CS465 Distributed Systems Problem Set 1

Sherwin Yu

September 28, 2011

1 Identical processes arranged in a torus

Consider two tori adhering to the problem description, of sizes $n_1 \times m_1$ and $n_2 \times m_2$, with $n_1 > m_1$ and $n_2 < m_2$. We will show that such an algorithm for computing whether $n > m$ cannot exist because it cannot distinguish between these two situations. (We are only considering the more general, nontrivial case where n and $m > 3$ because otherwise any process won't have 4 neighbors).

Consider any process i from the first torus and j from the second torus, and assume by way of contradiction that such an algorithm exists. Observe that these two processes are at the same initial state – namely, they all see their four neighbors and all have not received any message yet. Because all processes are identical, they must perform the same initial action. Because all the neighbors of i and j are also the same processes as i and j, they do the exact same action as well.

Now, because all processes and all their potential inputs (their neighbor) did the same thing, both i and j see the same thing. So they must perform identical next actions as well. Inductively, we see that process i from the first torus and process j from the second must always stay in the same state because at any step in which they are indistinguishable, they remain indistinguishable at the next step, and because they begin indistinguishable.

Thus, process i (and all processes from the first torus) must decide the same thing as process j (and all processes from the second torus). However, the first and second toruses cannot have the same answer as $n_1 > m_1$ and $n_2 < m_2$. Thus, we have a contradiction.

2 Finding cluster heads in an asynchronous network

I propose an algorithm similar to broadcast, but with improved message complexity. All heads begin by broadcasting their id and distances to neighbors. The only difference is that a node will wait from hearing from all its neighbors before rebroadcasting the closest head, whereas in broadcast algorithm, if further heads reached a node before a closer head did (due to asynchronicity), the node could broadcast multiple times. The problem here is that what do nodes that are no where near head do? They will never hear from their neighbors, and more importantly, even nodes one away from a cluster will never hear from a neighbor that is not touching a cluster head, so this algorithm can't make progress! Thus, we have to have nodes who haven't found a head yet to send blank messages that indicate their headlessness – this way, all nodes will eventually receive messages from its neighbors. The algorithm is as follows:

1. Heads have their id_{head} set to them selves and their *distance* set to 0. All other nodes have id_{head} initialized to -1 and *distance* to ∞
2. Head broadcast to their neighbors $(id, 0)$ id and distance tuples.
3. All non-heads that have their id_{head} set to -1 (i.e., haven't found a head yet) will broadcast to their neighbors (id, ∞) indicating they are headless. A node tracks how many blank messages it receives from a neighbor.
4. Upon receiving messages from all neighbors, a node sets its *distance* to $min(D) + 1$ where D is the set of all distances it received for this round (ties can be broken arbitrarily). It sets its id_{head} to the corresponding *id*. It then broadcasts $(id_head, distance)$ to its neighbors, excluding the neighbor from whom the node received the closest distance. If the node did not find a head this round, it broadcasts the empty message, (id, ∞) again.
5. This continues until all nodes have their id_{head} variable set.

We first show that the algorithm is correct. Observe that a node only decides its head once, and only when it has heard from all of its neighbors. Because we have all nodes wait for all neighbors before broadcasting, the asynchronous message passing system essentially becomes synchronize to individual steps. Everytime a node sends a blank message, it is essenitally telling its neighbors that another round has passed by without it finding a head. At the start of i, if a node is headless, then the we know that the node has no nodes within $i - 1$ (the shortest path to a head is $> i - 1$). Thus, if the node receives a message with distance $i - 1$ (implying that the node itself is distance i), the node knows this must be a shortest path to a head. Finally, the node waits until all messages from its neighbors before acting to avoid the case in the asynchronous system of settling on a message being passed from a head along a fast but longer path.

Assuming the graph is connected, this algorithm will terminate. Namely, the number of "steps" required for termination is d, the diameter of the graph because this is as far as a node can get from a head, and after the dth step, all nodes within distance d from a head will have a head.

The time complexity is therefore also $O(D)$. The length of a time unit is the longest time a message can take for each round, but the entire network is still synchroniozed to the nearest round, so a time unit is just the slowest message in a given round. Note that the entire network cannot get more than one message out of sync because all nodes require all neighbors to respond before broadcasting. Thus, the time complexity is $O(D)$ (which is also optimal, because if an algorithm worked faster than $O(D)$, it's possible a node hasn't heard from a head at all).

The message complexity is $O(ED)$, where E is the number of edges. There are D rounds, and each round, upto two messages are sent across each edge – either two blank messages, one blank and one distance, two distance updates, or fewer. Note that in the areas of the graph that already have decided their heads, no messages are sent, so $O(E)$ is an upper bound on the number of messages per round. $O(E)$ messages per round and $O(D)$ rounds yields a message complexity of $O(ED)$.

3 Merchant agreement

We define *agreement* for this problem as the two processes deciding on values such that $|P_a - P_b| <= 1$ at the end of execution. During an execution, the two processes are said to be in agreement if, at that point the execution terminated, the two processes would be in agreement. We make the following observations:

1. We note that the two processes must start in agreement because it is always possible that all messages are lost, and in this case, the two processes must terminate in agreement.

2. Next, we argue that one process cannot tell whether the other process terminated. This is because once a process has terminated, it must either send a message indicating its termination (which can be dropped), or it can stop sending messages (which are indistinguishable from its continued execution, but all of its subsequent messages are dropped). Specifically, when a process sees that it did not receive a message, it cannot assume anything – it can either be from termination or from a dropped message.

3. Next, we argue that if the algorithm is to guarantee success, the two processes must never deviate from agreement. This is because if they ever do, it is possible for all future messages to be dropped and it will be impossible for them to get back to agreement, and the algorithm will fail. In particular, because of the indistinguishability between termination and a lost message, the the algorithm cannot decide on a predetermined value when it detects a message loss (i.e. thinking "oh, a message was dropped so we can just both choose 0 and we're good") because it's possible that A and B have the same input and therefore the two processes must settle on a new value equivalent to their old value ($P_a = P_b$) – there's no way to know whether that lack of a message was termination or it was a dropped message!

4. Next, note if the two processes are currently settling on a value x at step i, their next value at step $i + 1$ (if any) must either be $x + 1$ or $x - 1$. Otherwise, one of the final messages could be dropped and the two processes would no longer be within 1 of another, therefore they would be

out of agreement. Thus, to stay in agreement, in the face of any message being dropped, the processes can change their values by at most 1 per round of message exchange.

5. Finally, note that in the case that messages go through, the two processes must end at the original $P_a = P_b$.

We now argue that any algorithm that satisfies these must take at least O(M) steps. The processes must start at predetermined value (by Observation 1). There is a case where the processes must move to $P_a = P_b$ (observation 5). And they can move by at most one per round of message exchange (Observation 4). Because P_a can be anywhere from 0 to M, it is $O(M)$. Moving from any valid fixed initial point to P_a at single increment steps, then, must take at least $O(M)$ steps, so our bound is $O(M)$ rounds. Note that if a message is dropped, this must occur before $O(M)$ steps because such an algorithm is done in $O(M)$ steps if no messages are dropped.

An example of such an algorithm is provided:

The processes agree on the initial value of M. Every round, they send their current values and decrement by 1. When a process reaches its original value (e.g., Process A reaches P_a), that process terminates. If a process does not receive a message in a round, it terminates as well.

Note that this algorithm will terminate in $O(M)$ in the worst case because P_a lies between 0 and M. Note that he processes are always in agreement and always stay in agreement. When a message is dropped, the intended receiver terminates and does not send further messages, triggerin the termination of the sender of the original dropped message. Thus, the processes will always decide values at most 1 apart, and hence will always be in agreement. When no messages are dropped and the initial values are different, the first process to reach its value terminates first and then the second process terminates in the next round, again still in agreement. When no processes are dropped, and the initial values are identical, the two processes terminate simultaneously at the initial value.