modifyvm Fedora-20-LXDE-32-bit --nic8 nat
```

(`--nic8` says 8th interfaces, and Fedora-20-LXDE-32-bit is the name of the VM that we use). You may use `--nic5,--nic6,--nic7` to get other three interfaces. Detailed usage of `VBoxManage modifyvm` is at `http://www.virtualbox.org/manual/ch08.html#vboxmanage-modifyvm`

(Paragraph addition 2014-10-23:) To make this change in windows, you need to find where your Virtual Box is installed, and specify the absolute path, for example:

`‘‘D:\Program Files\Oracle\VirtualBox\VBoxManage.exe’’ modifyvm Fedora-20-LXDE-32-bit --nic8 nat`

(We assume instructions are similar on the Mac, but have not yet verified.)

On VMWare Player on Windows: choose the VM, click “Player”, then choose “Manager”, then “Virtual Machine Settings”, then click “Add”, choose “Network Adapter”, click “Next”. In “Network connection” tab, make sure you choose “NAT” and in “Device status”, check “Connect at power on”. Then click “Finish”.

After you finish these steps, when you type ifconfig, you should be able to see two network interfaces (eth0, eth1). We changed this to use static names (eth1, etc.) to simplify things.)

To manage routing, we want the interfaces controlled by your application to be separate from those controlled by the VM guest OS. We will therefore put eth1 on a different /24 network. Let’s use 192.168.201/24 for eth1, assuming this is different from what you’re already using for your VM guest OS.

Then you need to configure the second ethernet with the following command:

```
sudo ifconfig eth1 192.168.201.2/24 up
```

This command both brings up eth1 with IP address 192.168.201.2, and it assigns 192.168.201/24 as a subnet that goes to eth1.

Finally, your router processes will need to talk through this interface out to the Internet through your host OS. They will talk to the world by reading and writing to raw IP sockets, bound explicitly to this network interface.

You will need to open a raw IP socket with the socket API: `socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)`. You may want to review the raw(7) man page (run `man 7 raw`, so you get manual page about raw sockets in the Linux manual section 7, not the one in section 8 about raw character devices). You will also need to bind your socket to the relevant network devices with `bind(2)` using the IP address of eth1. Finally, when you send out the raw socket, you need to use the `sendmsg(2)` system call (sendto(2) and write(2) don’t work; with sendmsg the kernel fills in the IP header for you).

In future stages, you will run multiple routers and you will need to make sure each gets a different, unique IP address and separate external ethN interface. However, for this stage we have only one router and eth1. You may want to have the proxy assign devices and IPs to the routers before it forks them.

4.1.3 Proxying

With all of this setup out of the way, you need to proxy. Your proxy should forward the ping packets it gets to the router. Your router should reply to any ping packets that are addressed to itself (just like in stage 2). But for packets that go elsewhere, it should rewrite them to its own IP address and then send them out a raw socket to the real world.

The router should also read from the raw socket for its network device, checking the destination address of each packet that arrives, discarding it if addressed elsewhere, otherwise picking it up and handling it.

Your router must then multiplex IO, reading UDP packets from the proxy, and reading from the raw interface. As with stage 2, a simple select loop should be sufficient for this multiplexing, if done properly. (You should not need to use threads.)

4.1.4 Samples

Sample input: Here is a sample configuration input for stage 3. (Yes, it's still boring.)

Clarification 2014-10-20: Project configuration files may have comments on any line in the file, and fields in non-comment lines may be separated by any amount of whitespace (tabs or spaces).

```
#This is a comment line, should be ignored
stage 3
num_routers 1
```

Sample output: After running your program and the above route commands, `ping -I $IP_OF_ETH0 -c 4 www.csail.mit.edu`. Your program should produce the following log files.

In `stage3.proxy.out`:

```
proxy port: 40064
router: 1, pid: 1674, port: 38305
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from port: 38305, src: 128.30.2.155, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from port: 38305, src: 128.30.2.155, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from port: 38305, src: 128.30.2.155, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from port: 38305, src: 128.30.2.155, dst: 192.168.97.131, type: 0
```

In `stage3.router1.out`:

```
router: 1, pid: 1674, port: 38305
ICMP from port: 40064, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from raw sock, src: 128.30.2.155, dst: 192.168.201.2, type: 0
ICMP from port: 40064, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from raw sock, src: 128.30.2.155, dst: 192.168.201.2, type: 0
ICMP from port: 40064, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from raw sock, src: 128.30.2.155, dst: 192.168.201.2, type: 0
ICMP from port: 40064, src: 192.168.97.131, dst: 128.30.2.155, type: 8
ICMP from raw sock, src: 128.30.2.155, dst: 192.168.201.2, type: 0
```

4.1.5 Writeup

Writeup: Please prepare a README for stage 3, called `README.stage3.txt`.

a) Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.) If you use the class timer code, you must say so here and describe any changes you had to make to it.

b) Complete: Did you complete this stage? If not, what works and what doesn't?

c) Addressing on the way out of your router: Why is it important to rewrite the source address in your proxy? (What would happen if you left it as the original address of the VM?)

c) Addressing on the way in to the VM: Why is it important that there is a separate network device for your router in the VM?

d) Addressing from the VM to the host: Both your routers and the VM use private addresses. What does the host OS (on the computer that is supporting the VM) do to outgoing packets?

4.2 Stage 4: Multiple Routers

In this stage, we add multiple routers and let your proxy pick between them.

The `num_routers` option can now take on values between 1 and 6. Your proxy should create as many routers as are called for, and keep track of them.

You will need to first create as many interfaces (eth1, eth2, eth3, etc.) in the VM as the number of routers you will have. Then you should bind each router to one interface. We recommend your proxy pass the IP address and interface to the router. We describe how to create interfaces, bind them, and proxy-router iteration in stage 3. After you create those interfaces, you need to assign IP addresses to them. Although you can assign any private addresses to those interfaces, we explicitly designate IP addresses for them here (this will help us evaluate your code). So we require you to use the following commands to configure them:

```
sudo ifconfig eth1 192.168.201.2/24 up
sudo ifconfig eth2 192.168.202.2/24 up
sudo ifconfig eth3 192.168.203.2/24 up
sudo ifconfig eth4 192.168.204.2/24 up
sudo ifconfig eth5 192.168.205.2/24 up
sudo ifconfig eth6 192.168.206.2/24 up
```

There is no specific order for binding interfaces to routers, but we suggest you follow the same order as the router index. (e.g. bind eth1 to router1, eth2 to router2, ...)

Each router should open its own log file (`stage4.routerN.out`, where N is its ID between 1 and the value given).

(Clarification 2014-10-26, in italics.) To provide consistent ~~paths~~ *first hops*, the proxy should pick a specific router for each destination IP, then stick with it. *For destinations that are routers, send them directly, otherwise do as follows:* We require that you hash the destination IP address across the routers that are provided. To do so, treat the destination IP address as a unsigned 32-bit number and take it MOD the number of routers, then add one. (So with 4 routers, IP address 1.0.0.5 will map to 2, because 16,777,221 MOD 4 is 1.) *(Note that stage 5 changes this policy and makes the whole path random.)*

We will send pings *through* your routers, as per the prior stages, or pings *to* specific routers, as per stage 2. You should forward traffic if necessary, and reply to traffic directed to the routers.

4.2.1 Samples

Sample input: Here is a sample configuration input for stage 4. Finally less boring, with numbers greater than 1.

```
#This is a comment line, should be ignored
stage 4
num_routers 4
```

Sample output: After running your program and the above route commands, followed by `ping -c 1 -I $IP_OF_ETH0 128.30.2.102; ping -c 2 -I $IP_OF_ETH0 128.30.2.103; ping -c 3 -I $IP_OF_ETH0 128.30.2.104`.

sample output for `stage4.proxy.out` (the order that routers show up might be different):

```
proxy port: 33398
router: 3, pid: 1861, port: 49186
router: 4, pid: 1862, port: 41568
router: 2, pid: 1860, port: 34541
router: 1, pid: 1859, port: 46578
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.102, type: 8
ICMP from port: 49186, src: 128.30.2.102, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.103, type: 8
ICMP from port: 41568, src: 128.30.2.103, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.103, type: 8
ICMP from port: 41568, src: 128.30.2.103, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.104, type: 8
ICMP from port: 46578, src: 128.30.2.104, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.104, type: 8
ICMP from port: 46578, src: 128.30.2.104, dst: 192.168.97.131, type: 0
ICMP from tunnel, src: 192.168.97.131, dst: 128.30.2.104, type: 8
ICMP from port: 46578, src: 128.30.2.104, dst: 192.168.97.131, type: 0
```

In `stage4.router3.out`, router 3:

```
router: 3, pid: 1861, port: 49186
ICMP from port: 33398, src: 192.168.97.131, dst: 128.30.2.102, type: 8
ICMP from raw sock, src: 128.30.2.102, dst: 192.168.203.2, type: 0
```

In `stage4.router4.out`, router 4:

```
router: 4, pid: 1862, port: 41568
ICMP from port: 33398, src: 192.168.97.131, dst: 128.30.2.103, type: 8
ICMP from raw sock, src: 128.30.2.103, dst: 192.168.204.2, type: 0
ICMP from port: 33398, src: 192.168.97.131, dst: 128.30.2.103, type: 8
ICMP from raw sock, src: 128.30.2.103, dst: 192.168.204.2, type: 0
```

In `stage4.router1.out`, router 1:

```
router: 1, pid: 1859, port: 46578
ICMP from port: 33398, src: 192.168.97.131, dst: 128.30.2.104, type: 8
ICMP from raw sock, src: 128.30.2.104, dst: 192.168.201.2, type: 0
ICMP from port: 33398, src: 192.168.97.131, dst: 128.30.2.104, type: 8
ICMP from raw sock, src: 128.30.2.104, dst: 192.168.201.2, type: 0
ICMP from port: 33398, src: 192.168.97.131, dst: 128.30.2.104, type: 8
ICMP from raw sock, src: 128.30.2.104, dst: 192.168.201.2, type: 0
```

In `stage4.router2.out`, router 2:

```
router: 2, pid: 1860, port: 34541
```

Your ports will of course be different from the sample output.

4.2.2 Writeup

Writeup: Please prepare a README for stage 4, called `README.stage4.txt`.

a) Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.) If you use the class timer code, you must say so here and describe any changes you had to make to it.

b) Complete: Did you complete this stage? If not, what works and what doesn't?

b) Router Selection: For the current approach to mapping traffic to routers, i) Why is that approach load balanced (at least statistically) over many flows to many targets? ii) Are there degenerate cases where your load could become unbalanced? (Just yes or no.) iii) What is one such case?

4.3 Stage 5: Onion Routing Without Encryption

Now that we have multiple routers, it's time for onion routing.

Rather than pick *one* router through which to route, the proxy should pick H different ones, and follow something like the TOR scheme of building up paths.

We will add a new configuration option `minitor_hops H`, which gives us how many hops all circuits will use. We promise that H will always be strictly less than the number of routers in the system, and at least 1.

On first receipt of a flow, the proxy should pick H routers in an ordered list. As with stage 4, it should strive to spread the load at least statistically randomly for many different targets.

The proxy should set up the circuit of H hops, step-by-step analogous to TOR, then after it's done that it should encapsulate the packet and send it down the circuit.

Clarification 2014-10-23: for Project B, only one circuit is required. (We may relax this and require one circuit per flow in Project C, but for Project B our test cases will have only one circuit.)

In order to track each router's IP address, we require you to log the IP address for each router. Please add the IP information to both your proxy's log and router's log: E.g. the first few lines of your proxy's log (stage5.proxy.out) should look like this:

```
proxy port: 44535
router: 4, pid: 2251, port: 37549, IP: 192.168.204.2
router: 2, pid: 2249, port: 41481, IP: 192.168.202.2
router: 3, pid: 2250, port: 44297, IP: 192.168.203.2
router: 1, pid: 2248, port: 33180, IP: 192.168.201.2
```

And the first line of router 1's log (stage5.router1.out) should look like this:

```
router: 1, pid: 2248, port: 33180, IP: 192.168.201.2
```

4.3.1 Control messages and circuit ids

To do this extra work, you will need Minitor control messages. All communication between your processes (the proxy and routers) is via UDP, and since Stage 2, we have embedded IP packets in the UDP packets. We will therefore make Minitor control messages packets with IP headers and distinguish them with an experimental IP protocol number: 253. The contents of the control message are then in the IP payload. (Although these are IP packets, they will be sent between proxy and routers over their UDP ports all inside the VM guest OS.) We don't actually use the contents of this IP header, but you must set both the source and destination IP addresses in it the loopback address, 127.0.0.1, fill in the protocol number (253), and leave everything else zero up to the payload.

The first byte of payload (after the IP header) will be the message type; and the second will be a two-byte, network-byte order circuit id.

Thus, the UDP packets you send around will have the following structure:

1. Real IP header: you won't see this, but it goes around your UDP messages.
2. UDP header: again, you won't see this, but it's there because you're sending UDP messages.
3. Minitor control message IP header: With source and destination IPs of 127.0.0.1, protocol number 253, and all other fields zero.
4. Minitor control message contents: As specified below. One byte of type, plus two bytes of circuit ID, plus different contents depending on type. For Minitor relay messages, included in here will be the encapsulated IP packet (with its own IP header and an ICMP message). For other Minitor messages there will be other contents.

Ordinarily TOR would assign circuit IDs randomly, but we will assign them in a special format to facilitate debugging. (Warning: this reduces the security.) Circuit IDs should be $i * 256 + s$, where i is the router id (1 to N), or 0 if it's the proxy, and s is a sequential number, starting at 1, of the number of circuits the assigner has created. Circuit IDs should always be printed in hexadecimal, with an 0x before them.

4.3.2 Creating Circuits

The proxy will build the circuit, by picking the hops, then building it, hop-by-hop. Your proxy should log each hop of the whole circuit: *hop: N, router: M.*

Each step, it adds one hop to the circuit, then it extends the circuit through the (partial) circuit to the next hop.

While our circuit setup is inspired by TOR, it is much simpler. Real Tor has many different kinds of control messages, include relay begin, teardown, extend, and reply messages sort of like what we do above, along with others. We instead get by with just two messages: circuit-extend and the reply, circuit-extend-done.

To build a circuit, the proxy will send *circuit-extend* control message with type 0x52, either to the first hop of a new circuit, or down the existing, partial circuit to extend it one more hop. This control message has 5 bytes of payload after the IP header: one byte for the message type, two bytes (network byte order) for the incoming circuit ID (0xIDi), and two for name (UDP port, in network byte order) of the next hop in the circuit (NEXT-NAME). NEXT-NAME should be 0xffff if this is the last hop of the circuit. (In a real system, it would need to include the IP address as well, but we can simplify things because we're on one VM.)

When a router gets circuit-extend for an ID it has not seen before, it should remember that it now knows about a new circuit. For anonymity, it then needs to generate its own, new, outgoing circuit ID, and log that as *new extend circuit: incoming: 0xIDi, outgoing: 0xIDo at NEXT-NAME*, where IDi and IDo are the incoming and outgoing IDs, and NEXT-NAME is next hop. (Note that it does not have to contact the next hop to "create" its part of the circuit!) Each router will need to keep the incoming and outgoing circuit IDs, and the NEXT-NAME (UDP port) of the previous and next hops.

Finally the router will reply back to the requester with a *circuit-extend-done* control message of type 0x53, with a two-byte, network-byte order circuit ID. The circuit-id that is returned should be the incoming id that was given to it, so the requester knows. (It should *not* return the next outgoing circuit id.)

4.3.3 Extending Circuits

The proxy wants to build multi-hop circuits, yet it must do so carefully and securely to preserve anonymity.

To extend a one-hop circuit, the proxy sends, to the first hop of an existing circuit, a circuit-extend message with an existing circuit ID and the NEXT-NAME of the new next hop.

Extending a circuit multiple hops works the same way: the proxy sends a circuit-extend message with the existing circuit-ID and the new next hop to the first hop of the circuit. This message passes through the circuit until it reaches the current end, which then uses it to add one more hop.

On the router side, when receiving a circuit-extend message, if the router already knows about that circuit ID (in other words, it remembers it and knows that there is an existing next hop), then that router will relay the message down the circuit, to the next-hop router it saved before. When doing so, it must map the incoming circuit ID (0xIDi) to the outgoing

circuit ID (0xIDo), and pass on NEXT-NAME for the new next hop. It should log *forwarding extend circuit: incoming: 0xIDi, outgoing: 0xIDo at NEXT-NAME.*

Eventually the circuit-extend message arrives at a router that has not seen this circuit ID before. This router will be one hop beyond the previous circuit's end. This router will process the message to learn about the next hop as described above (Section **??**). This message will travel back through the circuit to the proxy.

Routers must also forward circuit-extend-done messages, mapping the circuit ID and following the chain back to the proxy.

When the proxy receives the circuit-extend-done message, it should log *incoming extend-done circuit, incoming: 0xIDi from port: PORT*

4.3.4 Circuit Creation Example

Here's an example of how circuit creation works. Assume we have an Onion Proxy OP and three routers, OR1, OR2, and OR3.

The OP sends circuit-extend(ID: 0x01, next-hop: 10002) to OR1, where 10002 is the UDP port of OR2. It records a new circuit and assigns a new outgoing ID: (iID: 0x01, oID: 0x101), then replies circuit-extend-done(ID: 0x01) to the proxy.

The OP then adds one hop, sending circuit-extend(ID: 0x01, next-hop: 10003) to OR1. OR1 remaps that and sends circuit-extend(ID: 0x101, next-hop: 10003) to OR2. OR2 now records a new circuit and assigns it a new outgoing ID: (iID: 0x101, oID: 0x201). OR2 then replies to OR1: circuit-extend-done(ID: 0x101). OR1 then maps and forwards circuit-extend-done(ID: 0x01) to the proxy (using its saved UDP port).

The OP will finally build the third hop by sending circuit-extend(ID: 0x01, next-hop: 0xffff) to OR1. This message will eventually arrive at OR3.

4.3.5 Relay Data

Now that we've built circuits, we can send data over them.

To send a packet over a circuit, the proxy will embed the IP packet in a *relay data* control message. The relay-data should have type 0x51, with two bytes for the circuit ID, and then append the packet contents (the IP header and payload of the actual data). Routers should never get packets with an unknown circuit ID, but you should log this error case as *unknown incoming circuit: 0xIDi, src: S, dst: D* where S and D are the ~~payload~~ encapsulated packet's source and destination IP addresses.

If the circuit exists and has a non-exit next hop, the router should change the source IP [of the encapsulated packet] to its own, map the circuit ID from incoming to outgoing, and forward the packet to its neighbor over UDP with another relay-data control message. It should then log *relay packet, circuit incoming: 0xIDi, outgoing: 0xIDo, incoming src: Si, outgoing src: So, dst: D.* Si, So, and D are all for the encapsulated packet.

When the packet arrives at the end of the circuit, as indicated by a NEXT-NAME of 0xffff, the router should send the packet out to the Internet. To do so, it should decapsulating the packet, change the source IP address [of the encapsulated packet] to that of the router. Then write the packet out its raw ICMP socket. It should then log *outgoing packet, circuit incoming: 0xIDi, incoming src: Si, outgoing src: So, dst: D.* Incoming source Si is the

source IP address in the previously encapsulated packet. Outgoing source So should be the IP address of the router; this is the only internal IP that should appear in the outgoing packet. Destination is the IP address on the public Internet.

4.3.6 Relay Returning Data

Of course, one needs to get data back from the last hop to the proxy.

Each router will listen for data from the network on its raw device, as in prior stages. When this public-facing router gets an external packet addressed to it, it needs to return that to the proxy.

To return a packet to the proxy, the router will use a *relay-return-data* control message. The relay-return-data message will have type 0x54, with two bytes for the circuit ID and the appended packet contents, just as the relay-data message did.

In a real system, the router must determine which circuit a return packet belongs to. For now, we have only one circuit, so the router knows that incoming traffic belongs to that circuit.

Upon receiving the packet from the raw device, the router should log *incoming packet, src: S, dst: D, outgoing circuit: 0xIDo* where incoming source is the source IP in incoming packet (that is, the public IP address out the Internet), destination D is the IP address of the router's raw interface, and outgoing circuit is the Minitor circuit the router will return the packet on.

The public-facing and middle-hop routers will forward relay-return packets. Each hop, they will need to map the circuit ID backwards, and map the destination IP to the next hop in the circuit, working back to the proxy. Eventually this packet will arrive back at the proxy. Middle routers should log: *relay reply packet, circuit incoming: 0xIDi, outgoing: 0xIDo, src: S, incoming dst: Di, outgoing dst: Do.*

When the packet arrives at the proxy, the proxy will decapsulate the packet and send it back to the tunnel interface and the user's application. The router just previous to the proxy should have replaced the source IP address with that of the VM guest's ethernet interface. The proxy will then write the packet out the tunnel interface. It should then log *incoming packet, circuit incoming: 0xID, src: S, dst: D*, where S should be from the public Internet, and D the IP address of the VM guest's ethernet interface.

4.3.7 Additional logging

To help verify what is going on, the proxy and all routers also must log every packet they receive, with the source port, printing all payloads in hexadecimal, from the start of the Minitor control packet to the end-of-packet. Two exceptions: For the proxy, no need to log the packet from the tunnel interface. For the last hop router, no need to log the packet from the raw socket. Print the control packet byte-by-byte, as the raw bytestream that is sent "over the wire". The output format should be: *pkt from port: S, length: L, contents: 0xabcd.* For example, the first packet from section **??** is from the OP to OR1, circuit-extend(ID: 0x101, next-hop: 10002). The output will be `pkt from port: 10000, length: 5, contents: 0x5201012712`, since it has type 0x52, id 0x101, and port 10002 (== 0x2712 in hexadecimal).

4.3.8 Running things

Sample input 1: Let's do encapsulation:

```
#This is a comment line, should be ignored
stage 5
num_routers 4
minitor_hops 1
```

Sample output 1: After running your program and the above route commands, followed by `ping -I $IP_OF_ETH0 -c 1 128.30.2.155`, you should see packets going one hop and then out, just like in prior stages, but now they'll be encapsulated and you will see circuits being automatically created.

Please find the sample output under the Project Information page on moodle (stage5.sampleout1.tar.gz). (`https://moodle.ant.isi.edu/pluginfile.php/3972/mod_folder/content/0/stage5.sampleout1.tar.gz?forcedownload=1`)

Sample input 2: Let's do double hopping.

```
#This is a comment line, should be ignored
stage 5
num_routers 4
minitor_hops 2
```

Sample output 1: After running your program and the above route commands, followed by `ping -c 1 -I $IP_OF_ETH0 128.30.2.155`, you should see your router extend a circuit, which will both create it at the first hop and identify the next hop. You should then see the packet go through the hop and out the second hop to the network.

Please find the sample output in the Project Information page on moodle (stage5.sampleout2.tar.gz). (`https://moodle.ant.isi.edu/pluginfile.php/3972/mod_folder/content/0/stage5.sampleout2.tar.gz?forcedownload=1`)

4.3.9 Writeup

Please prepare a README for stage 5, called `README.stage5.txt`.

a) Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.) If you use the class timer code, you must say so here and describe any changes you had to make to it.

b) Complete: Did you complete this stage? If not, what works and what doesn't?

c) Why does Minitor extend circuits through partially built circuits, when the OP can contact the current hop at the end directly?

4.4 Stage 6: Onion Routing With Encryption

Finally, we add encryption. Unlike TOR, we will use simple, symmetric encryption without public-key node identities, and without TLS between hops. It should still be fun.

The idea to TOR-like encryption is that only the proxy and the last hop should know the key.

We will use AES (Advanced Encryption Standard) encryption, and the openssl library (already included in your Fedora 17 VM). Unfortunately, while openssl is a mature library, it has no manual pages for AES (yes, 10 years after it was established as a standard). (Wikipedia has a good (technical) description if you are interested, although you are not required to know the details.) We provide sample code on how to use openssl's AES routines on the Project Information website, with routines `class_AES_set_encrypt_key`, `class_AES_set_decrypt_key`, `class_AES_encrypt`, `class_AES_decrypt` that you must use. (If you can't use this library, please talk to us about why.)

4.4.1 Creating Circuits with Encryption

The proxy will build an encrypted circuit, by picking the hops, then building it, hop-by-hop, just like it did with unencrypted circuits.

However, the critical difference is that the *private portion* of *every* message sent down the circuit must be onion-encrypted. That is, the private parts of message m from the OP through hops B and C to D must be encrypted with each of their keys, so that the OP will take m and send $\{\{\{m\}_{KD}\}_{KC}\}_{KB}$, so that each hop can decrypt it and only the OP and D know the contents. Exactly which parts need to be private depends on the message type.

In real TOR, the first step of each hop of circuit extension is for the OP and the new hop H to establish a new session key unique to OP and H. In real TOR, this session key would be generated using a Diffie-Hellman key exchange. We're going to do something much simpler (and *not at all secure!*): the proxy should generate a random 128-bit AES key, then exclusive-OR it with the 16 copies of the router number of the last hop. (So if we're extending to router 2, XOR the key with 0x02020202020202020202020202020202.)

So the first step of adding a hop for the OP is therefore to send the new key to the new hop through the circuit (*not* directly). It does so with a *fake-diffie-hellman* message of type 0x65. This message will consist of one byte message type, two bytes (network byte order) for the incoming circuit ID (0xIDi), and finally the new session key, a 16 byte (128 bit) AES key. This key (but not the rest of the message) is *private* and must be onion encrypted if it goes over multiple hops. (Because we are establishing the key, it is not encrypted between the last hop and the penultimate hop. This step exposes the key to the penultimate hop and is therefore *insecure*; however, if it were a real Diffie-Helman exchange, it would not matter because of the properties of D-H to set up a new shared secret.) When the proxy generates a key, it should log *new-fake-diffie-hellman, router index: i, circuit outgoing: 0xIDo, key: 0xabcd...* where i is the router index number, and 0xabcd is the entire 128-bit key printed in hexadecimal.

When a router gets a fake-diffie-hellman message for a new circuit, it should remember the incoming circuit ID and the new session key. (It will later need to associate that key with the circuit.) It won't yet know where the next hop of the circuit is going, that doesn't happen

yet. It should then log *fake-diffie-hellman, new circuit incoming: 0xIDi, key: 0xabcd...* where 0xabcd is the entire 128-bit key printed in hexadecimal.

When a router gets a fake-diffie-hellman message for an *existing* circuit, it should onion-decrypt the key with its session key for the circuit, map the incoming circuit ID to its outgoing counterpart, then forward the message. It should then log *fake-diffie-hellman, forwarding, circuit incoming: 0xIDi, circuit outgoing: 0xIDi, key: 0xabcd...* where 0xabcd is the entire 128-bit key printed in hexadecimal. (Note that all but the penultimate hop will show some encrypted version of the key.)

There is no explicit acknowledgement message for the key establishment message, although in the real world there would need to be.

After the key is established, the OP should send a *encrypted-circuit-extend* control message with type 0x62. This message will consist of one byte message type, two bytes (network byte order) for the incoming circuit ID (0xIDi), and then the private (encrpyted) version of the name of the next hop in the circuit (NEXT-NAME). This private portion must be onion encrypted to the current end of the circuit. (For example, if the circuit is OP to OR1, send n_{K01} where $K01$ is the session key between OP and OR1. If it is to OR2 through OR1, send $\{\{n\}_{K02}\}_{K01}$, etc.)

When the router gets an encrypted-circuit-extend message, it should onion decrypt the NEXT-NAME and use this to fill in the next hop of the circuit. It will also generate an outgoing circuit ID, as before.

Finally the router will reply back to the request, through the circuit, with an *encrypted-circuit-extend-done* control message of type 0x63, with a two-byte, network-byte-order circuit ID. The circuit-id that is returned should be the incoming id that was given to it, so the requester knows. Note that the circuit-id is *not* private (it should not be onion-encrypted), since it will be changed each hop.

4.4.2 Extending Circuits with Encryption

The proxy wants to build multi-hop encrypted circuits, and as before, it must do so carefully and securely to preserve anonymity.

Extending an encrypted circuit works just like establishing it, except all communication happens down the circuit (not directly with the next hop router, which would give away the OP's identity). And all private communication must be onion-encrypted.

The bookkeeping is just like extending non-encrypted circuits, in that the routers build a chain of circuit IDs, to keep the identity of the OP private in one direction. But they also must keep track of session keys between the OP and each hop, to keep the OP and destination's identity private in the other direction.

4.4.3 Encrypted Circuit Creation Example

Here's an example of how circuit creation works. Assume we have an Onion Proxy OP and just two routers this time, OR1 and OR2.

The OP sends fake-diffie-hellman(ID: 0x01, key=0x01ab) to OR1, where 0x01ab is the 128-bit key. OR1 records that there is a new circuit with no current next hop and remembers the session key: (iID: 0x01, oID: unknown, key: 0x01ab). There is no reply.

The OP then sends a encrypted-circuit-extend message(ID: 0x01, $\{10002\}_{0x01ab}$). OR1 receives this message, decrypts the NEXT-NAME with the session key, generates the outgoing ID, and remembers the next hop: (iID: 0x01, oID: 0x101, next-hop: 10002, key: 0x01ab). It then replies through the circuit with an encrypted-circuit-extend-done(ID: 0x01).

The OP then adds one hop. First it makes a new key with OR2 through the circuit. That is, it sends fake-diffie-hellman(ID: 0x01, key=$\{0x02ab\}_{0x01ab}$) to OR1. OR1 gets this message, sees it's an existing circuit, unwraps one layer of the encryption and maps the circuit id, then forwards it to OR2 as fake-diffie-hellman(ID: 0x101, key=0x02ab). (Please don't peek at the key, OR1 :-) OR2 now remembers this information, just as above when OR1 heard a new circuit.

The OP then sends a encrypted-circuit-extend message(ID: 0x01, $\{\{0xffff\}_{0x02ab}\}_{0x01ab}$) to OR1. OR1 unwraps and forwards this message to OR2 as it did with the F-D-H message. OR2 receives this message, decrypts the NEXT-NAME, realizes and remembers it's the last hop. It then replies through the circuit with an encrypted-circuit-extend-done(ID: 0x101), sending this to OR1 who maps this to circuit ID 0x01 and sends back to the OP.

4.4.4 Relay Data with Encryption

We can now send data through our encrypted circuits.

To send a packet over a encrypted circuit, the proxy will embed the IP packet in a *relay-encrypted-data* control message type 0x61. This message has two bytes for the circuit ID, and then append the onion-encrypted packet contents. (That is, the IP header and payload of the actual, relayed packet, just like in stage 5's relay data.) To hide the identity of the originator, the OP should *zero* the source IP. (It should therefore remember this source IP and associate it with the circuit ID, so it can eventually send reply packets.)

After zeroing the source IP, the OP must onion-encrypt the contents of relayed packet. The proxy will encrypt the relayed packet with *each* of the AES keys along the path, starting with the last hop and working backwards.

When a packet arrives on a circuit that has a non-exit next hop, the router should decrypt one layer of onion routing, map the circuit ID from incoming to outgoing, and forward the packet to its neighbor over UDP with another relay-encrypted-data control message. It should then log *relay encrypted packet, circuit incoming: 0xIDi, outgoing: 0xIDo*. Note that you can no longer log source and destination since they're onion-encrypted.

When the packet arrives at the end of the circuit, as indicated by a NEXT-NAME of 0xffff, the router should send the packet out to the Internet. To do so, it should decrypt and decapsulate the packet, change the source IP address of the encapsulated packet to that of the router, then write the packet out its raw ICMP socket. It should then log *outgoing packet, circuit incoming: 0xIDi, incoming src: Si, outgoing src: So, dst: D*. Incoming source Si is the source IP address in the previously encapsulated packet and should always be zero. Outgoing source So should be the IP address of the router; this is the only internal IP that should appear in the outgoing packet. Destination is the IP address on the public Internet.

4.4.5 Relay Returning Data with Encryption

As before, one needs to get data back from the last hop to the proxy.

Each router will listen for data from the network on its raw device, as in stage 5.

To return a packet to the proxy, the router will use a *relay-return-encrypted-data* control message. The relay-return-encrypted-data message will have type 0x64, with two bytes for the circuit ID and the appended packet contents. It should zero the destination IP address, just as the OP zeroed the source IP address.

As in stage 5, upon receiving the packet from the raw device, the router should log *incoming packet, src: S, dst: D, outgoing circuit: 0xIDo* where incoming source is the source IP in incoming packet (that is, the public IP address out the Internet), destination D is the IP address of the router's raw interface, and outgoing circuit is the Minitor circuit the router will return the packet on.

As in stage 5, the public-facing and middle-hop routers will forward relay-return packets. Each hop, they will need to map the circuit ID backwards. They will also do reverse-onion-encryption. That is, each hop they need to encrypt the packet with the session key. Eventually this packet will arrive back at the proxy. Middle routers should log: *relay reply encrypted packet, circuit incoming: 0xIDi, outgoing: 0xIDo.*

When the packet arrives at the proxy, the proxy will decapsulate the packet, decrypt each layer of onion routing, fill in the destination IP with what it saved for the circuit, and send it back to the tunnel interface and the user's application. The proxy will then write the packet out the tunnel interface. It should then log *incoming packet, circuit incoming: 0xID, src: S, dst: D*, where S should be from the public Internet, and D the IP address of the VM guest's ethernet interface.

4.4.6 Encrypted Data Example

Continuing our example from before, with an OP, OR1, and OR2.

The OP wants to send packet p to the Internet, so it zeros the source IP, encapsulates the packet and sends OP to OR1, relay-encrypted-data(0x01,$\{\{m\}_{0x02ab}\}_{0x01ab}$). OR1 does one layer of decryption and mapping and sends to OR2, relay-encrypted-data(0x101,$\{m\}_{0x02ab}$). OR2 does final decryption, fills in its source address, and sends m to the world.

For the reply, OR2 picks up r off the net, zeros the destination, and sends OR2 to OR1, relay-return-encrypted-data(0x101,$\{r\}_{0x02ab}$). OR1 adds another layer and sends relay-return-encrypted-data(0x01,$\{\{r\}_{0x02ab}\}_{0x01ab}$) to the proxy. OP can then extracts r, fills in the destination with the application and writes it back into the tunnel interface.

4.4.7 Additional logging

To help verify what is going on, the proxy and all routers also must log every packet they receive (the same as the additional logging in stage 5), with the source port, printing all payload in hexadecimal, from the start of the Minitor control packet to the end-of-packet.

Sample input 1:

```
#This is a comment line, should be ignored
stage 6
num_routers 4
minitor_hops 1
```

Sample output 1: After running your program followed by `ping -c 1 -I $IP_OF_ETH0 128.30.2.134`, you should be able to see reply. Sample outputs are provided in the Project Information page on moodle (stage6.sampleout1.tar.gz) (`https://moodle.ant.isi.edu/pluginfile.php/3972/mod_folder/content/0/stage6.sampleout1.tar.gz?forcedownload=1`)

Sample input 2:

```
#This is a comment line, should be ignored
stage 6
num_routers 4
minitor_hops 3
```

Sample output 2: After running your program followed by `ping -c 1 -I $IP_OF_ETH0 128.30.2.134`. you should be able to see reply. Sample outputs are provided in the Project Information page on moodle (stage6.sampleout3.tar.gz) (`https://moodle.ant.isi.edu/pluginfile.php/3972/mod_folder/content/0/stage6.sampleout3.tar.gz?forcedownload=1`)

4.4.8 Writeup

Please prepare a README for stage 6, called `README.stage6.txt`.

a) Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.) If you use the class timer code, you must say so here and describe any changes you had to make to it.

b) Complete: Did you complete this stage? If not, what works and what doesn't?

5 Research Project B

For Research Project B, you will be expected to work on your research project regularly throughout the period of the assignment. In particular, you will be responsible for the following deliverables, as described in your project proposal submitted as part of Research Project A.

1. Regular meetings with your mentor. We expect these will be about weekly, so you should have about 4 of them (and at least 3) over this period. (You may have more if you want.)

2. Written weekly notes about your progress, sent to your mentor the day before your weekly meeting, commented on by the mentor at/after the meeting, and updated by you after the meeting. Of course, this requirement means that you are required to meet with your mentor roughly every week.

3. Code, data, and analysis results, provided when you submit this portion of the project. Additionally, you should include a README.txt file describing what you submitted.

4. A Project B report, probably 2–6 pages summarizing where you are as of the Project B deadline and what you plan for Research Project C (tentatively: noon Wed. 2013-10-29). The document should be a PDF and needs to include the following numbered sections:

Section 1 an introduction, giving an overview of what you're doing, and a statement about (a) why is it interesting research (to the field)? (b) why is it interesting for you (what will you learn)? (c) what about what you're doing is new.

Section 2 what is relationship to other work. List related work, with citations in your text, and for each piece of related work identify how it is alike or different from your work.

Section 3 a discussion of your work (possibly including subsections on goal, methodology/approach, data collection, and results)

Section 4 a description of what the next steps are for Research Project C. (You must include a description of what the next steps are even if you do not plan to do Research Project C but will switch to Triad Project C—knowing what's next is part of the process of doing research!)

i. what you would plan to do do for Research Project C. For example, if you did not yet evaluate your approach, the proposed work could describe your evaluation plan.

ii. if you want to do Research Project C.

iii. a checklist of specific deliverables for Project C (a) continuing to meet once a week with your mentor, (b) continued weekly notes about your progress, (c) what end results you expect to have (code or experiments), (d) that you will do a project C report

Section 5 a bibliography, with at least what you cite in related work

(We encourage you to write your document in LaTeX and build a PDF, but you can use whatever tool you prefer.)

Your Research Project A proposal will also serve as part of your grade for Research Project B. Since research is not always predictable, you will not be graded on executing exactly the plan in your Research Project A proposal, but it will serve as a guideline. However, the most important factor is to conduct quality work that adapts to what you discover about the problem as you go. Your mentor especially (as an expert on the problem), but also your TAs and professor, can help focus you on interesting directions throughout the semester.

6 Practical Project C: Extending Minitor

Project C is still under revision, but we are expecting it will include some extensions to your Minitor implementation, and perhaps some evaluation of how to detect Minitor.

7 Submitting and Evaluating Practical Projects

To submit each part the assignment, put *all* the files needed (Makefile, README, all source files, and source to any libraries) in a gzip'ed tar file and upload it to the class moodle with the filename `projb.tar.gz`. *Warning:* when you upload to the moodle, please be careful in that you must both do "Upload a file" *and* do "Save changes"! When you are done you should get a message "File uploaded successfully", and you should see a list with your file there and an option to "update this file". If you do *not* see "file uploaded successfully", then you have *not* completed uploading!

We *strongly* recommend that you confirm that you have included everything needed to build your project by extracting your tarfile in a different directory and building it yourself. (It's easy to miss something if you don't check.)

It is a project requirement that your project come with a Makefile and build with just the make command. To evaluate your project we will type the following command:

```
% make
```

Structure the Makefile such that it creates an executable called `projb` (Note: *not* `a.out`.) For more information please read the make(1) man page. (The Linux we use supports GNU make; you may use GNU make extensions if you wish.)

We will then run your program using a test configuration file. You can assume that the topology description will be syntactically correct. After running the program, we will grade your project based on the output files created by your program's processes.

It is a project requirement that your implementation be somewhat modular. This means that you should follow good programming practices—keep functions relatively short, use descriptive variable names. You must use at least one header file, and multiple files for different parts of your program code. (The whole project should be broken up into *at least* two C/C++ files. If you have a good file hierarchy in mind you can break 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.

We will consider external libraries, but **it is a project requirement that all external libraries must be explicitly approved by the professor**. Any libraries in the default VM image are suitable (including libc and STL). A list of approved libraries will be on the TA's webpage on the moodle.

Computer languages other than C or C++ will be considered but *must be approved ahead of time*; please contact the professor and TA if you have an alternative preference. The deadline for approving new computer languages is *one week* before the project due date, so get requests in early. The language must support sockets and process creation. (Please ask *before* you start, we don't want you waste your work.)

Although we provide a complete sample input file and output, final evaluation of your program will include other input sources. We therefore advise you to test your program with a range of inputs.

Although the exact output from your program may be different from the sample output we provide (due to events happening in different orders), your output should match ours in format.

It is a project requirement that your code build without warnings (when compiled with -Wall). We will be responsible for making sure there are no warnings in class-provided code (any warnings in class-provided code are our bugs and will not count against you).

8 Hints

The structure of the project is designed to help you by breaking it up into smaller chunks (compared to the size of the whole project). We strongly encourage you to follow this in your implementation, and do the stages in order, testing them as you go. In the past, some students have tried to read and implement the whole assignment all at once, almost always resulting in an unhappy result.

8.1 Common pitfalls

Please do not hardcode any directory path in your code! If you hardcode something like `//home/csci551/...` in your code to access something in your home directory and the grader cannot access these directories during grading, your code will not work (and this will be your fault)! If your code does not work, you get no credit! Instead, assume paths are given external to your program, and that you read and write files in the current directory (wherever that is).

8.2 Doing multiple things at once

Later stages of the project may require you to handle both timers and I/O at the same time. (Stage 1 is not complex enough to require timers.) One approach would be to use threads, but most operating systems and many network applications don't actually use threads because thread overhead can be quite large (not context switch cost, but more often memory cost—most threads take at least 8–24KB of memory, and on a machine with 1000s of active connections that adds up, and always in debugging time, in that you have to deal with synchronization and locking). Instead of threads, we strongly encourage you to use timers and event driven programming with a single thread of control. (See the talk "Why Threads Are A Bad Idea (for most purposes)" by John Ousterhout, `http://www.stanford.edu/~ouster/cgi-bin/papers/threads.pdf` for a more careful explanation of why.)

Creating a timer library from scratch is interesting, but non-trivial. We will provide a timer library that makes it easy to schedule timers in a single-threaded process. You may download this code from the TA web page. There is *no* requirement to use this code, but you may if you want. If you want to use it, download it from the class web page. There is no external documentation, but please read the comments in the `timers.hh` and look at `test-app.cc` as an example. If you do use the code, you must add it to your Makefile and you must document how you used it in your README.

8.3 Other sources of help

You should see the Unix man pages for details about socket APIs, fork, and Makefiles. Try `man foo` where foo is a function or program.

The TAs can provide *some* help about Unix APIs and functionality, but it is not their job to read the manual page for you, nor is it their job to teach how to log in and use vi or emacs.

You may wish to get the book *Unix Network Programming*, Volume 1, by W. Richard Stevens, as a reference for how to use sockets and fork (it's a great book). We will *not* cover this material in class.

The README file should not just be 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 than for known deficiencies that you document.)