CS425/CS525 Final Exam

December 15th, 2005

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are three problems on this exam, each worth 20 points, for a total of 60 points. You have approximately three hours to complete this exam.

1 Consensus by attrition (20 points)

Suppose you are given a *bounded fetch-and-subtract* register that holds a non-negative integer value and supports an operation fetch-and-subtract(k) for each $k > 0$ that (a) sets the value of the register to the previous value minus k, or zero if this result would be negative, and (b) returns the previous value of the register.

Determine the consensus number of bounded fetch-and-subtract under the assumptions that you can use arbitrarily many such objects, that you can supplement them with arbitrarily many multiwriter/multireader read/write registers, that you can initialize all registers of both types to initial values of your choosing, and that the design of the consensus protocol can depend on the number of processes N.

Solution

The consensus number is 2.

To implement 2-process wait-free consensus, use a single fetch-and-subtract register initialized to 1 plus two auxiliary read/write registers to hold the input values of the processes. Each process writes its input to its own register, then performs a fetch-and-subtract(1) on the fetch-and-subtract register. Whichever process gets 1 from the fetch-and-subtract returns its own input; the other process (which gets 0) returns the winning process's input (which it can read from the winning process's read/write register.)

To show that the consensus number is at most 2, observe that any two fetch-and-subtract operations commute: starting from state x, after fetch-and-subtract(k_1) and fetch-and-subtract(k_2) the value in the fetch-and-subtract register is $\max(0, x - k_1 - k_2)$ regardless of the order of the operations.

2 Long-distance agreement (20 points)

Consider an asynchronous message-passing model consisting of N processes $p_1 \dots p_N$ arranged in a line, so that each process i can send messages only to processes $i-1$ and $i+1$ (if they exist). Assume that there are no failures, that local computation takes zero time, and that every message is delivered at most 1 time unit after it is sent no matter how many messages are sent on the same edge.

Now suppose that we wish to solve agreement in this model, where the agreement protocol is triggered by a local *input* event at one or more processes and it terminates when every process executes a local *decide* event. As with all agreement problems, we want Agreement (all processes decide the same value), Termination (all processes eventually decide), and Validity (the common decision value previously appeared in some input). We also want no false starts: the first action of any process should either be an *input* action or the receipt of a message.

Define the time cost of a protocol for this problem as the worst-case time between the first *input* event and the last *decide* event. Give the best upper and lower bounds you can on this time as function of N. Your upper and lower bounds should be *exact*: using no asymptotic notation or hidden constant factors. Ideally, they should also be equal.

Solution

Upper bound

Because there are no failures, we can appoint a leader and have it decide. The natural choice is some process near the middle, say $p_{\lfloor (N+1)/2 \rfloor}$. Upon receiving an input, either directly through an *input* event or indirectly from another process, the process sends the input value along the line toward the leader. The leader takes the first input it receives and broadcasts it back out in both directions as the decision value. The worst case is when the protocol is initiated at p_N; then we pay $2(N - \lfloor (N+1)/2 \rfloor)$ time to send all messages out and back, which is N time units when N is even and $N-1$ time units when N is odd.

Lower bound

Proving an almost-matching lower bound of $N-1$ time units is trivial: if p_1 is the only initiator and it starts at time t_0, then by an easy induction argument,in the worst case p_i doesn't learn of any input until time $t_0+(i-1)$, and in particular p_N doesn't find out until after $N-1$ time units. If p_N nonetheless decides early, its decision value will violate validity in some executions.

But we can actually prove something stronger than this: that N time units are indeed required when N is odd. Consider two slow executions Ξ_0 and Ξ_1, where (a) all messages are delivered after exactly one time unit in each execution; (b) in Ξ_0 only p_1 receives an input and the input is 0; and (c) in Ξ_1 only p_N receives an input and the input is 1. For each of the executions, construct a causal ordering on events in the usual fashion: a send is ordered before a receive, two events of the same process are ordered by time, and other events are partially ordered by the transitive closure of this relation.

Now consider for Ξ_0 the set of all events that precede the *decide(0)* event of p_1 and for Ξ_1 the set of all events that precede the *decide(1)* event of p_N. Consider further the sets of processes S_0 and S_1 at which these events occur; if these two sets of processes do not overlap, then we can construct an execution in which both sets of events occur, violating Agreement.

Because S_0 and S_1 overlap, we must have $|S_0| + |S_1| \geq N+1$, and so at least one of the two sets has size at least $\lceil (N+1)/2 \rceil$, which is $N/2+1$ when N is even. Suppose that it is S_0. Then in order for any event to occur at $p_{N/2+1}$ at all some sequence of messages must travel from the initial input to p_1 to process $p_{N/2+1}$ (taking $N/2$ time units), and the causal ordering implies that an additional sequence of messages travels back from $p_{N/2+1}$ to p_1 before p_1 decides (taking and additional $N/2$ time units). The total time is thus N.

3 Mutex appendages (20 points)

An *append* register supports standard read operations plus an append operation that appends its argument to the list of values already in the register. An *append-and-fetch* register is similar to an append register, except that it returns the value in the register after performing the append operation. Suppose that you have an failure-free asynchronous system with anonymous deterministic processes (i.e., deterministic processes that all run exactly the same code). Prove or disprove each of the following statements:

1. It is possible to solve mutual exclusion using only append registers.
2. It is possible to solve mutual exclusion using only append-and-fetch registers.

In either case, the solution should work for arbitrarily many processes—solving mutual exclusion when $N = 1$ is not interesting. You are also not required in either case to guarantee lockout-freedom.

Clarification given during exam

1. If it helps, you may assume that the processes know N. (It probably doesn't help.)

Solution

1. Disproof: With append registers only, it is not possible to solve mutual exclusion. To prove this, construct a failure-free execution in which the processes never break symmetry. In the initial configuration, all processes have the same state and thus execute either the same read operation or the same append operation; in either case we let all N operations occur in some arbitrary order. If the operations are all reads, all processes read the same value and move to the same new state. If the operations are all appends, then no values are returned and again all processes enter the same new state. (It's also the case that the processes can't tell from the register's state which of the identical append operations went first, but we don't actually need to use this fact.)

   Since we get a fair failure-free execution where all processes move through the same sequence of states, if any process decides it's in its critical section, all do. We thus can't solve mutual exclusion in this model.

2. Since the processes are anonymous, any solution that depends on them having identifiers isn't going to work. But there is a simple solution that requires only appending single bits to the register.

   Each process trying to enter a critical section repeatedly executes an append-and-fetch operation with argument 0; if the append-and-fetch operation returns either a list consisting only of a single 0 or a list whose second-to-last element is 1, the process enters its critical section. To leave the critical section, the process does append-and-fetch(1).