Message-passing homework: *Crimson Instant Messenger*

CS 257

October 21, 2005

Administrative details

This will be the only homework in CS 257. It combines the two homework assignments that were proposed at the beginning of class. The work is **due Thursday, November 17** at 11:59PM. Writeups can be left in NR's mailbox or emailed to `cs257@eecs.harvard.edu`. Code should be emailed. Plan to *demonstrate your work in class* Friday, November 18.

The goal of the assignment is to help you learn some basic techniques for building concurrent programs using message passing. The essential elements of this assignment are

- To use model checking to validate critical properties of an implementation
- To build and use working code in a language that supports message passing

Inessential elements include the exact choices of model checker, programming language, and problem. If any inessential element displeases you, please propose an alternative for the course staff to consider.

A satisfying solution to this problem should be well within the reach of a single student. If you wish to work with a group of other students, that is possible, but you will need to propose, and have the course staff approve, two adjustments to the assignment:

- Additional technical elements that justify the extra manpower
- A "management plan" that ensures all students will gain needed expertise

For example, it is not acceptable to divide work between two students such that one student does no model checking.

You have a long lead time and no intermediate milestones. This structure should give you the kind of flexibility that is appropriate for graduate students, whose lives may be disrupted with paper deadlines, qualifying examinations, external visitors, and other high-priority demands. Don't make the mistake of starting a week before the deadline; your life will be unpleasant.

Technical details

Design, model, and implement *Crimson Instant Messenger*.

- At minimum, you should support entering and leaving chats, creating new chats, sending private messages, and so on.
- You are welcome to crib the high-level design from another system such as AIM, Google Talk, Skype, etc.
- Your model and implementation should include both clients and servers. It's not sufficient simply to write a server for an existing client.
- Keep use of a central server to a reasonable minimum. In particular, there should be no central server through which all chat messages pass. (Either create a server per chat, or do chats in peer-to-peer fashion.)
- A single-process solution is acceptable, but you will have more fun if your solution works with multiple processes on multiple machines. If you do tackle a multiple-machine solution, we urge you to use the TCP protocol and to assume reliable message delivery. Please do not expend any effort attempting to make your system robust against machine failure or link failure.[1]

Your implementation should be written in Newsqueak, Concurrent Haskell, Concurrent ML, or (if you are feeling adventurous) Erlang. If you want to use a different programming language, you'll need to have the course staff approve it.

Your models should be written in Promela and validated with SPIN. Promela can model Newsqueak's and CML's channels and Concurrent Haskell's `CVar`s fairly directly. For other abstractions, such as Concurrent Haskell's `MVar`s, CML's first-class events, and Erlang's process-based message passing, you'll need to do some thinking. If you want to use a model checker other than SPIN, you'll need to have the course staff approve it.

It should be obvious what to do with your implementation: build a system you can demo in class. You may have questions about the user interface. A line-oriented user interface is fine, or if you prefer, you can build a graphical user interface. A more interesting thing to do, especially if you are intrigued by some of the concurrent GUI work, is to build a small text-based windowing system on top of `ncurses`. (Rob would have kittens.)

In case it's not obvious what to do with your models, they should address the following problems:

1. Build a model of the incorrect single-chat server designed in class on October 7, and show that it can deliver messages out of order.

[1] On the other hand, if you have taken CS 262 and can assemble a suitable group of students, robustness against failure would be a very interesting property to model, validate, and implement. If you have not taken CS 262, we recommend against it.

2. Show that your system never delivers messages out of order.
3. Show that your system does not drop messages.
4. Show that every message is eventually delivered (or more precisely, that eventually either the message is delivered or the client leaves the chat).
5. Show whether your system cannot guarantee delivery if a client leaves the chat, provided the message is sent before the client leaves. (This will be true if you define both "sending" and "delivery" to be user-interface actions.)
6. Show that a private message is never delivered to anyone other than its intended recipient.
7. Show that distinct clients see a consistent view of the system.
8. Show that your system doesn't deadlock.
9. Show that an unresponsive client doesn't prevent other clients from communicating.
10. Show that your system correctly recovers resources when a client leaves.
11. Show any other property you think is useful or important

Several of these problems are specified quite informally; you'll have to think carefully about how to formalize them in a way that's useful for model checking.

You may not want to use the same model for each property. In particular, for recovery of resources, you may have to model the garbage collector.

Your submission should include all of your models and your implementation as well as a suitable Makefile or mkfile. It should also include a writeup that covers the following points:

- A high-level description of your design
- An explanation of your models
- A description of any techniques or tricks you used to match the modeling language to your intended implementation language and its abstractions
- Descriptions of the more interesting bugs you found using your models
- Descriptions of nontrivial bugs that were not part of your models but that you found in your implementation
- An explanation of your implementation
- An analysis of how the modeling tool and concurrent language support did or did not help you solve this problem

For the in-class demo, please be prepared to give a short demo of the features of your system. Please also be prepared to show us one or two of the most interesting parts of your models or code. You will have about ten minutes. Because this is such a short time, it is imperative that you *practice* your demo before class, so that you will know just what you can cover in the time you have available. (If you are approved to work in a group, the course staff will let you know how much time will be available for your demo.)