CS425/CS525 Final Exam

May 10th, 2010

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

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

1 Anti-consensus (20 points)

A wait-free *anti-consensus* protocol satisfies the conditions:

Wait-free termination Every process decides in a bounded number of its own steps.

Non-triviality There is at least one process that decides different values in different executions.

Disagreement If at least two processes decide, then some processes decide on different values.

Show that there is no deterministic wait-free anti-consensus protocol using only atomic registers for two processes and two possible output values, but there is one for three processes and three possible output values.

Clarification: You should assume processed have distinct identities.

Solution

No protocol for two: turn an anti-consensus protocol with outputs in $\{0, 1\}$ into a consensus protocol by having one of the processes always negate its output.

A protocol for three: Use a splitter.

Alternatively, have the first two processes always output 0 and 1 (satisfying disagreement), have the second process write to some register, and have the third process output 0 if it doesn't see the second process's write and 1 otherwise (satisfying non-triviality). The existence of this protocol suggests that non-triviality should probably be stronger: every process may output different values in different executions. (But that is not how the question was written.)

2 Odd or even (20 points)

Suppose you have a protocol for a synchronous message-passing ring that is anonymous (all processes run the same code) and uniform (this code is the same for rings of different sizes). Suppose also that the processes are given inputs marking some, but not all, of them as leaders. Give an algorithm for determining if the size of the ring is odd or even, or show that no such algorithm is possible.

Clarification: Assume a bidirectional, oriented ring and a deterministic algorithm.

Solution

Here is an impossibility proof. Suppose there is such an algorithm, and let it correctly decide "odd" on a ring of size $2k + 1$ for some k and some set of leader inputs. Now construct a ring of size $4k + 2$ by pasting two such rings together (assigning the same values to the leader bits in each copy) and run the algorithm on this ring. By the usual symmetry argument, every corresponding process sends the same messages and makes the same decisions in both rings, implying that the processes incorrectly decide the ring of size $4k + 2$ is odd.

3 Implementing atomic snapshot arrays using message-passing (20 points)

Consider the following variant of Attiya-Bar-Noy-Dolev for obtaining snapshots of an array instead of individual register values, in an asynchronous message-passing system with $t < n/4$ crash failures. The data structure we are simulating is an array a consisting of an atomic register $a[i]$ for each process i, with the ability to perform atomic snapshots.

Values are written by sending a set of $\langle i, v, t_i \rangle$ values to all processes, where i specifies the segment $a[i]$ of the array to write, v gives a value for this segment, and t_i is an increasing timestamp used to indicate more recent values. We use a set of values because (as in ABD) some values may be obtained indirectly.

To update segment $a[i]$ with value v, process i generates a new timestamp t_i, sends $\{\langle i, v, t_i \rangle\}$ to all processes, and waits for acknowledgments from at least $3n/4$ processes.

Upon receiving a message containing one or more $\langle i, v, t_i \rangle$ triples, a process updates its copy of $a[i]$ for any i with a higher timestamp than previously seen, and responds with an acknowledgment (we'll assume use of nonces so that it's unambiguous which message is being acknowledged).

To perform a snapshot, a process sends SNAPSHOT to all processes, and waits to receive responses from at least $3n/4$ processes, which will consist of the most recent values of each $a[i]$ known by each of these processes together with their timestamps (it's a set of triples as above). The snapshot process then takes the most recent versions of $a[i]$ for each of these responses and updates its own copy, then sends its entire snapshot vector to all processes and waits to receive at least $3n/4$ acknowledgments. When it has received these acknowledgments, it returns its own copy of $a[i]$ for all i.

Prove or disprove: The above procedure implements an atomic snapshot array in an asynchronous message-passing system with $t < n/4$ crash failures.

Solution

Disproof: Let s_1 and s_2 be processes carrying out snapshots and let w_1 and w_2 be processes carrying out writes. Suppose that each w_i initiates a write of 1 to $a[w_i]$, but all of its messages to other processes are delayed after it updates its own copy $a_{w_i}[w_i]$. Now let each s_i receive responses from $3n/4 - 1$ processes not otherwise mentioned plus w_i. Then s_1 will return a vector with $a[w_1] = 1$ and $a[w_2] = 0$ while s_2 will return a vector with $a[w_1] = 0$ and $a[w_2] = 1$, which is inconsistent. The fact that these vectors are also disseminated throughout at least $3n/4$ other processes is a red herring.

4 Priority queues (20 points)

Let Q be a priority queue whose states are multisets of natural numbers and that has operations enq(v) and deq(), where enq(p) adds a new value v to the queue, and deq() removes and returns the smallest value in the queue, or returns null if the queue is empty. (If there is more than one copy of the smallest value, only one copy is removed.)

What is the consensus number of this object?

Solution

The consensus number is 2. The proof is similar to that for a queue.

To show we can do consensus for $n = 2$, start with a priority queue with a single value in it, and have each process attempt to dequeue this value. If a process gets the value, it decides on its own input; if it gets null, it decides on the other process's input.

To show we can't do consensus for $n = 3$, observe first that starting from any states C of the queue, given any two operations x and y that are both enqueues or both dequeues, the states Cxy and Cyx are identical. This means that a third process can't tell which operation went first, meaning that a pair of enqueues or a pair of dequeues can't get us out of a bivalent configuration in the FLP argument. We can also exclude any split involving two operations on different queues (or other objects) But we still need to consider the case of a dequeue operation d and an enqueue operation e on the same queue Q. This splits into several subcases, depending on the state C of the queue in some bivalent configuration:

1. $C = \{\}$. Then $Ced = Cd = \{\}$, and a third process can't tell which of d or e went first.

2. C is nonempty and $e = \text{enq}(v)$, where v is greater than or equal to the smallest value in C. Then Cde and Ced are identical, and no third process can tell which of d or e went first.

3. C is nonempty and $e = \text{enq}(v)$, where v is less than any value in C. Consider the configurations Ced and Cde. Here the process p_d that performs d can tell which operation went first, because it either obtains v or some other value $v' \neq v$. Kill this process. No other process in Ced or Cde can distinguish the two states without dequeuing whichever of v or v' was not dequeued by p_d. So consider two parallel executions $Ced\sigma$ and $Cde\sigma$ where σ consists of an arbitrary sequence of operations ending with a deq on Q by some process p (if no process ever attempts to dequeue from Q, then we have already won, since the survivors can't distinguish Ced from Cde). Now the state of all objects is the same after $Ced\sigma$ and $Cde\sigma$, and only p_d and p have different states in these two configurations. So any third process is out of luck.