COSC 364
Internet Technologies and Engineering
Assignment

Andreas Willig
Dept. of Computer Science and Software Engineering
University of Canterbury
email: andreas.willig@canterbury.ac.nz

February 6, 2016

---

1 Administrivia

This assignment is part of the COSC 364 assessment process. It is worth 25% of the final grade. It is a programming project in which you implement parts of the RIP routing protocol [1] and run several instances of a RIP demon under the Linux operating system on the same machine (no virtual machines involved!) – each instance runs as a separate process, and processes communicate through local sockets. You will emulate a small network and explore the response of the RIP protocol to different types of faults.

You will work in groups of two persons, it is not an option to work alone or in larger groups, unless you have **very** convincing reasons (mere convenience is not convincing) and have obtained **explicit approval** by me (normally by e-mail). Submissions by individuals or larger groups without such approval will not be marked. I will help with pairing.

You can use Python, Java or C, and you have to use the socket interface implementation that is available for your programming language. Your program has to be executable under Linux.

It is quite likely that the problem description below leaves a lot of things unclear to you. Please do not hesitate to use the "Question and Answer Forum" on the Learn platform for raising and discussing any unclear issues.

Important: Before reading the following problem description in more detail, it is absolutely essential that you read the RIP specification [1]. The scope of this project is essentially given by Section 3 of [1], none of the extensions in Section 4 is to be included (if you do nonetheless, no credit will be given).

2 Problem Description

You will implement a "routing demon" as a normal userspace program under Linux. Instead of sending its routing packets over real network interfaces, your routing demon will communicate with its peer demons (which run in parallel on the same machine) through local sockets.[1] Your program should be a text mode program, no credit will be given for providing a graphical user interface.

Roughly speaking, your program will operate in three stages:

- At the beginning it reads a configuration file (whose name has to be supplied as a command line parameter – please avoid any interactive reading of the filename) which contains a unique identification for the routing demon instance, the port numbers on which the demon receives routing packets from peer demons (**input ports**), and specifications of the **outputs** towards neighbored routers. Clearly, any output port declared for one router should be an input port of another router. The information in the configuration file is only meant to inform demons about links, the demons internal routing table is still empty at this stage.

- Next the demon creates as many UDP sockets as it has input ports and binds one socket to each input port – no sockets are created for outputs, these only refer to input ports of neighbored routers. One of the input sockets can be used for sending UDP datagrams to neighbors.

- Finally, you will enter an infinite loop in which you react to incoming events (check out the `select()` system call or its equivalent in your favorite programming language.[2]) An incoming event is either a routing packet received from a peer (upon which you might need to update your own routing table, print it on the screen, or possibly send own routing messages to your peers), or it is a timer event upon which you send an unsolicited RIP response message to your peers. Please ensure that you handle each event atomically, i.e. the processing of one event (e.g. a received packet) must not be interrupted by processing another event (e.g. a timer).

All the routing demons will be instantiated on the same Linux machine, so you will have to use the IP address for the local host (`127.0.0.1`). You should use the UDP protocol for routing messages.

One hint for your design: many protocol specifications (and even more so their implementations) are expressed using the formalism of extended finite state

[1] It is not absolutely and strictly necessary to run each demon as a separate process, it is also legitimate to have several threads in the same process. However, it **is** strictly necessary to have the ability to stop and re-start individual routing demons at arbitrary times, and have them loose all their state. Separate processes appear to be the simplest way to achieve this.

[2] One of `select()`'s important properties is that it *blocks* while you wait for events, i.e. your program does not consume CPU time. This property is important and your program needs to have it. Note that `select()` blocks an entire process – if your process contains several threads (e.g. one per router) then *all* threads are blocked, not only the calling one.

machines or finite automata (i.e. state machines / automata that have variables added and where transitions can be conditioned on both an event and the value of one or more variables). This can be a very useful design framework. You would have to identify all possible events, the set of variables, and the set of states. You then have to specify for each state what will happen when when an event comes in. This has to be done for each event.

2.1 The Configuration File

Your program should be a single executable which can be started with a single command line parameter. This parameter will be the filename of a configuration file. The configuration file should be an ASCII file that can be **read and edited by humans**. I recommend to choose a very simple syntax for the configuration file and to keep the parser very easy, no extra credits will be given for an elaborate syntax (nothing XMLish or any other similarly baroque format!), fancy parsing code or the like. However, basic syntax and consistency checks (when a number is expected it should be checked that indeed a number is present and is in the allowed range) **need to be made**. Each instance of the router demon will be started with a separate configuration file.

The configuration file should allow users to set at least the following parameters (one parameter per line, please allow for empty lines and comments):

- Router id, which is the unique id of this router. A suggested format for this parameter is:

  ```
  router-id 1
  ```

  The router id is a positive integer between 1 and 64000. Each router must have its own unique router-id, and thus each router configuration file must have a different value for this parameter.

- Input port numbers: this is the set of port numbers (and underlying sockets) on which the instance of the routing demon will listen for incoming routing packets from peer routing demons. There needs to be a separate input port for each neighbor the router has. A suggested format for this parameter is:

  ```
  input-ports 6110, 6201, 7345
  ```

  i.e. the port numbers could be separated by commas. You do not have to follow precisely this format, but your parser for the configuration should ensure that:

  - All port numbers are positive integers x with $1024 \leq x \leq 64000$ (find out why 1024 was chosen as lower bound!)
  - You can specify arbitrarily many such entries, but all in one line
  - Each port number occurs at most once.

- Outputs: here you specify the "contact information" for neighboured routers, i.e. routers to which a direct link should exist and with which you exchange routing information. A suggested format for this parameter is:

  ```
  outputs 5000-1-1, 5002-5-4
  ```

  As before, different outputs are separated by commas. For one output, for example `5002-5-4`, the first number `5002` specifies the input port number of the peer router, i.e. the port number on which the peer router will listen to packets from the current instance. The second number `5` represents the metric value for the link to the peer router. The third number `4` represents the router-id of the peer router, so that your instance knows which router is really behind this link. The port numbers given here must satisfy the same conditions as the port numbers given for the `input-ports` parameter. Furthermore, none of the port numbers given for the `outputs` parameter should be listed for the `input-ports` parameter. The metric values should conform to the conditions given in [1].

- Values for the timers you are using in your implementation. It is wise to use timer values (for periodic updates and timeouts) smaller than the ones given in the standard, since this can shorten experimentation times. However, please ensure that the ratio of the timer values remains the same ($180/30 = 6$ for timeout vs. periodic, similarly for garbage collection).

The first three parameters (`router-id`, `input-ports`, `outputs`) are mandatory, if one of them is missing your program should stop with an error message. When you set up several configuration files (one for each router), you must ensure manually that the following conditions are met:

- The `router-id`'s of all routers are distinct
- When you want two routers A and B to be neighbored, you should:
  - provide an input port x at B that is also listed as output port of A
  - provide an input port y at A that is also listed as output port of B
  - ensure no other host than A has listed port x as an output port
  - ensure no other host than B has listed port y as an output port
  - ensure no other host than A has listed port y as an input port
  - ensure no other host than B has listed port x as an input
  - and finally, ensure that the metrics that A specifies for its output with port number x is the same as the metric that B specifies for its output port y.

2.2 Exchanging Routing Packets and Route Computations

In this section it is assumed that you are already intimately familiar with the RIP protocol specification [1]. Here we will just give a few hints on which parts of the specification can be ignored, should be simplified, or need to be modified.

- Scope of implementation is essentially Section 3 of [1]. None of the extensions in section 4 is to be included (no extra credit).
- Please implement split-horizon with poisoned reverse. This means that for any update message or unsolicited response a separate packet needs to be created for each neighbor.
- Implement triggered updates only when routes become invalid (i.e. when a router sets the routes metric to 16 for whatever reason, compare end of page 24 and beginning of page 25 in [1]), not for other metric updates or new routes.
- Do not use the UDP port specified in [1], but the port numbers from the configuration file.
- Do not implement request messages, you should use only the periodic and triggered updates.
- The version number is always 2.
- You do not route to subnetworks and do not handle IPv4 addresses, but you only route to the various routers (as identified by their `router-id`). You also do not need to take care of default addresses, host routes or subnet masks.
- RIP response packets always include the entire routing table, and to each neighbor a separate copy of this table is sent (according to the split-horizon-with poisoned reverse rule, which implies that each neighbor might get a different view of the table).
- Do not multicast response messages, just send them to the intended peer through its input port.
- Packet format:
  - Ignore the issue of network byte order
  - Please use the 16-bit wide all-zero field in the RIP packet common header for the `router-id` (since otherwise you would have no identification of the router sending out an update. Furthermore, you should also work with `router-id`'s instead of IPv4 addresses.
  - Please perform consistency checks on incoming packets: have fixed fields the right values? Is the metric in the right range? Non-conforming packets should be dropped (and perhaps a message printed, this helps your debugging).

- Regarding periodic updates: You need to find out how to generate periodic timer events and how to write own handlers for this event. This depends on your choice of programming language. It could also be useful to introduce some randomness to the periodic updates, for example by setting the timer in question randomly to a value that is uniformly distributed over the range $[0.8 \cdot \text{period}, 1.2 \cdot \text{period}]$. Why could such a randomization be useful?

Some other things:

- If a demon has several input ports, it needs to listen to all of them simultaneously. This can be achieved through the `select()` system call or its equivalent in your chosen programming language.

It goes without saying that you should perform various tests with your setup. These tests should show that your routing protocol design and implementation converge to the "right" routing tables (which you will have to establish from inspection) and respond "in the right way" to certain failure events (e.g. one process is shut down and re-started later). To properly conduct these tests it is also important to formulate **in advance** which behaviour your implementation **should** show and to compare the actual outcome against it.

3 Deliverables

To receive marks for the assignment, each pair of students has to do two things:

- Demonstrate your program and explain the source code to me
- Submit a **single `.pdf` file** which includes answers to questions (see below), the source code of your program and the configuration files used in the demonstration. It is sufficient when one member of a team submits this.

3.1 The Demonstration

The demonstrations will take place shortly after assignment submission, dates will be arranged shortly before. A demonstration session will have the following structure:

- The first ten minutes are a live presentation of the program:
  - First we will start up all router instances and check whether the routing tables converge and the resulting routes are "minimum hop". Please make sure that your program generates sufficient output for me to check this – for example, you could print the routing table after each change or each periodic update.
  - Next, we will switch off one or more of the routers and expect that the remaining network converges again into a minimum hop state after a flurry of activity.

  - Finally we will switch the failed router(s) on again and expect to see that the network converges back into the initial state.

- The second ten minutes are devoted to a code walk-through of your program, to be done by the first partner. The second partner of a team will wait outside. **Note**: each partner should be able to explain the entire program!
- The other partner will do the code walk-through in the third ten minutes.

The marking will be based on the following factors:

- The achieved functionality as shown in the first ten minutes
- The contribution percentage of either partner
- The understanding of the code shown in the walk-through

For the demonstration you need to set up configuration files for the example network shown in Figure 1. In the figure, the vertices represent routers and their router-id, the edges represent bi-directional links. The edges are labeled with cost values for the respective link.

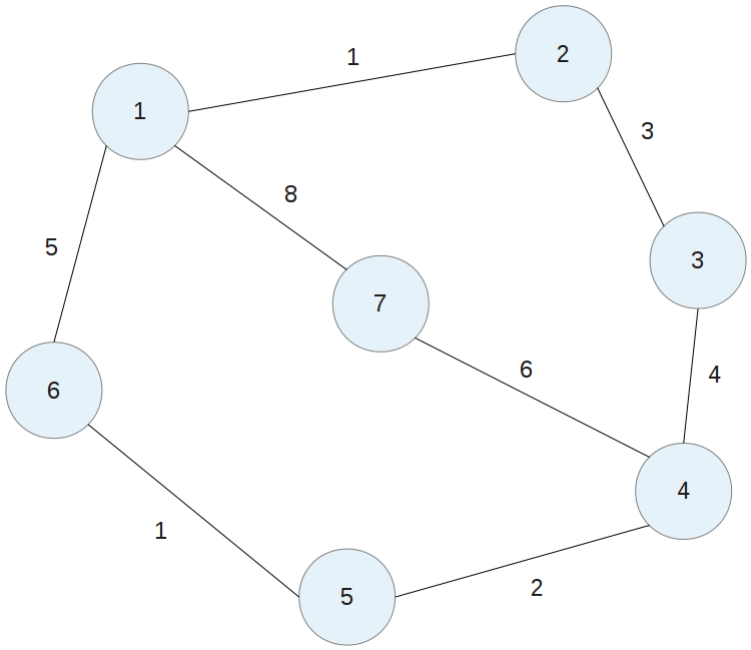

Figure 1: Example network for demonstration

3.2 The Documentation

The documentation needs to be a **single `.pdf` file** for each pair. The file needs to include:

- A title page listing name and student-id of both partners.
- Answers to the following questions (please answer each question with one or two paragraphs at most):
  - A percentage contribution of each partner to the overall project. **Note**: this must be agreed upon by you and your partner, the relative weights will influence grading.
  - Which aspects of your overall program (design or implementation) do you consider particularly well done?
  - Which aspects of your overall program (design or implementation) could be improved?
  - How have you ensured atomicity of event processing?
- There should be substantial discussion (much more than two paragraphs) about the tests you have performed and which conclusions these tests allow about the correctness of your design and implementation.
- The source code of your program. I would appreciate if you use a pretty-printer to generate the listing.
- All configuration files for the example network of Figure 1.

Note: Please make sure that your `.pdf` file contains all fonts. This condition is checked upon submission.

The documentation has to be submitted to the departmental coursework dropbox, see `http://cdb.cosc.canterbury.ac.nz`. The submission deadline is **Friday, May 6, 2016, 11.59pm**. Late submissions are **not** accepted. To be on the safe side, it is recommended to submit preliminary versions, you can submit multiple versions to the course dropbox. I will take the last version before the deadline.

4 Marking

Marking will be based on the following factors:

- Achieved functionality (70%). This includes: convergence to correct routing tables, robustness against station failures, syntax and consistency checks (e.g. correct packet format, correct values and ranges for numbers, handling of read accesses beyond the end of the file etc.), CPU load, ability to read filename parameter from the command line. For this factor first a "raw value" is determined based purely on achieved functionality. Following this, these raw marks will be weighted against the relative contributions of each group member and individual marks will be determined.

- The quality of the submitted responses / documentation, in particular your comments about testing (15%).
- Your ability to explain the source code, its architecture and major functions. This is determined individually and normally weighted with 15%, but if you give the impression that you have not even been remotely present when the software has been created, you will receive significant reductions in marks or even fail the assignment, depending on the case at hand.
- When after a quick glance I find that your source code is particularly ugly and un-organized, I will apply deductions of up to 10% of the achievable marks. These will be applied to both team members equally.

References

[1] G. Malkin. RIP Version 2. *RFC 2453*, 1998.