Design and Analysis of Algorithms May 1, 2015
Massachusetts Institute of Technology 6.046J/18.410J
Profs. Erik Demaine, Srini Devadas and Nancy Lynch Recitation 10

Distributed Algorithms

1 Review

We are going to review content from lecture that might be confusing.

Some key ideas from lecture include:

1. Synchronous vs. asynchronous network models (based on undirected graphs). In the synchronous model all nodes move in lockstep. They all send messages, then receive messages and then do computation. Then, after computation they send messages again and re-start the cycle. In the asynchronous model any finite amount of time can pass in between rounds. This finite amount of time can be different for different nodes. This allows for many possible interleavings. (Note: analysis of the time for asynchronous models tends to add some assumption of nodes finishing a round by time t for some t. However, this *is not used for correctness*. For correctness any amount of time can pass, for time analysis we need some kind of bound.)

2. There are many possible ways to judge how costly a protocol is. We can count number of messages, number of bits, number of rounds or do asynchronous time analysis.

3. There were many proof methods discussed in class. E.G., invariants, probabilistic methods for breaking symmetry, etc.

4. Synchronous Leader Election: A canonical symmetry-breaking problem. In symmetric graphs like the clique, this is impossible if the processes are identical and deterministic, but has solutions if the processes have UIDs or can use randomness.

5. Synchronous Maximal Independent Set: Another interesting symmetry-breaking problem. The goal is to have the set of nodes that are on by the end of execution be a maximal independent set. Luby's algorithm works for this. Every round nodes randomly pick an ID from $[1, n^5]$. Then every node sends to all neighbors the ID. A node that has a higher ID than all of its neighbors selects to be in the set. It tells all of its neighbors that it is in the MIS, those neighbors are then not in the MIS. We repeat rounds after that. This will terminate in $O(\lg(n))$ time.

6. Synchronous Breadth-First Spanning Tree construction.
   Synchronous: v_0 sends out a message to all neighbors. Those neighbors send out messages, etc. Nodes will save, from the first node they get a message from that that node is their parent.
   This is the simple algorithm from class. The message complexity is $O(|E|)$.

7. Termination using convergecast. Go over this in detail.

8. Synchronous Shortest-Paths Tree construction. Algorithm (uses relaxation). Correctness. Analysis. Child pointers. Simple applications.

9. Termination using convergecast, with corrections.

10. Asynchronous distributed algorithms. Go over the model assumptions. Process automata, channel automata, compose them. Max computation example.

11. Asynchronous BFS tree construction. What goes wrong if we just run the synchronous algorithm asynchronously. A correct algorithm (uses relaxation). Correctness, analysis. Termination using convergecast.

12. Asynchronous SP tree construction. Algorithm (with lots of relaxation). Analysis. Surprising back worst-case example.

2 New Questions

2.1 Leader election in a ring with UIDs

Assume that the processes have UIDs and a consistent sense of direction (ports labeled "left" (clockwise) or "right"(counterclockwise)).

Consider the synchronous setting first.

Simple algorithm: Everyone sends their UID clockwise. When they receive their own back, they know they have received all the UIDs. Then the largest UID process can declare itself the leader.

Time: n Messages: n^2

Q: What if we want to save messages?

Idea: Throw away any arriving message that is less than your own, or less than the maximum you ever saw. But these don't help the order of magnitude in the worst case. Construct an example.

Q: Can we beat $O(n^2)$?

Hirshberg-Sinclair is one solution, involving searching to successively doubled distances. Specifically, each process does the following.

- Send out a message with its UID and a message hop count h and direction d, (UID, h, d).

- If you receive another nodes message (UID, h, d) forward $(UID, h-1, d)$ in the direction d only if the hop count is greater than 0 and its UID is greater than or equal to any other UID you have seen before. Whenever the message is forwarded, its hop count is decremented by one. If it is greater than any other UID you have seen and the hop count is 0, forward it in the opposite direction of d.

- If a node doesn't receive a message it sent out back, then it sends no new messages.

- If a node receives its $(UID, h, left)$ message on its right and its $(UID, h, right)$ message on its left then these messages made it all the way around the circle and thus the UID is the largest in the circle. This node becomes the leader.

How many messages does this send in the worst case? By the end of the round where the hop lengths are 2^r there can be at most $n/2^r$ nodes that still send messages. Each node sending messages in round r will have hop counts of 2^r resulting in at most 2^{r+1} messages sent. Then the total number of messages can take

$$\sum_{i=0}^{\lg(n)} 2^{i+2} n/2^i = 4n \lg(n).$$

This is a big improvement! $O(n \lg(n))$ instead of $O(n^2)$.
Q: What about an asynchronous ring?
The above still works.

2.2 An asynchronous algorithm for counting the nodes

Assume that the graph has at least two nodes. Assume a root process i_0 at graph vertex v_0.

Q: Give a simple algorithm that allows process i_0 to compute the total number of nodes in the network.

Set up a spanning tree, e.g., using the simple synchronous spanning tree run asynchronously. In the asynchronous setting, this produces some spanning tree, which is not guaranteed to produce the BFS spanning tree. But it is OK for our purpose here (counting the nodes.) (This also won't have the optimal time complexity, because of the timing anomaly). Finally, add a convergecast that also accumulates the sum on the way up.

Work with the class to develop the code. Do this in three stages: 1. The asynchronous (non-BF) spanning tree code from the lecture slides. 2. Add actions for child pointers. 3. Then add actions for convergecasting the number of nodes.

Analyze time cost: $O(|E|)$.

All the processes will use the strategy of putting messages into FIFO send queues for their neighbors, and then have separate output actions that actually send the messages, starting from the heads of the queues. Thus, all processes will have the following transition definition. We will omit them later.

output $send(m)_{u,v}$, m a message, $v \in \Gamma(u)$
Precondition: $m = head(send(v))$
Effect: remove head of $send(v)$

2.2.1 Part 1: Setting up a tree

First search message becomes your parent. Pass search messages on.

Process v_0

State variables:
for each $v \in \Gamma(v_0)$, $send(v)$, a queue, initially $(search)$

Transitions:

input $receive(search)_{v,v_0}$, $v \in \Gamma(v_0)$
Effect: none

Process u**,** $u \neq v_0$

State variables:
$parent \in \Gamma(u) \cup \{\perp\}$, initially $\perp$
for each $v \in \Gamma(u)$, $send(v)$, a queue, initially empty

Transitions:

input $receive(search)_{v,u}$, $v \in \Gamma(u)$
Effect:
if $parent = \perp$ then
    $parent := v$
    for each $v \in \Gamma(u)$, add $search$ to $send(v)$

2.2.2 Part 2: Adding child pointers

Now send parent(true) or parent(false) responses. Keep track of who has responded, and which of these have responded parent(true)—the latter are the children.

Process v_0

State variables:
$responded \subseteq \Gamma(v_0)$, initially $\emptyset$
$children \subseteq \Gamma(v_0)$, initially $\emptyset$
for each $v \in \Gamma(v_0)$, $send(v)$, a queue, initially $(search)$

Transitions:

input $receive(search)_{v,v_0}$, $v \in \Gamma(v_0)$
Effect: add $parent(false)$ to $send(v)$

input $receive(parent(b))_{v,v_0}$, b a Boolean, $v \in \Gamma(v_0)$
Effect:
if b then
    $children := children \cup \{v\}$
$responded := responded \cup \{v\}$

Process u**,** $u = v_0$

State variables:
$parent \in \Gamma(u) \cup \{\perp\}$, initially $\perp$
$responded \subseteq \Gamma(u)$, initially $\emptyset$
$children \subseteq \Gamma(u)$, initially $\emptyset$
for each $v \in \Gamma(u)$, $send(v)$, a queue, initially empty

Transitions:

input $receive(search)_{v,u}$, $v \in \Gamma(u)$
Effect:
if $parent = \perp$ then
    $parent := v$
    add $parent(true)$ to $send(v)$
else add $parent(false)$ to $send(v)$

input $receive(parent(b))_{v,u}$, b a Boolean, $v \in \Gamma(u)$
Effect:
if b then
    $children := children \cup \{v\}$
$responded := responded \cup \{v\}$

2.2.3 Part 3: Adding the convergecast for the count

Now convergecast the count up the tree.

Process v_0

State variables:
$responded \subseteq \Gamma(v_0)$, initially $\emptyset$
$children \subseteq \Gamma(v_0)$, initially $\emptyset$
$done \subseteq \Gamma(v_0)$, initially $\emptyset$
$total$, a nonnegative integer, initially 0
for each $v \in \Gamma(v_0)$, $send(v)$, a queue, initially $(search)$

Transitions:

input $receive(search)_{v,v_0}$, $v \in \Gamma(v_0)$
Effect: add $parent(false)$ to $send(v)$

input $receive(parent(b))_{v,v_0}$, b a Boolean, $v \in \Gamma(v_0)$
if b then
    $children := children \cup \{v\}$
$responded := responded \cup \{v\}$

input $receive(done(k))_{v,v_0}$, k a nonnegative integer, $v \in \Gamma(v_0)$
Effect:
$done := done \cup \{v\}$
$total := total + k$
if $done = \Gamma(v_0)$ then (the final output is the value in $total$).

Process u**,** $u \neq v_0$

State variables:
$parent \in \Gamma(u) \cup \{\perp\}$, initially $\perp$
$responded \subseteq \Gamma(u)$, initially $\emptyset$
$children \subseteq \Gamma(u)$, initially $\emptyset$
$done \subseteq \Gamma(u)$, initially $\emptyset$
$total$, a nonnegative integer, initially 0
for each $v \in \Gamma(u)$, $send(v)$, a queue, initially empty

Transitions:

input $receive(search)_{v,u}$, $v \in \Gamma(u)$
Effect:
if $parent = \perp$ then
    $parent := v$
    add $parent(true)$ to $send(v)$
else add $parent(false)$ to $send(v)$

input $receive(parent(b))_{v,u}$, b a Boolean, $v \in \Gamma(u)$
Effect:
if b then
    $children := children \cup \{v\}$
$responded := responded \cup \{v\}$

if $responded = \Gamma(u)$ and $children = \emptyset$ then

add $done(1)$ to $send(parent)$

input $receive(done(k))_{v,u}$, k a nonnegative integer, $v \in \Gamma(u)$
Effect:
$done := done \cup \{v\}$
$total := total + k$
if $responded = \Gamma(u)$ and $done = children$ then
add $done(total + 1)$ to $send(parent)$

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015