CS425/ECE428 — Distributed Systems — Spring 2013

Machine Problem 1 — Multicast

Due: March 11, 2013, 5 p.m.

Your goal will be to create a distributed chat system. Each process will take input messages from a user and then multicast them to other processes. It will also receive messages received via multicast from other users and display them to the user. Your job will be to ensure reliable and ordered transmission in the face of an unreliable network and process failures.

You will be writing a reliable, ordered multicast protocol. As such, you have to implement the multicast interface; in particular, the multicast protocol must support the `multicast`(*message*) call to send a message to a multicast group and `deliver`(*source*,*message*) to deliver a message (labeled with its source) to the process. Your protocol can make use of a basic unicast send protocol: `usend`(*destination*,*message*) and `receive`(*source*,*message*). Note that `deliver` and `receive` act as callbacks.

Requirements

Your multicast implementation must have the following properties:

- **Causal ordering.** The messages must be delivered to each process respecting causal ordering. Recall that for causal ordering, the relevant events are calls to `multicast` and `deliver`. We recommend that you follow the vector timestamp algorithm in the textbook.
- **Reliable multicast.** Your multicast must ensure the three reliable multicast conditions (integrity, validity, and agreement). Note that we will give points for efficiency of your implementations, so a solution such as the reliable multicast algorithm in §15.9 in the text will not get full credit. We recommend that you incorporate ideas from the piggybacked acknowledgment algorithm described in the text instead.
- **Failure detection.** You should implement failure detection to ensure that the multicast algorithm proceeds correctly even if some of the processes crash. You should implement a failure detector and use it to modify your algorithm for causal ordering. You may want to create an extra thread for sending periodic heartbeat / ping messages.

You will be expected to hand in a design document (1–2 pages) that describes the algorithms you used to implement the above functionality and how you made particular design choices. Your implementation code should be well documented and will be graded for clarity.

Skeleton Code

You will be given a skeleton implementation of the chat program, composed of 3 source files. `chat.cc` will implement the user interface for obtaining input from the user and displaying messages from others; it will use the multicast implementation in `mcast.cc`. The skeleton multicast implementation will implement basic multicast (B-multicast), using unicast send functionality provided by `unicast.cc`. The initial implementation of unicast send provides reliable and ordered delivery of messages, but we will provide you with an implementation that introduces delays and loses messages for testing. The implementation can be found in `/class/ece428/mp1` on the EWS machines.

To run the code, first copy the implementation into your directory and then build the program using '`make`':

```
$ cd
$ cp -r /class/ece428/mp1 .
$ cd mp1
$ make
```

You can then run instances of the chat program by typing '`./chat`'. You can run multiple instances in different terminal windows; the underlying code will automatically discover other group members. Note that for this to work, all instances must be running in the same directory and on the same host. (In particular, if you are using `remlnx.ews`, take care that your terminal windows do not get mapped to different hosts, e.g., `linux1` and `linux3`.) You can type messages in one instance and observe that they are displayed in all others.

When you are finished, you can type '`/quit`' in each instance. You should also delete the file `GROUPLIST` that is used internally to keep track of multicast members.

You should modify *only* `mcast.cc`; the interface and unicast implementation should be left as is. This will ensure that we can test your code properly.

Handing In

To hand in your code, you will need to run '`/class/ece428/Handin/handin 1`' from inside your mp1 directory. Please run '`make clean`' in your directory so that you are only submitting source files. Place a copy of your design document, called '`design.pdf`' in your mp1 directory.

You may run hand in multiple times until the deadline; we will grade the latest submission. Submissions after the deadline will not be allowed. Only one submission per group is necessary.

Grading Scheme

- Design document — 20 pts.
- Functionality — 70 pts.
  Your functionality will be tested in the following scenarios:
  - Single node — 10 pts.
  - Reliable, ordered network – 10 pts.
  - Reliable, but out-of-order network – 10 pts.
  - Unreliable, but in-order network – 10 pts.
  - Process failures on reliable, in-order network – 10 pts.
  - Full functionality – 20 pts.
- Code clarity, documentation — 10 pts.