CSc72010

General Asynchronous Networks

Reading

Lynch, Sections 15.2-15.4

Introduction

Basic tasks:

- Broadcast
- Convergecast
- Loop-free communication

In this section of the course, we look at important proof techniques on simple algorithms. Assume a general directed graph for the network. Processes know who their neighbors are and can tell whether an incoming and outgoing edge go to the same neighbor. (This could also be done with an initial round).

Leader Election

Assume a *strongly connected* digraph and UID's with comparisons only. Both leaders and non-leaders will output status. Processes know an upper bound on the diameter of the network graph (d).

Requirements for leader election algorithm:

- Elect the process with the max UID by flooding the max UID throughout the network.
- Each process maintains the max UID seen so far and sends over all outgoing edges on each round.
- After d rounds, compare max-uid to own uid. If equal, declare self leader; otherwise, declare self non-leader.

Correctness Conditions

Very strong assumption: Assume that the network runs synchronously, ie, a node sends a message and then every other node for which a send was enabled gets to send. Subsequently, every channel that has a message in it delivers the message.

We'll call each such sequence of events a "round".

Suppose that
u is a process's UID;
max-uid is the maximum known uid;
status $\in$ {unknown, leader, non-leader};

Assertion: After d rounds,

$status_{i_{max}}$ = leader

$status_j$ = non-leader for $j \neq i_{max}$

Invariant assertion: For 0 <= r <= d and for every j, if distance(i_{mzx}, j) <= r then max-uid_j = u_{max}.

That is, u_{max} propagates to all nodes within distance r by the end of r rounds.

Complexity

Time: d

Messages: O(d |E|)

Optimization

Don't send the same value more than once. Use a flag to say whether the current value is new.

Correctness is proved by a *simulation relationship* between the algorithms. The idea of simulation relationships:

- Run the algorithms side-by-side (call the original the "specification," the optimized algorithm the "implementation").
- The specification is known (or at least assumed) to be correct; the implementation is not known to be correct, but must be proved correct.
- Prove an invariant relating states of the processes (this is called the *simulation relation*).
- Use the simulation relation to prove correctness of the implementation.

For this example, the invariant says that the state variables u, max-uid, status, rounds are the same in both algorithms after r rounds.

Prove by induction on the length of the execution.

Induction base: Algorithms have the same initialization.

Induction hypothesis: For r < r' rounds, the states are the same.

Induction step: After round r', we must show that max-uid_j is the same in both states, for all j.

By hypothesis, max-uid_j is the same before r', and so is max-uid_i neighbor i of j.

Broadcast (BFS)

This translates familiar sequential BFS to a distributed version.

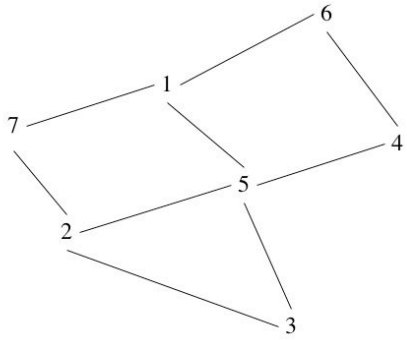
Algorithm idea

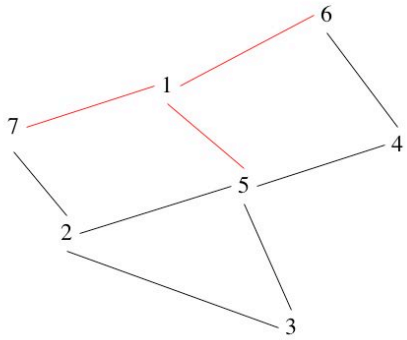
Initially, a start node is marked.
All newly marked nodes send search messages to all outgoing neighbors.
If an unmarked node receives a search message, it marks itself and sends a search message to outgoing neighbors.
The node from which the first message arrives is the parent of the receiving node..

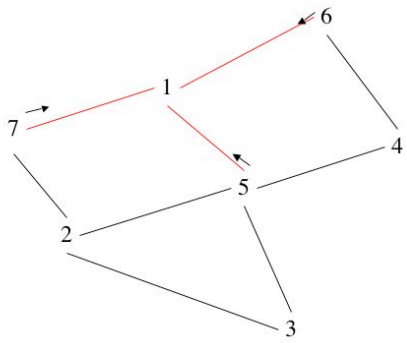
```
vocabulary Messages
      types
            Types enumeration [search, notparent, parent],
            Message = tuple [type:Types, distance:Int]
end

imports Messages
automaton BFS (p:Nat, size:Nat, neighbors:Set[Nat])
signature
input receive(m:Message, n:Nat, const p:Nat)
output send(m:Message, const p:Nat, n:Nat)

states
newMark:Boolean := true for start node, false elsewhere
parent:Int := -1,
newm:Message := [p, 0] for root [p, -1] otherwise
distance: Int := 0 (for root) –1 otherwise
reject:Array[Int, Bool] := constant(false)

transitions
input receive(m, n, i)
      if (parent ~= -1) then
            for i \in neighbors do
                  newMark[i] := true
            od;
            newm := [m.type, m.distance+1];
            parent := n
      else
            reject[n] := true
      fi;
      if (m.distance = size-1) then
            done := true
      fi

output send(m, n, i)
pre
      ~done /\ ((newMark[i] /\ m=newm)
                  \/ reject[n]
eff
      newMark[i] := false
```

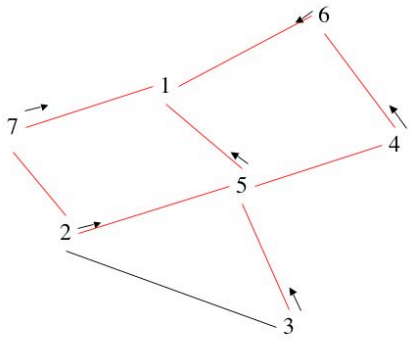
Safety property: No process has two parents.
Safety property: Once a parent has been set, it stays the same.
Liveness property: Every process eventually has a parent.

BFS Example, assuming more or less synchronous sending and receiving of messages:


6
1
7
4
5
2
3


6
1
7
4
5
2
3


6
1
7
4
5
2
3



Note that each process at distance d appears at depth d in the BFS tree.

But, if processes are running at very different rates, the tree could end up a straight line. This clearly takes more steps to build.

We can force processes to operate synchronously with trajectories:

trajectories
trajdef round
    stop when now = stoptime
    evolve d(now) = 1

Then make preconditions include now<stoptime

Invariant assertion: After d rounds, every node within d of that start node has a parent node.

Complexity:
Time: diam
Messages: |E|

Applications

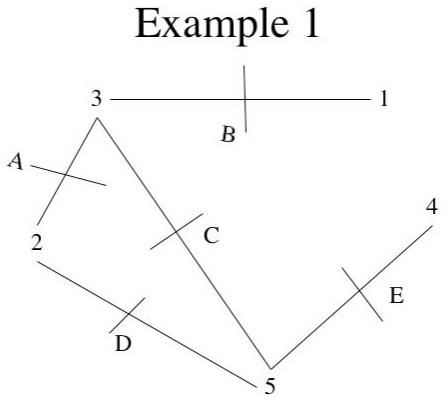
Broadcast: Piggyback the actual message on the search messages
Child pointers: When receiving a search message, respond with either a parent or non-parent message.
This can be used for return messages.
If communication is not always bi-directional, may need to run another instance of SynchBFS to reply.
Broadcast/convergecast: When leaves receive a message, send to parents – eventually, start node gets all replies.

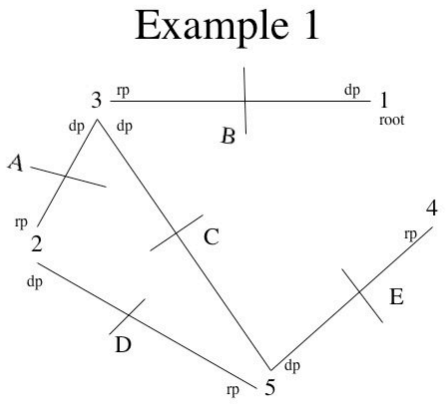
Leader election: Use broadcast/convergecast to determine max (or min) UID in network.
Computing the diameter: All processes do BFS, determine max-$dist_i$ from i to any other process by piggybacking distance information. Then broadcast/convergecast to get the largest distance in the graph.

Spanning Tree

Note that BFS computes a spanning tree (the parent pointers identify the edges). How many edges are there in a spanning tree?

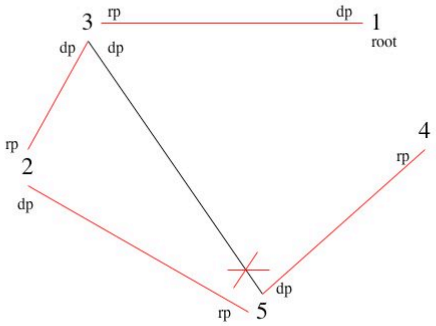
The Cisco STP uses this.





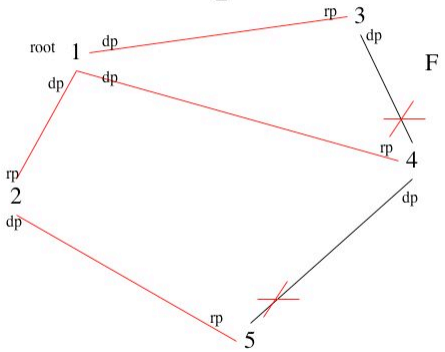
Assume ports are numbered clockwise starting at 12

Example 1



Assume ports are numbered clockwise starting at 12

Example 2



Assume ports are numbered clockwise starting at 12

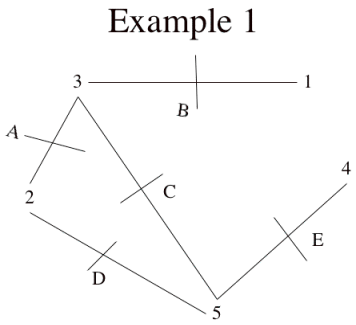
What does the Cisco STP compute

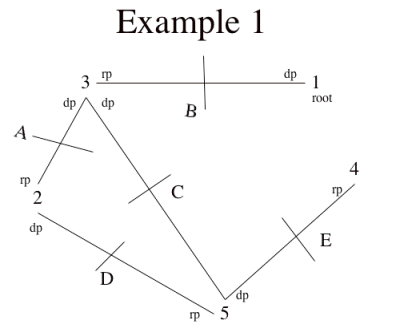
When the protocol stabilizes, the state should be as follows:

1) **Root bridge**: The process (switch) with the lowest MAC address (or lowest combined priority+MAC address) is the root. This uses a leader election algorithm.
2) **Root ports**: Each bridge has one root port. The root port on each bridge is the port of the bridge with the smallest distance from the root. If two ports are equidistant from the root, then the one going to the bridge with the lower MAC address is the root port. This uses a breadth-first search, if we assume rounds; however, if the network is asyncrhonous, it's more complicated.
3) **Designated ports**: Each network segment (connecting bridges) has a designated port. Messages put on that network segment are forwarded to the rest of the network through the designated port. The designated port is on the bridge closest to the root. If there is a tie, it is on the bridge with the lowest MAC address. If the

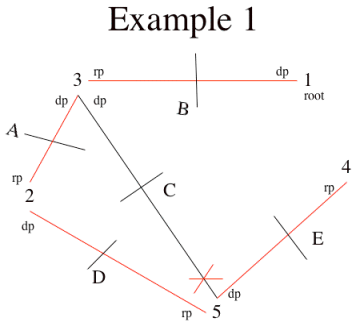
bridge selected by this rule has multiple ports on a network, it is the port with the lowest id.

Examples

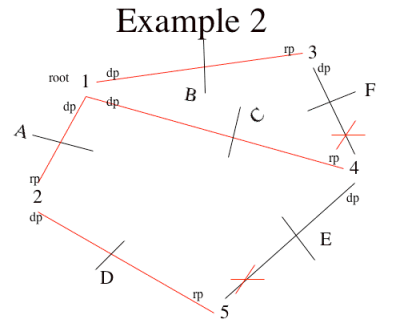




Assume ports are numbered clockwise starting at 12



Assume ports are numbered clockwise starting at 12



Assume ports are numbered clockwise starting at 12

Cisco Version

All processes send a BPDU (Basic Protocol Data Unit) at each round (actually, default is every 2 seconds – but we will describe this as a synchronous algorithm, running in rounds). The BPDU contains the id (MAC address) of the sending process, the id of the process it thinks is the root, and the distance from the sending process to its presumed root.

id:Int
id:Int
cost:Int

When a process receives a BPDU, it compares the ID of the root (designated by the neighbor that sent the BPDU) to the local value for the ID of the root. If the new root ID is lower, it replaces its local root ID with the new one and adds one to the cost in the incoming BPDU and makes that its cost to the root. If it receives different BPDU's

having different roots, it uses the “best,” i.e., the one with the lowest root ID and the lowest distance (if root Ids are the same).
Finally it designates the port to which the sending neighbor is connected as the root port. (Effectively choosing its parent)

Note that this is essentially BFS.

Designated ports are then chosen: for each network it is on, the bridge checks incoming BPDUs. If its own cost is lowest, or if it is tied with another bridge on cost and its id is smaller, it is the designated bridge, and its port on that network segment is the designated port. If it has multiple ports on the network segment, it chooses the one with the lowest id.