CMPUT 313 - Lab #3 (7%)

Experiments with Flooding Algorithms
(first draft)

Due: Friday, December 4, 2015, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you more experience with understanding, developing, and simulating distributed routing algorithms using the *cnet* discrete event simulation environment. As well, the assignment gives experience with obtaining simulation results suitable for plotting performance graphs for the developed algorithms.

Context

Routing algorithms that utilize controlled flooding are useful in a number of networking situations. For example, flooding mechanisms may be used for constructing (or reconstructing) routing tables in a network, achieving reliable routing for delivering a given type of traffic messages, and implementing multicast operations. This lab assignment explores in Part 1 certain features of the flooding algorithms included in the `flooding` project directory. In Part 2, the assignment asks for modifying the `flooding2.c` program to incorporate some potential improvements.

Part 1: Exploring the Supplied Flooding Protocols

Perform the suggested lab activities described in the document posted on the course's web page and answer the following questions.

1. In `flooding2.c`, what information is computed by `"ALL_LINKS & ~(1<<arrived_on)"`? How does the program use the expression?

2. In `flooding2.c`, what variables are used in implementing a network layer stop-and-wait protocol? What are the initial values of such variables? How does the implementation initialize such variables?

3. In `flooding2.c`, does varying `MAXHOPS` over the range $[1, 4]$ have an effect on the performance of the algorithm? If yes, identify at least two specific quantities that can be used to describe the resulting effect. Each quantity should be observable either from the global network statistics produced by *cnet*, or the node-specific statistics displayed when a node is clicked (or when any debugging button in a node's window is pressed). Explain your answer.

4. For `flooding2.c`, comment on the behaviour of the program in the following two situations:

   (a) The call to function `CNET_disable_application` is commented out, and the program runs on a network with 8 or more nodes for 5 or more minutes.

(b) The unmodified program runs on a network of 20 or more nodes for 5 or more minutes.

Explain the observed behaviour(s).

5. For `flooding3.c`, explain how a node processes a *relayed* frame (i.e., the node is neither the source nor the destination of the frame) in each of the following cases:

   (a) The node has **not** previously received a frame with a matching source address.

   (b) The node has previously received a frame with a matching source address.

   (c) The node has previously received a frame with a matching destination address.

Part 2: Handling Lost Packets

Note that changing the `dll_basic.c` file to use `CNET_write_physical` instead of `CNET_write_physical_reliable` makes the continuous operation of the three flooding programs dependent on the absence of packet loss events. In this part you are asked to modify the `flooding2.c` program so that the protocol maintains in-order delivery of application layer messages in the presence of packet loss events (e.g., when we set `wan-probframeloss= 1`).

One way to provide the required packet loss recovery feature is to use a stop-and-wait protocol between the two end nodes of each flow stream. Note that the current implementation of `flooding2.c` employs an *incomplete* form of stop-and-wait between the end nodes of each flow stream. The purpose of this incomplete version of the protocol is to control the amount of traffic passing through the network. So, your task is to extend this incomplete implementation to enable recovery from packet loss events. Here are more required implementation details.

1. Implement the above feature as modifications to the `flooding2.c` program. Call the modified program `lab3.c`.

2. When processing a relatively large network, it is possible that `flooding2.c` (or the modified program) to terminate with `ER_TOOBUSY` error. The occurrence of this error indicates the occurrence of packet loss events. The modifications described below allow a program to continue execution in the presence of such packet loss events. The number of errors occurred is kept in a counter, called `count_toobusy`, that is is printed when desired.

   (a) **File dll_basic.h**: Add `"extern int count_toobusy;"`

   (b) **File dll_basic.c**: Declare the counter, and replace `CHECK(CNET_write_physical_reliable(...))` with

   ```
   if (CNET_write_physical(link, (char *)packet, &length) < 0) {
       if (cnet_errno == ER_TOOBUSY) count_toobusy++;
       else CNET_exit(__FILE__, __func__, __LINE__);
   }
   ```

   (c) **File dll_basic.c**: Add `count_toobusy= 0` to function `reboot_DLL(void)`.

   (d) **Printing count_toobusy**: see **DEBUG1 Events** below.

3. For simplicity, you may assume that the number of nodes in the input network is at most 32.

4. **DEBUG0 Events.** In `flooding2.c`, pressing the `DEBUG0` button causes the node to print the contents of its `NL_table`. To help in debugging the program when the command line interface is used, add printing the current simulation time before printing the contents of the `NL_table`.

5. **DEBUG1 Events.** In the modified program, label the `DEBUG1` button in a node's window `"TIMERS info"`. Pressing the button should cause the program to (a) clear the screen by calling function `CNET_clear`, (b) print the node's name and address, (c) print the contents of `count_toobusy`, and (d) for each packet p that is being timed out by the protocol (to recover from packet loss), print the following information: $p.src$, $p.dest$, $p.kind$, $p.seqno$, $p.length$, and the timer identifier (obtained by calling `CNET_start_timer`) associated with monitoring this packet. Use a suitable printing layout of your choice.

6. **PERIODIC Events.** To facilitate monitoring the execution of your program without using the GUI, add an `EV_PERIODIC` handler in `lab3.c`. Invoking the handler should cause a node to print the information printed when the `DEBUG0` button is pressed followed by pressing the `DEBUG1` button.

7. (a) Generate two random connected networks with 10 and 15 nodes, respectively.

   (b) Run `flooding2.c` on each network using the following 4 combinations of parameters: simulation time $\in [10\text{min.}, 30\text{min.}]$, and probframeloss $\in [0, 1]$. In each of the 8 experiments, record the number of (Messages generated, Messages delivered) from the global statistics obtained at the end of the simulation. Present the results in a tabular form, as shown below.

   (c) Repeat the experiments in (b) using your `lab3.c` program, and fill in a similar table.

Network Size	flooding2.c	
	probframeloss= 0	probframeloss= 1
10	10 min. $(\cdots, \cdots)$	10 min. $(\cdots, \cdots)$
	30 min. $(\cdots, \cdots)$	30 min. $(\cdots, \cdots)$
15	10 min. $(\cdots, \cdots)$	10 min. $(\cdots, \cdots)$
	30 min. $(\cdots, \cdots)$	30 min. $(\cdots, \cdots)$

Network Size	lab3.c	
	probframeloss= 0	probframeloss= 1
10	10 min. $(\cdots, \cdots)$	10 min. $(\cdots, \cdots)$
	30 min. $(\cdots, \cdots)$	30 min. $(\cdots, \cdots)$
15	10 min. $(\cdots, \cdots)$	10 min. $(\cdots, \cdots)$
	30 min. $(\cdots, \cdots)$	30 min. $(\cdots, \cdots)$

Deliverables

1. All submitted programs should compile and run on the lab machines.

2. Typeset a project report (two to four pages either in text, HTML, or PDF) with the following (minimal set of) sections:

   – **Objectives:** describe the project objectives and value from your point of view

   – **Answers to Part 1 Questions**

- **Design Overview:** highlight in point-form the important ideas and features of your `lab3.c` program
- **Project Status:** describe the status of your designed program, mention difficulties encountered in the implementation
- **Testing and Results:** Explain how you tested your implementation, and discuss the obtained results
- **Acknowledgments:** acknowledge sources of assistance

3. Combine the project report, with files `lab3.c`, `flooding2.c`, `dll_basic.*`, `nl_table.*`, other possible readme, script/makefile files, and topology files into a single tar archive `'submit.tar'`. Make your submission **self-contained** and ready for testing by the marker when extracted in a fresh directory.

4. Upload your tar archive using the **Lab #3 submission/feedback** link on the course's web page. Late submission (through the above link) is available for 24 hours for a penalty of 10%.

5. It is strongly suggested that you **submit early and submit often**. Only your **last successful submission** will be used for grading.

Marking

Roughly speaking, the breakdown of marks is as follows:

35% : correct answers to Part 1 questions

50% : implementation of the features mentioned in Part 2, and correct program compilation and operation

15% : quality of the project report for Part 2
