Appendix B

Sample assignments from Fall 2022

Assignments should be uploaded to Canvas in PDF format.

Questions about assignments can be sent to the instructor directly at james.aspnes@gmail.com, or posted to the course Discord.

Do not include any identifying information in your submissions. This will allow grading to be done anonymously.

Make sure that your submissions are readable. You are strongly advised to use LaTeX, Microsoft Word, Google Docs, or similar software to generate typeset solutions. Scanned or photographed handwritten submissions often come out badly, and submissions that are difficult for the grader to read will be penalized.

Sample solutions will appear in this appendix after the assignment is due.

Questions about assignments can be sent to the instructor directly at james.aspnes@gmail.com, or posted to the course Discord.

B.1 Assignment 1: due Thursday 2022-09-22, at 23:59 Eastern US time

B.1.1 Leader election using broadcast

In the usual asynchronous message-passing model, each process can choose to send a message to any of its neighbors. To make our system super-anonymous, suppose that we eliminate the need for a process to know what neighbors it has by replacing these point-to-point channels with a **broadcast channel** where any message that is sent is eventually delivered to every process

(including the sender). This is equivalent to requiring in the standard model that whenever a process sends a message, it sends n copies of the message, one to each possible recipient. As in the standard model, we assume that every copy of a message is delivered after at most 1 time unit, but by default impose no other constraints on the time at which each copy of a message is delivered.

We would like to solve leader election in this model, under various assumptions. By leader election, we mean a protocol in which exactly one process eventually sets its `leader` bit to 1. For each of the conditions below, give an algorithm for solving leader election, prove its correctness, and compute its message complexity and running time; or prove that no such algorithm is possible.

1. An anonymous system in which all processes run the same code and do not have unique IDs.

2. A uniform system with IDs, where uniformity means that the code for each process depends only on its ID and not on the size of the system.

3. A non-uniform system with IDs, where the processes know n.

4. A uniform system with IDs, but where the broadcast channel is replaced by an **ordered broadcast** channel that guarantees for each pair of messages m_1 and m_2, that if m_1 is sent before m_2, each process receives m_1 before it receives m_2.

Solution

For computing message complexity, there is an ambiguity in the problem description: does sending a single broadcast count as n messages or one message? Below, we assume a broadcast counts as n messages, but one message is also a reasonable interpretation, so either assumption is acceptable as long as it is clear.

1. Not possible. Construct a synchronous execution in which we alternate between having all n processes take steps until each sends a message then having all n^2 messages delivered. The usual symmetry argument shows that each process updates to the same state and sends the same messages in each round, so either no process ever declares itself the leader, or they all do.

2. Not possible. Consider a system with two processes p_1 and p_2. Run p_1 but do not deliver any of its messages to p_2. Since this execution is indistinguishable from an execution in which p_1 is the only process, it must eventually set its leader bit. Now run p_2 without delivering any of its messages to or from p_1. It also must eventually set its leader bit. We can now satisfy admissibility by delivering all the undelivered messages, but it's too late: we already have two leaders.

3. Possible. Have each process broadcast its ID then wait to collect n IDs. The process with the smallest ID among these n IDs sets its leader bit. Message complexity is n^2 and time complexity is 1.

4. Possible. Have each process broadcast its ID. If a process receives its own ID before any others, it sets its leader bit. Since the broadcast channel is ordered, only the first process to do a broadcast wins. Message complexity is n^2 and time complexity is 1.

B.1.2 Discovery by flooding

In the usual message-passing model, it is assumed that every process has the ability to communicate directly only with its immediate neighbors in the communication graph. For this problem we will consider model closer to the current Internet, where (in principle) any machine in the network can send a message to any other machine, provided it knows the other machine's IP address.

For each process p_i, let S_i be the set of processes p_j such that p_i knows p_j's address, and let $G = (V, E)$ be the directed graph whose vertices V are all processes and which contains an edge $ij \in E$ for each pair p_i, p_j such that $p_j \in S_i$ in the initial configuration. Assume that p_i knows about itself, so that G includes all the self-loops ii.

We'd like the processes to exchange messages until this graph is complete, with an edge for every pair of processes. The protocol is simple: In each (synchronous) round, every process p_i sends its current list S_i to every process in S_i, then updates S_i to be the union of every message it receives.

Show that if the initial graph G is weakly-connected, then after at most $O(\log n)$ rounds, this protocol reaches a configuration where $S_i = V$ for all i.

Solution

For each r, let S_i^r be the value of S_i after r rounds of messages. Define $G^r = (V, E^r)$ as the graph where V is the set of processes and $ij \in E^r$ if and

only if $p_j \in S_i$. From the definition we have $G^0 = G$.

It is convenient to work with undirected graphs. Let H^r be the *undirected* graph that contains an edge ij if and only if ij and ji are both edges in G^r. Note that H^r is always a subgraph of G^r.

Claim: H^1 is connected. Proof: For each edge $ij \in G^0$, p_i sends $p_i \in S_i$ to p_j, so p_j updates S_j^1 to include ji. So H^1 contains the undirected version of G^0 as a subgraph. Since G^0 is weakly connected, H^0 is connected.

Because H^1 is connected, there is a path in H^1 between any two nodes, and the diameter $d(H^1)$ of H^1 is at most $n-1$. We now show that each round of the protocol reduces the diameter of H by roughly half.

Claim: If uv and vw are both edges in H^r, then uw is an edge in H^{r+1}. Proof: From the definition of H^r, we have $\{u, w\} \subseteq S_v^r$. So both of u and w add the other upon receiving S_v^r from v.

Now consider arbitrary $u, v \in H^r$ with $d(u, v) = m$. This means that there is a path $u = u_0 u_1 \dots u_m = v$ in H^r. From the claim, we have that $u = u_0 u_2 u_4 \dots u_m = v$ is a path in H^{r+1} if m is even, and $u = u_0 u_2 u_4 \dots u_{m-1} u_m = v$ is a path in H^{r+1} if m is odd. In either case we have $d_{H_{r+1}}(u, v) \le \lceil m/2 \rceil$. It follows that $d(H^{r+1}) = \max_{u,v} d_{H_{r+1}}(u, v) \le \max_{u,v} \lceil d_{H_r}(u, v)/2 \rceil \le \lceil d(H^r)/2 \rceil$.

A simple induction on r shows that if $d(H^1) \le 2^k$, then $d(H^r) \le \min(1, 2^{k-r+1})$. In particular for $r = \lceil \lg n \rceil + 1$ we have $d(H^r) \le 1$, which shows that there is an edge between every pair of nodes in H^r. Since H^r is defined to contain ij if and only if ij and ji are edges in G^r, it follows that G^r is complete for $r = \lceil \lg n \rceil + 1 = O(\log n)$.

B.2 Assignment 2: due Thursday 2022-10-06, at 23:59 Eastern US time

B.2.1 Maximum consensus

Suppose you have a synchronous message-passing system with n processes that may experience up to f crash failures. Each process p_i starts with an input x_i that is an arbitrarily-large natural number. What is the minimum number of rounds needed to solve each of the following problems in the worst case as a function of f? In each case, provide matching upper and lower bounds for sufficiently large n.

1. Each non-faulty process p_i outputs a value y_i such that (a) $y_i = x_j$ for some process p_j, and (b) $y_i \ge x_j$ for all non-faulty processes p_j.

2. As above, but in addition $y_i = y_j$ for all non-faulty processes i and j.

Solution

1. One round is enough. Each process sends x_i to all processes (including itself), and each process returns y_i equal to the largest of all x_j it received.

   Condition (a) follows immediately from y_i being equal to some x_j. For (b), if p_j is non-faulty, p_i receives x_j from p_j, so it returns either x_j or some larger $x_{j'}$.

   For the lower bound, if a protocol uses zero rounds, then no messages are sent. If process p_i decides x_i in some execution, then for $n \geq 2$ there exists an execution indistinguishable to p_i from this one, where some non-faulty p_j with $j \neq i$ has $x_j > x_i$, violating (b). Similarly, if p_i decides a value $y_i \neq x_i$, there exists an indistinguishable execution where no process has y_i as its input value, violating (a).

2. Here we need $f + 1$ rounds. For the lower bound we can reduce from synchronous consensus and apply Dolev-Strong ([DS83]; see also §9.3). To solve consensus using this problem, have each process p_i decide on y_i. This satisfies validity from (a) and agreement from the added condition that $y_i = y_j$ for all non-faulty i and j. So if we have an algorithm that uses less than $f + 1$ rounds, we get an algorithm for consensus that also uses less than $f + 1$ rounds, contradicting the known lower bound for consensus.

   For the upper bound, we can use the flooding mechanism from Dolev-Strong ([DS83]; see also §9.2). This guarantees that after $f + 1$ rounds, every non-faulty process obtains the same set S of input values, which includes the inputs of all non-faulty processes. So taking $\max S$ gives a common return value for all non-faulty processes that satisfies both (a) and (b).

B.2.2 Colorful Byzantine agreement

Consider a synchronous system with n processes, each of which is labeled with one of four colors: red, green, blue, or yellow. The processes have unique IDs that are known to all the other processes, and all processes know which processes have which color.

The adversary can turn as many processes as it likes Byzantine, provided that all the processes corrupted by the adversary are of the same color.

Prove or disprove: It is possible to solve Byzantine agreement in this system for any number of processes $n \geq 4$ using any assignment of colors that gives at least one process of each color.

Solution

Possible. The idea is to reduce the problem to four processes of which at most one is Byzantine, then use any Byzantine agreement algorithm that tolerates $f < n/3$ Byzantine faults to solve agreement. One possibility would be exponential information gathering [PSL80] (see §10.2.1), since we don't particularly care about anything but fault tolerance and 4 is a constant anyway.

For each color group, let the process with maximum ID represent the group (this does not require any rounds of communication under the assumption that all IDs are known to all processes). We then have four representatives that can execute EIG in $f + 1 = 2$ rounds to solve Byzantine agreement among themselves. Each representative then broadcasts its decision value to all n processes, and each non-faulty process decides on the value broadcast by the majority of representatives. (Note that it is not enough for a process to follow its own representative, because there may be non-faulty processes within the faulty group.)

We would like to show that this algorithm solves Byzantine agreement for all n processes. Termination is immediate. For validity, if all non-faulty processes have the same input v, then so do the three non-faulty representatives; validity in the four-process protocols implies that all three non-faulty representatives broadcast this value and thus all non-faulty processes decide it. Agreement is similar: because all three non-faulty representatives agree on the same value v, each non-faulty process will see a majority for v and decide on v.

B.3 Assignment 3: due Thursday 2022-10-27, at 23:59 Eastern US time

B.3.1 A census of failure

Suppose we have an asynchronous message passing system with crash failures, and we want to implement an oracle that returns a count of the number of processes that haven't crashed yet. Define a **census protocol** to be a protocol that stores at every point in the execution a value c_i at each process p_i, such that (a) $c_i \geq n - f$ always, where n is the number of processes in

the system and f is the number of processes that have crashed so far, and (b) once f converges to a fixed value, c_i eventually converges to $n - f$. These properties should hold for every non-crashed process p_i.

Prove or disprove each of the following statements. In each case assume that we have an asynchronous message-passing system with a complete communication graph, deterministic processes, and crash failures modeled as explicit crash events, and that any implementation must work for arbitrarily large n (which is known to the processes).

1. It is possible to implement a census protocol without using a failure detector.

2. It is possible to implement a census protocol using an eventually perfect ($\Diamond P$) failure detector.

3. It is possible to implement a census protocol using a perfect (P) failure detector.

Solution

1. Disproof: With no failure detector, consider two executions of a two-process system. In one execution, process p_1 takes no steps because it crashes immediately. In the other, p_1 takes no steps for a very long time.

   If p_2 eventually sets c_2 to 1, this violations $c_2 \geq n - f$ in the execution where p_1 has not crashed.

   If p_2 does not eventually set c_2 to 1, this violations c_2 converging to $n - f$ in the execution where p_1 has crashed.

2. Disproof: Consider the two executions in the previous case, and suppose that $\Diamond P$ correctly suspects p_1 throughout the crash execution and incorrectly suspects p_1 in the no-crash execution.

   If p_2 sets c_2 to 1, it violates (a) again in the no-crash execution, and afterwards we can both wake up p_1 and have $\Diamond P$ stop suspecting p_1.

   If p_2 doesn't set c_2 to 1, it violates (b) in the crash execution.

3. Proof: Recall that P eventually permanently suspects every crashed process and never suspects a process before it crashes. So have each process p_i set c_i to $n - f_i$, where f_i is the number of processes that p_i's instance of P currently suspects. Because P only suspects crashed

processes, $f_i \leq f$ and thus $c_i = n - f_i \geq n - f$, satisfying (a). Because P eventually permanently suspects all crashed processes, once every process that will crash has crashed, P will eventually suspect all of them at each p_i. This gives $f_i = f$ and $c_i = n - f_i = n - f$.

B.3.2 Distributed shared memory with Byzantine servers

Consider the following modification to the usual asynchronous message-passing model:

1. There are m clients, and any of them may crash at any time.
2. There are n servers. These do not crash, but up to f of them may be Byzantine.

We would like to have a linearizable implementation of a single-writer multi-reader register in this model, where the single writer and multiple readers are all clients, and any operation by a non-faulty client eventually finishes. Show that there is a constant c such that this is possible for $n \geq cf + 1$.

Solution

We can do this when $n \geq 4f + 1$ by modifying ABD (see §17.2).

To make things easier, we will assume that the honest servers keep track of every timestamp-value pair $\langle t, v \rangle$ they have every received, instead of just the one with the maximum timestamp. Upon receiving a `read`(u) message, the server responds with its entire list (including $\langle t, v \rangle$ if it wasn't there already).

To perform a write operation with value v, the writer increments its local timestamp t, sends `write`(t, v) to all servers, and waits for $n - f$ acknowledgments.

To perform a read operation, a reader sends `read`(u) to all servers, waits for $n - f$ replies, and then chooses a pair $\langle t, v \rangle$ that (a) is sent by at least $f + 1$ servers, and (b) has the largest t out of all such pairs. If there is no pair sent by $f + 1$ servers, the reader returns the default initial register value $\perp$. Otherwise, it sends `write`(t, v) to all servers, waits for $n - f$ acknowledgments, then returns v.

To show this gives a linearizable implementation of a single-writer multi-reader register, we will largely follow the original proof for ABD, constructing an explicit linearization of any complete execution. We start with a simple invariant:

Lemma B.3.1. *Let $\langle t, v\rangle$ be a pair that is (a) in some honest server's list, (b) in a* `write`(t, v) *message, or (c) adopted by a reader. Then $\langle t, v\rangle$ was previously sent by the writer.*

Proof. It is easy to see that if (b) and (c) hold in some configuration, then (a) and (b) hold in any successor configuration, since we can only add a tuple to an honest server if it was in a `write`(t, v) message and we can only generate a `write`(t, v) message if $\langle t, v\rangle$ is sent by the writer or was previously adopted by a reader. To show that (c) holds, observe that if a reader adopts $\langle t, v\rangle$, it must first receive it from $f + 1$ servers. At least one of these servers is honest, so (a) applies. □

For any operation a, let $t(a)$ be the timestamp of the pair $\langle t, v\rangle$ that a sends in its `write`(t, v) messages. Observe that if a finishes, then it receives acknowledgements from $n - f$ servers of which at least $n - 2f$ are not faulty: this implies that by the time a finishes, at least $n - f$ servers have $\langle t, v\rangle$ in their lists. If b is a read operation with $a <_H b$, then b receives responses from at least $n - 3f$ of these servers. With $n \geq 4f + 1$, this is at least $f + 1$. So b either adopts $\langle t, v\rangle$ or adopts some other $\langle t', v'\rangle$ with $t' > t$. So whenever $a <_H b$, $t(a) \leq t(b)$.

To define $<_S$, a before b if (1) $t(a) < t(b)$ (which we've just shown is consistent with $<_H$); or (2) $t(a) = t(b)$, a is a write, and b is a read (which is consistent with $<_H$ by Lemma B.3.1); or (3) $t(a) = t(b)$, both operations are reads, and $a <_H b$ (definitely consistent with $<_H$!). Then extend the resulting partial order to a total order. As in the original ABD algorithm, we get a sequence of blocks of operations where all operations in a block have the same $\langle t, v\rangle$ pair, and the first operation in each block (except possibly the first block) is a write of v and the rest are reads that return v. So the resulting sequential execution is consistent both with H and the specification of a register, and we have shown that the implementation is linearizable.

B.4 Assignment 4: due Thursday 2022-11-10, at 23:59 Eastern US time

B.4.1 Arithmetic registers

An **arithmetic register** holds an integer value and supports operations `read`(), `add`(x), and `multiply`(x), where `read`() returns the current value of the register; `add`(x) updates the current value by adding x to it; and

`multiply`(x) updates the current value by multiplying it by x. The `add` and `multiply` operations do not return a value.

Suppose that arithmetic registers come in two flavors: a **signed** arithmetic register can hold any integer value and allows any integer argument to `add` or `multiply`, while an **unsigned** arithmetic register holds only non-negative integer values and allows only non-negative integer arguments.

Prove or disprove: There exists a deterministic, wait-free, linearizable implementation of a signed arithmetic register from unsigned arithmetic registers and ordinary atomic registers.

Solution

Proof: We'll show that an unsigned arithmetic register implements consensus for any fixed number of processes n, then use universality of consensus to get an implementation of a signed arithmetic register.

The consensus construction follows a similar argument of Ellen *et al.* [EGSZ20] for registers supporting multiplication and decrement, but we have to be a little careful to only use non-negative values. Start with a single unsigned arithmetic register r initialized to 1. A process with input 0 applies `add`(1) to r. A process with input 1 applies `multiply`($n+2$) to r, where n is the number of processes.

Consider some sequence of operations s, and let a_i be the number of calls to `add`(1) in s that are followed by exactly i calls to `multiply`($n+2$) in s. Let k be the total number of calls to `multiply`($n+2$) in s. Then it is easily shown by induction on the length of s that the value of the register at the end of s is given by $r = (a_k + 1)(n+2)^k + \sum_{i=0}^{k-1} a_i(n+2)^i$.

Since each coefficient in this expansion is at most $n+1$, we can recover the expansion uniquely from r. The value a_k will be nonzero if and only if the first operation on the register was `add`(1). Since this holds for any sequence of operations, any process reading the register can determine whether an adder or multiplier went first, and so all processes can return 0 in the first case and 1 in the second.

Now apply Herlihy's universal construction to implement a signed arithmetic register.

(With some tinkering, we can even drop the requirement for atomic registers by showing that they can be implemented from unsigned arithmetic registers, but this is not required by the problem.)

B.4.2 Counting to two

Let us say that we can count to k with m registers for n processes if there is a deterministic, wait-free, linearizable, one-shot implementation of a k-bounded counter from m registers that works for n processes. A k-bounded register starts at 0, has a read operation that returns its current value, and has an increment operation that increases the value by 1 unless it is already k. It is one-shot if each process is only allowed to call the increment operation at most once.

It is easy to show that we can count to 1 for any number of processes using 1 register: start with a 0 in the register, and implement an increment by writing 1. It is also straightforward to count to any value k for n processes using n registers: give a register to each process; implement increment by writing 1 to my register; and sum over a collect to get a number of increments s, returning $\min(k, s)$ to enforce k-boundedness.

Prove or disprove: We can count to 2 with 3 registers for 4 processes.

Solution

Proof: In fact, we can do this for any n, not just $n = 4$.

Use two of the three registers to build a splitter (Algorithm 18.6). The third register, initially 0, will be a flag indicating at least two increments.

To do an increment: Try to win the splitter. If I win, I am done. If not, write 1 to the flag.

To do a read: Check door. If it's open, assume no increments have finished yet and return 0. If it's closed, use the flag to decide whether to return 1 or 2.

Code is given in Algorithm B.1.

Since each operation does at most a constant number of steps, this is clearly wait-free. But we need to show that it is linearizable. We'll use linearization points.

For a read that returns 0: linearize it at the point where it reads door and sees open.

For any other read: linearize it at the point where it reads flag.

This orders all reads that return 0 (when the door is still open) before all reads that return 1 or 2; and orders all reads that return 1 (when the flag is not yet set) before all reads that return 2. So now we just need to fit in some increments to justify the changes.

If there is an increment I_1 that wins the splitter and does not set the flag, assign its linearization point to the step where the door closes (whether

```
   shared data:
 1 atomic register race, big enough to hold an ID, initially ⊥
 2 atomic register door, big enough to hold a bit, initially open
 3 atomic register flag, big enough to hold a bit, initially 0
 4 procedure increment(id)
 5     race ← id
 6     if door = closed then
 7         flag ← 1
 8     door ← closed
 9     if race ≠ id then
10         flag ← 1
11 procedure read
12     if door = open then
13         return 0
14     else if flag = 0 do
15         return 1
16     else
17         return 2
```

Algorithm B.1: Counting to 2 with a splitter

I_1 closes the door or not). Then I_1 linearizes between all reads that return 0 and all reads that return 1 or 2. Because every other increment loses the splitter, every other increment sets the flag; make each such increment's linearization point be the step where it sets the flag. The first such increment I_2 linearizes between all reads that return 0 or 1 and all reads that return 2.

If no increment wins the splitter, then no increment finishes before setting the flag, at the point where the flag is first set there are at least two increments in progress and none have already finished. Let I_1 be one of these increments that starts before the door closes, and assign its linearization point to the step where the door closes. Let I_2 be any other increment in progress when the flag is first set, and assign its linearization point to when the flag is first set. Assign the linearization points of any other increment anywhere during its execution interval that is after I_2's. Again we get one increment linearized between the 0 and 1 reads, and at least one between the 1 and 2 reads. We are done.

B.5 Assignment 5: due Monday 2022-12-05, at 23:59 Eastern US time

B.5.1 A hidden counter

Consider a system with n processes; n single-writer multi-reader atomic registers, one for each process; and a counter that can be incremented by any process but that can be read by nobody. We would like a wait-free protocol that results in the counter being incremented by at least $f(n)$ using as few total operations, across all processes, as possible, counting both increment operations on the counter and read and write operations on the registers.

In this context, wait-freedom means that a process can only return when it is sure that $f(n)$ increments have been done, which may, in the worst case, require it to do all $f(n)$ increments by itself. A process that returns is scheduled for no more operations.

1. Show that $O(n^2)$ total operations are sufficient to increment the counter at least n^2 times.

2. Show that $T(n)$ total operations are sufficient to increment the counter at least n times, for some $T(n) = o(n^2)$.

Solution

1. We'll have each process i alternate between incrementing the counter and writing out the total number of increments it has done so far to its register r_i.

   After n increments, the process will read all the registers r_j, and if $\sum r_j \geq n^2$, return.

   This gives an amortized cost of 3 operations per increment, so as long as we only do $O(n^2)$ increments, we are fine. To show this, observe that once the total value in the registers exceeds n^2, each process does at most n increments before it re-reads the registers, for at most n^2 extra increments.

2. There are a number of ways to do this. One simple approach is to divide the processes into groups of size $k = \sqrt{n}$, and have each group independently do at least $n = k^2$ increments using the algorithm from the previous case. This costs $O(n)$ operations per group, or $O(n^{3/2})$ operations total.

B.5.2 One register to rule them all

This problem was nearly identical to Problem C.5.1 from 2020 and has been withdrawn. Any submission for this assignment will be graded as if a complete solution to this problem had been provided.

B.6 CPSC 565 student presentations

Students taking CPSC 565, the graduate version of the class, are expected to give a 5-minute presentation on a recent paper in the theory of distributed computing.

The choice of paper to present should be made in consultation with the instructor. To a first approximation, any paper from PODC, DISC, or a similar conference in the last two or three years (that is not otherwise covered in class) should work.

Please use the form at https://forms.gle/cmVwdhTF6oheWJbr9 to submit your proposed paper and preferred presentation time.

To get you started, the proceedings for PODC 2022 can be found at https://dl.acm.org/doi/proceedings/10.1145/3519270. The detailed

program for DISC 2021 can be found at http://www.disc-conference.org/wp/disc2021/program/#detailed-program.[1]

Because of the limited presentation time, you are not required to get into all of the technical details of the paper, but your presentation should include:[2]

1. Title, authors, and date and venue of publication of the paper.
2. A high-level description of a central result. Unlike a talk for a general audience, you can assume that your listeners know at least everything that we've talked about so far in the class.
3. A description of where this result fits into the literature (e.g., solves an open problem previously proposed in [...], improves on the previous best running time for an algorithm from [...], gives a lower bound or impossibility result for a problem previously proposed by [...], opens up a new area of research for studying [...]), and why it is interesting and/or hard.
4. A description (also possibly high-level) of the main technical mechanism(s) used to get the result.

You do not have to prepare slides for your presentation if you would prefer to use the chalkboard (or the whiteboard feature in Zoom), but you should make sure to practice it in advance to make sure it fits in the allocated time. The instructor will be happy to offer feedback on draft versions of slides if available far enough before the actual presentation date.

Relevant dates:

2022-11-28 Paper selection due.

2022-12-06, 2022-12-07, and 2022-12-08 Presentations. Tuesday and Thursday presentations will be during the normal class time. Wednesday presentations will be in the Zoom overflow session.

[1] DISC 2022 has a program up, but doesn't seem to have paper downloads available yet. Looking in Google Scholar may be helpful if you see any papers you like.

[2] Literary theorists will recognize this as a three-act structure (preceded by a title card): introduce the main character, make their life difficult, then resolve their problems in time for the final curtain. This is not the only way to organize a talk, but if done right it has the advantage of keeping the audience awake.

Appendix C

Sample assignments from Spring 2020

C.1 Assignment 1: due Wednesday, 2020-09-23, at 5:00pm Eastern US time

C.1.1 A token-passing game

Suppose we have an asynchronous bidirectional message-passing network in the form of a connected graph, where initially m of the n nodes possess a token, represented by a local variable hasToken being set to true. We'd like to be able to move the tokens around, while preserving the total number of tokens.

1. Show that no algorithm that allows tokens to move can guarantee that there are exactly m tokens in any reachable configuration.

2. Give an algorithm that satisfies the following two properties, starting with a configuration with m tokens:

   (a) *Safety:* In any reachable configuration, there are at most m tokens. You should give an explicit invariant that implies this, and show that any transition of your algorithm preserves the invariant.

   (b) *Liveness:* From any reachable configuration C_0, for any subset S of the processes with $|S| = m$, there exists an execution starting in C_0 that ends with a configuration in which every process in S has a token.[1]

[1] Strictly speaking, this is a lot weaker than the usual definition of liveness, because it

To keep things simple, you may assume that the processes can make non-deterministic choices. For example, a process p might choose arbitrarily between sending a message to a neighbor q or to a different neighbor r, and each choice leads to a different possible execution.

Solution

1. Suppose that we are preserving total tokens. Consider some transition between configurations C_1 and C_2. If some process switches hasToken from 1 to 0 between these configurations, then some other process must switch hasToken from 0 to 1. But the definition of delivery events in the asynchronous message-passing model only allows one process at a time to change its state. It follows that no process can change hasToken from 1 to 0 in any transition, so tokens can't move.

2. Consider the following algorithm:

   - At any time, a process with hasToken $= 1$ may send a message takeThis to any of its neighbors and set hasToken $= 0$.
   - A process that receives takeThis when hasToken $= 1$ sends takeThis to any of its neighbors. A process that receives takeThis when hasToken $= 0$ may either set hasToken $= 1$ or send takeThis to any of its neighbors.

   Let us show that this has the desired properties:

   (a) *Safety:* Our invariant will be that the sum of the number of processes with hasToken $= 1$ plus the number of takeThis messages in transit will be m.
   The invariant holds in the initial configuration because there are exactly m processes with hasToken $= 1$ and no message in transit.
   It is preserved by transitions, because in each possible transition, either:
      i. Some process changes hasToken $= 1$ to hasToken $= 0$ and generates a takeThis message;
      ii. Some process changes hasToken $= 0$ to hasToken $= 1$ while consuming a takeThis message; or

effectively assumes that the adversary is cooperating with us. In retrospect I should have written this as "for any admissible adversary strategy, there is a sequence of nondeterministic choices by the algorithm that causes the execution to reach a desired configuration." But I didn't write this, and so it's fine to answer the problem I did write.

iii. Some process consumes a takeThis message but generates a new takeThis message.

In each case, the total number of tokens plus messages is preserved.

(b) *Liveness:* For any configuration C, let $T(C)$ be the set of processes with hasToken $= 1$. We will argue that if $T(C) \neq S$, there exists a partial execution that increases $|T(C) \cap S|$ by 1. First pick some $p \in S \setminus T(C)$. Now consider two cases:

i. If there is at least one takeThis message t in transit, apply the following strategy. Deliver t. If the recipient of t is p, set p.hasToken $= 1$. If not, have the recipient send takeThis to some neighbor that is closer to p than it is. Repeat this process until a takeThis message reaches p.

ii. If there is no takeThis message in transit, generate a takeThis message at some $q \in T(C) \setminus S$ while setting q.hasToken to 0. Then apply the previous case.

For any configuration with $T(C) \neq S$, at least one of these two conditions will hold because of the safety property.
Each partial execution defined above increases $|T(C) \cap S|$ by one, and we can only increase $|T(C) \cap S|$ at most m times because $|S| = m$, so after at most m of these partial executions we reach a configuration with $T(C) = S$.

C.1.2 A load-balancing problem

Consider a two-way message-passing ring with $n = mk$ nodes, where $m > 1$ and k is odd. Nodes at positions $0, k, 2k, \ldots, (m-1)k$ are initially marked as leaders, while nodes at other positions are followers. All nodes have a sense of direction, and can distinguish their left neighbor from their right, but they do not have any other ID information.

Algorithm C.1 is intended to allow the leaders to recruit followers. It is not hard to show that every follower eventually adds itself to a tree of parent pointers rooted at some leader. We would like all of these trees to contain roughly the same number of nodes.

1. Suppose we run this algorithm in a synchronous system. What is the minimum and maximum possible size of a tree?

2. Suppose instead we run the algorithm in an asynchronous system. Now what is the minimum and maximum possible size of a tree?

```
1  initially do
2      if I am a leader then
3          parent ← id
4          send recruit to both neighbors
5      else
6          parent ← ⊥
7  upon receiving recruit from p do
8      if parent = ⊥ then
9          parent ← p
10         send recruit to my neighbor who is not p
```

Algorithm C.1: Recruiting algorithm for Problem C.1.2.

3. Give an algorithm for the asynchronous version of this model that guarantees that all trees are the same size.

Solution

1. In a synchronous execution, we can prove by induction that for each t with $0 \leq t \leq \frac{k-1}{2}$, and each $0 \leq i \leq m-1$, each node at position $ik \pm t$ joins the tree rooted at ik at time t. This puts exactly k nodes in each tree.

2. In an asynchronous execution, by rushing messages from 0, we can recruit all nodes in the range $[-k+1, k-1]$ to the 0 tree before any other messages are delivered. Conversely, each leader ik can't recruit nodes $(i-1)k$ or $(i+1)k$, because these are leaders. So the maximum size of any tree is $2k-1$.

   For the minimum size, suppose we rush all messages from nodes k and $(m-1)k$. Then nodes 1 and $m-1$ are recruited into the trees rooted at these nodes before either message from 0 is delivered. This shows that there are executions with a minimum tree size of 1.

3. The easiest fix may be to have each leader initially send just one recruit message to the right. For each i, this recruits all agents $ik, \ldots, ik+(k-1)$ to a tree of size k rooted at ik.

C.2 Assignment 2: due Wednesday, 2020-10-07, at 5:00pm Eastern US time

C.2.1 Synchronous agreement with limited broadcast

Suppose that we are given a synchronous message passing system on a complete network in which messages are replaced by k-way broadcasts, where the recipient is replaced by a recipient list of up to k processes. Suppose further that when a process crashes in round r, each of its round-r messages is either delivered to all of the processes on the message's recipient list or to none of them. A process can send as many messages as it likes to as many groups as it likes, but if it crashes in some round, any subset of the messages sent in that round may be lost.

Show that for it is possible to solve agreement in this model in $O(f/k)$ rounds, assuming $n > f$.

Solution

We'll use the flooding algorithm of Dolev and Strong [DS83] (see §9.2), but replace sending S to all n processes in each round with sending S to all $\binom{n}{k}$ possible recipient lists. As in the original algorithm, we want to prove that after some round with few enough failures, all the non-faulty processes have the same set.

Let S_i^r be the set stored by process i after r rounds. Suppose there is some round $r+1$ in which fewer than k processes fail. Then every recipient list in round r includes a process that does not fail in round $r+1$. Let L be the set of processes that successfully deliver a message to at least one recipient list in round r, and let $S = \cup_{i \in L} S_i^r$. Then for each value $v \in S$, there is some process that receives v during round r, does not crash in round $r+1$, and so retransmits v to all processes in round $r+1$, causing it to be added to S_i^{r+2}. On the other hand, for any $v \notin S$, v is not transmitted to any recipient list in round r, which means that no non-faulty process i includes v in S_i^{r+1}. So $S \subseteq S_i^{r+2} \subseteq \cup_j S_j^{r+1} \subseteq S$ for all i, and the usual induction argument shows that $S_i^{r'}$ continues to equal S for all non-faulty i and all $r' \geq r+2$.

We can have at most $\lfloor f/k \rfloor$ rounds with $\geq k$ crashes before we run out, so the latest possible round in which we have fewer than k crashes is is $r = \lfloor f/k \rfloor + 1$, giving agreement after $\lfloor f/k \rfloor + 2$ rounds (since we don't need to send any messages in round $r+2$).

(With some tinkering, it is not too hard to adapt the Dolev-Strong lower

bound to get a $\lfloor f/k \rfloor + 1$ lower bound for this model. The main issue is now we have to crash k processes fully in round $r + 1$ before we can remove one outgoing broadcast from a process in round r, which means we need to budget tk failures to break a t-round protocol. The details are otherwise pretty much the same as described in §9.3.)

C.2.2 Asynchronous agreement with limited failures

Algorithm C.2 describes an algorithm for asynchronous agreement with f crash failures in a fully-connected message-passing network. The idea is to collect values from $n - f$ other processes in each of m rounds, and then decide on the smallest value collected.

```
1 preference ← input
2 for i ← 1 to m do
3 |   send ⟨i, preference⟩ to all processes
4 |   wait to receive ⟨i, v⟩ from n − f processes
5 |   for each ⟨i, v⟩ received do
6 |   |   preference ← min(preference, v)
7 decide preference
```

Algorithm C.2: Candidate algorithm for asynchronous agreement

The value m is a parameter of the algorithm and may depend on n and f.

As usual, when waiting for messages from round i, any messages delivered for with other round numbers will be buffered internally and processed when the algorithm is ready for them.

Note that when a process sends a message to all process, that includes itself.

Show that, for any n and $0 < f < n/2$, there exists a value of m such that Algorithm C.2 satisfies agreement, termination, and validity; or show how to construct an execution for any n, $0 < f < n/2$, and m that causes Algorithm C.2 to fail at least one of these requirements.

Solution

We know from the FLP bound ([FLP85], Chapter 11) that Algorithm C.2 can't work. So the only question is how to find an execution that shows it doesn't work.

It's not too hard to see that Algorithm C.2 satisfies both termination and validity. So we need to find a problem with agreement.

The easiest way I can see to do this is to pick a patsy process p and give it input 0, while giving all the other processes input 1. Now run Algorithm C.2 while delaying all outgoing messages $\langle i, v \rangle$ from p until after the receiver has finished the protocol. Because each other process is waiting for $n - f \leq n - 1$ messages, this will not prevent the other processes from finishing. But all the other processes have input 1, so we have an invariant that messages in transit from processes other than p and preferences of processes other than p will be 1 that holds as long as no messages from p are delivered. This results in the non-p processes all deciding 1. We can then run p to completion, at which point it will decide 0.

C.3 Assignment 3: due Wednesday, 2020-10-21, at 5:00pm Eastern US time

C.3.1 Too many Byzantine processes

The phase king algorithm (Algorithm 10.2) described in §10.2.2 solves Byzantine agreement for $f < n/4$ processes. For larger values of f, it may fail by violating one or more of the properties of termination, validity, or agreement.

For this algorithm:

1. How big does f need to be to prevent termination?
2. How big does f need to be to prevent validity?
3. How big does f need to be to prevent agreement?

Assume that the processes know the new bound on f, and any thresholds in the algorithm that use f are adjusted to correspond to this new bound.

Solution

1. Termination: The algorithm always terminates in $f + 1$ synchronous rounds, so f doesn't matter.
2. Validity: To violate validity, we need to convince some non-faulty process to decide on the wrong value when all non-faulty processes have the same input.

   Suppose all the non-faulty processes have input 0, and we want to introduce a 1 somewhere. Each process updates its preference in

each round to be either the majority value it sees, if this value has multiplicity greater than $n/2 + f$, or the kingMajority broadcast by the phase king otherwise.

If $f < n/2$, it's going to be hard to show a process a bogus majority. But a Byzantine phase king gives us more options. Suppose that all the f Byzantine processes send out 1 in all rounds. Then for $f \geq n/4$, the multiplicity of the correct value 0 will be $n - f \leq (3/4)n$, while the required multiplicity to ignore the phase king will be strictly greater than $n/2 + f \geq (3/4)n$. So at $f = n/4$, all non-faulty processes adopt the phase king's bad value 1. In any subsequent round, we can just run the algorithm with the Byzantine agents pretending to be non-faulty processes with preference 1, and eventually all processes incorrectly decide 1.

3. Agreement: Now we need to get two non-faulty processes to decide different values. Wait to the last round, and use $f = n/4$ Byzantine processes to prevent the non-faulty processes from seeing a high enough multiplicity on any majority value to accept it, and use a Byzantine phase king to transmit different kingMajority values to different non-faulty processes. So again, the algorithm fails at $f = n/4$.

C.3.2 Committee election

Consider the following **committee election** problem in an asynchronous message-passing system with $f < n/2$ crash failures. Each process runs a committee election protocol, at the end of which it receives a value 1 (elected) or 0 (not elected). The requirements of the protocol are:

1. Nonempty committee: If no processes fail, at least one process receives 1.

2. No latecomers: In any execution, if some process p finishes the protocol before another process q starts the protocol, then q receives 0.

Give an algorithm that solves this problem, and show that it satisfies these requirements.

(For the purpose of defining when a process starts or ends the protocol, imagine that it uses explicit invoke and respond events. Your protocol should have the property that all non-faulty processes eventually terminate.)

Solution

The easiest way to do this may be to use ABD (see §17.2). Algorithm C.3 has each process read the simulated register, which we assume is initialized to 1, then write a 0 before returning the value it read.

```
1 Let r be an ABD register initialized to 1.
2 procedure elect
3     onCommittee ← r
4     r ← 0
5     return onCommittee
```

Algorithm C.3: Committee election using ABD

This satisfies nonempty committee, because the first operation in the linearization of the register must be a read operation that returns 1. It satisfies no latecomers, because if p finishes before q starts, then p's write finishes before q's read starts, and linearizability of ABD implies q reads a 0.

This takes 3 round-trips to finish (2 for the ABD read and 1 for the ABD write). It is not too hard to reduce this to 2 round-trips by replacing the embedded write in the ABD read operation with a write of 1, but this requires a more detailed correctness argument.

C.4 Assignment 4: due Wednesday, 2020-11-04, at 5:00pm Eastern US time

C.4.1 Counting without snapshots

Algorithm C.4 gives a wait-free implementation of a generalized counter using a collect. The `inc`(v) procedure adjusts the value of the counter by v: if it was x before `inc`(v), it should be $x + v$ after. The `read` procedure returns the current value of the counter. Assume that the initial value of the counter is 0, as are the initial values of the registers $A[i]$ that implement it.

This counter implementation is not linearizable in all executions, but it may be linearizable if we restrict the allowed values v that can be supplied as arguments to an `inc` operations. For each of the following sets V, show that any execution in which all increments are elements of V is linearizable, or show that there exists an execution with increments in V that is not.

1. $V = \{0, 1\}$.

```
1 procedure inc(v)
2     A[i] ← A[i] + v
3 procedure read()
4     s ← 0
5     for j ← 1 to n do
6         s ← s + A[j]
7     return s
```

Algorithm C.4: An alleged counter. Code for process i.

2. $V = \{-1, 1\}$.
3. $V = \{1, 2\}$.

Solution

1. The $\{0, 1\}$ case is linearizable. Given an execution S of Algorithm C.4, we assign to a linearization point to each `inc` operation at the step where it writes to A, and assign a linearization point to each `read` operation ρ that returns s at the later of the first step that leaves $\sum_j A[j] = s$ or the first step of ρ. Since this may assign the same linearization point to some write operation π and one or more read operations $\rho_1, \ldots, \rho_k$, when this occurs, we order the write before the reads and the reads arbitrarily.

   Observe that:

   (a) The value of each $A[j]$ individually is non-decreasing over time, and increases by at most one at each step.

   (b) The same holds for $\sum_{j=1}^{n} A[j]$.

   These are easily shown by induction on the steps of the execution, since each `inc` operation only changes at most one $A[j]$ and only changes it by increasing it by 1.

   The first condition implies that the value v_j of $A[j]$ used by a particular `read` operation ρ lies somewhere between the minimum and maximum values of $A[j]$ during the operation's interval, which implies the same about the total $\sum_j A[j]$. In particular, if ρ returns s the value of $\sum_j A[j]$ is no greater than s, and it reaches s no later than the end of ρ.

Because $\sum_j A[j]$ increases by at most one per step, this means that either $\sum_j A[j] = s$ at the first step of ρ, or $\sum_j A[j] = s$ at some step within the execution interval of ρ. In either case, ρ is assigned an execution point within its interval that follows exactly s non-trivial increments. This means that the return values of all `read` operations are consistent with a sequential generalized counter execution, and because both `read` and `inc` operations are ordered consistently with the execution ordering in S, we have a linearization of S.

2. For increments in $\{-1, 1\}$, there are executions of Algorithm C.4 that are not linearizable. We will construct a specific bad execution for $n = 3$. Let p_1 perform `inc`(1) and p_2 perform `inc`(2), where p_1 finishes its operation before p_2 starts. Because the `inc`(1) must be linearized before the `inc`(-1), the values of the counter in any linearization will be $0, 1, 0$ in this order.

   Now add a `read` operation by p_3 that is concurrent with both `inc` operations. Suppose that in the execution, the follow operations are performed on the registers $A[1]$ through $A[3]$:

   (a) p_3 reads 0 from $A[1]$.
   (b) p_1 writes 1 to $A[1]$.
   (c) p_2 writes -1 to $A[2]$.
   (d) p_3 reads -1 from $A[2]$.
   (e) p_3 reads 0 from $A[3]$.

   Now p_3 returns -1. There is no point in the sequential execution at which this is the correct return value, so there is no linearization of this execution.

3. For increments in $\{1, 2\}$, essentially the same counterexample works. Here we let p_1 do `inc`(1) and p_2 do `inc`(2), while p_3 again does a concurrent read. The bad execution is:

   (a) p_3 reads 0 from $A[1]$.
   (b) p_1 writes 1 to $A[1]$.
   (c) p_2 writes 2 to $A[2]$.
   (d) p_3 reads 2 from $A[2]$.
   (e) p_3 reads 0 from $A[3]$.

   Now p_3 returns 2, but in any linearization of the two write operations, the values in the counter are $0, 1, 3$.

C.4.2 Rock-paper-scissors

Define a **rock-paper-scissors object** as having three states 0 (rock), 1 (paper), and 2 (scissors), with a `read` operation that returns the current state and a `play`(v) operation for $v \in \{0, 1, 2\}$ that changes the state from s to v if $v = (s + 1) \pmod 3$ and has no effect otherwise.

Prove or disprove: There exists a deterministic wait-free linearizable implementation of a rock-paper-scissors object from atomic registers.

Solution

Proof: We will show how to implement a rock-paper-scissors object using an unbounded max register, which can be built from atomic registers using snapshots. The idea is to store a value v such that $v \bmod 3$ gives the value of the rock-paper-scissors object. Pseudocode for both operations is given in Algorithm C.5.

```
1 Let m be a shared max register.
2 procedure play(v)
3     s ← m
4     if v = ((s + 1) mod 3) then
5         m ← s + 1
6 procedure read()
7     return (m mod 3)
```

Algorithm C.5: Implementation of a rock-paper-scissors object

Linearize each `play` operation that does not write m at the step at which it reads m.

Linearize each `play` operation that writes $s + 1$ to m at the first step at which $m \geq s + 1$. If this produces ties, break first in order of increasing $s + 1$ and then arbitrarily. Since each such operation has $m \leq s$ when the operation starts and $m \geq s + 1$ when it finishes, these linearization points fit within the intervals of their operations.

Linearize each `read`() operation at the step where it reads m.

Since each of these linearization points is within the corresponding operation's interval, this preserves the observed execution ordering.

Observe that the `play` operations that write are linearized in order of increasing values written, and there are no gaps in this sequence because no process writes $s + 1$ without first seeing s. (This actually shows there is

no to break ties by value.) So the sequence of values in the max register, taken mod 3, iterates through the values $0, 1, 2, 0, \ldots$ in sequence, with each value equal mod 3 to some argument to a `play` operation. So we can take these values mod 3 as the actual value of the register for the purposes of `read` operations, meaning the `read` operations all return correct values. The `play` operations that don't write are linearized at a point where they would have no effect on the state of the rock-paper-scissors object, which is also consistent with the sequential specification.

It follows that Algorithm C.5 is a linearizable implementation of a rock-paper-scissors object from max registers. It is also wait-free, since each operation is implemented using a constant number of max-register operations. By implementing max registers using snapshots, we get a wait-free linearizable implementation from atomic registers.

C.5 Assignment 5: due Wednesday, 2020-11-18, at 5:00pm Eastern US time

C.5.1 Randomized consensus with one max register

Prove or disprove: A single max register, with no other objects, is sufficient to solve randomized wait-free binary consensus for two processes against an oblivious adversary.

Solution

We'll disprove it.

Let p_0 and p_1 be the two processes. The idea is to consider, for each $i \in \{0, 1\}$ some nonzero-probability solo terminating execution ξ_i of p_i with input i, then show that ξ_0 and ξ_1 can be interleaved to form a two-process execution ξ that is indistinguishable by each p_i from ξ_i.

The oblivious adversary will simply choose to schedule the processes for ξ. Since the processes flip a finite number of coins in this execution, there is a nonzero chance that the adversary gets lucky and they flip their coins exactly the right way.

Fix ξ_0 and ξ_1 as above. Partition each ξ_i as $\alpha_i\beta_{i1}\beta_{i2}\ldots\beta_{ik_i}$ where α_i contains only read operations and each β_{ij} starts with a write operation of a value v_{ij} strictly larger than any previous write operation.

Let $\xi = \alpha_0\alpha_1\beta_{i_1j_1}\beta_{i_2j_2}\ldots\beta_{i_kj_k}$ where $k = k_0 + k_1$ and the blocks $\beta_{i_\ell j_\ell}$ are the blocks $\{\beta_{0j}\}$ and $\{\beta_{1j}\}$ sorted in order of non-decreasing v_{ij}. Then each block $\beta_{i_\ell j_\ell}$ in ξ starts with a write of a value no smaller than the previous

value in the max register, causing each read operation within the block to return the value of this write, just as in the solo execution ξ_{i_ℓ}. Assuming both processes flip their coins as in the solo executions, they both perform the same operations and return the same values. These values will either violate agreement in ξ or validity in at least one of ξ_0 or ξ_1.

C.5.2 A plurality object

Consider a shared-memory object with operations `vote`(v) and `winner`(), where `winner`() returns the value v that appeared in the largest number of previous `vote` operations, or $\perp$ if there is no such unique v. For example, in a sequential execution with votes $a, b, b, c, c, c, a, a, a$, the value returned by a `winner` operation following each vote will be $a, \perp, b, b, \perp, c, c, \perp, a$.

Pick one of these statements, and show that it is true:

1. There is a deterministic wait-free linearizable implementation of this object for n processes that uses $o(n)$ registers.

2. There is such an implementation that uses $O(n)$ registers, but not $o(n)$ registers.

3. There is no such implementation using $O(n)$ registers.

Solution

Case (2) holds.

To implement the object, use a snapshot array to hold the total votes from each process, and have the `winner` operation take a snapshot, add up all the votes and return the correct result. This can be done using n registers.

To show that it can't be done with $o(n)$ registers, use the JTT bound (see Chapter 21). We need to argue that the object is perturbable. Let $\Lambda\Sigma\pi$ be an execution that needs to be perturbed, and let m be the maximum number of `vote`(v) operations that start in Λ for any value v. Then a sequence γ of $m+1$ votes for some v' that does not appear in Λ will leave the object with v' as the plurality value, no matter how the remaining operations are linearized. Since v' did not previously appear in Λ, this gives a different return value for π in $\Lambda\gamma\Sigma\pi$ from $\Lambda\Sigma\pi$ as required. The JTT bound now implies that any implementation of the object requires at least $n-1$ registers.

C.6 Presentation (for students taking CPSC 565): due Tuesday, 2020-12-01, or Thursday, 2020-12-03, in class; paper selection due Friday, 2020-11-20.

Students taking CPSC 565, the graduate version of the class, are expected to give a 10-minute presentation on a recent paper in the theory of distributed computing.

The choice of paper to present should be made in consultation with the instructor. To a first approximation, any paper from PODC, DISC, or a similar conference in the last two or three years (that is not otherwise covered in class) should work.

To get you started, the program for PODC 2020 can be found at https://www.podc.org/podc2020/program/. The web page for DISC 2020 can be found at http://www.disc-conference.org/wp/disc2020/. Both include links to papers published in the conference and to videos of conference presentations.

Because of the limited presentation time, you are not required to get into all of the technical details of the paper, but your presentation should include:[2]

1. Title, authors, and date and venue of publication of the paper.

2. A high-level description of a central result. Unlike a talk for a general audience, you can assume that your listeners know at least everything that we've talked about so far in the class.

3. A description of where this result fits into the literature (e.g., solves an open problem previously proposed in [...], improves on the previous best running time for an algorithm from [...], gives a lower bound or impossibility result for a problem previously proposed by [...], opens up a new area of research for studying [...]), and why it is interesting and/or hard.

4. A description (also possibly high-level) of the main technical mechanism(s) used to get the result.

[2] Literary theorists will recognize this as a three-act structure (preceded by a title card): introduce the main character, make their life difficult, then resolve their problems in time for the final curtain. This is not the only way to organize a talk, but if done right it has the advantage of keeping the audience awake.

You do not have to prepare slides for your presentation if you would prefer to use the whiteboard feature in Zoom, but you should make sure to practice it in advance to make sure it fits in the allocated time. The instructor will be happy to offer feedback on draft versions if available far enough before the actual presentation date.

Relevant dates:

2020-11-20 Paper selection due.

2020-11-27 Last date to send draft slides or arrange for a practice presentation with the instructor if you want guaranteed feedback.

2020-12-01 and 2020-12-03 Presentations, during the usual class time.

Appendix D

Sample assignments from Spring 2019

D.1 Assignment 1: due Wednesday, 2019-02-13, at 5:00pm

D.1.1 A message-passing bureaucracy

Alice and Bob are communicating with each other by alternately exchanging messages. But Bob finds Alice's messages alarming, and whenever he responds to Alice, he also forwards a copy of Alice's message to his good friend Charlie 1, a secret policeman. Charlie 1 reports to Charlie 2, but following the rule that "once is happenstance, twice is coincidence, the third time it's enemy action," [Fle59] Charlie 1 only sends a report to Charlie 2 after receiving three messages from Bob. Similarly, Charlie 2 only sends a message to his supervisor Charlie 3 after receiving three messages from Charlie 2, and so on up until the ultimate boss Charlie n. Pseudocode for each participant is given in Algorithm D.1.

Assuming we are in a standard asynchronous message-passing system, that Alice sends her first message at time 0, and that the protocol finishes as soon as Charlie n receives a message, what is the worst-case time and message complexity of this protocol as a function of n?

Solution

Time complexity Observe that Alice sends at least k messages by time $2k - 2$. This is easily shown by induction on k, because Alice sends at least 1 message by time 0, and if Alice has sent at least $k - 1$ message by time

```
1  Alice:
2  initially do
3      send message to Bob
4  upon receiving message from Bob do
5      send message to Bob
6  Bob:
7  upon receiving message from Alice do
8      send message to Alice
9      send message to Charlie 1
10 Charlie i, for i < n:
11 initially do
12     c ← 0
13 upon receiving message from Bob or Charlie i − 1 do
14     c ← c + 1
15     if c = 3 then
16         c ← 0
17         send message to Charlie i + 1
```

Algorithm D.1: Reporting Alice's alarming messages

$2k-4$, the last of these is received by Bob no later than time $2k-3$, and Bob's response is received by Alice no later than time $2k-2$.

Because each message from Alice prompts a message from Bob at most one time unit later, this implies that Bob sends at least k messages by time $2k-1$.

Write $T_0(k) = 2k-1$ for the maximum time for Bob to send k messages. Write $T_i(k)$ for the maximum time for Charlie i to send k messages, for each $0 < i < n$. In order for Charlie i to send k messages, it must receive $3k$ messages from Bob or Charlie $i-1$ as appropriate. These messages are sent no later than $T_{i-1}(3k)$, and the last of them is received no later than $T_{i-1}(3k)+1$. So we have the recurrence

$$T_i(k) = T_{i-1}(3k) + 1$$
$$T_0(k) = 2k - 1$$

with the exact solution

$$T_i(k) = (2 \cdot 3^i \cdot k - 1) + k.$$

For $i = n-1$ and $k = 1$, this is $2 \cdot 3^{n-1} - 1 + n - 1 = 2 \cdot 3^{n-1} + n = O(3^n)$. We can get the exact time to finish by adding one more unit to account for the delay in delivering the message from Charlie $n-1$ to Charlie n. This gives $2 \cdot 3^{n-1} + n + 1$ time exactly in the worst case, or $O(3^n)$ if we want an asymptotic bound.

Message complexity Message complexity is easier: there is no bound on the number of messages that may be sent before Charlie n receives his first message. This is because in an asynchronous system, Alice and Bob can send an unbounded (though finite) number of messages to each other even before Bob's first message to Charlie 0 is delivered, without violating fairness.

D.1.2 Algorithms on rings

In Chapter 5, we saw several leader election algorithms for rings. But nobody builds rings. However, it may be that an algorithm for a ring can be adapted to other network structures.

1. Suppose you have a network in the form of a d-dimensional hypercube Q^d. This means we have $n = 2^d$ nodes, where each node is labeled by a d-bit coordinate vector, and two nodes are adjacent if their vectors

differ in exactly one coordinate. We also assume that each node knows its own coordinate vector and those of its neighbors.

Show that any algorithm for an asynchronous ring can be adapted to an asynchronous d-dimensional hypercube with no increase in its time or message complexity.

2. What difficulties arise if we try to generalize this to an arbitrary graph G?

Solution

1. The idea is to embed the ring in the hypercube, so that each node is given a clockwise and counterclockwise neighbors, and any time the ring algorithm asks to send a message clockwise or counterclockwise, we send to the appropriate neighbor in the hypercube. We can then argue that for any execution of the hypercube algorithm there is a corresponding execution of the ring algorithm and vice versa; this implies that the worst-case time and message-complexity in the hypercube is the same as in the ring.

   It remains only to construct an embedding. For $d = 0$, $d = 1$, and $d = 2$, the ring and hypercube are the same graph, so it's easy. For larger d, split the hypercube into two subcubes Q^{d-1}, consisting of nodes with coordinate vectors of the form $0x$ and $1x$. Use the previously constructed embedding for $d - 1$ to embed a ring on each subcube, using the same embedding for both. Pick a pair of matching edges $(0x, 0y)$ and $(1x, 1y)$ and remove them, replacing them with $(0x, 1x)$ and $(0y, 1y)$. We have now constructed an undirected Hamiltonian cycle on Q^d. Orient the edges to get a directed cycle, and we're done.

2. There are a several problems that may come up:

   (a) Maybe G is not Hamiltonian.

   (b) Even if G is Hamiltonian, finding an Hamiltonian cycle in an arbitrary graph is **NP**-hard. This could be trouble for a practical algorithm.

   (c) Even if we can find a Hamiltonian cycle for G (maybe because G is a nice graph of some kind, or maybe by taking advantage of the unbounded computational power of processes assumed in the standard message-passing model), the processes don't necessarily know what G looks like at the start. So they would need some

initial start-up cost to map the graph, adding to the time and message complexity of the ring algorithm.

D.1.3 Shutting down

Suppose we want to be able to stop a running protocol in an asynchronous message-passing system prematurely. Define a **shutdown mechanism** as a modification to an existing protocol in which any process can nondeterministically issue a `stop` order that eventually causes all processes to stop sending messages. We would like such a shutdown mechanism to satisfy two properties:

1. **Termination.** If some process issues a `stop` order at time t, no process sends a message at time $t + \Delta$ or later, for some finite bound Δ that may depend on the structure of the network.

2. **Non-interference.** If no process issues a `stop` order, the protocol carries out an execution identical to some execution of the underlying protocol without a shutdown mechanism.

Show how to implement a shutdown mechanism, and prove tight upper and lower bounds on Δ as a function of the structure of the network.

Solution

This is pretty much the same as a Chandy-Lamport snapshot [CL85], as described in §6.3. The main difference is that instead of recording its state upon receiving a `stop` message, a process shuts down the underlying protocol. Pseudocode is given in Algorithm D.2. We assume that the initial `stop` order takes the form of a `stop` message delivered by a process to itself.

```
1 initially do
2     stopped ← false
3 upon receiving stop do
4     if ¬stopped then
5         stopped ← true
6         send stop to all neighbors
7         replace all events in underlying protocol with no-ops
```

Algorithm D.2: Shutdown mechanism based on Chandy-Lamport

An easy induction argument shows that if p receives a `stop` message at time t, then any process q at distance d from p receives a `stop` message no later than time $t + d$. It may be that q sends `stop` messages in responds to this `stop` message, but these are the last messages q ever sends. It follows that no process sends a message later than time $t + D$, where D is the diameter of the graph. This gives an upper bound on Δ.

For the lower bound, we can apply an indistinguishability argument. Let p and q be processes at distance D from each other, and suppose that the underlying protocol involves processes sending messages to their neighbors at every opportunity. Consider two synchronous executions: X, an execution in which no `stop` order is ever issued, and X_t, an execution in which p delivers a `stop` message to itself at time t.

We can show by induction on d that any process r at distance d from p carries out the same steps in both X and X_t up until time $t+d-1$. The base case is when $d = 0$, and we are simply restating that p runs the underlying protocol before time t. For the induction step, we observe for any time $t' < t + d - 1$, any message sent to r from some neighbor s was sent at time $t' - 1 < t + d - 2$, and since $d(p, s) \geq d - 1$, the induction hypothesis gives that s sends the same messages at $t' - 1$ in both X and X_t.

It follows that q sends the same message in X and X_t at time $t + D - 1$. If it sends a message, then we have $\Delta > D - 1$. If it does not send a message, then the mechanism violates the non-interference condition. So any correct shutdown mechanism requires exactly $\Delta = D$ time to finish in the worst case.

D.2 Assignment 2: due Wednesday, 2019-03-06, at 5:00pm

D.2.1 A non-failure detector

Consider the following vaguely monarchist leader election mechanism for an asynchronous message-passing system with crash failures. Each process has access to an oracle that starts with the value 0 and may increase over time. The oracle guarantees:

1. No two processes ever see the same nonzero value.

2. Eventually some non-faulty process is given a fixed value that is larger than the values for all other processes for the rest of the execution.

As a function of the number of processes n, what is the largest number of crash failures f for which it is possible to solve consensus using this oracle?

Solution

We need $f < n/2$.

To show that $f < n/2$ is sufficient, observe that we can use the oracle to construct an eventually strong ($\Diamond S$) failure detector.

Recall that $\Diamond S$ has the property that there is some non-faulty process that is eventually never suspected, and every fault process is eventually permanently suspected. Have each process broadcast the current value of its leader oracle whenever it increases; when a process p receives i from some process q, it stops suspecting q if i is greater than any value p has previously seen, and starts suspecting all other processes. The guarantee that eventually some non-faulty q gets a maximum value that never changes ensures that eventually q is never suspected, and all other processes (including faulty processes) are suspected. We can now use Algorithm 13.2 to solve consensus.

To show that $f < n/2$ is necessary, apply a partition argument. In execution Ξ_0, processes $n/2+1$ through n crash, and processes 1 through $n/2$ run with input 0 and with the oracle assigning value 1 to process 1 (and no others). In execution Ξ_1, processes 1 through $n/2$ crashes, and processes $n/2+1$ through n run with input 1 and with the oracle assigning value 2 to process n (and no others). In each of these executions, termination and validity require that eventually the processes all decide on their respective input values 0 and 1.

Now construct an execution Ξ_2, in which both groups of processes run as in Ξ_0 and Ξ_1, but no messages are exchanged between the groups until after both have decided (which must occur after a finite prefix because this execution is indistinguishable to the processes from Ξ_0 or Ξ_1). We now violate agreement.

D.2.2 Ordered partial broadcast

Define **ordered partial broadcast** as a protocol that allows any process to broadcast a message, with the guarantees that, for messages sent through the broadcast mechanism:

1. Any message sent by a non-faulty process is received by at least one process;
2. Any message that is received by at least one process is received by at least k processes; and

3. If two processes p and q both receive messages m_1 and m_2 from the protocol, then either p and q both receive m_1 before m_2, or they both receive m_2 before m_1.

Give an implementation of ordered partial broadcast with $k = 3n/4$ that works for sufficiently large n in a fully-connected asynchronous message-passing system with up to $f = n/6$ crash failures, or show that no such implementation is possible.

Solution

No such implementation is possible. The proof is by showing that if some such implementation could work, we could solve asynchronous consensus with 1 crash failure, contradicting the Fischer-Lynch-Patterson bound [FLP85] (see Chapter 11).

An implementation of consensus based on totally-ordered partial broadcast for $k = 3n/4$ is given in Algorithm D.3. In fact, $k = 3n/4$ is overkill when $f = 1$; $k > n/2 + f$ is enough.

```
1  first ← ⊥
2  for i ← 1 to n do
3      count[i] ← 0
4      value[i] ← ⊥
5  broadcast ⟨i, input⟩
6  upon receiving ⟨j, v⟩ do
7      if first = ⊥ then
8          first ← ⟨j, v⟩
9          send received(⟨j, v⟩) to all processes
10 upon receiving received(⟨j, v⟩) do
11     count[j] ← count[j] + 1
12     value[j] ← v
13     if count[j] = k − f then
14         decide value[j]
```

Algorithm D.3: Consensus from totally-ordered partial broadcast. Code for process i.

The idea of the algorithm is to use the broadcast mechanism to chose a decision value, by looking at which value is delivered first. Since not every process will see the same value delivered first, this requires a second round

of communication in which processes retransmit their first incoming message. The following lemma shows that this is enough to get agreement:

Lemma D.2.1. *In any execution of Algorithm D.3 with $k > n/2 + f$. there is a unique pair $\langle j, v\rangle$ such that at least $k - f$ non-faulty processes resend* `received`$(\langle j, v\rangle)$.

Proof. Because all processes that receive messages m_1 and m_2 through the broadcast mechanism receive them in the same order, we can define a partial order on messages by letting $m_1 < m_2$ if any process receives m_1 before m_2.

There are only finitely many messages, so there is at least one pair $\langle j, v\rangle$ that is minimal in this partial order. This message is received by at least k processes, of which at least $k - f$ are non-faulty. Each such process receives $\langle j, v\rangle$ before any other broadcast messages, so it sets first to $\langle j, v\rangle$ and resends `received`$(\langle j, v\rangle)$.

To show that $\langle j, v\rangle$ is unique, observe that $k - f > n/2$ implies that if there is some other pair $\langle j', v'\rangle$ that is resent by $k - f$ non-faulty processes, then there is some process that resends both $\langle j, v\rangle$ and $\langle j', v'\rangle$. But each process resends at most one pair. □

Lemma D.2.1 immediately gives agreement, because a process only decides on a value v after receiving `received`$(\langle j, v\rangle)$ from $k - f$ processes, and only one such pair is sent by so many. Termination follows from the existence of such a pair: eventually every non-faulty process receives $\langle j, v\rangle$ from $k - f$ processes. Validity is immediate from the fact that v is j's input.

It follows that Algorithm D.3 solves consensus whenever $k > n/2 + f$, which includes the case $k = 3n/4$ and $f = n/6$. If an implementation of ordered partial broadcast with these parameters exists in the standard message-passing model, this would give a protocol for asynchronous consensus with $f = n/6 \geq 1$ when $n \geq 6$. This contradicts FLP, showing that such an implementation is impossible.

D.2.3 Mutual exclusion using a counter

Algorithm D.4 gives a modified version of Peterson's two-process mutual exclusion algorithm (§18.5.1) that replaces the present bits with an atomic counter count. This object supports read, increment, and decrement operations, where increment and decrement increase and decrease the value in the counter by one, respectively. Unlike the present array, count doesn't depend on the number of processes n. So in principle this algorithm might work for arbitrary n.

```
   shared data:
 1 waiting, atomic register, initially arbitrary
 2 count, atomic counter, initially 0
 3 Code for process i:
 4 while true do
       // trying
 5     increment count
 6     waiting ← i
 7     while true do
 8         if count = 1 then
 9             break
10         if waiting = i + 1 (mod n) then
11             break
       // critical
12     (do critical section stuff)
       // exiting
13     decrement count
       // remainder
14     (do remainder stuff)
```

Algorithm D.4: Peterson's mutual exclusion algorithm using a counter

Show that Algorithm D.4 provides starvation-free mutual exclusion for two processes, but not for three processes.

Solution

The proof that this works for two processes is essentially the same as in the original algorithm. The easiest way to see this is to observe that process p_i sees count $= 1$ in Line 8 under exactly the same circumstances as it sees present$[\neg i] = 0$ in Line 8 in the original algorithm; and similarly with two processes waiting is always set to the same value as waiting in the original algorithm. So we can map any execution of Algorithm D.4 for two processes to an execution of Algorithm 18.5, and all of the properties of the original algorithm carry over to the modified version.

To show that the algorithm doesn't work for three processes, we construct an explicit bad execution:

1. p_0 increments count
2. p_1 increments count
3. p_2 increments count
4. p_0 writes 0 to waiting
5. p_1 writes 1 to waiting
6. p_2 writes 2 to waiting
7. p_0 reads 3 from count
8. p_1 reads 3 from count
9. p_2 reads 3 from count
10. p_1 reads 2 from waiting and enters the critical section.
11. p_1 leaves the critical section and decrements count.

At this point we have count $= 2$ and waiting $= 2$, with both p_0 and p_2 at the start of the loop to check these variables. Suppose that p_1 doesn't come back. Because neither p_0 nor p_2 changes count or waiting, both variables remain at 2 forever. But then neither p_0 nor p_2 enters the critical section, because count is never 1 and waiting is never equal to $0+1$ or $2+1 \pmod 3$.

D.3 Assignment 3: due Wednesday, 2019-04-17, at 5:00pm

D.3.1 Zero, one, many

Consider a counter supporting `inc` and `read` operations that is capped at 2. This means that after the first two calls to `inc`, any further calls to `inc` have no effect: a `read` operation will return 0 if it follows no calls to `inc`, 1 if it follows exactly one call to `inc`, and 2 if it follows two or more calls to `inc`.

There is a straightforward implementation of this object using snapshot. This requires $O(n)$ space and $O(n)$ steps per operation in the worst case.

Is it possible to do better? That is, can one give a deterministic, wait-free, linearizable implementation of a 2-bounded counter from atomic registers that uses $o(n)$ space and $o(n)$ steps per operation in the worst case?

Solution

One possible implementation is given in Algorithm D.5. This requires $O(1)$ space and $O(1)$ steps per call to `inc` or `read`.

```
1  procedure inc
2      if c[1] = 1 then
           // somebody already did inc
3          c[2] ← 1
4      else
5          c[1] ← 1
           // maybe somebody else is doing inc
6          if splitter returns right or down then
7              c[2] ← 1
8  procedure read
9      if c[2] = 1 then
10         return 2
11     else if c[1] = 1 do
12         return 1
13     else
14         return 0
```

Algorithm D.5: A 2-bounded counter

The implementation uses two registers $c[1]$ and $c[2]$ to represent the value of the counter. Two additional registers implement a splitter object as in Algorithm 18.6.[1]

Claim: For any two calls to `inc`, at least one sets $c[2]$ to 1. Proof: Suppose otherwise. Then both calls are by different processes p and q (or else the second call would see $c[1] = 1$) and both execute the splitter. Since a splitter returns stop to at most one process, one of the two processes gets right or down, and sets $c[2]$.

It is also straightforward to show that a single `inc` running alone will set $c[1]$ but not $c[2]$, since in this case the splitter will return stop.

Now we need to argue linearizability. We will do so by assigning linearization points to each operation.

If some `inc` does not set $c[2]$, assign it the step at which it sets $c[1]$. Assign each other `inc` the step at which it first sets $c[2]$.

If every `inc` sets $c[2]$, assign the first `inc` to set $c[1]$ the step at which it does so, and assign all others the first point during its execution interval at which $c[2]$ is nonzero.

For a `read` operation that returns 2, assign the step at which it reads $c[2]$. For a `read` operation that returns 1, assign the first point in the execution interval after it reads $c[2]$ at which $c[1] = 1$. For a `read` operation that returns 0, assign the step at which it reads $c[1]$.

This will assign the same linearization point to some operations; in this case, put `inc`s before `read`s and otherwise break ties arbitrarily.

These choices create a linearization which consists of (a) a sequence of `read` operations that return 0, all of which are assigned linearization points before the first step at which $c[1] = 1$; (b) the first `inc` operation that sets $c[1]$; (c) a sequence of `read` operations that return 1, all of which are linearized after $c[1] = 1$ but before $c[2] = 1$; (c) some `inc` that is either the first to set $c[2]$ or spans the step that sets $c[2]$; and (d) additional `inc` operations together with `read` operations that all return 2. Since each `read` returns the minimum of 2 and the number of `inc`s that precede it, this is a correct linearization.

D.3.2 A very slow counter

Consider a **slow counter** object with operations `inc` and `read`, where the value v of a counter starts at 0 and increases by 1 as the result of each call

[1] It may be possible shave off a register by breaking the splitter abstraction and using the race or door register in place of $c[1]$, but I haven't worked out the linearizability for this case.

to `inc`, but `read` returns $\log^* v$ instead of v.

Suppose we want a deterministic, wait-free, linearizable implementation of a slow counter as defined above from atomic registers. Give tight bounds on the worst-case step complexity of operations on such an implementation.

Solution

The worst-case step complexity of an operation is $\Theta(n)$.

For the upper bound, implement a counter on top of snapshots (or just collect), and have `read` compute $\log^*$ of whatever value is read.

For the lower bound, observe that a slow counter has the perturbability property needed for the JTT proof. Given an execution of the form $\Lambda_k \Sigma_k \pi$ as described in Chapter 21, we can always insert some sequence of `inc` operations between Λ_k and Σ_k that will change the return value of π. The number of `inc`s needed will be the number needed to raise $\log^* v$, plus an extra n to overcome the possibility of pending `inc`s in Σ_k being linearized before or after π. Since this object is perturbable, and the atomic registers we are implementing it from are historyless, JTT applies and gives an $\Omega(n)$ lower bound on the cost of `read` in the worst case.

D.3.3 Double-entry bookkeeping

Consider an object that implements an unbounded collection of accounts $A_1, A_2, \ldots$, each of which holds an integer value, and that provides three operations:

- The operation `read`(i) returns the current value of A_i.
- The operation `transfer`(i, j, n) moves n units from A_i to A_j; if A'_i and A'_j are the new values, then $A'_i = A_i - n$ and $A'_j = A_j + n$.
- The operation `close`(i, j) sets A_i to zero and adds the previous value to A_j. It is equivalent to atomically executing `transfer`$(i, j,$ `read`$(i))$.

1. What is the consensus number of this object?
2. What is the consensus number of a restricted version of this object that provides only the `read` and `transfer` operations?

Solution

1. The consensus number of the object is infinite. Initialize A_0 to 1 and the remaining A_i to 0. We can solve ID consensus by having process i (where $i > 0$ execute `close`$(0, i)$ and then applying `read` to scan all the A_j values for itself and other processes. Whichever process gets the 1 wins.

2. The consensus number without `close` is 1. Proof: Observe that `transfer` operations commute.

D.4 Presentation (for students taking CPSC 565): due Monday, 2019-04-22, or Wednesday, 2019-04-24, in class

Students taking CPSC 565, the graduate version of the class, are expected to give a 10-minute presentation on a recent paper in the theory of distributed computing.

The choice of paper to present should be made in consultation with the instructor. To a first approximation, any paper from PODC, DISC, or a similar conference in the last two or three years (that is not otherwise covered in class) should work.

To get you started, the 2018 PODC proceedings can be found at https://dl.acm.org/citation.cfm?ID=3212734&picked=prox, and the 2018 DISC proceedings can be found at http://drops.dagstuhl.de/opus/portals/lipics/index.php?semnr=16090.

Because of the limited presentation time, you are not required to get into all of the technical details of the paper, but your presentation should include:[2]

1. Title, authors, and date and venue of publication of the paper.

2. A high-level description of a central result. Unlike a talk for a general audience, you can assume that your listeners know at least everything that we've talked about so far in the class.

[2] Literary theorists will recognize this as a three-act structure (preceded by a title card): introduce the main character, make their life difficult, then resolve their problems in time for the final curtain. This is not the only way to organize a talk, but if done right it has the advantage of keeping the audience awake.

3. A description of where this result fits into the literature (e.g., solves an open problem previously proposed in [...], improves on the previous best running time for an algorithm from [...], gives a lower bound or impossibility result for a problem previously proposed by [...], opens up a new area of research for studying [...]), and why it is interesting and/or hard.

4. A description (also possibly high-level) of the main technical mechanism(s) used to get the result.

You do not have to prepare slides for your presentation if you would prefer to use the blackboard, but you should make sure to practice it in advance to make sure it fits in the allocated time. The instructor will be happy to offer feedback on draft versions if available far enough before the actual presentation date.

Relevant dates:

2019-04-10 Paper selection due.

2019-04-17 Last date to send draft slides or arrange for a practice presentation with the instructor if you want guaranteed feedback.

2019-04-22 and 2019-04-24 Presentations, during the usual class time.

D.5 CS465/CS565 Final Exam, May 7th, 2019

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.

D.5.1 A roster (20 points)

A **roster** object has operations `announce` and `read`, where `read` returns a list of the identities of all processes that have previously called `announce` at least once.

Suppose you want a wait-free, linearizable implementation of this object from multiwriter atomic registers. As a function of the number of processes n, give the best upper and lower bound you can on the number of registers you will need.

You may assume that the set of process identities is fixed for each n and that each process knows its own identity.

Solution

You will need exactly n registers ($\Theta(n)$ is also an acceptable answer).

For the upper bound, have each process write its ID to its own register, and use a double-collect snapshot to read all of them. This uses exactly n registers. The double-collect snapshot is wait-free because after each process has called `announce` once, the contents of the registers never change, so `read` finishes after $O(n)$ collects or $O(n^2)$ register reads. It's linearizable because double-collect snapshot returns the exact contents of the registers at some point during its execution.[3]

For the lower bound, use a covering argument.[4]

Have the processes $p_1, \ldots, p_n$ run `announce` in order, stopping each process when it covers a new register. This will give sequence of partial executions Ξ_i where at the end of Ξ_i, there is a set of i registers $r_1 \ldots r_i$ that are covered by $p_1 \ldots p_i$, and no other operations are in progress.

To show this works, we need to argue that each p_{i+1} does in fact cover a register $r_{i+1} \notin \{r_1, \ldots, r_i\}$. If not, then we can extend Ξ_i by running p_{i+1}'s `announce` operation to completion, then delivering all the covering writes by $p_1 \ldots p_i$. Now any subsequent `read` will fail to return p_{i+1}, violating the specification. (If we have a spare process, we can have it do the bad `read`; otherwise we can run p_1 to completion and let it do it.)

At the end of Ξ_n, we have covered n distinct registers, proving the lower bound.

D.5.2 Self-stabilizing consensus (20 points)

Consider a model with n processes $p_0, \ldots, p_{n-1}$ organized in a ring, where p_i can directly observe both its own state and that of $p_{(i-1) \bmod n}$. Suppose that the state x_i of each process p_i is always a natural number.

[3] If we only do a single collect, the implementation is not linearizable. An example of a bad execution is one where a reader reads r_1, then p_1 writes to r_1 (starting and finishing its `announce` operation), then p_2 writes to r_2 (starting an finishing its `announce` operation), and finally the reader reads r_2. In this execution the reader will return $\{p_2\}$ only, which is inconsistent with the observed ordering that puts `announce`(p_1) before `announce`(p_2).

[4] It's tempting to use JTT [JTT00] here, but the roster object is not perturbable. Once all IDs of p_1 through p_{n-1} have been registered, subsequent `announce` operations by p_1 through p_{n-1} have no effect.

The subtlety here is that in the JTT argument, we probably won't choose γ when perturbing $\Lambda_k\Sigma_k$ to include more than one new `announce`, but replacing γ by γ' to hit the first possible uncovered register in π might involve an arbitrary sequence of operations by p_1 through p_{n-1}, including a sequence where all of them call `announce` at least once. Once this happens, we get a $\Lambda_{k+1}\Sigma_{k+1}$ execution that can no longer be perturbed.

At each step, an adversary chooses a process p_i to run. This process then updates its own state based on its previous state x_i and the state $x_{(i-1) \bmod n}$ of its counterclockwise neighbor $p_{(i-1) \bmod n}$. The adversary is required to run every process infinitely often but is not otherwise constrained.

Is there a protocol for the processes that guarantees that, starting from an arbitrary initial configuration, they eventually reach a configuration where (a) all processes have the same state $x \in \mathbb{N}$; and (b) no process ever changes its state as the result of taking additional steps?

Give such a protocol and prove that it works, or show that no such protocol is possible.

Solution

It turns out that this problem is a good example of what happens if you don't remember to include some sort of validity condition. As pointed in several student solutions, having each process pick a fixed constant x_i the first time it updates works.

Here is a protocol that also works, and satisfies the validity condition that the common output was some process's input (which was not required in the problem statement). When p_i takes a step, it sets x_i to $\max(x_i, x_{(i-1) \bmod n})$.

To show that this works, we argue by induction that the maximum value eventually propagates to all processes. Let $x = x_i$ be the initial maximum value. The induction hypothesis is that for each $j \in \{0, \ldots, n-1\}$, eventually all processes in the range i through $i + j \pmod n$ hold value x forever.

Suppose that the hypothesis holds for j; to show that it holds for $j + 1$, start in a configuration where x_i through x_{i+j} are all x. No transition can change any of these values, because taking the max of x and any other value yields x. Because each process is scheduled infinitely often, eventually p_{i+j+1} takes a step; when this happens, x_{i+j+1} is set to $\max(x, x_{i+j+1}) = x$.

Since the hypothesis holds for all $j \in \{0, \ldots, n-1\}$, it holds for $j = n-1$; but this just says that eventually all n processes hold x forever.

D.5.3 All-or-nothing intermittent faults (20 points)

Recall that in the standard synchronous message-passing model with crash failures, a faulty process runs correctly up until the round in which it crashes, during which it sends out some subset of the correct messages, and after which it sends out no messages at all.

Suppose instead we have intermittent faults, where any process may fail to send outgoing messages in a particular round, but these are all-or-nothing

faults in the sense that a process either sends all of its messages in a given round or no messages in that round. To avoid shutting down a protocol completely, we require that in every round, there is at least one process that sends all of its messages. We also allow a process to send a message to itself.

If we wish to solve agreement (that is, get agreement, termination, and validity) in this model, what is the minimum number of rounds we need in the worst case?

Solution

We need one round. Every process transmits its input to all processes, including itself. From the all-or-nothing property, all processes receive the same set of messages. From the assumption that some process is not faulty in this round, this set is nonempty. So the processes can reach agreement by applying any consistent rule to choose an input from the set.

D.5.4 A tamper-proof register (20 points)

Consider a **tamper-proof register**, which is a modified version of a standard multiwriter atomic register for which the `read` operation returns $\perp$ if no `write` operation has occurred, v if exactly one `write`(v) operation has occurred, and fail if two or more `write` operations have occurred.

What is the consensus number of this object?

Solution

The consensus number is 1.

Proof: We can implement it from atomic snapshot, which can be implemented from atomic registers, which have consensus number 1.

For my first `write`(v) operation, write v to my component of the snapshot; for subsequent `write`(v) operations, write fail. For a `read` operation, take a snapshot and return (a) $\perp$ if all components are empty; (b) v if exactly one component is non-empty and has value v; and (c) fail if more than one component is non-empty or any component contains fail.

Appendix E

Sample assignments from Spring 2016

E.1 Assignment 1: due Wednesday, 2016-02-17, at 5:00pm

Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Whether you are taking the course as CPSC 465 or CPSC 565.
4. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

E.1.1 Sharing the wealth

A kindergarten consists of n children in a ring, numbered 0 through $n-1$, with all arithmetic on positions taken mod n.

In the initial configuration, child 0 possesses n cookies. The children take steps asynchronously, and whenever child i takes a step in a configuration

where they have a cookie but child $i+1$ does not, child i gives one cookie to child $i+1$. If child $i+1$ already has a cookie, or child i has none, nothing happens. We assume that a fairness condition guarantees that even though some children are fast, and some are slow, each of them takes a step infinitely often.

1. Show that after some finite number of steps, every child has exactly one cookie.

2. Suppose that we define a measure of time in the usual way by assigning each step the largest possible time consistent with the assumption that that no child ever waits more than one time unit to take a step. Show the best asymptotic upper bound you can, as a function of n, on the time until every child has one cookie.

3. Show the best asymptotic lower bound you can, as a function of n, on the worst-case time until every child has one cookie.

Solution

1. First observe that in any configuration reachable from the initial configuration, child 0 has k cookies, $n-k$ of the remaining children have one cookie each, and the rest have zero cookies. Proof: Suppose we are in a configuration with this property, and consider some possible step that changes the configuration. Let i be the child that takes the step. If $i=0$, then child i goes from k to $k-1$ cookies, and child 1 goes from 0 to 1 cookies, increasing the number of children with one cookie to $n-k+1$. If $i>0$, then child i goes from 1 to 0 cookies and child $i+1$ from 0 to 1 cookies, with k unchanged. In either case, the invariant is preserved.

   Now let us show that k must eventually drop as long as some cookie-less child remains. Let i be the smallest index such that the i-th child has no cookie. Then after finitely many steps, child $i-1$ takes a step and gives child i a cookie. If $i-1=0$, k drops. If $i-1>0$, then the leftmost 0 moves one place to the left. It can do so only finitely many times until $i=1$ and k drops the next time child 0 takes a step. It follows that after finitely many steps, $k=1$, and by the invariant all $n-1$ remaining children also have one cookie each.

2. Number the cookies 0 through $n-1$. When child 0 takes a step, have it give the largest-numbered cookie it still possesses to child 1. For each

cookie i, let x_i^t be the position of the i-th cookie after t asynchronous rounds, where an asynchronous round is the shortest interval in which each child takes at least one step.

Observe that no child $j > 0$ ever gets more than one cookie, since no step adds a cookie to a child that already has one. It follows that cookie 0 never moves, because if child 0 has one cookie, so does everybody else (including child 1). We can thus ignore the fact that the children are in a cycle and treat them as being in a line $0 \ldots n-1$.

We will show by induction on t that, for all i and t, $x_i^t \geq y_i^t = \max(0, \min(i, z_i^t))$ where $z_i^t = t + 2(i - n + 1)$.

Proof: The base case is when $t = 0$. Here $x_i^t = 0$ for all i. We also have $z_i^t = 2(i - n + 1) \leq 0$ so $y_i^t = \max(0, \min(i, z_i^t)) = \max(0, z_i^t) = 0$. So the induction hypothesis holds with $x_i^t = y_i^t = 0$.

Now suppose that the induction hypothesis holds for t. For each i, there are several cases to consider:

(a) $x_i^t = x_{i+1}^t = 0$. In this case cookie i will not move, because it's not at the top of child 0's stack. But from the induction hypothesis we have that $x_{i+1}^t = 0$ implies $z_{i+1}^t = t + 2(i + 1 - n + 1) \leq 0$, which gives $z_i^t = z_{i+1}^t - 2 \leq -2$. So $z_i^{t+1} \leq z_{i+1}^t + 1 \leq -1$ and $y_i^{t+1} = 0$, and the induction hypothesis holds for x_i^{t+1}.

(b) $x_i^t = i$. Then even if cookie i doesn't move (and it doesn't), we have $x_i^{t+1} \geq x_i^t \geq \min(i, z_i^t)$.

(c) $x_i^t < i$ and $x_{i+1}^t = x_i^t + 1$. Again, even if cookie i doesn't move, we still have $x_i^{t+1} \geq x_i^t = x_{i+1}^t - 1 \geq y_{i+1}^t - 1 \geq t + 2(i+1-n+1) - 1 = t + 2(i - n + 1) + 1 > y_i^t$.

(d) $x_i^t < i$ and $x_{i+1}^t > x_i^t + 1$. Nothing is blocking cookie i, so it moves: $x_i^{t+1} = x_i^t + 1 \geq t + 2(i - n + 1) + 1 = (t + 1) + 2(i - n + 1) = y_i^{t+1}$.

It follows that our induction hypothesis holds for all t. In particular, at $t = 2n - 2$ we have $z_i^t = 2n - 2 + 2(i - n + 1) = 2i - 1 \geq i$ for all $i > 0$. So at time $2n - 2$, $x_i^t \geq y_i^t = i$ for all i and every child has one cookie. This gives an asymptotic upper bound of $O(n)$.

3. There is an easy lower bound of $n - 1$ time. Suppose we run the processes in round-robin order, i.e., the i-th step is taken by process $i \bmod n$. Then one time unit goes by for every n steps, during which each process takes exactly one step. Since process 0 reduces its count

by at most 1 per step, it takes at least $n-1$ time to get it to 1. This gives an asymptotic lower bound of $\Omega(n)$, which is tight.

I believe it should be possible to show an exact lower bound of $2n-2$ time by considering a schedule that runs in reverse round-robin order $n-1, n-2, \ldots, 0, n-1, n-2, \ldots$, but this is more work and gets the same bound up to constants.

E.1.2 Eccentricity

Given a graph $G = (V, E)$, the **eccentricity** $\epsilon(v)$ of a vertex v is the maximum distance $\max_{v'} d(v, v')$ from v to any vertex in the graph.

Suppose that you have an anonymous[1] asynchronous message-passing system with no failures whose network forms a tree.

1. Give an algorithm that allows each node in the network to compute its eccentricity.

2. Safety: Prove using an invariant that any value computed by a node using your algorithm is in fact equal to its eccentricity. (You should probably have an explicit invariant for this part.)

3. Liveness: Show that every node eventually computes its eccentricity in your algorithm, and that the worst-case message complexity and time complexity are both within a constant factor of optimal for sufficiently large networks.

Solution

1. Pseudocode is given in Algorithm E.1. For each edge vu, the algorithm sends a message d from v to u, where d is the maximum length of any simple path starting with uv. This can be computed as soon as v knows the maximum distances from all of its other neighbors $u' \neq u$.

2. We now show correctness of the values computed by Algorithm E.1. Let $d_v[u]$ be the value of $d[u]$ at v. Let $\ell_v[u]$ be the maximum length of any simple path starting with the edge vu. To show that the

[1] Clarification added 2016-02-13: Anonymous means that processes don't have global IDs, but they can still tell their neighbors apart. If you want to think of this formally, imagine that each process has a **local identifier** for each of its neighbors: a process with three neighbors might number them $1, 2, 3$ and when it receives or sends a message one of these identifiers is attached. But the local identifiers are arbitrary, and what I call you has no relation to what you call me or where either of us is positioned in the network.

```
   local data: d[u] for each neighbor u, initially ⊥
   notified[u] for each neighbor u, initially false

 1 initially do
 2     notify ()
 3 upon receiving d from u do
 4     d[u] ← d
 5     notify ()
 6 procedure notify ()
 7     foreach neighbor u do
 8         if ¬notified[u] and d[u'] ≠ ⊥ for all u' ≠ u then
 9             Send 1 + max_{u'≠u} d[u'] to u
10             notified[u] ← true
11     if notified[u] = true for all neighbors u then
12         ε ← max_u d[u]
```

Algorithm E.1: Computing eccentricity in a tree

algorithm computes the correct values, we will prove the invariant that $d_v[u] \in \{\bot, \ell_v[u]\}$ always, and for any message d in transit from u to v, $d = \ell_v[u]$.

In the initial configuration, $d_v[u] = \bot$ for all v and u, and there are no messages in transit. So the invariant holds.

Now let us show that calling `notify` at some process v preserves the invariant. Because `notify`() does not change d_v, we need only show that the messages it sends contain the correct distances.

Suppose `notify`() causes v to send a message d to u. Then $d = 1 + \max_{u' \neq u} d_v[u'] = 1 + \max_{u' \neq u} \ell_v[u']$, because $d_v[u'] \neq \bot$ for all neighbors $u' \neq u$ by the condition on the if statement and thus $d_v[u'] = \ell_v[u']$ for all $u' \neq u$ by the invariant.

So the invariant will continue to hold in this case provided $\ell_u[v] = 1 + \max_{u' \neq u} \ell_v[u']$. The longest simple path starting with uv either consists of uv alone, or is of the form $uvw\ldots$ for some neighbor w of v with $w \neq u$. In the former case, v has no other neighbors u', in which case $d = 1 + \max_{u' \neq u} \ell_v[u'] = 1 + 0 = 1$, the correct answer. In the latter case, $d = 1 + \max_{u' \neq u} \ell_v[u'] = 1 + \ell_v[w]$, again the length of the longest path starting with uv.

This shows that `notify` preserves the invariant. We must also show that assigning $d_v[u] \leftarrow d$ upon receiving d from u does so. But in this case we know from the invariant that $d = \ell_v[u]$, so assigning this value to $d_v[u]$ leaves $d_v[u] \in \{\perp, \ell_v[u]\}$ as required.

3. First let's observe that at most one message is sent in each direction across each edge, for a total of $2|E| = 2(n-1)$ messages. This is optimal, because if in some execution we do not send a message across some edge uv, then we can replace the subtree rooted at u with an arbitrarily deep path, and obtain an execution indistinguishable to v in which its eccentricity is different from whatever it computed.

   For time complexity (and completion!) we'll argue by induction on $\ell_v[u]$ that we send a message across uv by time $\ell_v[u] - 1$.

   If $\ell_v[u] = 1$, then u is a leaf; as soon as `notify` is called in its initial computation event (which we take as occurring at time 0), u notices it has no neighbors other than v and sends a message to v.

   If $\ell_v[u] > 1$, then since $\ell_v[u] = 1 + \max_{v' \neq v} \ell_u[v']$, we have $\ell_u[v]' \leq \ell_v[u] - 1$ for all neighbors $v' \neq v$ of u, which by the induction hypothesis means that each such neighbor v' sends a message to u no later than time $\ell_v[u] - 2$. These messages all arrive at u no later than time $\ell_v[u] - 1$; when the last one is delivered, u sends a message to v.

   It follows that the last time a message is sent is no later than time $\max_{uv}(\ell_v[u] - 1)$, and so the last delivery event occurs no later than time $\max_{uv} \ell_v[u]$. This is just the diameter D of the tree, giving a worst-case time complexity of exactly D.

   To show that this is optimal, consider an execution of some hypothetical algorithm that terminates by time $D - 1$ in the worst case. Let u and v be nodes such that $d(u, v) = D$. Then there is an execution of this algorithm in no chain of messages passes from u to v, meaning that no event of u is causally related to any event of v. So we can replace u with a pair uw of adjacent nodes with $d(w, v) = d(u, v) + 1$, which changes $\epsilon(v)$ but leaves an execution that is indistinguishable to v from the original. It follows that v returns an incorrect value in some executions, and this hypothetical algorithm is not correct. So time complexity D is the best possible in the worst case.

E.1.3 Leader election on an augmented ring

Suppose that we have an asynchronous ring where each process has a distinct identity, but the processes do not know the size n of the ring. Suppose also that each process i can send messages not only to its immediate neighbors, but also to the processes at positions at positions $i-3$ and $i+3 \pmod n$ in the ring.

Show that $\Theta(n \log n)$ messages are both necessary and sufficient in the worst case to elect a unique leader in this system.

Solution

For sufficiency, ignore the extra edges and use Hirschberg-Sinclair [HS80] (see §5.2.2).

For necessity, we'll show that an algorithm that solves leader election in this system using at most $T(n)$ messages can be modified to solve leader election in a standard ring without the extra edges using at most $3T(n)$ messages. The idea is that whenever a process i attempts to send to $i+3$, we replace the message with a sequence of three messages relayed from i to $i+1$, $i+2$, and then $i+3$, and similarly for messages sent in the other direction. Otherwise the original algorithm is unmodified. Because both systems are asynchronous, any admissible execution in the simulated system has a corresponding admissible execution in the simulating system (replace each delivery event by three delivery events in a row for the relay messages) and vice versa (remove the initial two relay delivery events for each message and replace the third delivery event with a direct delivery event). So in particular if there exists an execution in the simulating system that requires $\Omega(n \log n)$ messages, then there is a corresponding execution in the simulated system that requires at least $\Omega(n \log n/3) = \Omega(n \log n)$ messages as well.

E.2 Assignment 2: due Wednesday, 2016-03-09, at 5:00pm

E.2.1 A rotor array

Suppose that you are given an object that acts as described in Algorithm E.2. A `write` operation on this object writes to location $A[r]$ and increments r mod n. A `read` operation by process i (where $i \in \{0 \dots n-1\}$) returns $A[i]$. Initially, $r = 0$ and $A[i] = \perp$ for all i.

What is the consensus number of this object?

```
1 procedure write(A, v)
2     atomically do
3         A[r] ← v; r ← (r + 1) mod n
4 procedure read(A)
5     return A[i]
```

Algorithm E.2: Rotor array: code for process i

Solution

First let's show that it is at least 2, by exhibiting an algorithm that uses a single rotor array plus two atomic registers to solve 2-process wait-free consensus.

```
1 procedure consensus(v)
2     input[i] ← v
3     write(A, i)
4     i′ ← read(A)
5     if i′ = i then
          // Process 0 wrote first
6         return input[0]
7     else
          // Process 1 wrote first
8         return input[1]
```

Algorithm E.3: Two-process consensus using a rotor array

The algorithm is given as Algorithm E.3. Each process i first writes its input value to a single-writer register input$[i]$. The process then writes its ID to the rotor array. There are two cases:

1. If process 0 writes first, then process 0 reads 0 and process 1 reads 1. Thus both processes see $i' = i$ and return input$[0]$, which gives agreement, and validity because input$[0]$ is then equal to 0's input.

2. If process 1 writes first, then process 0 reads 1 and process 1 reads either 0 (if 0 wrote quickly enough) or $\perp$ (if it didn't). In either case, both processes see $i' \neq i$ and return input$[1]$.

Now let us show that a rotor array can't be used to solve wait-free consensus with three processes. We will do the usual bivalence argument,

and concentrate on some bivalent configuration C and pair of operations π_0 and π_1 such that $C\pi_i$ is i-valent for each i.

If π_0 and π_1 are operations on different objects or operations on an atomic register, then they either commute or the usual analysis for atomic registers gives a contradiction. So the interesting case is when π_0 and π_1 are both operations on a single rotor array object A.

If either operation is a `read`, then only the process that carries out the `read` knows whether it occurred. The same argument as for atomic registers applies in this case. So the only remaining case is when both operations are `write`s.

Consider the configurations $C\pi_0\pi_1$ (which is 0-valent) and $C\pi_1\pi_0$ (which is 1-valent). These differ in that there are two locations j and $(j+1) \bmod n$ (which we will just write as $j+1$) that contain values v_0 and v_1 in the first configuration and v_1 and v_0 in the second. Suppose that we stop processes j and $j+1$, and let some other process run alone until it decides. Because this third process can't observe either locations j or $j+1$, it can't distinguish between $C\pi_0\pi_1$ and $C\pi_1\pi_0$, and thus decides the same value starting from either configuration. But this contradicts the assumption that $C\pi_i$ is i-valent. It follows that there is no escape from bivalence with three processes, and the rotor array plus atomic registers cannot be used to solve three-process wait-free consensus.

The consensus number of this object is 2.

E.2.2 Set registers

Suppose we want to implement a **set register** S in a message-passing system, where a set register provides operations `insert`(S, v), which inserts a new element v in S, and `read`(S), which returns the set of all elements previously inserted into S. So, for example, after executing `insert`$(S, 3)$, `insert`$(S, 1)$, and `insert`$(S, 1)$; `read`(S) would return $\{1, 3\}$.

1. Give an algorithm for implementing a linearizable set register where all operations terminate in finite time, in a deterministic asynchronous message-passing system with $f < n/2$ crash failures and no failure detectors, or show that no such algorithm is possible.

2. Suppose that we change the `read`(S) operation to return a list of all the elements of S in the order they were first inserted (e.g., $[3, 1]$ in the example above). Call the resulting object an **ordered set register**.

   Give an algorithm for implementing a linearizable ordered set register

under the same conditions as above, or show that no such algorithm is possible.

Solution

1. It's probably possible to do this with some variant of ABD, but getting linearizability when there are multiple concurrent `insert` operations will be tricky.

   Instead, we'll observe that it is straightforward to implement a set register using a shared-memory snapshot: each process writes to $A[i]$ the set of all values it has ever inserted, and a `read` consists of taking a snapshot and then taking the union of the values. Because we can implement snapshots using atomic registers, and we can implement atomic registers in a message-passing system with $f < n/2$ crash failures using ABD, we can implement this construction in a message-passing system with $f < n/2$ failures.

2. This we can't do. The problem is that an ordered set register can solve agreement: each process inserts its input, and the first input wins. But FLP says we can't solve agreement in an asynchronous message-passing system with one crash failure.

E.2.3 Bounded failure detectors

Suppose you have a deterministic asynchronous message-passing system equipped with a failure detector that is eventually weakly accurate and k-bounded strongly complete, meaning that at least $\min(k, f)$ faulty processes are eventually permanently suspected by all processes, where f is the number of faulty processes.

For what values of k, f, and n can this system solve agreement?

Solution

We can solve agreement using the k-bounded failure detector for $n \geq 2$ processes if and only if $f \leq k$ and $f < n/2$.

Proof:

If $k \geq f$, then every faulty process is eventually permanently suspected, and the k-bounded failure detector is equivalent to the $\Diamond S$ failure detector. The Chandra-Toueg protocol [CT96] then solves consensus for us provided $f < n/2$.

If $f \geq n/2$, the same partitioning argument used to show impossibility with $\Diamond P$ applies to the k-bounded detector as well (as indeed it applies to any failure detector that is only eventually accurate).

If $k < f$, then if we have an algorithm that solves agreement for n processes, then we can turn it into an algorithm that solves agreement for $n - k$ processes with $f - k$ failures, using no failure detector at all. The idea is that the $n - k$ processes can pretend that there are an extra k faulty processes that send no messages and that are permanently suspected. But this algorithm runs in a standard asynchronous system with $f - k$ failures, and FLP says we can't solve agreement in such a system with $n \geq 2$ and $f \geq 1$. So this rules out solving agreement in the original system if $k < f$ and $k \leq n - 2$.

There is one remaining case, where $k = n - 1$ and $f = n$. Here we can actually solve consensus when $n = 1$ (because we can always solve consensus when $n = 1$). For larger n, we have $f \geq n/2$. So there is only one exception to the general rule that we need $f \leq k$ and $f < n/2$.

E.3 Assignment 3: due Wednesday, 2016-04-20, at 5:00pm

E.3.1 Fetch-and-max

```
1  procedure fetchAndMax(r, 0 : x)
2      if switch = 0 then
3          return 0 : fetchAndMax(left, x)
4      else
5          return 1 : fetchAndMax(right, 0)
6  procedure fetchAndMax(r, 1 : x)
7      v ← fetchAndMax(right, x)
8      if TAS(switch) = 0 then
9          return 0 : fetchAndMax(left, 0)
10     else
11         return 1 : v
```

Algorithm E.4: Max register modified to use a test-and-set bit

Algorithm E.4 replaces the switch bit in the max register implementation from Algorithm 22.2 with a test-and-set, and adds some extra machinery to

return the old value of the register before the write.

Define a fetch-and-max register as a RMW object that supports a single operation `fetchAndMax`(x) that, as an atomic action, (a) replaces the old value v in the register with the maximum of x and v; and (b) returns the old value v.

Suppose that left and right are both linearizable wait-free k-bit fetch-and-max registers. Show that Algorithm E.4 implements a linearizable wait-free $(k+1)$-bit fetch-and-max register, or give an example of an execution that violates linearizability.

Solution

Here is a bad execution (there are others). Let $k = 1$, and let π_1 do `fetchAndMax`(01) and π_2 do `fetchAndMax`(10). Run these operations concurrently as follows:

1. π_1 reads switch and sees 0.
2. π_2 does `fetchAndMax`$(\mathsf{right}, 0)$.
3. π_2 does `TAS`(switch) and sees 0.
4. π_2 does `fetchAndMax`$(\mathsf{left}, 0)$ and sees 0.
5. π_1 does `fetchAndMax`$(\mathsf{left}, 1)$ and sees 0.

Now both π_1 and π_2 return 00. But in the sequential execution $\pi_1\pi_2$, π_2 returns 01; and in the sequential execution $\pi_2\pi_1$, π_1 returns 10. Since π_1 and π_2 return the values they return in the concurrent execution in neither sequential execution, the concurrent execution is not linearizable.

E.3.2 Median

Define a **median register** as an object r with two operations `addSample`(r, v), where v is any integer, and `computeMedian`(r). The `addSample` operation adds a sample to the multiset M of integers stored in the register, which is initially empty. The `computeMedian` operations returns a median of this multiset, defined as a value x with the property that (a) x is in the multiset; (b) at least $|M|/2$ values v in the multiset are less than or equal to x; (c) at least $|M|/2$ values v in the multiset are greater than or equal to x.

For example, if we add the samples $1, 1, 3, 5, 5, 6$, in any order, then a subsequent `computeMedian` can return either 3 or 5.

Suppose that you wish to implement a linearizable wait-free median register using standard atomic registers and resettable test-and-set bits. Give tight (up to constants) asymptotic upper and lower bounds on the number of such objects you would need. You may assume that the atomic registers may hold arbitrarily large values.

Solution

For the upper bound, we can do it with $O(n)$ registers using any linear-space snapshot algorithm (for example, Afek *et al.* [AAD+93]). Each process stores in its own segment of the snapshot object the multiset of all samples added by that process; `addSample` just adds a new sample to the process's segment. For `computeMedian`, take a snapshot, then take the union of all the multisets, then compute the median of this union. Linearizability and wait-freedom of both operations are immediate from the corresponding properties of the snapshot object.

For the lower bound, use JTT [JTT00]. Observe that both atomic registers and resettable test-and-sets are historyless: for both types, the new state after an operation doesn't depend on the old state. So JTT applies if we can show that the median register is perturbable.

Suppose that we have a schedule $\Lambda_k \Sigma_k \pi$ in which Λ_k consists of an arbitrary number of median-register operations of which at most k are incomplete, Σ_k consists of k pending base object operations (writes, test-and-sets, or test-and-set resets) covering k distinct base objects, and π is a read operation by a process not represented in $\Lambda_k \Sigma_k$. We need to find a sequence of operations γ that can be inserted between Λ_k and Σ_k that changes the outcome of π.

Let S be the multiset of all values appearing as arguments to `addSample` operations that start in Λ_k or Σ_k. Let $x = \max S$ (or 0 if S is empty), and let γ consist of $|S| + 1$ `addSample`$(r, x + 1)$ operations. Write T for the multiset of $|S| + 1$ copies of $x + 1$. Then in any linearization of $\Lambda_k \gamma \Sigma_k \pi$, the multiset U of samples contained in r when π executes includes at least all of T and at most all of S; this means that a majority of values in U are equal to $x + 1$, and so the median is $x + 1$. But $x + 1$ does not appear in S, so π can't return it in $\Lambda_k \Sigma_k \pi$. It follows that a median register is in fact perturbable, and JTT applies, which means that we need at least $\Omega(n)$ base objects to implement a median register.

E.3.3 Randomized two-process test-and-set with small registers

Algorithm E.5 gives an implementation of a randomized one-shot test-and-set for two processes, each of which may call the procedure at most once, with its process ID (0 or 1) as an argument.

The algorithm uses two registers, a_0 and a_1, that are both initialized to 0 and hold values in the range $0 \dots m-1$, where m is a positive integer. Unfortunately, whoever wrote it forgot to specify the value of m.

```
1  procedure TAS(i)
2      myPosition ← 0
3      while true do
4          otherPosition ← read(a_¬i)
5          x ← myPosition − otherPosition
6          if x ≡ 2 (mod m) then
7              return 0
8          else if x ≡ −1 (mod m) do
9              return 1
10         else if fair coin comes up heads do
11             myPosition ← (myPosition + 1) mod m
12         write(a_i, myPosition)
```

Algorithm E.5: Randomized one-shot test-and-set for two processes

For what values of m does Algorithm E.5 correctly implement a one-shot, probabilistic wait-free, linearizable test-and-set, assuming:

1. An oblivious adversary?
2. An adaptive adversary?

Solution

For the oblivious adversary, we can quickly rule out $m < 5$, by showing that there is an execution in each case where both processes return 0:

- When $m = 1$ or $m = 2$, both processes immediately return 0, because the initial difference 0 is congruent to 2 mod m.
- When $m = 3$, there is an execution in which p_0 writes 1 to a_0, p_1 reads this 1 and computes $x = -1 \equiv 2 \pmod 3$ and returns 0, then p_0 reads

0 from a_0, computes $x = 1$, advances a_0 to 2, then re-reads 0 from a_0, computes $x = 2$, and returns 0.

- When $m = 4$, run p_0 until it writes 2 to a_0. It then computes $x = 2$ and returns 0. If we now wake up p_1, it computes $x = -2 \equiv 2 \pmod 4$ and also returns 0.

When $m \geq 5$ and the adversary is oblivious, the implementation works. We need to show both linearizability and termination. We'll start with linearizability.

Observe that in a sequential execution, first process to perform `TAS` returns 0 and the second 1. So we need to show (a) that the processes between them return both values, and (b) that if one process finishes before the other starts, the first process returns 0.

It is immediate from Algorithm E.5 that in any reachable configuration, $\mathsf{myPosition}_i \in \{a_i, a_i + 1\}$, because process i can only increment myPosition at most once before writing its value to a_i.

Below we will assume without loss of generality that p_0 is the first process to perform its last read before returning.

Suppose that p_0 returns 0. This means that p_0 observed $a_1 \equiv a_0 - 2 \pmod m$. So at the time p_0 last read a_1, $\mathsf{myPosition}_1$ was congruent to either $a_0 - 1$ or $a_0 - 2$. This means that on its next read of a_0, p_1 will compute x congruent to either -1 or -2. Because m is at least 5, in neither case will it mistake this difference for 2. If it computed $x \equiv -1$, it returns 1; if it computed $x \equiv -2$, it does not return immediately, but eventually it will flip its coin heads, increment $\mathsf{myPosition}_1$, and return 1. In either case we have that exactly one process returns each value.

Alternatively, suppose that p_0 returns 1. Then p_0 reads $a_1 \equiv a_0 + 1$, and at the time of this read, $\mathsf{myPosition}_1$ is either congruent to $a_0 + 1$ or $a_0 + 2$. In the latter case, p_1 returns 0 after its next read; in the former, p_1 eventually increments $\mathsf{myPosition}_1$ and then returns 0. In either case we again have that exactly one process returns each value.

Now suppose that p_0 runs to completion before p_1 starts. Initially, p_0 sees $a_0 \equiv a_1$, but eventually p_0 increments a_0 enough times that $a_0 - a_1 \equiv 2$; p_0 returns 0.

To show termination (with probability 1), consider any configuration in which neither process has returned. During the next $2k$ steps, at least one process takes k steps. Suppose that during this interval, this fast process increments myPosition at every opportunity, while the other process does not increment myPosition at all (this event occurs with nonzero probability

for any fixed k, because the coin-flips are uncorrelated with the oblivious adversary's choice of which process is fast). Then for k sufficiently large, the fast process eventually sees $a_0 - a_1$ congruent to either 2 or -1 and returns. Since this event occurs with independent nonzero probability in each interval of length $2k$, eventually it occurs.[2]

Once one process has terminated, the other increments myPosition infinitely often, so it too eventually sees a gap of 2 or -1.

For the adaptive adversary, the adversary can prevent the algorithm from terminating. Starting from a state in which both processes are about to read and $a_0 = a_1 = k$, run p_0 until it is about to write $(k+1) \bmod m$ to a_0 (unlike the oblivious adversary, the adaptive adversary can see when this will happen). Then run p_1 until it is about to write $(k+1) \bmod m$ to a_1. Let both writes go through. We are now in a state in which both processes are about to read, and $a_0 = a_1 = (k+1) \bmod m$. So we can repeat this strategy forever.

E.4 Presentation (for students taking CPSC 565): due Wednesday, 2016-04-27

Students taking CPSC 565, the graduate version of the class, are expected to give a 15-minute presentation on a recent paper in the theory of distributed computing.

The choice of paper to present should be made in consultation with the instructor. To a first approximation, any paper from PODC, DISC, or a similar conference in the last two or three years (that is not otherwise covered in class) should work.

Because of the limited presentation time, you are not required to get into all of the technical details of the paper, but your presentation should include[3]

1. Title, authors, and date and venue of publication of the paper.

2. A high-level description of the main result. Unlike a talk for a general

[2] The fancy way to prove this is to invoke the second Borel-Cantelli lemma of probability theory. Or we can just argue that the probability that we don't terminate in the first ℓ intervals is at most $(1-\epsilon)^{\ell}$, which goes to zero in the limit.

[3] Literary theorists will recognize this as a three-act structure (preceded by a title card): introduce the main character, make their life difficult, then resolve their problems in time for the final curtain. This is not the only way to organize a talk, but if done right it has the advantage of keeping the audience awake.

audience, you can assume that your listeners know at least everything that we've talked about so far in the class.

3. A description of where this result fits into the literature (e.g., solves an open problem previously proposed in [...], improves on the previous best running time for an algorithm from [...], gives a lower bound or impossibility result for a problem previously proposed by [...], opens up a new area of research for studying [...]), and why it is interesting and/or hard.

4. A description (also possibly high-level) of the main technical mechanism(s) used to get the main result.

You do not have to prepare slides for your presentation if you would prefer to use the blackboard, but you should make sure to practice it in advance to make sure it fits in the allocated time. The instructor will be happy to offer feedback on draft versions if available far enough before the actual presentation date.

Relevant dates:

2016-04-13 Paper selection due.

2016-04-22 Last date to send draft slides or arrange for a practice presentation with the instructor if you want guaranteed feedback.

2016-04-27 Presentations, during the usual class time.

E.5 CS465/CS565 Final Exam, May 10th, 2016

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.

E.5.1 A slow register (20 points)

Define a **second-to-last register** as having a read operation that always returns the second-to-last value written to it. For example, after `write` (1), `write` (2), `write` (3), a subsequent `read` operation will return 2. If fewer that two `write` operations have occurred, a `read` will return $\perp$.

What is the consensus number of this object?

Solution

The consensus number of this object is 2.

For two processes, have each process i write its input to a standard atomic register $r[i]$, and then write its ID to a shared second-to-last-value register s. We will have whichever process writes to s first win. After writing, process i can detect which process wrote first by reading s once, because it either sees $\perp$ (meaning the other process has not written yet) or it sees the identity of the process that wrote first. In either case it can return the winning process's input.

For three processes, the usual argument gets us to a configuration C where all three processes are about to execute operations x, y, and z on the same object, where each operation moves from a bivalent to a univalent state. Because we know that this object can't be a standard atomic register, it must be a second-to-last register. We can also argue that all of x, y, and z are writes, because if one of them is not, the processes that don't perform it can't tell if it happened or not.

Suppose that Cx is 0-valent and Cy is 1-valent. Then $Cxyz$ is 0-valent and Cyz is 1-valent. But these configurations are indistinguishable to any process but x. It follows that the second-to-last register can't solve consensus for three processes.

E.5.2 Two leaders (20 points)

Assume that you are working in an asynchronous message-passing system organized as a connected graph, where all processes run the same code except that each process starts with an ID and the knowledge of the IDs of all of its neighbors. Suppose that all of these IDs are unique, except that the smallest ID (whatever it is) might appear on either one or two processes.

Is it possible in all cases to detect which of these situations hold? Either give an algorithm that allows all processes to eventually correctly return whether there are one or two minimum-id processes in an arbitrary connected graph, or show that no such algorithm is possible.

Solution

Here is an algorithm.

If there are two processes p and q with the same ID that are adjacent to each other, they can detect this in the initial configuration, and transmit this fact to all the other processes by flooding.

If these processes p and q are not adjacent, we will need some other mechanism to detect them. Define the extended ID of a process as its own ID followed by a list of the IDs of its neighbors in some fixed order. Order the extended IDs lexicographically, so that a process with a smaller ID also has a smaller extended ID.

Suppose now that p and q are not adjacent and have the same extended ID. Then they share the same neighbors, and each of these neighbors will see that p and q have duplicate IDs. So we can do an initial round of messages where each process transmits its extended ID to its neighbors, and if p and q observe that their ID is a duplicate, they can again notify all the processes to return that there are two leaders by flooding.

The remaining case is that p and q have distinct extended IDs, or that only one minimum-process ID exists. In either case we can run any standard broadcast-based leader-election algorithm, using the extended IDs, which will leave us with a tree rooted at whichever process has the minimum extended ID. This process can then perform convergecast to detect if there is another process with the same ID, and perform broadcast to inform all processes of this fact.

E.5.3 A splitter using one-bit registers (20 points)

Algorithm E.6 implements a splitter-like object using one-bit registers. It assumes that each process has a unique ID ID consisting of $k = \lceil \lg n \rceil$ bits $\mathsf{ID}_{k-1}\mathsf{ID}_{k-2}\ldots\mathsf{ID}_0$. We would like this object to have the properties that (a) if exactly one process executes the algorithm, then it wins; and (b) in any execution, at most one process wins.

Show that the algorithm has these properties, or give an example of an execution where it does not.

Solution

The implementation is correct.

If one process runs alone, it sets $A[i][\mathsf{ID}_i]$ for each i, sees 0 in door, then sees 0 in each location $A[i][\neg\mathsf{ID}_i]$ and wins. So we have property (a).

Now suppose that some process with ID p wins in an execution that may involve other processes. Then p writes $A[i][p_i]$ for all i before observing 0 in door, which means that it sets all these bits before any process writes 1 to door. If some other process q also wins, then there is at least one position i where $p_i = \neg q_i$, and q reads $A[i][p_i]$ after writing 1 to door. But then q sees 1 in this location and loses, a contradiction.

```
   shared data:
 1 one-bit atomic registers A[i][j] for i = 0 ... ⌈lg n⌉ − 1 and j ∈ {0, 1}, all
   initially 0
 2 one-bit atomic register door, initially 0
 3 procedure splitter(ID)
 4     for i ← 0 to k − 1 do
 5         A[i][ID_i] ← 1
 6     if door = 1 then
 7         return lose
 8     door ← 1
 9     for i ← 0 to k − 1 do
10         if A[i][¬ID_i] = 1 then
11             return lose
12     return win
```

Algorithm E.6: Splitter using one-bit registers

E.5.4 Symmetric self-stabilizing consensus (20 points)

Suppose we have a synchronous system consisting of processes organized in a connected graph. The state of each process is a single bit, and each process can directly observe the number of neighbors that it has and how many of them have 0 bits and how many have 1 bits. At each round, a process counts the number of neighbors k_0 with zeros, the number k_1 with ones, and its own bit b, and chooses a new bit for the next round $f(b, k_0, k_1)$ according to some rule f that is the same for all processes. The goal of the processes is to reach consensus, where all processes have the same bit forever, starting from an arbitrary initial configuration. An example of a rule that has this property is for f to output 1 if $b = 1$ or $k_1 > 0$.

However, this rule is not symmetric with respect to bit values: if we replace all ones by zeros and vice versa, we get different behavior.

Prove or disprove: There exists a rule f that is symmetric, by which we mean that $f(b, k_0, k_1) = \neg f(\neg b, k_1, k_0)$ always, such that applying this rule starting from an arbitrary configuration in an arbitrary graph eventually converges to all processes having the same bit forever.

Solution

Disproof by counterexample: Fix some f, and consider a graph with two processes p_0 and p_1 connected by an edge. Let p_0 start with 0 and p_1 start with 1. Then p_0's next state is $f(0,0,1) = \neg f(1,1,0) \neq f(1,1,0)$, which is p_1's next state. So either p_0 still has 0 and p_1 still has 1, in which case we never make progress; or they swap their bits, in which case we can apply the same analysis with p_0 and p_1 reversed to show that they continue to swap back and forth forever. In either case the system does not converge.

Appendix F

Sample assignments from Spring 2014

F.1 Assignment 1: due Wednesday, 2014-01-29, at 5:00pm

Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

F.1.1 Counting evil processes

A connected bidirectional asynchronous network of n processes with identities has diameter D and may contain zero or more evil processes. Fortunately, the evil processes, if they exist, are not Byzantine, fully conform to RFC 3514 [Bel03], and will correctly execute any code we provide for them.

Suppose that all processes wake up at time 0 and start whatever protocol we have given them. Suppose that each process initially knows whether it is

evil, and knows the identities of all of its neighbors. However, the processes do not know the number of processes n or the diameter of the network D.

Give a protocol that allows every process to correctly return the number of evil processes no later than time D. Your protocol should only return a value once for each process (no converging to the correct answer after an initial wrong guess).

Solution

There are a lot of ways to do this. Since the problem doesn't ask about message complexity, we'll do it in a way that optimizes for algorithmic simplicity.

At time 0, each process initiates a separate copy of the flooding algorithm (Algorithm 3.1). The message $\langle p, N(p), e\rangle$ it distributes consists of its own identity, the identities of all of its neighbors, and whether or not it is evil.

In addition to the data for the flooding protocol, each process tracks a set I of all processes it has seen that initiated a protocol and a set N of all processes that have been mentioned as neighbors. The initial values of these sets for process p are $\{p\}$ and $N(p)$, the neighbors of p.

Upon receiving a message $\langle q, N(q), e\rangle$, a process adds q to I and $N(q)$ to N. As soon as $I = N$, the process returns a count of all processes for which $e =$ **true**.

Termination by D: Follows from the same analysis as flooding. Any process at distance d from p has $p \in I$ by time d, so I is complete by time D.

Correct answer: Observe that $N = \bigcup_{i \in I} N(i)$ always. Suppose that there is some process q that is not in I. Since the graph is connected, there is a path from p to q. Let r be the last node in this path in I, and let s be the following node. Then $s \in N \setminus I$ and $N \neq I$. By contraposition, if $I = N$ then I contains all nodes in the network, and so the count returned at this time is correct.

F.1.2 Avoiding expensive processes

Suppose that you have a bidirectional but not necessarily complete asynchronous message-passing network represented by a graph $G = (V, E)$ where each node in V represents a process and each edge in E connects two processes that can send messages to each other. Suppose further that each process is assigned a weight 1 or 2. Starting at some initiator process, we'd like to construct a shortest-path tree, where each process points to one of

its neighbors as its parent, and following the parent pointers always gives a path of minimum total weight to the initiator.[1]

Give a protocol that solves this problem with reasonable time, message, and bit complexity, and show that it works.

Solution

There's an ambiguity in the definition of total weight: does it include the weight of the initiator and/or the initial node in the path? But since these values are the same for all paths to the initiator from a given process, they don't affect which is lightest.

If we don't care about bit complexity, there is a trivial solution: Use an existing BFS algorithm followed by convergecast to gather the entire structure of the network at the initiator, run your favorite single-source shortest-path algorithm there, then broadcast the results. This has time complexity $O(D)$ and message complexity $O(DE)$ if we use the BFS algorithm from §4.3. But the last couple of messages in the convergecast are going to be pretty big.

A solution by reduction: Suppose that we construct a new graph G' where each weight-2 node u in G is replaced by a clique of nodes $u_1, u_2, \ldots u_k$, with each node in the clique attached to a different neighbor of u. We then run any breadth-first search protocol of our choosing on G', where each weight-2 node simulates all members of the corresponding clique. Because any path that passes through a clique picks up an extra edge, each path in the breadth-first search tree has a length exactly equal to the sum of the weights of the nodes other than its endpoints.

A complication is that if I am simulating k nodes, between them they may have more than one parent pointer. So we define $u.\mathsf{parent}$ to be $u_i.\mathsf{parent}$ where u_i is a node at minimum distance from the initiator in G'. We also re-route any incoming pointers to $u_j \neq u_i$ to point to u_i instead. Because u_i was chosen to have minimum distance, this never increases the length of any path, and the resulting modified tree is a still a shortest-path tree.

Adding nodes blows up $|E'|$, but we don't need to actually send messages between different nodes u_i represented by the same process. So if we use the §4.3 algorithm again, we only send up to D messages per real edge, giving $O(D)$ time and $O(DE)$ messages.

If we don't like reductions, we could also tweak one of our existing algorithms. Gallager's layered BFS (§4.2) is easily modified by changing the

[1] Clarification added 2014-01-26: The actual number of hops is not relevant for the construction of the shortest-path tree. By shortest path, we mean path of minimum total weight.

depth bound for each round to a total-weight bound. The synchronizer-based BFS can also be modified to work, but the details are messy.

F.2 Assignment 2: due Wednesday, 2014-02-12, at 5:00pm

F.2.1 Synchronous agreement with weak failures

Suppose that we modify the problem of synchronous agreement with crash failures from Chapter 9 so that instead of crashing a process forever, the adversary may jam some or all of its outgoing messages for a single round. The adversary has limited batteries on its jamming equipment, and can only cause f such one-round faults. There is no restriction on when these one-round jamming faults occur: the adversary might jam f processes for one round, one process for f rounds, or anything in between, so long as the sum over all rounds of the number of processes jammed in each round is at most f. For the purposes of agreement and validity, assume that a process is non-faulty if it is never jammed.[2]

As a function of f and n, how many rounds does it take to reach agreement in the worst case in this model, under the usual assumptions that processes are deterministic and the algorithm must satisfy agreement, termination, and validity? Give the best upper and lower bounds that you can.

Solution

The par solution for this is an $\Omega(\sqrt{f})$ lower bound and $O(f)$ upper bound. I don't know if it is easy to do better than this.

For the lower bound, observe that the adversary can simulate an ordinary crash failure by jamming a process in every round starting in the round it crashes in. This means that in an r-round protocol, we can simulate k crash failures with kr jamming faults. From the Dolev-Strong lower bound [DS83] (see also Chapter 9), we know that there is no r-round protocol with $k = r$ crash failures faults, so there is no r-round protocol with r^2 jamming faults. This gives a lower bound of $\lfloor\sqrt{f}\rfloor + 1$ on the number of rounds needed to solve synchronous agreement with f jamming faults.[3]

[2] Clarifications added 2014-02-10: We assume that processes don't know that they are being jammed or which messages are lost (unless the recipient manages to tell them that a message was not delivered). As in the original model, we assume a complete network and that all processes have known identities.

[3] Since Dolev-Strong only needs to crash one process per round, we don't really need

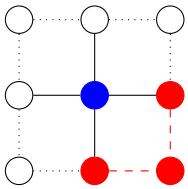

Figure F.1: Connected Byzantine nodes take over half a cut

For the upper bound, have every process broadcast its input every round. After $f+1$ rounds, there is at least one round in which no process is jammed, so every process learns all the inputs and can take, say, the majority value.

F.2.2 Byzantine agreement with contiguous faults

Suppose that we restrict the adversary in Byzantine agreement to corrupting a connected subgraph of the network; the idea is that the faulty nodes need to coordinate, but can't relay messages through the non-faulty nodes to do so.

Assume the usual model for Byzantine agreement with a network in the form of an $m \times m$ torus. This means that each node has a position (x, y) in $\{0, \ldots, m-1\} \times \{0, \ldots, m-1\}$, and its neighbors are the four nodes $(x+1 \bmod m, y)$, $(x-1 \bmod m, y)$, $(x, y+1 \bmod m)$, and $(x, y-1 \bmod m)$.

For sufficiently large m,[4] what is the largest number of faults f; that this system can tolerate and still solve Byzantine agreement?

Solution

The relevant bound here is the requirement that the network have enough connectivity that the adversary can't take over half of a vertex cut (see §10.1.3). This is complicated slightly by the requirement that the faulty nodes be contiguous.

The smallest vertex cut in a sufficiently large torus consists of the four neighbors of a single node; however, these nodes are not connected. But we can add a third node to connect two of them (see Figure F.1).

By adapting the usual lower bound we can use this construction to show that $f = 3$ faults are enough to prevent agreement when $m \geq 3$. The question

the full r jamming faults for processes that crash late. This could be used to improve the constant for this argument.

[4] Problem modified 2014-02-03. In the original version, it asked to compute f for all m, but there are some nasty special cases when m is small.

then is whether $f = 2$ faults is enough.

By a case analysis, we can show that any two nodes in a sufficiently large torus are either adjacent themselves or can be connected by three paths, where no two paths have adjacent vertices. Assume without loss of generality that one of the nodes is at position $(0, 0)$. Then any other node is covered by one of the following cases:

1. Nodes adjacent to $(0, 0)$. These can communicate directly.

2. Nodes at $(0, i)$ or $(i, 0)$. These cases are symmetric, so we'll describe the solution for $(0, i)$. Run one path directly north: $(0, 1), (0, 2), \ldots, (0, i - 1)$. Similarly, run a second path south: $(0, -1), (0, -2), \ldots (0, i + 1)$. For the third path, take two steps east and then run north and back west: $(1, 0), (2, 0), (2, 1), (2, 2), \ldots, (2, i), (1, i)$. These paths are all non-adjacent as long as $m \geq 4$.

3. Nodes at $(\pm 1, i)$ or $(i, \pm 1)$, where i is not -1, 0, or 1. Suppose the node is at $(1, i)$. Run one path east then north through $(1, 0), (1, 1), \ldots, (1, i - 1)$. The other two paths run south and west, with a sideways jog in the middle as needed. This works for m sufficiently large to make room for the sideways jogs.

4. Nodes at $(\pm 1, \pm 1)$ or (i, j) where neither of i or j is -1, 0, or 1. Now we can run one path north then east, one east then north, one south then west, and one west then south, creating four paths in a figure-eight pattern centered on $(0, 0)$.

F.3 Assignment 3: due Wednesday, 2014-02-26, at 5:00pm

F.3.1 Among the elect

The adversary has decided to be polite and notify each non-faulty processes when he gives up crashing it. Specifically, we have the usual asynchronous message-passing system with up to f faulty processes, but every non-faulty process is eventually told that it is non-faulty. (Faulty processes are told nothing.)

For what values of f can you solve consensus in this model?

Solution

We can tolerate $f < n/2$, but no more.

If $f < n/2$, the following algorithm works: Run Paxos, where each process i waits to learn that it is non-faulty, then acts as a proposer for proposal number i. The highest-numbered non-faulty process then carries out a proposal round that succeeds because no higher proposal is ever issued, and both the proposer (which is non-faulty) and a majority of accepters participate.

If $f \geq n/2$, partition the processes into two groups of size $\lfloor n/2 \rfloor$, with any leftover process crashing immediately. Make all of the processes in both groups non-faulty, and tell each of them this at the start of the protocol. Now do the usual partitioning argument: Run group 0 with inputs 0 with no messages delivered from group 1 until all processes decide 0 (we can do this because the processes can't distinguish this execution from one in which the group 1 processes are in fact faulty). Run group 1 similarly until all processes decide 1. We have then violated agreement, assuming we didn't previously violate termination of validity.

F.3.2 Failure detectors on the cheap

Suppose we do not have the budget to equip all of our machines with failure detectors. Instead, we order an eventually strong failure detector for k machines, and the remaining $n - k$ machines get fake failure detectors that never suspect anybody. The choice of which machines get the real failure detectors and which get the fake ones is under the control of the adversary.

This means that every faulty process is eventually permanently suspected by every non-faulty process with a real failure detector, and there is at least one non-faulty process that is eventually permanently not suspected by anybody. Let's call the resulting failure detector $\Diamond S_k$.

Let f be the number of actual failures. Under what conditions on f, k, and n can you still solve consensus in the usual deterministic asynchronous message-passing model using $\Diamond S_k$?

Solution

First observe that $\Diamond S$ can simulate $\Diamond S_k$ for any k by having $n - k$ processes ignore the output of their failure detectors. So we need $f < n/2$ by the usual lower bound on $\Diamond S$.

If $f \geq k$, we are also in trouble. The $f > k$ case is easy: If there exists a consensus protocol for $f > k$, then we can transform it into a consensus protocol for $n - k$ processes and $f - k$ failures, with no failure detectors at all, by pretending that there are an extra k processes with real failure detectors

that crash immediately. The FLP impossibility result rules this out.

If $f = k$, we have to be a little more careful. By immediately crashing $f - 1$ processes with real failure detectors, we can reduce to the $f = k = 1$ case. Now the adversary runs the FLP strategy. If no processes crash, then all $n - k + 1$ surviving process report no failures; if it becomes necessary to crash a process, this becomes the one remaining process with the real failure detector. In either case the adversary successfully prevents consensus.

So let $f < k$. Then we have weak completeness, because every faulty process is eventually permanently suspected by at least $k - f > 0$ processes. We also have weak accuracy, because it is still the case that some process is eventually permanently never suspected by anybody. By boosting weak completeness to strong completeness as described in §13.2.3, we can turn out failure detector into $\Diamond S$, meaning we can solve consensus precisely when $f < \min(k, n/2)$.

F.4 Assignment 4: due Wednesday, 2014-03-26, at 5:00pm

F.4.1 A global synchronizer with a global clock

Consider an asynchronous message-passing system with n processes in a bidirectional ring with no failures. Suppose that the processes are equipped with a global clock, which causes a local event to occur simultaneously at each process every c time units, where as usual 1 is the maximum message delay. We would like to use this global clock to build a global synchronizer. Provided c is at least 1, a trivial approach is to have every process advance to the next round whenever the clock pulse hits. This gives one synchronous round every c time units.

Suppose that c is greater than 1 but still $o(n)$. Is it possible to build a global synchronizer in this model that runs more than a constant ratio faster than this trivial global synchronizer in the worst case?

Solution

No. We can adapt the lower bound on the session problem from §7.4.2 to apply in this model.

Consider an execution of an algorithm for the session problem in which each message is delivered exactly one time unit after it is sent. Divide it as in the previous proof into a prefix β containing special actions and a suffix δ containing no special actions. Divide β further into segments

$\beta_1, \beta_2, \beta_3, \ldots, \beta_k$, where each segment ends with a clock pulse. Following the standard argument, because each segment has duration less than the diameter of the network, there is no causal connection between any special actions done by processes at opposite ends of the network that are in the same segment β_i. So we can causally shuffle each β_i to get a new segment β_i' where all special actions of process p_0 (say) occur before all special actions of process $p_{n/2}$. This gives at most one session per segment, or at most one session for every c time units.

Since a globally synchronous system can do one session per round, this means that our global synchronizer can only produce one session per c time units as well.

F.4.2 A message-passing counter

A **counter** is a shared object that support operations `inc` and `read`, where `read` returns the number of previous `inc` operations.

Algorithm F.1 purports to implement a counter in an asynchronous message-passing system subject to $f < n/2$ crash failures. In the algorithm, each process i maintains a vector c_i of contributions to the counter from all the processes, as well as a nonce r_i used to distinguish responses to different read operations from each other. All of these values are initially zero.

Show that the implemented counter is linearizable, or give an example of an execution where it isn't.

Solution

This algorithm is basically implementing an array of ABD registers [ABND95], but it omits the second phase on a `read` where any information the reader learns is propagated to a majority. So we expect it to fail the same way ABD would without this second round, by having two `read` operations return values that are out of order with respect to their observable ordering.

Here is one execution that produces this bad outcome:

1. Process p_1 starts an `inc` by updating $c_1[1]$ to 1.

2. Process p_2 carries out a `read` operation in which it receives responses from p_1 and p_2, and returns 1.

3. After p_2 finishes, process p_3 carries out a `read` operation in which it receives responses from p_2 and p_3, and returns 0.

```
1  procedure inc
2      c_i[i] ← c_i[i] + 1
3      Send c_i[i] to all processes.
4      Wait to receive ack(c_i[i]) from a majority of processes.
5  upon receiving c from j do
6      c_i[j] ← max(c_i[j], c)
7      Send ack(c) to j.
8  procedure read
9      r_i ← r_i + 1
10     Send read(r_i) to all processes.
11     Wait to receive respond(r_i, c_j) from a majority of processes j.
12     return ∑_k max_j c_j[k]
13 upon receiving read(r) from j do
14     Send respond(r, c_i) to j
```

Algorithm F.1: Counter algorithm for Problem F.4.2.

If we want to be particularly perverse, we can exploit the fact that p_2 doesn't record what it learns in its first `read` to have p_2 do the second `read` that returns 0 instead of p_3. This shows that Algorithm F.1 isn't even sequentially consistent.

The patch, if we want to fix this, is to include the missing second phase from ABD in the `read` operation: after receiving values from a majority, I set $c_i[k]$ to $\max_j c_j[k]$ and send my updated values to a majority. That the resulting counter is linearizable is left as an exercise.

F.5 Assignment 5: due Wednesday, 2014-04-09, at 5:00pm

F.5.1 A concurrency detector

Consider the following optimistic mutex-like object, which we will call a **concurrency detector**. A concurrency detector supports two operations for each process i, $\texttt{enter}_i$ and $\texttt{exit}_i$. These operations come in pairs: a process enters a critical section by executing $\texttt{enter}_i$, and leaves by executing $\texttt{exit}_i$. The behavior of the object is undefined if a process calls $\texttt{enter}_i$ twice without an intervening $\texttt{exit}_i$, or calls $\texttt{exit}_i$ without first calling $\texttt{enter}_i$.

Unlike mutex, a concurrency detector does not enforce that only one

process is in the critical section at a time; instead, $\texttt{exit}_i$ returns 1 if the interval between it and the previous $\texttt{enter}_i$ overlaps with some interval between a $\texttt{enter}_j$ and corresponding $\texttt{exit}_j$ for some $j \neq i$, and returns 0 if there is no overlap.

Is there a deterministic linearizable wait-free implementation of a concurrency detector from atomic registers? If there is, give an implementation. If there is not, give an impossibility proof.

Solution

It is not possible to implement this object using atomic registers.

Suppose that there were such an implementation. Algorithm F.2 implements two-process consensus using a two atomic registers and a single concurrency detector, initialized to the state following $\texttt{enter}_1$.

```
1  procedure consensus1(v)
2      r1 ← v
3      if exit1() = 1 then
4          return r2
5      else
6          return v
7  procedure consensus2(v)
8      r2 ← v
9      enter2()
10     if exit2() = 1 then
11         return v
12     else
13         return r1
```

Algorithm F.2: Two-process consensus using the object from Problem F.5.1

Termination is immediate from the absence of loops in the code.

To show validity and termination, observe that one of two cases holds:

1. Process 1 executes $\texttt{exit}_1$ before process 2 executes $\texttt{enter}_2$. In this case there is no overlap between the interval formed by the implicit $\texttt{enter}_1$ and $\texttt{exit}_1$ and the interval formed by $\texttt{enter}_2$ and $\texttt{exit}_2$. So the $\texttt{exit}_1$ and $\texttt{exit}_2$ operations both return 0, causing process 1 to return its own value and process 2 to return the contents of r_1. These

will equal process 1's value, because process 2's read follows its call to $\texttt{enter}_2$, which follows $\texttt{exit}_1$ and thus process 1's write to r_1.

2. Process 1 executes $\texttt{exit}_1$ after process 2 executes $\texttt{enter}_2$. Now both `exit` operations return 1, and so process 2 returns its own value while process 1 returns the contents of r_2, which it reads after process 2 writes its value there.

In either case, both processes return the value of the first process to access the concurrency detector, satisfying both agreement and validity. This would give a consensus protocol for two processes implemented from atomic registers, contradicting the impossibility result of Loui and Abu-Amara [LAA87].

F.5.2 Two-writer sticky bits

A **two-writer sticky bit** is a sticky bit that can be read by any process, but that can only be written to by two specific processes.

Suppose that you have an unlimited collection of two-writer sticky bits for each pair of processes, plus as many ordinary atomic registers as you need. What is the maximum number of processes for which you can solve wait-free binary consensus?

Solution

If $n = 2$, then a two-writer sticky bit is equivalent to a sticky bit, so we can solve consensus.

If $n \geq 3$, suppose that we maneuver our processes as usual to a bivalent configuration C with no bivalent successors. Then there are three pending operations x, y, and z, that among them produce both 0-valent and 1-valent configurations. Without loss of generality, suppose that Cx and Cy are both 0-valent and Cz is 1-valent. We now consider what operations these might be.

1. If x and z apply to different objects, then $Cxz = Czx$ must be both 0-valent and 1-valent, a contradiction. Similarly if y and z apply to different objects. This shows that all three operations apply to the same object O.

2. If O is a register, then the usual case analysis of Loui and Abu-Amara [LAA87] gives us a contradiction.

3. If O is a two-writer sticky bit, then we can split cases further based on z:

(a) If z is a read, then either:
   i. At least one of x and y is a read. But then $Cxz = Czx$ or $Cyz = Czy$, and we are in trouble.
   ii. Both x and y are writes. But then Czx (1-valent) is indistinguishable from Cx (0-valent) by the two processes that didn't perform z: more trouble.

(b) If z is a write, then at least one of x or y is a read; suppose it's x. Then Cxz is indistinguishable from Cz by the two processes that didn't perform x.

Since we reach a contradiction in all cases, it must be that when $n \geq 3$, every bivalent configuration has a bivalent successor, which shows that we can't solve consensus in this case. The maximum value of n for which we can solve consensus is 2.

F.6 Assignment 6: due Wednesday, 2014-04-23, at 5:00pm

F.6.1 A rotate register

Suppose that you are asked to implement a concurrent m-bit register that supports in addition to the usual `read` and `write` operations a `RotateLeft` operation that rotates all the bits to the left; this is equivalent to doing a left shift (multiplying the value in the register by two) followed by replacing the lowest-order bit with the previous highest-order bit.

For example, if the register contains 1101, and we do `RotateLeft`, it now contains 1011.

Show that if m is sufficiently large as a function of the number of processes n, $\Theta(n)$ steps per operation in the worst case are necessary and sufficient to implement a linearizable wait-free m-bit shift register from atomic registers.

Solution

The necessary part is easier, although we can't use JTT (Chapter 21) directly because having write operations means that our rotate register is not perturbable. Instead, we argue that if we initialize the register to 1, we get a mod-m counter, where increment is implemented by `RotateLeft` and read is implemented by taking the log of the actual value of the counter. Letting $m \geq 2n$ gives the desired $\Omega(n)$ lower bound, since a mod-$2n$ counter *is* perturbable.

For sufficiency, we'll show how to implement the rotate register using snapshots. This is pretty much a standard application of known techniques [AH90b, AM93], but it's not a bad exercise to write it out.

Pseudocode for one possible solution is given in Algorithm F.3.

The register is implemented using a single snapshot array A. Each entry in the snapshot array holds four values: a timestamp and process ID indicating which write the process's most recent operations apply to, the initial write value corresponding to this timestamp, and the number of rotate operations this process has applied to this value. A write operation generates a new timestamp, sets the written value to its input, and resets the rotate count to 0. A rotate operation updates the timestamp and associated write value to the most recent that the process sees, and adjusts the rotate count as appropriate. A read operation combines all the rotate counts associated with the most recent write to obtain the value of the simulated register.

```
1  procedure write(A, v)
2      s ← snapshot(A)
3      A[id] ← ⟨max_i s[i].timestamp + 1, id, v, 0⟩
4  procedure RotateLeft(A)
5      s ← snapshot(A)
6      Let i maximize ⟨s[i].timestamp, s[i].process⟩
7      if s[i].timestamp = A[id].timestamp and
       s[i].process = A[id].process then
           // Increment my rotation count
8          A[id].rotations ← A[id].rotations + 1
9      else
           // Reset and increment my rotation count
10         A[id] ← ⟨s[i].timestamp, s[i].process, s[i].value, 1⟩
11 procedure read(A)
12     s ← snapshot(A)
13     Let i maximize ⟨s[i].timestamp, s[i].process⟩
14     Let
       r = ∑_{j, s[j].timestamp=s[i].timestamp ∧ s[j].process=s[i].process} s[j].rotations
15     return s[i].value rotated r times.
```

Algorithm F.3: Implementation of a rotate register

Since each operation requires one snapshot and at most one update, the cost is $O(n)$ using the linear-time snapshot algorithm of Inoue *et al.* [IMCT94].

Linearizability is easily verified by observing that ordering all operations by the maximum timestamp/process tuple that they compute and then by the total number of rotations that they observe produces an ordering consistent with the concurrent execution for which all return values of reads are correct.

F.6.2 A randomized two-process test-and-set

Algorithm F.4 gives pseudocode for a protocol for two processes p_0 and p_1. It uses two shared unbounded single-writer atomic registers r_0 and r_1, both initially 0. Each process also has a local variable s.

```
1  procedure TAS_i()
2      while true do
3          with probability 1/2 do
4              r_i ← r_i + 1
5          else
6              r_i ← r_i
7          s ← r_¬i
8          if s > r_i then
9              return 1
10         else if s < r_i − 1 do
11             return 0
```

Algorithm F.4: Randomized two-process test-and-set for F.6.2

1. Show that any return values of the protocol are consistent with a linearizable, single-use test-and-set.

2. Will this protocol always terminate with probability 1 assuming an oblivious adversary?

3. Will this protocol always terminate with probability 1 assuming an adaptive adversary?

Solution

1. To show that this implements a linearizable test-and-set, we need to show that exactly one process returns 0 and the other 1, and that if one process finishes before the other starts, the first process to go returns 1.

Suppose that p_i finishes before $p_{\neg i}$ starts. Then p_i reads only 0 from $r_{\neg i}$, and cannot observe $r_i < r_{\neg i}$: p_i returns 0 in this case.

We now show that the two processes cannot return the same value. Suppose that both processes terminate. Let i be such that p_i reads $r_{\neg i}$ for the last time before $p_{\neg i}$ reads r_i for the last time. If p_i returns 0, then it observes $r_i \geq r_{\neg i} + 2$ at the time of its read; $p_{\neg i}$ can increment $r_{\neg i}$ at most once before reading r_i again, and so observed $r_{\neg i} < r_i$ and returns 1.

Alternatively, if p_i returns 1, it observed $r_i < r_{\neg i}$. Since it performs no more increments on r_i, p_i also observes $r_i < r_{\neg i}$ in all subsequent reads, and so cannot also return 1.

2. Let's run the protocol with an oblivious adversary, and track the value of $r_0^t - r_1^t$ over time, where r_i^t is the value of r_i after t writes (to either register). Each write to r_0 increases this value by 1/2 on average, with a change of 0 or 1 equally likely, and each write to r_1 decreases it by 1/2 on average.

   To make things look symmetric, let Δ^t be the change caused by the t-th write and write Δ^t as $c^t + X^t$ where $c^t = \pm 1/2$ is a constant determined by whether p_0 or p_1 does the t-th write and $X^t = \pm 1/2$ is a random variable with expectation 0. Observe that the X^t variables are independent of each other and the constants c^t (which depend only on the schedule).

   For the protocol to run forever, at every time t it must hold that $\left|r_0^t - r_1^t\right| \leq 3$; otherwise, even after one or both processes does its next write, we will have $\left|r_0^{t'} - r_1^{t'}\right|$ and the next process to read will terminate. But

$$\begin{aligned}\left|r_0^t - r_1^t\right| &= \left|\sum_{s=1}^{t} \Delta^s\right| \\ &= \left|\sum_{s=1}^{t} (c_s + X_s)\right| \\ &= \left|\sum_{s=1}^{t} c_s + \sum_{s=1}^{t} X_s\right|.\end{aligned}$$

   The left-hand sum is a constant, while the right-hand sum has a binomial distribution. For any fixed constant, the probability that a binomial distribution lands within ± 2 of the constant goes to zero in

the limit as $t \to \infty$, so with probability 1 there is some t for which this event does not occur.

3. For an adaptive adversary, the following strategy prevents agreement:

   (a) Run p_0 until it is about to increment r_0.

   (b) Run p_1 until it is about to increment r_1.

   (c) Allow both increments to proceed and repeat.

   The effect is that both processes always observe $r_0 = r_1$ whenever they do a read, and so never finish. This works because the adaptive adversary can see the coin-flips done by the processes before they act on them; it would not work with an oblivious adversary or in a model that supported probabilistic writes.

F.7 CS465/CS565 Final Exam, May 2nd, 2014

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.

F.7.1 Maxima (20 points)

Some deterministic processes organized in an anonymous, synchronous ring are each given an integer input (which may or may not be distinct from other processes' inputs), but otherwise run the same code and do not know the size of the ring. We would like the processes to each compute the maximum input. As usual, each process may only return an output once, and must do so after a finite number of rounds, although it may continue to participate in the protocol (say, by relaying messages) even after it returns an output.

Prove or disprove: It is possible to solve this problem in this model.

Solution

It's not possible.

Consider an execution with $n = 3$ processes, each with input 0. If the protocol is correct, then after some finite number of rounds t, each process returns 0. By symmetry, the processes all have the same states and send the same messages throughout this execution.

Now consider a ring of size $2(t+1)$ where every process has input 0, except for one process p that has input 1. Let q be the process at maximum distance from p. By induction on r, we can show that after r rounds of communication, every process that is more than $r+1$ hops away from p has the same state as all of the processes in the 3-process execution above. So in particular, after t rounds, process q (at distance $t+1$) is in the same state as it would be in the 3-process execution, and thus it returns 0. But—as it learns to its horror, one round too late—the correct maximum is 1.

F.7.2 Historyless objects (20 points)

Recall that a shared-memory object is **historyless** if any operation on the object either (a) always leaves the object in the same state as before the operation, or (b) always leaves the object in a new state that doesn't depend on the state before the operation.

What is the maximum possible consensus number for a historyless object? That is, for what value n is it possible to solve wait-free consensus for n processes using some particular historyless object but not possible to solve wait-free consensus for $n+1$ processes using any historyless object?

Solution

Test-and-sets are (a) historyless, and (b) have consensus number 2, so n is at least 2.

To show that no historyless object can solve wait-free 3-process consensus, consider an execution that starts in a bivalent configuration and runs to a configuration C with two pending operations x and y such that Cx is 0-valent and Cy is 1-valent. By the usual arguments x and y must both be operations on the same object. If either of x and y is a read operation, then (0-valent) Cxy and (1-valent) Cyx are indistinguishable to a third process p_z if run alone, because the object is left in the same state in both configurations; whichever way p_z decides, it will give a contradiction in an execution starting with one of these configurations. If neither of x and y is a read, then x overwrites y, and Cx is indistinguishable from $Cyx$to p_z if p_z runs alone; again we get a contradiction.

F.7.3 Hams (20 points)

Hamazon, LLC, claims to be the world's biggest delivery service for canned hams, with guaranteed delivery of a canned ham to your home anywhere on Earth via suborbital trajectory from secret launch facilities at the North

and South Poles. Unfortunately, these launch facilities may be subject to crash failures due to inclement weather, trademark infringement actions, or military retaliation for misdirected hams.

For this problem, you are to evaluate Hamazon's business model from the perspective of distributed algorithms. Consider a system consisting of a client process and two server processes (corresponding to the North and South Pole facilities) that communicate by means of asynchronous message passing. In addition to the usual message-passing actions, each server also has an irrevocable `launch` action that launches a ham at the client. As with messages, hams are delivered asynchronously: it is impossible for the client to tell if a ham has been launched until it arrives.

A ham protocol is correct provided (a) a client that orders no ham receives no ham; and (b) a client that orders a ham receives exactly one ham. Show that there can be no correct deterministic protocol for this problem if one of the servers can crash.

Solution

Consider an execution in which the client orders ham. Run the northern server together with the client until the server is about to issue a `launch` action (if it never does so, the client receives no ham when the southern server is faulty).

Now run the client together with the southern server. There are two cases:

1. If the southern server ever issues `launch`, execute both this and the northern server's `launch` actions: the client gets two hams.

2. If the southern server never issues `launch`, never run the northern server again: the client gets no hams.

In either case, the one-ham rule is violated, and the protocol is not correct.[5]

[5]It's tempting to try to solve this problem by reduction from a known impossibility result, like Two Generals or FLP. For these specific problems, direct reductions don't appear to work. Two Generals assumes message loss, but in this model, messages are not lost. FLP needs any process to be able to fail, but in this model, the client never fails. Indeed, we can solve consensus in the Hamazon model by just having the client transmit its input to both servers.

F.7.4 Mutexes (20 points)

A swap register s has an operation `swap`(s, v) that returns the argument to the previous call to `swap`, or $\perp$ if it is the first such operation applied to the register. It's easy to build a mutex from a swap register by treating it as a test-and-set: to grab the mutex, I swap in 1, and if I get back $\perp$ I win (and otherwise try again); and to release the mutex, I put back $\perp$.

Unfortunately, this implementation is not starvation-free: some other process acquiring the mutex repeatedly might always snatch the $\perp$ away just before I try to swap it out. Algorithm F.5 uses a swap object s along with an atomic register r to try to fix this.

```
1 procedure mutex()
2     predecessor ← swap(s, myId)
3     while r ≠ predecessor do
4         try again
      // Start of critical section
5     ...
      // End of critical section
6     r ← myId
```

Algorithm F.5: Mutex using a swap object and register

Prove that Algorithm F.5 gives a starvation-free mutex, or give an example of an execution where it fails. You should assume that s and r are both initialized to $\perp$.

Solution

Because processes use the same ID if they try to access the mutex twice, the algorithm doesn't work.

Here's an example of a bad execution:

1. Process 1 swaps 1 into s and gets $\perp$, reads $\perp$ from r, performs its critical section, and writes 1 to r.

2. Process 2 swaps 2 into s and gets 1, reads 1 from r, and enters the critical section.

3. Process 1 swaps 1 into s and gets 2, and spins waiting to see 2 in r.

4. Process 3 swaps 3 into s and gets 1. Because r is still 1, process 3 reads this 1 and enters the critical section. We now have two processes in the critical section, violating mutual exclusion.

I believe this works if each process adopts a new ID every time it calls `mutex`, but the proof is a little tricky.[6]

[6]The simplest proof I can come up with is to apply an invariant that says that (a) the processes that have executed `swap`(s, myId) but have not yet left the while loop have predecessor values that form a linked list, with the last pointer either equal to $\perp$ (if no process has yet entered the critical section) or the last process to enter the critical section; (b) r is $\perp$ if no process has yet left the critical section, or the last process to leave the critical section otherwise; and (c) if there is a process that is in the critical section, its predecessor field points to the last process to leave the critical section. Checking the effects of each operation shows that this invariant is preserved through the execution, and (a) combined with (c) show that we can't have two processes in the critical section at the same time. Additional work is still needed to show starvation-freedom. It's a good thing this algorithm doesn't work as written.

Appendix G

Sample assignments from Fall 2011

G.1 Assignment 1: due Wednesday, 2011-09-28, at 17:00

Bureaucratic part

Send me email! My address is aspnes@cs.yale.edu.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

G.1.1 Anonymous algorithms on a torus

An $n \times m$ **torus** is a two-dimensional version of a ring, where a node at position (i, j) has a neighbor to the north at $(i, j-1)$, the east at $(i+1, j)$, the south at $(i, j+1)$, and the west at $(i-1, j)$. These values wrap around modulo n for the first coordinate and modulo m for the second; so $(0, 0)$ has neighbors $(0, m-1)$, $(1, 0)$, $(0, 1)$, and $(n-1, 0)$.

Suppose that we have a synchronous message-passing network in the form of an $n \times m$ torus, consisting of anonymous, identical processes that do not know n, m, or their own coordinates, but do have a sense of direction (meaning they can tell which of their neighbors is north, east, etc.).

Prove or disprove: Under these conditions, there is a deterministic[1] algorithm that computes whether $n > m$.

Solution

Disproof: Consider two executions, one in an $n \times m$ torus and one in an $m \times n$ torus where $n > m$ and both n and m are at least 2.[2] Using the same argument as in Lemma 5.1.1, show by induction on the round number that, for each round r, all processes in both executions have the same state. It follows that if the processes correctly detect $n > m$ in the $n \times m$ execution, then they incorrectly report $m > n$ in the $m \times n$ execution.

G.1.2 Clustering

Suppose that k of the nodes in an asynchronous message-passing network are designated as cluster heads, and we want to have each node learn the identity of the nearest head. Given the most efficient algorithm you can for this problem, and compute its worst-case time and message complexities.

You may assume that processes have unique identifiers and that all processes know how many neighbors they have.[3]

Solution

The simplest approach would be to run either of the efficient distributed breadth-first search algorithms from Chapter 4 simultaneously starting at all cluster heads, and have each process learn the distance to all cluster heads at once and pick the nearest one. This gives $O(D^2)$ time and $O(k(E + VD))$ messages if we use layering and $O(D)$ time and $O(kDE)$ messages using local synchronization.

We can get rid of the dependence on k in the local-synchronization algorithm by running it almost unmodified, with the only difference being the attachment of a cluster head ID to the exactly messages. The simplest way to show that the resulting algorithm works is to imagine coalescing

[1] Clarification added 2011-09-28.

[2] This last assumption is not strictly necessary, but it avoids having to worry about what it means when a process sends a message to itself.

[3] Clarification added 2011-09-26.

all cluster heads into a single initiator; the clustering algorithm effectively simulates the original algorithm running in this modified graph, and the same proof goes through. The running time is still $O(D)$ and the message complexity $O(DE)$.

G.1.3 Negotiation

Two merchants A and B are colluding to fix the price of some valuable commodity, by sending messages to each other for r rounds in a synchronous message-passing system. To avoid the attention of antitrust regulators, the merchants are transmitting their messages via carrier pigeons, which are unreliable and may become lost. Each merchant has an initial price p_A or p_B, which are integer values satisfying $0 \leq p \leq m$ for some known value m, and their goal is to choose new prices p'_A and p'_B, where $|p'_A - p'_B| \leq 1$. If $p_A = p_B$ and no messages are lost, they want the stronger goal that $p'_A = p'_B = p_A = p_B$.

Prove the best lower bound you can on r, as a function of m, for all protocols that achieve these goals.

Solution

This is a thinly-disguised version of the Two Generals Problem from Chapter 8, with the agreement condition $p'_A = p'_B$ replaced by an **approximate agreement** condition $|p'_A - p'_B| \leq 1$. We can use a proof based on the indistinguishability argument in §8.2 to show that $r \geq m/2$.

Fix r, and suppose that in a failure-free execution both processes send messages in all rounds (we can easily modify an algorithm that does not have this property to have it, without increasing r). We will start with a sequence of executions with $p_A = p_B = 0$. Let X_0 be the execution in which no messages are lost, X_1 the execution in which A's last message is lost, X_2 the execution in which both A and B's last messages are lost, and so on, with X_k for $0 \leq k \leq 2r$ losing k messages split evenly between the two processes, breaking ties in favor of losing messages from A.

When i is even, X_i is indistinguishable from X_{i+1} by A; it follows that p'_A is the same in both executions. Because we no longer have agreement, it may be that $p'_B(X_i)$ and $p'_B(X_{i+1})$ are not the same as p'_A in either execution; but since both are within 1 of p'_A, the difference between them is at most 2. Next, because X_{i+1} to X_{i+2} are indistinguishable to B, we have $p'_B(X_{i+1}) = p'_B(X_{i+2})$, which we can combine with the previous claim to get $|p'_B(X_i) - p'_B(X_{i+2})|$. A simple induction then gives $p'_B(X_{2r}) \leq 2r$, where

X_{2r} is an execution in which all messages are lost.

Now construct executions X_{2r+1} and X_{2r+2} by changing p_A and p_B to m one at a time. Using essentially the same argument as before, we get $|p'_B(X_{2r}) - p'_B(X_{2r+2})| \leq 2$ and thus $p'_B(X_{2r+2}) \leq 2r + 2$.

Repeat the initial $2r$ steps backward to get to an execution X_{4r+2} with $p_A = p_B = m$ and no messages lost. Applying the same reasoning as above shows $m = p'_B(X_{4r+2}) \leq 4r + 2$ or $r \geq \frac{m-2}{4} = \Omega(m)$.

Though it is not needed for the solution, it is not too hard to unwind the lower bound argument to extract an algorithm that matches the lower bound up to a small constant factor. For simplicity, let's assume m is even.

The protocol is to send my input in the first message and then use $m/2-1$ subsequent acknowledgments, stopping immediately if I ever fail to receive a message in some round; the total number of rounds r is exactly $m/2$. If I receive s messages in the first s rounds, I decide on $\min(p_A, p_B)$ if that value lies in $[m/2 - s, m/2 + s]$ and the nearest endpoint otherwise. (Note that if $s = 0$, I don't need to compute $\min(p_A, p_B)$, and if $s > 0$, I can do so because I know both inputs.)

This satisfies the approximate agreement condition because if I see only s messages, you see at most $s+1$, because I stop sending once I miss a message. So either we both decide $\min(p_A, p_B)$ or we choose endpoints $m/2 \pm s_A$ and $m/2 \pm s_B$ that are within 1 of each other. It also satisfies the validity condition $p'_A = p'_B = p_A = p_B$ when both inputs are equal and no messages are lost (and even the stronger requirement that $p'_A = p'_B$ when no messages are lost), because in this case $[m/2 - s, m/2 + s]$ is exactly $[0, m]$ and both processes decide $\min(p_A, p_B)$.

There is still a factor-of-2 gap between the upper and lower bounds. My guess would be that the correct bound is very close to $m/2$ on both sides, and that my lower bound proof is not quite clever enough.

G.2 Assignment 2: due Wednesday, 2011-11-02, at 17:00

G.2.1 Consensus with delivery notifications

The FLP bound (Chapter 11) shows that we can't solve consensus in an asynchronous system with one crash failure. Part of the reason for this is that only the recipient can detect when a message is delivered, so the other processes can't distinguish between a configuration in which a message has or has not been delivered to a faulty process.

Suppose that we augment the system so that senders are notified immediately when their messages are delivered. We can model this by making the delivery of a single message an event that updates the state of both sender and recipient, both of which may send additional messages in response. Let us suppose that this includes attempted deliveries to faulty processes, so that any non-faulty process that sends a message m is eventually notified that m has been delivered (although it might not have any effect on the recipient if the recipient has already crashed).

1. Show that this system can solve consensus with one faulty process when $n = 2$.

2. Show that this system cannot solve consensus with two faulty processes when $n = 3$.

Solution

1. To solve consensus, each process sends its input to the other. Whichever input is delivered first becomes the output value for both processes.

2. To show impossibility with $n = 3$ and two faults, run the usual FLP proof until we get to a configuration C with events e' and e such that Ce is 0-valent and $Ce'e$ is 1-valent (or vice versa). Observe that e and e' may involve two processes each (sender and receiver), for up to four processes total, but only a process that is involved in both e and e' can tell which happened first. There can be at most two such processes. Kill both, and get that $Ce'e$ is indistinguishable from Cee' for the remaining process, giving the usual contradiction.

G.2.2 A circular failure detector

Suppose we equip processes $0 \dots n-1$ in an asynchronous message-passing system with n processes subject to crash failures with a failure detector that is strongly accurate (no non-faulty process is ever suspected) and causes process $i+1 \pmod n$ to eventually permanently suspect process i if process i crashes. Note that this failure detector is not even weakly complete (if both i and $i+1$ crash, no non-faulty process suspects i). Note also that the ring structure of the failure detector doesn't affect the actual network: even though only process $i+1 \pmod n$ may suspect process i, any process can send messages to any other process.

Prove the best upper and lower bounds you can on the largest number of failures f that allows solving consensus in this system.

Solution

There is an easy reduction to FLP that shows $f \leq n/2$ is necessary (when n is even), and a harder reduction that shows $f < 2\sqrt{n} - 1$ is necessary. The easy reduction is based on crashing every other process; now no surviving process can suspect any other survivor, and we are back in an asynchronous message-passing system with no failure detector and 1 remaining failure (if f is at least $n/2 + 1$).

The harder reduction is to crash every $(\sqrt{n})$-th process. This partitions the ring into $\sqrt{n}$ segments of length $\sqrt{n} - 1$ each, where there is no failure detector in any segment that suspects any process in another segment. If an algorithm exists that solves consensus in this situation, then it does so even if (a) all processes in each segment have the same input, (b) if any process in one segment crashes, all $\sqrt{n} - 1$ process in the segment crash, and (c) if any process in a segment takes a step, all take a step, in some fixed order. Under this additional conditions, each segment can be simulated by a single process in an asynchronous system with no failure detectors, and the extra $\sqrt{n} - 1$ failures in $2\sqrt{n} - 1$ correspond to one failure in the simulation. But we can't solve consensus in the simulating system (by FLP), so we can't solve it in the original system either.

On the other side, let's first boost completeness of the failure detector, by having any process that suspects another transmit this submission by reliable broadcast. So now if any non-faulty process i suspects $i + 1$, all the non-faulty processes will suspect $i + 1$. Now with up to t failures, whenever I learn that process i is faulty (through a broadcast message passing on the suspicion of the underlying failure detector, I will suspect processes $i + 1$ through $i + t - f$ as well, where f is the number of failures I have heard about directly. I don't need to suspect process $i + t - f + 1$ (unless there is some intermediate process that has also failed), because the only way that this process will not be suspected eventually is if every process in the range i to $i + t - f$ is faulty, which can't happen given the bound t.

Now if t is small enough that I can't cover the entire ring with these segments, then there is some non-faulty processes that is far enough away from the nearest preceding faulty process that it is never suspected: this gives us an eventually strong failure detector, and we can solve consensus using the standard Chandra-Toueg $\Diamond S$ algorithm from §13.4 or [CT96]. The inequality I am looking for is $f(t - f) < n$, where the left-hand side is maximized by setting $f = t/2$, which gives $t^2/4 < n$ or $t < \sqrt{2n}$. This leaves a gap of about $\sqrt{2}$ between the upper and lower bounds; I don't know which one can be improved.

I am indebted to Hao Pan for suggesting the $\Theta(\sqrt{n})$ upper and lower bounds, which corrected an error in my original draft solution to this problem.

G.2.3 An odd problem

Suppose that each of n processes in a message-passing system with a complete network is attached to a sensor. Each sensor has two states, *active* and *inactive*; initially, all sensors are off. When the sensor changes state, the corresponding process is notified immediately, and can update its state and send messages to other processes in response to this event. It is also guaranteed that if a sensor changes state, it does not change state again for at least two time units. We would like to detect when an odd number of sensors are active, by having at least one process update its state to set off an alarm at a time when this condition holds.

A correct protocol for this problem should satisfy two conditions:

No false positives If a process sets of an alarm, then an odd number of sensors are active.

Termination If at some time an odd number of sensors are active, and from that point on no sensor changes its state, then some process eventually sets off an alarm.

For what values of n is it possible to construct such a protocol?

Solution

It is feasible to solve the problem for $n < 3$.

For $n = 1$, the unique process sets off its alarm as soon as its sensor becomes active.

For $n = 2$, have each process send a message to the other containing its sensor state whenever the sensor state changes. Let s_1 and s_2 be the state of the two process's sensors, with 0 representing inactive and 1 active, and let p_i set off its alarm if it receives a message s such that $s \oplus s_i = 1$. This satisfies termination, because if we reach a configuration with an odd number of active sensors, the last sensor to change causes a message to be sent to the other process that will cause it to set off its alarm. It satisfies no-false-positives, because if p_i sets off its alarm, then $s_{\neg i} = s$ because at most one time unit has elapsed since $p_{\neg i}$ sent s; it follows that $s_{\neg i} \oplus s_i = 1$ and an odd number of sensors are active.

No such protocol is possible for $n \geq 3$. Make p_1's sensor active. Run the protocol until some process p_i is about to enter an alarm state (this occurs

eventually because otherwise we violate termination). Let p_j be one of p_2 or p_3 with $j \neq i$, activate p_j's sensor (we can do this without violating the once-per-time-unit restriction because it has never previously been activated) and then let p_i set off its alarm. We have now violated no-false-positives.

G.3 Assignment 3: due Friday, 2011-12-02, at 17:00

G.3.1 A restricted queue

Suppose you have an atomic queue Q that supports operations `enq` and `deq`, restricted so that:

- `enq`(Q) always pushes the identity of the current process onto the tail of the queue.
- `deq`(Q) tests if the queue is nonempty and its head is equal to the identity of the current process. If so, it pops the head and returns **true**. If not, it does nothing and returns **false**.

The rationale for these restrictions is that this is the minimal version of a queue needed to implement a starvation-free mutex using Algorithm 18.2.

What is the consensus number of this object?

Solution

The restricted queue has consensus number 1.

Suppose we have 2 processes, and consider all pairs of operations on Q that might get us out of a bivalent configuration C. Let x be an operation carried out by p that leads to a b-valent state, and y an operation by q that leads to a $(\neg b)$-valent state. There are three cases:

- Two `deq` operations. If Q is empty, the operations commute. If the head of the Q is p, then y is a no-op and p can't distinguish between Cx and Cyx. Similarly for q if the head is q.
- One `enq` and one `deq` operation. Suppose x is an `enq` and y a `deq`. If Q is empty or the head is not q, then y is a no-op: p can't distinguish Cx from Cyx. If the head is q, then x and y commute. The same holds in reverse if x is a `deq` and y an `enq`.

- Two `enq` operations. This is a little tricky, because Cxy and Cyx are different states. However, if Q is nonempty in C, whichever process isn't at the head of Q can't distinguish them, because any `deq` operation returns false and never reaches the newly-enqueued values. This leaves the case where Q is empty in C. Run p until it is poised to do $x' = \texttt{deq}(Q)$ (if this never happens, p can't distinguish Cxy from Cyx); then run q until it is poised to do $y' = \texttt{deq}(Q)$ as well (same argument as for p). Now allow both `deq` operations to proceed in whichever order causes them both to succeed. Since the processes can't tell which `deq` happened first, they can't tell which `enq` happened first either. Slightly more formally, if we let α be the sequence of operations leading up to the two `deq` operations, we've just shown $Cxy\alpha x'y'$ is indistinguishable from $Cyx\alpha y'x'$ to both processes.

In all cases, we find that we can't escape bivalence. It follows that Q can't solve 2-process consensus.

G.3.2 Writable fetch-and-increment

Suppose you are given an unlimited supply of atomic registers and fetch-and-increment objects, where the fetch-and-increment objects are all initialized to 0 and supply *only* a fetch-and-increment operation that increments the object and returns the old value. Show how to use these objects to construct a wait-free, linearizable implementation of an augmented fetch-and-increment that also supports a `write` operation that sets the value of the fetch-and-increment and returns nothing.

Solution

We'll use a snapshot object a to control access to an infinite array f of fetch-and-increments, where each time somebody writes to the implemented object, we switch to a new fetch-and-increment. Each cell in a holds (timestamp, base), where base is the starting value of the simulated fetch-and-increment. We'll also use an extra fetch-and-increment T to hand out timestamps.

Code is in Algorithm G.1.

Since this is all straight-line code, it's trivially wait-free.

Proof of linearizability is by grouping all operations by timestamp, using $s[i]$.timestamp for `FetchAndIncrement` operations and t for `write` operations, then putting `write` before `FetchAndIncrement`, then ordering `FetchAndIncrement` by return value. Each group will consist of a `write`(v)

```
1 procedure FetchAndIncrement()
2     s ← snapshot(a)
3     i ← arg max_i(s[i].timestamp)
4     return f[s[i].timestamp] + s[i].base

5 procedure write(v)
6     t ← FetchAndIncrement(T)
7     a[myId] ← (t, v)
```

Algorithm G.1: Resettable fetch-and-increment

for some v followed by zero or more `FetchAndIncrement` operations, which will return increasing values starting at v since they are just returning values from the underlying `FetchAndIncrement` object; the implementation thus meets the specification.

To show consistency with the actual execution order, observe that timestamps only increase over time and that the use of snapshot means that any process that observes or writes a timestamp t does so at a time later than any process that observes or writes any $t' < t$; this shows the group order is consistent. Within each group, the `write` writes $a[\mathsf{myId}]$ before any `FetchAndIncrement` reads it, so again we have consistency between the `write` and any `FetchAndIncrement` operations. The `FetchAndIncrement` operations are linearized in the order in which they access the underlying $f[\dots]$ object, so we win here too.

G.3.3 A box object

Suppose you want to implement an object representing a $w \times h$ box whose width (w) and height (h) can be increased if needed. Initially, the box is 1×1, and the coordinates can be increased by 1 each using `IncWidth` and `IncHeight` operations. There is also a `GetArea` operation that returns the area $w \cdot h$ of the box.

Give an obstruction-free deterministic implementation of this object from atomic registers that optimizes the worst-case individual step complexity of `GetArea`, and show that your implementation is optimal by this measure up to constant factors.

Solution

Let b be the box object. Represent b by a snapshot object a, where $a[i]$ holds a pair $(\Delta w_i, \Delta h_i)$ representing the number of times process i has executed `IncWidth` and `IncHeight`; these operations simply increment the appropriate value and update the snapshot object. Let `GetArea` take a snapshot and return $(\sum_i \Delta w_i)(\sum_i \Delta h_i)$; the cost of the snapshot is $O(n)$.

To see that this is optimal, observe that we can use `IncWidth` and `GetArea` to represent `inc` and `read` for a standard counter. The Jayanti-Tan-Toueg bound applies to counters, giving a worst-case cost of $\Omega(n)$ for `GetArea`.

G.4 CS465/CS565 Final Exam, December 12th, 2011

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.

General clarifications added during exam Assume all processes have unique IDs and know n. Assume that the network is complete in the message-passing model.

G.4.1 Lockable registers (20 points)

Most memory-management units provide the ability to control access to specific memory pages, allowing a page to be marked (for example) read-only. Suppose that we model this by a **lockable register** that has the usual register operations `read`(r) and `write`(r, v) plus an additional operation `lock`(r). The behavior of the register is just like a normal atomic register until somebody calls `lock`(r); after this, any call to `write`(r) has no effect.

What is the consensus number of this object?

Solution

The consensus number is ∞; a single lockable register solves consensus for any number of processes. Code is in Algorithm G.2.

Termination and validity are trivial. Agreement follows from the fact that whatever value is in r when `lock`(r) is first called will never change, and thus will be read and returned by all processes.

```
1 write(r, input)
2 lock(r)
3 return read(r)
```

Algorithm G.2: Consensus using a lockable register

G.4.2 Byzantine timestamps (20 points)

Suppose you have an asynchronous message passing system with exactly one Byzantine process.

You would like the non-faulty processes to be able to acquire an increasing sequence of timestamps. A process should be able to execute the timestamp protocol as often as it likes, and it should be guaranteed that when a process is non-faulty, it eventually obtains a timestamp that is larger than any timestamp returned in any execution of the protocol by a non-faulty process that finishes before the current process's execution started.

Note that there is no bound on the size of a timestamp, so having the Byzantine process run up the timestamp values is not a problem, as long as it can't cause the timestamps to go down.

For what values of n is it possible to solve this problem?

Solution

It is possible to solve the problem for all n except $n = 3$. For $n = 1$, there are no non-faulty processes, so the specification is satisfied trivially. For $n = 2$, there is only one non-faulty process: it can just keep its own counter and return an increasing sequence of timestamps without talking to the other process at all.

For $n = 3$, it is not possible. Consider an execution in which messages between non-faulty processes p and q are delayed indefinitely. If the Byzantine process r acts to each of p and q as it would if the other had crashed, this execution is indistinguishable to p and q from an execution in which r is correct and the other is faulty. Since there is no communication between p and q, it is easy to construct and execution in which the specification is violated.

For $n \geq 4$, the protocol given in Algorithm G.3 works.

The idea is similar to the Attiya, Bar-Noy, Dolev distributed shared memory algorithm [ABND95]. A process that needs a timestamp polls $n - 1$ other processes for the maximum values they've seen and adds 1 to it; before returning, it sends the new timestamp to all other processes and waits to

```
1  procedure getTimestamp()
2      c_i ← c_i + 1
3      send probe(c_i) to all processes
4      wait to receive response(c_i, v_j) from n − 1 processes
5      v_i ← (max_j v_j) + 1
6      send newTimestamp(c_i, v_i) to all processes
7      wait to receive ack(c_i) from n − 1 processes
8      return v_i

9  upon receiving probe(c_j) from j do
10     send response(c_j, v_i) to j

11 upon receiving newTimestamp(c_j, v_j) from j do
12     v_i ← max(v_i, v_j)
13     send ack(c_j) to j
```

Algorithm G.3: Timestamps with $n \geq 3$ and one Byzantine process

receive $n - 1$ acknowledgments. The Byzantine process may choose not to answer, but this is not enough to block completion of the protocol.

To show the timestamps are increasing, observe that after the completion of any call by i to `getTimestamp`, at least $n - 2$ non-faulty processes j have a value $v_j \geq v_i$. Any call to `getTimestamp` that starts later sees at least $n - 3 > 0$ of these values, and so computes a max that is at least as big as v_i and then adds 1 to it, giving a larger value.

G.4.3 Failure detectors and k-set agreement (20 points)

Recall that in the k-set agreement problem we want each of n processes to choose a decision value, with the property that the set of decision values has at most k distinct elements. It is known that k-set agreement cannot be solved deterministically in an asynchronous message-passing or shared-memory system with k or more crash failures.

Suppose that you are working in an asynchronous message-passing system with an eventually strong ($\Diamond S$) failure detector. Is it possible to solve k-set agreement deterministically with f crash failures, when $k \leq f < n/2$?

Solution

Yes. With $f < n/2$ and $\Diamond S$, we can solve consensus using Chandra-Toueg [CT96]. Since this gives a unique decision value, it solves k-set agreement for any $k \geq 1$.

G.4.4 A set data structure (20 points)

Consider a data structure that represents a set S, with an operation $\texttt{add}(S, x)$ that adds x to S by setting $S \leftarrow S \cup \{x\}$), and an operation $\texttt{size}(S)$ that returns the number of distinct[4] elements $|S|$ of S. There are no restrictions on the types or sizes of elements that can be added to the set.

Show that any deterministic wait-free implementation of this object from atomic registers has individual step complexity $\Omega(n)$ for some operation in the worst case.

Solution

Algorithm G.4 implements a counter from a set object, where the counter read consists of a single call to $\texttt{size}(S)$. The idea is that each increment is implemented by inserting a new element into S, so $|S|$ is always equal to the number of increments.

```
1 procedure inc(S)
2     nonce ← nonce + 1
3     add(S, ⟨myId, nonce⟩).

4 procedure read(S)
5     return size(S)
```

Algorithm G.4: Counter from set object

Since the Jayanti-Tan-Toueg lower bound [JTT00] gives a lower bound of $\Omega(n)$ on the worst-case cost of a counter read, there exists an execution in which $\texttt{size}(S)$ takes $\Omega(n)$ steps.

(We could also apply JTT directly by showing that the set object is perturbable; this follows because adding an element not added by anybody else is always visible to the reader.)

[4] Clarification added during exam.