CSci551 Spring 2004 Project A

Assigned: February 18, 2003. Due: noon, March 28.

You are welcome to discuss your project with other students. You may reuse algorithms from textbooks. You may possibly reuse functions from libraries, but if you're using anything other than 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. Otherwise, each student is expected write *all* of his or her code independently. All programs will be subject automated checking for similarity. Any cases of plagiarism will result in an F for the entire course.

Please note: you should expect to spend at least 12 or more hours on this assignment, and probably 20 hours or more. Please plan accordingly. If you leave all the work until the week before it is due you are unlikely to have a successful outcome.

Changes: 17-Feb-04: 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.)

24-Feb-04: two small clarifications about comments and LS packet size.

25-Feb-04: fixed accidental deletion of most of the text. (Don't clarify the project late at night :-)

16-Mar-04: clarification about when LSA messages must be sent.

16-Mar-04: clarification about TTL handling.

23-Mar-04: clarification about route priorities.

24-Mar-04: clarification about dropping.

25-Mar-04: clarification about LS message byte ordering and about the definitions of messages-sent and last-ls-msg-sent.

1 Overview

The purpose of this project is to apply some of what we've studied about routing. It also has a secondary purpose of implementing a program using network programming (sockets) and processes (fork) and Unix development tools (make).

This homework will be done in two parts. The first part will involve implementing a link-state IGP routing protocol for an AS with several routers.

The second part will add some external ASes and policy routing. Some of these routers will also receive routes from external ASes. External messages may include policy routing with an approach similar to BGP's Multi-exit discriminator (MED) path attribute.

The program will be implemented as multiple processes running at the same time. A *manager* process will be responsible for starting the experiment and synchronizing it between phases. It will create *child* processes that represent each router in the AS. You will need to use the Unix fork() API to create processes.

The manager process will communicate with the children via TCP. You should let the system assign the master's port number dynamically (see `getsockname()`). The master can then pass this port to the childern via shared memory (during the fork) or some other means if you prefer.

Children will communicate with each other via UDP. You will need to use the Unix socket APIs to create these connections and the select() call to multiplex traffic from your several neighbors.

2 Building the Network

The first stage of your program will create and configure processes to represent your network. You will create a manager program that is responsible for forking the router processes. The manager should also bind and listen to a TCP port that it will use for communication with the routers (a separate TCP connection for each router).

The manager program will create a TCP port to communicate with its routers. It should then read configuration information from stdin. The first non-comment line will indicate the desired type of output. Initially, this will be "stage1". The second non-comment line will indicate the number of routers to create (by forking). The third non-comment will indicate the simulated packet loss rate. (Initially that will be zero, but you will change that for stage 2.)

It should then fork a process to represent the routers. The routers should each open a TCP connection with the master and read its configuration information from the master. It should read this configuration information over TCP from the master in a format that you can specify. This configuration information should include the router's id (numbered 1 to n), the prefix it represents (a 32-bit IPv4 address and CIDR mask), how many nodes are in the AS, what the loss rate should be (used in Section 4, zero for now) and the UDP ports and the router ids of any immediately connected neighbors.

Note that you may not send information like the entire topology in the configuration message, routers must discover this themselves by exchanging routing updates.

For more details please refer to example manager configuration input in Section 6.

Output for this stage: If the first line of the configuration information is "stage1" then produce the following output and stop.

At the end of this stage, the manager should create the output file "s1.out" with one line for each router. Each line should list the router id, process id, prefix, and UDP port number for the router.

3 Link-State Routing

The goal of the second part is to implement the link state routing algorithm. At the beginning of the routing stage each router knows only its neighboring routers (including external routers). Each router needs to send an LSA message informing the other routers about its connectivity. Format for the LSA is specified below.

Each router, after it receives its connectivity table from the manager, should send routing updates to its peers. It should send out LSA updates whenever its link-state changes. (*Clarification 16-Mar-04*: with LS protocols you only need to flood out new announcements when your local link state changes, *not* whever your routing table changes. That was a specification error.) (You need not implement periodic update routing messages.)

Each router should then contact its neighbors over UDP and send them an LS update message to each of its IGP peer routers. That message will contain a type field and its current routing information (what prefixes it has routes to, their paths), and a sequence number. The message must be sent via *reliable flooding*. The receiver should ACK (via UDP) the message and sequence number when it gets it. The sender must sent a timer and resend the message if it was not ACKed. For simplicity we'll assume 5 second fixed timers.

Please read and consider the protocol processing rules carefully. You will probably find it useful to work through the routing algorithm by hand on different simple topologies.

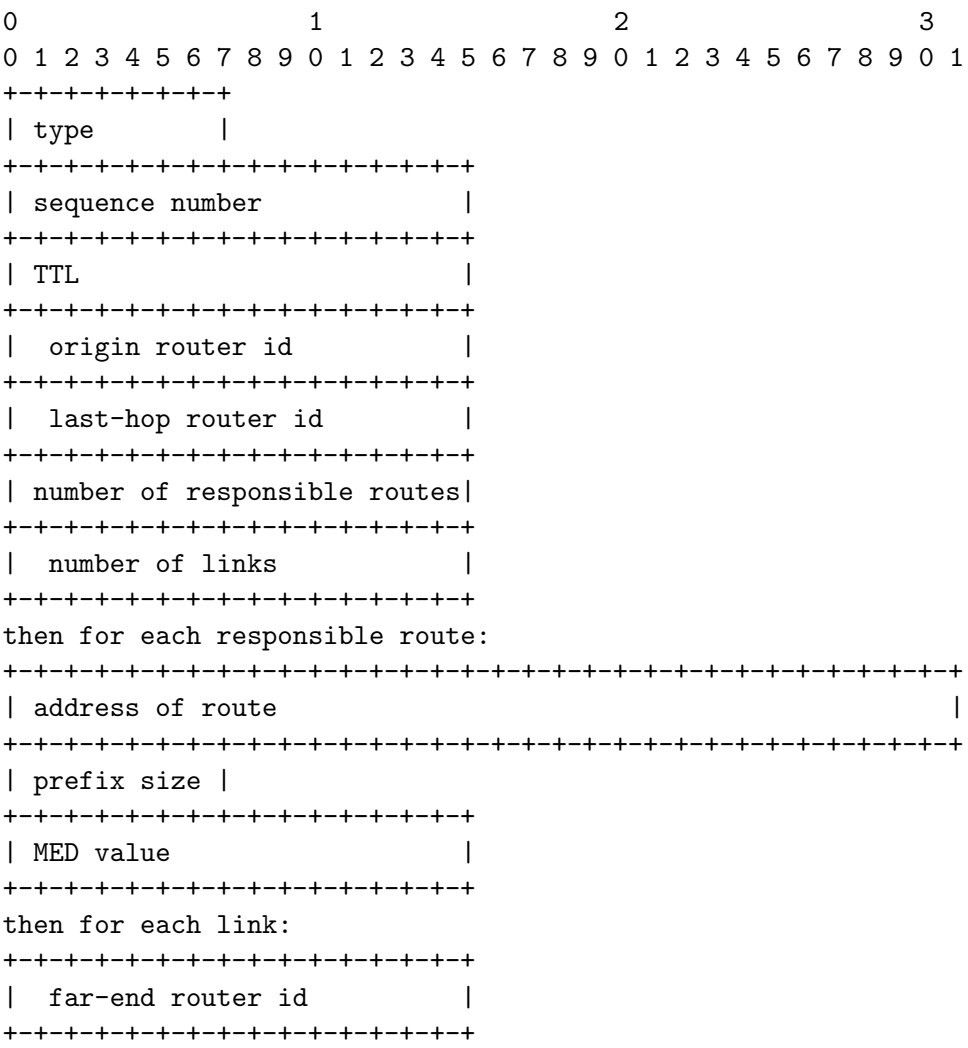
Even if routers know the port numbers and/or ids of their peers, they are not allowed to assume anything about routing from that peer until they've actually heard a UDP message from that peer. Thus, the first message a router can send to its peers is a message with 0 neighboring links. (Hopefully it will be able to send another more positive message quickly. :-)

Things you should think about: how do I insure that routing updates don't create more routing updates that result in packets flowing infinitely in circles? *Clarification 24-Feb-04*: previously this said, incorrectly, that router messages get longer. That was incorrect.)

Things you do *not* need to think about: what if the LS message is larger than a UDP packet?

You should figure out a way to determine when the routing exchange has converged. This can be as simple as waiting for several seconds.

The LS message format should be:



For stage 2 of this project MED value will always be zero.

You can use the type field however you want.

For stages 2 and 3, the "responsible" routes are whichever routes are assigned to that origin router. For stage 3 you may relax this to include routes injected from other ASes if

you'd like.

Clarification 25-Mar-04: You are encouraged to encode your fields in network byte order, but you are not required to do so. (Since it wasn't specified in the original project specification.)

If you need to add fields to the LS message, you may do so, but you must document what fields you add and why in the "message format" section of your README.

As each router gets routing information it needs to run the shortest-path routing algorithm (Dijkstra's algorithm) to create its routing table. (It will run this algorithm multiple times as LS messages arrive.)

Prioritization of routes is as follows:

1. MED discriminators (added in stage 4 of this project)
2. shortest path in hops
3. lowest id of either router on the last-hop link

Clarification 23-Mar-04: A different way of stating the third part of prioritization is that, given otherwise equivalent routes, break ties in favor of lowest router id.

Also, make each router count how many messages it gets.

Clarification 16-Mar-04: The TTL field should be initialized to 16 when a message is sent and decremented by one each hop. Packets with a TTL less than 1 should be dropped. We will not have topologies bigger than this in our test cases—the purpose is that if you have bugs in your code that cause packet looping, you don't want the packets going around infinitely!

Output for this stage: If the first line of the configuration information is "stage2", then produce the following output and stop. (You may assume that the drop rate will be zero in the configuration file as well.)

For each router, there should be a file containing the routing table after convergence. The name of this file should be "s2-X.out" where X is the id of the corresponding router. Hence, there will be N such files. This file should contain the complete routing table of the corresponding router for *each* step in the evolution of the routing table. Each line in the file is a routing table entry for a router X. The format of a line is:

```
id prefix/size next-hop distance (med=x)
```

where id and prefix/size is the final destination network and router, next-hop is the id of the next-hop router to get to that node, and distance is the distance to that router in number of hops. Med=x specifies the med value (if any). For stages1 through 3, the med value will always be zero.

After each version of the routing table, print a line with three dashes.

After the last routing table (after the routing has converged), show the line:

```
messages-sent A
messages-received B
messages-dropped C
last-ls-msg-sent DDDD...
```

where A is the number of LS messages that router X sent (*Clarification 25-Mar-04:* "sent" means either originated or forwarded). B the number successfully received (don't count a dropped message as received, but do count it as sent), and C the number it dropped on receipt. For Stage 2, it should be the case that C is 0, since dropping has not yet been added. Finally, you need to print, in hexadecimal (the DDDD), the contents of the last LS message that node originated. (*Clarification 25-Mar-04:* "originated" does not include forwarded messages.)

4 Packet Loss

Link-state routing must use *reliable* flooding. This part looks at what happens when packets are lost in the network.

To simulate packet loss, modify your input file to handle a non-zero discard rate in the configuration input.

You should simulate packet discards randomly on a per-packet basis. For example, with a 5% discard rate, every packet must suffer a 5% chance of being dropped. Different runs of your program should drop different packets. You should implement this randomly, not deterministically. (For example with a counter that drops every 20th packet is deterministic, not random.) Dropping should be implemented on the *receive* side (i.e., on receipt, randomly drop or keep the packet), not on the sending side.

You should simulate drops only for LS traffic, not for configuration manager TCP traffic.

Clarification 24-Mar-04: However, all traffic sent over UDP should be subject to possible dropping. (I.e., *both* LS messages and ACKs can be dropped.)

Output for this stage: If the first line of the configuration information is "stage3", then produce the following output and stop. (You may assume that the drop rate will be non-zero as well.)

The output for this stage is just like the output in Section 3, but written to the file "s3-X.out".

Hint: what differences should you see between your s2 files and your s3 files?

5 Policy Routing

We want to observe a simple form of policy routing, so this section describes how we will use the MED field.

Assume your routing protocol also gets routes injected from other ASes. (This is similar to how an I-BGP routing mesh can get routes from E-BGP connections.)

An additional part of the configuration file will specify all external routes. Each line will list the (internal) router id where the route will be injected, an id representing the external router, the prefix and mask of the external route, and a MED value for that route.

You may assume all ids (for both internal and external routers) are in the range [0,2^15], and that they don't overlap with internal router IDs. (This way don't need to worry about encoding of signed numbers.)

You should not simulate the external routers as separate processes. Instead, the master program should send these routes to your router processes as part of their configuration. This should be sent over the TCP connection in some format that you specify.

Once your routers have these external routes they should propagate them just as their own routes (using reliable flooding).

All routers must then compute routing tables not only for internal routes, but also for external routes, using the same prioritization rules as before. Note: you will have to propagate these external routes across your AS in LS update messages. I suggest you have the peer router add them to its list of "local" routes.

Note: there could be packet losses for this stage as well depending on the input file.

Output for this stage: If the first line of the configuration information is "stage4", then produce the following output and stop.

The output for this stage is just like the output in Section 3, but written to the file "s4-X.out".

Note: when generating the output for external routes, the *id* portion can correspond to either the external router's id or the peer it talks to, and the *distance* field can be include the hop to the destination router or not at your discretion.

6 Sample configuration input

Your program should read and ignore any lines beginning with "#". You should be prepared to handle comments *anywhere* in the input configuration. (*Clarification 24-Feb-04*: comment lines are any lines that have # as the first non-space character.) Note that any amount of whitespace (spaces or tabs) can separate words. *Please check your program to confirm that it handles both these constraints*—we will post several sample input files, but we will *not* give students a chance to update their program to handle sample input after the assignment is over.

```
stage1
# number of nodes
5
# loss rate, in percentage
# (zero for for stage 1, 10 for stage 2)
0
# edges, termining with "0 0"
1 2
1 3
2 4
3 4
3 5
4 5
0 0
# addresses and masks for each internal router, in order
# in the form a.b.c.d e where e is the prefix length.
# last entry is 0.0.0.0 0
# The first line is for router 1, 2nd for router 2, etc.
10.1.0.0        16
10.2.0.0        16
192.168.123.0   24
```

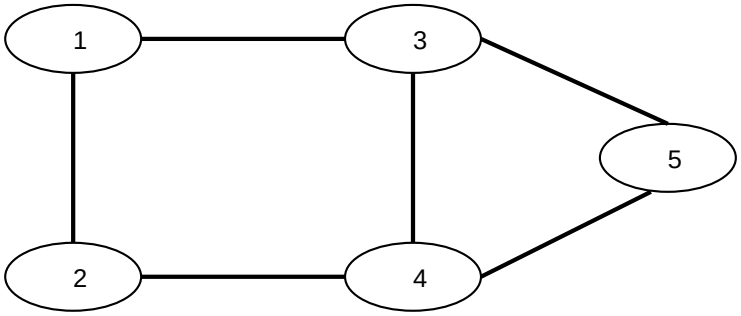


Figure 1: Sample topology shown in Section 6. (This figure appears only in the PDF version.)

```
192.168.101.0   24
192.168.200.128 25
0.0.0.0         0
# external routes needed for policy routing
# you may assume that all the config files for stages 0-2
# have this line in them and no external routes
0               0 0.0.0.0  0     0
```

Shows 5 nodes and their connectivity. A picture of this topology is in Figure 1.

Here's the same topology with external connectivity to other ASes added for policy routing (see Section 5).

```
# number of nodes
5
# loss rate, in percentage
# (zero for for stage 1, 10 for stage 2)
0
# edges, termining with "0 0"
1 2
1 3
2 4
3 4
3 5
4 5
0 0
# addresses and masks for each internal router, in order
# in the form a.b.c.d e where e is the prefix length.
# last entry is 0.0.0.0 0
10.1.0.0        16
10.2.0.0        16
192.168.123.0   24
192.168.101.0   24
192.168.200.128 25
0.0.0.0         0
# external routes
```

```
# PeerRouter ExternalID Prefix   Mask  MED value
1             11     80.0.0.0 16    0
1             11     81.0.0.0 8     0
1             11     82.0.0.0 8     5
5             12     79.0.0.0 16    0
5             12     81.0.0.0 8     0
5             12     82.0.0.0 8     10
0             0     0.0.0.0  0     0
```

Sample output for these inputs will be posted sometime soon.

7 Project file layout and writeup

Your project must have the following:

Makefile: The project *must* use a Makefile for compiling the programs.

The Makefile should have the following targets:

all : build all the programs

clean : remove the old `*.o` files and all the binary files

and anything else you need to make your project. For more information please read the make(1) man page. (Your program must run with Solaris make, so be careful not to use any make extensions such as those in GNU make (gmake).)

Your program must compile into a file `proja`.

header file(s): This file contains all the declarations of the data structure, `#includes` and `#define`. This header file is then included in other C files.

C/C++ files: 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.

README: This file describes your project, and must include the following sections.

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.

Message format: Did you make any changes to the LS message format? Did you add any fields? If so, what and why?

Message counts: How did the message counts compare with and without loss rates? Were they what you expected?

Termination method: How did you determine that the computation had terminated? What was good or bad about this mechanism?

Idiosyncrasies: Are there any idiosyncrasies of your project? It should list under what conditions the project fails, it any. What input limitations does it have?

Surprises: Did you find any surprising things implementing this project?

The README file should not just be a few sentences. You need to take some time to describe what you did and especally 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.)

Computer languages other than C or C++ will be considered; please contact the professor and TA if you have an alternative preference. The language must support sockets and process creation.

8 Submitting and evaluating your project

Submit the project using submit command on aludra. The exact syntax of the command is the following. You need to submit *all* the files you created, i.e. C/C++ files, `*.h` files, `README` and the `Makefile`. You *should not* submit files that are automatically generated, such as `*.o` files, executables, or output from test runs.

List each file on the command line, do *not* put your program into a single tar/zip file.

```
% submit -user csci551 -tag proj1 file1 [ file2 ... ]
```

To evaluate your project we will type the following command:

```
% make
```

Structure the Makefile such that it creates an executable called `proja`. We will then run the manager 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 the manager and the routers.

By February 28 we will provide a complete sample input file and sample output from that file. However, final evaluation of your program will include other input sources.

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.

9 Hints

Please test and debug your processes on the Suns in SAL 127 and LVL library. Please do not run jobs with many processes on aludra/nunki; there are process limits that may prevent them from running. (Hint: make sure you check the exit status of fork() so you're not surprised when something doesn't run.)

You will need to think about how you're going to handle timers and I/O at the same time. 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). Insted 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://home.pacbell.net/ouster/threads.ppt` for a more careful explanation of why.)

Creating a timer library from scratch is interesting, but non-trivial. We will providing 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.

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 TA can provide *some* help about Unix APIs and functionality, but it is not his job to read the manual page for you, nor is it his 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.

Please be very careful with fork()—starting too many processes can bring an entire computer to its knees.

Although the final program must be run on Solaris machines, you are encouraged to do initial testing on your own computer. If you have a Linux or BSD computer, porting to Solaris should not be a problem (everything in this assignment should be completely portable). If you're running MS-Windows, you might consider trying Cygwin, a Unix emulation package for Windows. (We cannot support or answer questions about it, however.) If you're running a Macintosh, MacOS X includes a complete BSD Unix that should be very similar to Solaris.

One thing to watch out for on Solaris is that all the useful bits of the network are in libraries "socket" and "nsl". You will need to link against these libraries on Solaris, but not on other versions of Unix.