CSC2221: Introduction to Distributed Algorithms **Fall 2017**

Lecture 1: Introductions

Lecturer: Faith Ellen *Scribe: Lily Li*

1.1 Introductions

Many different models of Distributed computing: synchronous (rounds of operation, send one round and pass in the next) vs asynchronous (very little assumption of timing) — finally semi-synchronous with some bounds on time.

Note: fault tolerance. Many different ways in which this can take place: corrupted registered, byzantine fault model etc. Faults can be transient or permanent.

Distributed computing is *different* from parallel computing — there is a little bit of overlap. The difference is with the type of problems being solved. In the latter, problem does not depend on system, everyone shares one processor work towards a common goal, inputs available to everyone. In the former, everyone is doing their own thing, there maybe competition for resources.

Sources of uncertainty include: limited knowledge of the system (not a complete view of the global system), asynchrony, faults (as noted above).

Example 1.1 *Let p_0, p_1 be two processes. There is a shared variable n set initially to zero. Further, each p_i has a local variable x_i. We assume that no faults occur. Let $k > 1$ be an integer. Each process will preform the following operation:*

for $1, ..., k$ ***do***
  $x_i \leftarrow read(n)$
  $n \leftarrow write(n)$
end for

Can you see how n can be any value between 2 and $2k$ if read and write are not bundled in an atomic operation? Lets define the R_i to be the read of process i and W_i to be the write of process i. Observe that the sequence

$$R_0 \quad (R_1W_1)^{k-1} \quad W_0 \quad R_1 \quad (R_0W_0)^{k-1} \quad W_1$$

Results in a value of 2 in the shared register n.

Further can you see why no other values are possible? Hint: here are some observations. Each time a write occurs the resulting value of n read by p_i is exactly *one larger than the value previously read by p_i. Each read (except for possibly the first read by each process) returns a positive integer. Since $k > 1$, the last write into n must be last least 2. If i writes occur the value of n is at most i. Formality is* **crucial**. *If necessary induction, or contradiction, is the way to go.*

1.2 Message Passing Models

There are n processes $p_0, ..., p_{n-1}$ with communication channels — two way link communication between two processes. Undirected graphs are the standard model where the nodes are the processes and the edges

are the channels. Processes are modeled as state machine. The state of p_i is the triple: the locate state, in-buffer $\mathsf{inbuf}_i[e]$ for each incident edge (messages delivered but not yet processed), and contents of an out buffer $\mathsf{outbuf}_i[e]$ (prepared but not yet delivered messages).

A **configuration** at a point in an execution consists of the states of every process at that point.

The execution of an asynchronous message passing model is a finite or infinite sequence of alternating configurations and events.

$$C_0\phi_1C_1\phi_2\cdots$$

Where C_0 is the initial configuration and ϕ_i are the events. ϕ_k is either

$\mathsf{del}(i,j,m)$: delivery of message m from $\mathsf{outbuf}_i[e]$ to $\mathsf{inbuf}_j[e]$ where $e=\{p_i,p_j\}$.

$\mathsf{comp}(i)$: (1) Read all (some) messages in $\mathsf{inbuf}_i[e]$ for all (some) incident edge e; remove read messages. (2) Local state of p_i changes according to the state transition function. (3) Writes ≥ 1 messages to each of its out buffers. Together these consists of one computational step of processor p_i.

If σ is a sequence of events and let C be a configuration, then $C\sigma$ is the configuration obtained from C by applying events in σ in order. Inputs + outputs are encoded in states of the process.

First we consider non-faulty systems: every sent message will eventually be delivered. Processes will take steps until explicitly terminated (going to a special *end* state) or continue for an infinite number of execution events. This defines an admissible execution.

An algorithm for an asynchronous model is typically thought of as being scheduled by an adversary which controls the input and the order in which events occurs. Conversely for a synchronous model there are alternating sequence of configurations and rounds (sequence of events). A round consists of delivering a message that is each out buffer followed by a single step by every process which has not yet terminated (if message order is important, then adversary controls that). *Round count is important, the definition is slightly different from that in the book.*

1.2.1 Complexity Measure

These include the number of round in a synchronous execution (round complexity or possibly step complexity — maximum steps by any process on any message; does not make sense if you have infinite processes), number of messages passed (message complexity), the complexity of the messages sent (bit complexity).

Example 1.2 *Recall that a spanning tree is a tree whose vertex set is the set of all processes and whose edge set are communication links in the network.*

First lets discuss some uses of such a spanning tree: let the spanning tree be rooted at p_r and contain n processes. Each process knows which of its neighbors are its parent and children. Further p_i knows its own process ID and that of all of its neighbors. Spanning tree algorithms include:

1. ***Broadcast by** p_r**:** when a node receives a message, pass it along to its children. The message complexity is $n-1$. The round complexity is the height of the tree. Both the synchronous and asynchronous models work here. In the synchronous model we need $t+1$ rounds (first round nothing happens).*
2. ***Collecting information:** collect information at the root, further each node forward messages to its parent after receiving all messages from its children. Both the message and round complexity are pretty much the same as before.*

To construct such a spanning tree we start at p_r and perform a broadcast-like operation (called flooding) with the caveat that we only accept the first message received and we need to send back a confirmation message. Do we need a message back if we are in the asynchronous model?

In the synchronous model, it is simply a breadth first search. In the an asynchronous system the tree could be a line (why is this a problem? It isn't really a problem, it just shows that the resulting tre is not a BFS). What is the message complexity? $O(m)$. What about the time (round) complexity? $O(D)$ where D is the diameter of the tree.

Next we should prove that this construction is correct.

Theorem 1.3 *If (V, G) is connected then all processes eventually terminate and when they do (V, T) is a spanning tree of (V, G) when $T = \{(p_i, p_j) : p_i$ is the parent of $p_j\}$ (and p_j sends p_i back a* "I am your child" *message).*

Proof: *First let us define the termination state for each node: this happens when every message sent and received a message from all of its neighbors. Suppose that the master message is M. We can show that all processes eventually get M by contradiction (please work this out on your own and be* very precise*). Note that $|T| = n - 1$ since every child — that's everyone except p_r — sends back a* "I am your child" *message. Finally we need to show that T is acyclic. I trust that you can do that on your own.* ■

Next we can consider the case where p_r is not specified. The gist of the algorithm is as follows: every node will try to build a DFS spanning tree with itself as the root while passing along its ID. If ever a node gets two requests, it will join the DFS tree whose root has a higher ID.

It turns out that is it *impossible* to find a spanning tree deterministically if the nodes were not labeled. To see this consider the triangle. Since it is impossible to break symmetry, a tree cannot be established.

Next it is useful consider if $\Omega(m)$ is indeed a lower bound for the number of messages (remember m is the number of edges). Actually it is not, but in-order to do so you have to cheat bit complexity a bit and pass along messages with more information.

1.3 Homework

Construct a BFS tree starting from a specified root r in an asynchronous message passing system (using $O(\log n)$ bit messages — can't just tack on everything you know). Using $O(m + nD)$ messages where $D = \max_u \mathsf{dist}(r, u)$. Prove correctness of the algorithm and that it runs in the appropriated time.