

CSci551/651 Spring 2019 Project A

Practical Project A Assigned: Thursday 2019-01-17

Practical Project A Due: **noon, Wed. 2019-02-06** (not a class day). (*Correction:* previously the year was listed as 2018, although they day of month and week was correct.)

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: 2019-01-05: none yet. *But*, you should *expect* that there will be changes and clarifications to this project, and code accordingly. (Start early, but be able to change things slightly as the work goes along.)

2019-01-21: clarified how to handle `num_routers`, and fixed the router number in sample output.

2019-01-22: undid change in router number in sample output, and corrected the word “proxy” in several places to “primary router”.

1 Overview

There is one one type of project this year. In years with a joint CSci551/651 class we have an option of a Research Path. This year we have a *Practical Path* project that involves implementing a network-centric system. Everyone following this path solves the same problem, but of course in slightly different ways.

(If a student is very interested in doing a research project as an alternative, please talk before the the third week of class.)

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, C++, or Go 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 *Octane*, a simplified version of an Ethane-like system as described in the paper by Casado et al. (see [Casado09a] 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 implementation with additional capabilities, and using it to explore some network use cases. It will probably be due the last week of class.

1.2 Research Path 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.

We will have a poster session at the end of the semester where students can present their research path results. All students are encouraged to attend (not just students doing Research-Path projects).

Research Projects 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 may 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, independent writeup.)

1.3 Academic Integrity and Your Project

You are welcome to discuss your project with other students at the conceptual level.

You are welcome to use external libraries or code *after getting permission* and with disclosure about code sources in your report. (Many projects start from libraries or modify

existing programs, and that is generally fine as long as it is clear to everyone what you got elsewhere and what you did.)

The assumption is that any unattributed work in your report or code in your project is assumed to be your creation, and you are required to attribute (cite text, describe any code, *and* indicate in the code source comments) anything that you did not originate.

If you have any questions about policies about academic integrity, please talk to the professor. Any cases of plagiarism or misrepresentation will result in an F for the entire course.

2 Practical Project A: Getting Started

2.1 Virtual Machines and your Development Environment

To provide a consistent environment that you can develop your class code on, we will provide you a virtual machine image with Fedora 29 (Linux 4.19.8-300 or so) installed upon it. You can download it from “Project Information” on the class moodle (at https://ant.isi.edu/csci551_sp2019/). The fedora29 VM is a few gigabytes in size to download, and about 64 GB when unpacked. You are welcome to develop your code on any platform (if you want), but your code *must* build and run on this platform, as it is the only one we will use for evaluation.

There are virtual machines available that can be run with VirtualBox (available at no cost for Windows, Linux, and Mac from <https://www.virtualbox.org>). With some work you could also probably run our VM image under other VM hypervisors (VMWare Player, available at no cost for Windows and Linux from <http://www.vmware.com/products/player/>, or kvm under Linux, or Hyper-V on Windows), although we don’t directly support things other than VirtualBox. (In the past we had some trouble with subtly different operation under different hypervisors, so you should be careful.) Please see the TA’s Project Information page for more information about VMs.

We provide a VM image. If anyone does *not* have access to a computer that can run VirtualBox, please let us know immediately and we will check on alternatives. (We believe the computers in SAL127 support VMs, but in the past they only supported VMWare Player. We will make sure everyone has a development environment.)

Please note that when you submit your code, you will submit *only* your code, not the entire virtual machine. So your code must run without any OS changes.

Accounts in the VM: In the virtual machine, we have created a **user account csci551**, with **password spring!bloom**. We recommend you do all your development as this user (not as root, since it’s very bad security practice to run as root). We also recommend you **change the password after you log in**. Some parts of the project will require root access. You may use the **sudo** program to get root access using this user password. (We are not giving out the root password, but, of course, with full sudo access, you can certainly change that if you insist, although you should not need to do so.)

You will need to **get files in to your VM**, and later you will need to get them out (say, to submit your assignment). The VM has networking turned on, so you should be able to **scp**

and ftp in and out, as well as contact the class moodle from within the VM to submit your project. You may also find it helpful to use a “share folder” to move files between the host computer and the VM. Other ways are possible; a search engine can help you find them if you want.

The VM will start in graphics mode (running the GNOME with Wayland). You will need to run multiple processes on the VM for the project, so you will probably want multiple windows. There are several ways to get multiple windows. You can use the graphics on the console of the virtual machine. Alternatively, you can start separate terminals on the host computer and ssh in to the VM from each.

Other tools: You are welcome to install other development tools if you want. A web search on “installing Fedora software” will suggest how to do so. Please remember, though, any libraries must be pre-approved, as must any development tools that are required to build your program. If your program requires any libraries or non-pre-installed development tools, you must document that in your program’s README, described below.

Please see Section 5 for other requirements about your project source code.

2.2 Stage 1: Starting Processes

In the first stage you will experiment with process forking and socket communication, and network plumbing, and you will do this all in a virtual machine.

You will create two Octane Routers. The first, the primary router, will read a configuration file that will confirm we’re running stage 1 (and in future stages, will do more). It will fork of a second router. The first router will hook in to the networking system through a *tunnel* network interface that lets it scoop up network traffic. We will show that you can ping to the second router through the first.

Later stages will have more elaborate networks and processing.

We will give you the exact commands to hook up network traffic to the primary router and a framework for handling packet processing in the (*clarification 2019-01-22: was proxy*) primary router. You will need to implement packet processing at the router.

Your program will begin by reading a configuration file. The name of the configuration file will be the first argument to the program (in C, this is `argv[1]`), so the command to run you code should be like: `sudo ./proja config_file` (Subsequent stages will give the configuration file in the same way, but will likely pass a different name in the first argument.)

The contents of the configuration file is in a simple format with comments and non-comment lines. Any line that starts with a hash character (“#”) is a comment and should be skipped. Comments can be on any line of the input file. Non-comment lines will be one or more fields separated by whitespace (one or more spaces or tabs). The first non-comment line will indicate what stage is being run. Initially, this will be **stage 1**. The second non-comment line will contain the text `num_routers N`, where N will be replaced by an integer indicating how many routers to create. For stage 1 (and, in fact, for all stages in Project A), N will *always* be 1, but we will change that in later stages. (*Clarification 2019-01-21: note that, although it is called num_routers, it actually indicates the number of secondary routers you should have. The primary router is implicit.*)

The primary router will create a log file `stage1.r0.out` where it will write information we will use to evaluate your assignment. We will ignore anything printed to standard output or standard error; we *only* evaluate the log file.

The primary router should create a dynamic (operating-system assigned) UDP port. (See the `getsockname()` man page or the Steven's book on socket programming in the class syllabus for details about how to do dynamic port allocation if you're not familiar with it.) After starting the Routers the primary router will wait on this port for return traffic. The primary router should print the dynamic port to its log file by writing the line "primary port: PORT" where PORT is the number of the dynamic port.

After reading the configuration file, your primary router should `fork()` to create the second router. The code for both your primary and secondary router need to be in the same program, so that after forking, the secondary router should run a router subroutine and the primary router can call a function for its job. (*Clarification 2019-01-22*: previously the last sentence talked about ~~proxy~~ several times where it mean primary router.) You should not execute another program (you don't need to, and it would just complicate things). You need to use independent processes, not threads.

The primary router should pass its dynamically allocated UDP port process to the child through a global variable. A copy of this global variable will be available in the child process, and this variable will inform it how to contact the primary router after starting its secondary router duties. You may also pass the stage number and the router index numbers through global variables, if you want.

The primary router will then wait for the secondary router to send it an "I'm up" message on its UDP port. You may select whatever format you want for this message.

When the secondary router starts, it should create a log file called `stage1.r1.out`. It should then get its process id (see the `getpid()` man page) and write the line "router: X, pid: N, port: M" to this file. It should then send an "I'm up" message (Just send router's PID) to the primary router's UDP socket.

After forking, the primary router will wait passively on its UDP socket for packets from secondary routers. When the secondary router sends "I'm up" message to the primary router, the primary router should print the line "router: X, pid: N, port: M" to its log file (the primary router's log file). Finally, print to the log file when a router is closed, either by a function call or the end of the program.

Sample input: Here is a sample configuration input for stage 1.

```
#This is a comment line, should be ignored
stage 1
# There can be other comments, or variable numbers of spaces or tabs in lines.
# Clarification 2019-01-21: 1 router means one secondary router,
# in addition to the implicit primary router.
num_routers      1
```

Sample output: Here is a sample output for stage 1.

In `stage1.r0.out`: (*Correction 2019-01-21*: ~~the primary router should be number 0 in the output.~~ *Re-correction 2019-01-22*: actually, the router number in the "router:" line is

the number of the secondary router, so “1” is correct. The number on the “closed” line is the number of the primary router.)

```
primary port: 44051
router: 1, pid: 3104, port: 43031
router 0 closed
```

In `stage1.r1.out`:

```
router: 1, pid: 3104, port: 43031
router 1 closed
```

Writeup: In addition to turning in your code, you need to write up what you learned. Put this information in a text file called `README.stage1.txt`, and include these details (include the letters, too, so we know what part of your README answers which of our questions):

- 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) **Portable:** Will your code always work if the primary and secondary router were on different computers with different CPU architectures, like an IBM PowerPC and an AMD x86-64? (*Clarification 2019-01-22:* changed `proxy` to primary router.) If so, why (what specifically did you do to support that case)? If not, why not (what problem would occur if run between different types of computers)? (Note that in your Project A, they are always on the same computer architecture because you’re just forking another process, not connecting to another computer. This question is about the hypothetical case of if they were on different computers with different CPU architectures.)

Other comments: Even at this stage your program needs to be modular, using multiple files for logically different parts of the source code. One strategy might be to put the primary and secondary router functions each in different files. (*Clarification 2019-01-22:* changed `proxy` to primary router.) You can choose other organizations.

2.3 Stage 2: Sending Traffic

In this stage, we add traffic and hook your primary router into the VM’s network.

Your primary router will hook into the networking system. To do this, we first need to create a tunnel and configure it. To do so, run the following commands in a terminal:

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

The first command creates a new tunnel interface named `tun1`. The second command sets the IP address of the other end of the tunnel to 10.5.51.2, and brings it up. (These commands will be run by hand from the terminal; they should not be run by your program.) After doing these steps, all the traffic sent to the 10.5.51/24 address block will go into `tun1`. So if you ping address 10.5.51.x from a new terminal, all the ICMP packets will go into `tun1`.

After you create and configure the tunnel, you need to write code to receive tunneled packets. To do this, your program will open the tunnel interface and then busy loop reading packets from the tunnel device. Each packet will be returned in a separate read call.

The code to handle this process is fairly standard; we will provide a framework on moodle that we recommend you use (see https://ant.isi.edu/csci551_sp2019/sample_tunnel.c). (Our framework is derived from the tutorial at <http://backreference.org/2010/03/26/tuntap-interface-tutorial/>, if you want more detailed information about how to open a tunnel interface and how to read data from and write data to it.)

Our framework only supports setting up and reading from the tunnel. You must extend this framework to read *both* from the tunnel *and* to get packets over UDP from the router. To handle both, you will need to replace the busy loop reading the tunnel with a `select()` statement that handles both the tunnel interface and the UDP socket. You hopefully had experience with `select` in EE450 or CS350; if not, the book *Unix Network Programming: Volume 1: The Sockets Networking APIs*, by Stevens, Fenner, and Rudoff, or any number of web-based tutorials can get you started.

After you get an ICMP ECHO packet from tunnel interface, your primary router will need to send it to the secondary router. To do this, you must send the packet via the UDP socket you created in stage 1. *Clarification 2019-01-22:* Normally, when sending an IP packet, one has to encapsulate it in some other protocol. Several standards exist for encapsulation, like (IP in IP, IPsec, or PPP). For this project, you should just send the IP packet from the primary to the secondary in UDP. (That is, the primary read the IP packet from the tunnel interface, getting a its IP header and payload, then send that information [the IP header and payload from the tunnel] from the primary router to the secondary via UDP.)

Your secondary router should receive the ping packet and generate an ICMP echo reply, and send it back to primary via the UDP socket. (*Clarification 2019-01-22:* changed `proxy` to primary router.) You will need to use the standard ICMP packet format for echo reply as defined in RFC-792 (a copy is at <http://tools.ietf.org/html/rfc792>). You will also need to compute packet checksums (if you don't the kernel will silently drop your packet, assuming it's corrupted.) We provide example code for checksum computation at https://ant.isi.edu/csci551_sp2019/icmp_checksum.c.

Clarification 2019-01-22: A simple way for the secondary to generate the reply is just to reverse the source and destination address, change the ICMP type field, and regenerate the checksum.

Finally, your primary router should write the ICMP echo reply packet to the tunnel interface.

The primary router must log each ICMP packet it forwards (both to the user and to the router) to its log file, with the format "ICMP from tunnel, src: 10.5.51.2, dst: 10.5.51.3, type: 8", for traffic from the tunnel. Or if from a secondary router, it should say "...from port: N..."

After you did all of the above steps, when you ping any address in 10.5.51/24, you

should be able to see replies. **Note:** your program must be run as root user or superuser privileges (for example, by `sudo ./proja config_file`), since Linux requires root privileges to manipulate tunnel interface.

Your primary router should exit after 15 seconds of inactivity closing your program (hint: set the timeout parameter in your `select()` statement).

Sample input: Here is a sample configuration input for stage 2.

```
#This is a comment line, should be ignored
stage 2
num_routers 1
```

Sample output: After running your program and then running `ping -c 4 10.5.51.3` in another window:

In `stage2.r0.out`: (*Correction 2019-01-21*: the primary router should be number 0 in the output. *Re-correction 2019-01-22*: same comment as the last recorection.)

```
primary port: 44051
router: 1, pid: 3104, port: 43031
ICMP from tunnel, src: 10.5.51.2, dst: 10.5.51.3, type: 8
ICMP from port: 43031, src: 10.5.51.3, dst: 10.5.51.2, type: 0
ICMP from tunnel, src: 10.5.51.2, dst: 10.5.51.3, type: 8
ICMP from port: 43031, src: 10.5.51.3, dst: 10.5.51.2, type: 0
ICMP from tunnel, src: 10.5.51.2, dst: 10.5.51.3, type: 8
ICMP from port: 43031, src: 10.5.51.3, dst: 10.5.51.2, type: 0
ICMP from tunnel, src: 10.5.51.2, dst: 10.5.51.3, type: 8
ICMP from port: 43031, src: 10.5.51.3, dst: 10.5.51.2, type: 0
router 1 closed
```

In `stage2.r1.out`:

```
router: 1, pid: 3104, port: 43031
ICMP from port: 44051, src: 10.5.51.2, dst: 10.5.51.3, type: 8
ICMP from port: 44051, src: 10.5.51.2, dst: 10.5.51.3, type: 8
ICMP from port: 44051, src: 10.5.51.2, dst: 10.5.51.3, type: 8
ICMP from port: 44051, src: 10.5.51.2, dst: 10.5.51.3, type: 8
router 1 closed
```

Writeup: In addition to turning in your code, you need to write up what you learned. Put this information in a text file called `README.stage2.txt`, and include these details (include the letters, too, so we know what part of your README answers which of our questions):

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) **Portable:** Will your code always work if the primary and the router were on different computers with different CPU architectures, like an IBM PowerPC and an AMD x86-64? If so, why (what specifically did you do to support that case)? If not, why not (what problem would occur if run between different types of computers)? (Note that in your Project A, they are always on the same computer architecture because you're just forking another process, not connecting to another computer. This question is about the hypothetical case of if they were on different computers with different CPU architectures.)

Debugging suggestion: When you're debugging, you will want to look at where packets go. You should explore tools like `wireshark`, `tshark`, and `tcpdump` which are GUI- and terminal-based tools that let you capture and display network packets. You will likely want to run these against different interfaces.

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

3 Practical Project B: Octane

Project B will be to implement Octane, a simplified version of an Ethane-like software-defined networking system as described in the paper by Casado et al. (see [Casado09a] in the class syllabus). You may wish to review that paper to prepare for the project. (You should not need to read that paper for Project A.)

4 Practical Project C: Extending Octane

5 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 `proja.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!

Note that you will submit *one* tar.gz file, with *one* program in it. That program must run all stages. The project is designed so the input tells you what stage is running, allowing you to do different things if necessary. (However, it's also designed so that you shouldn't need to do too much differently.)

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 `proja`. (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, C++, and Go 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).

6 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.

6.1 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).

6.2 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.

6.3 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.)