CS 603 (Park) **Assignment III** **Due: Feb. 25 (Wed.), 1998**

PROBLEM 1

Read Sections 3.2–3.5 from Tanenbaum. the paper.

PROBLEM 2

(a) Give a sketch of the correctness proof (i.e., safety/liveness) of Ricart & Agrawala's distributed mutual exclusion algorithm. First and foremost, make sure to have a rigorous proof *structure*. The details, where straightforward, can be glanced over.

(b) Problems 3.3, 3.8, and 3.13 from Tanenbaum.

PROBLEM 3

Implement the vector time stamp causal order multicast algorithm discussed in class. Create two library routines

`causal_order_multicast_send`(), `causal_order_multicast_recv`(),

through which an application interfaces to the multicast enabling middleware. Both functions take three arguments, the first indicating a 4-byte integer group ID, the second indicating a pointer to a read/write buffer, and the third indicating the size of the multicast message to be read/written. `causal_order_multicast_send`() is a nonblocking call, and `causal_order_multicast_recv`() is going to be used inside a user-defined function `any_name_the_user_likes`() which is invoked when SIGUSR1 occurs. SIGUSR1 is generated by a daemon process, `com_server_d`, running at every multicast group member. The daemon processes incoming multicast messages enqueueing them in a holdback queue when necessary.

The multicast enabling middleware consists of the library routines and `com_server_d`; thus, as part of the interface specification, the user is required to register `any_name_the_user_likes`() such that it is asynchronously invoked whenever SIGUSR1 occurs. `causal_order_multicast_send`() interacts with `com_server_d` when sending a multicast message. All communication are to use UDP with a timeout mechanism suitably set to trigger retransmission of lost messages.

Test your application in the following set-up. The driver part of the application to be tested has the basic form

for $i \leftarrow 1$ to 10 do

block SIGUSR1

print("outgoing i")

buf $\leftarrow$ "hostname:i"

`causal_order_multicast_send`(10, buf, |buf|)

unblock SIGUSR1

The receive part, i.e., `any_name_the_user_likes`(), has the form

`causal_order_multicast_recv`(10, buf, |buf|)

print(buf)

Run five multicast group clients on xinu_i, xinu_{i+1}, xinu_{i+2}, xinu_{i+3}, xinu_{i+4}. Hardcode the group member IP/port numbers as before. Perform the experiment three times and verify that causal order multicast is correctly achieved. To verify correctness, show the output generated by the print instructions—per multicast group member (i.e., host)—and indicate via arrows the causal relations between different messages. In so doing, show that causally related messages are perceived in the same order (i.e., *printed* in the same order) at every host. Also, clearly demark concurrent messages.

Note: For your implementation to work correctly, care must be given to block certain signals from interrupting the various signal handlers.

PROBLEM 4

(a) Implement the *bully algorithm* for leader election discussed in class. Use as group members xinu_i, xinu_{i+1}, xinu_{i+2}, xinu_{i+3}, xinu_{i+4}, xinu_{i+5}, where their IDs are used to give a linear order (smaller ID has higher rank). At start-up, hardcode the fact that xinu_i is the group leader and print out its ID at every host and the current time stamp. Subsequently, each group member j polls the current group leader of its up status by sending a *are-you-alive* message every 5 seconds. The group leader, if up, is to respond immediately. If a response is not received in 1 second, two more tries are made resulting in a total elapsed time of about 3 seconds before new leader election is initiated. When a new election is initiated, the initiator is to print whether the initiation of new leader election is due to its own observation of the current leader's (potential) failure, or whether it is due to its receiving a new election message from a lower priority group member. When a new leader has been confirmed (by the broadcast of this fact by the new leader), each host is to print out the new leader's ID along with the current time stamp. Give a high-level sketch of the structure of your system including the names and relations of the principal functions used.

In the first test run, make the process running on xinu_i fail (i.e., `exit`) after 10 seconds after start-up. Make the group member on xinu_{i+1} fail after 15 seconds. Similarly for xinu_{i+2} after 20 seconds. Explain using the traces what is going on.

In the second run, make xinu_i and xinu_{i+1} fail after 10 seconds, and make xinu_{i+2} fail after 14 seconds. Give an interpretation of your trace.

(b) Can you think of a failure event sequence (perhaps involving just a single failure) where the present algorithm may work incorrectly, e.g., by getting stuck without ever electing a new leader? If so, what is this malfunction attributable to? Is it an instrinsic problem of the bully algorithm or an *implementation* feature? How can it be fixed?

Can you think of other failure sequences that may be problematic? What can you conclude about the "fault-tolerance" of the bully algorithm?