Csc72010

Parallel and Distributed Computation and Advanced Operating Systems

Lecture 4

February 16, 2006

Spanning Tree Protocol

Broadcast (BFS)

This translates familiar sequential BFS to a distributed version.
See Lynch, 15.4.

Algorithm idea

Initially, a start node is marked.
All newly marked nodes are enabled to 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 (out-nbrs).
The node from which the first message arrives is the parent of a process.

The following *corrected* algorithm avoids most multiple sends over the same edge.
However... we can't avoid them all: what if two processes send to each other at the same time? Any thoughts about what's the best we can do?

```
automaton BFS(p:Int, n:Set[Int])

signature
  input receive(m:Int, i:Int, const p),
        init(i:Int)
  output send(m:Int, const p, j:Int)

states
  newMark:Bool := false,
  parent:Int := -1,
  nbrs:Set[Int]:=n,
  sentMark:Set[Int] := {},
  root:Bool := false

transitions
  input init(i)
  eff
    root := true;
    if i=p then newMark := true fi
```

```
  input receive(m, i, j) where m=1 %/\ i \in n
  eff
    if ~root /\ parent = -1 then   % this is the first message
      parent := i;
      newMark := true
    fi;
    sentMark := insert(i,sentMark)   % no need to send message back

  output send(m, i, j) where m=1 %/\ j \in n
  pre newMark /\ ~(j \in sentMark) /\ j \in nbrs
  eff sentMark := insert(j,sentMark)   % don't send another message
```

Representation of state: = <false, -1, {}>, <true, 3, {1,4}>
That is, <*newMark*, *parent*, *sentMark*>

Channel

```
automaton Channel(i:Int , j:Int)
  signature
    input SEND (m: Int , const i : Int, const j : Int)
    output RECEIVE(m : Int, const i : Int ,const j : Int)

  states
    queue : Seq[Int] := {}

  transitions
    input SEND (m, i, j)
      eff queue := queue |- m

    output RECEIVE(m,i,j)
      pre head(queue) = m
      eff queue := tail(queue)
```

Representaton of state: {}, {4}, {4,7}
That is, *queue*

Composition

```
automaton BFSNetwork
  components

   P1: Process(1, {2,4});
   P2: Process(2, {1,3});
   P3: Process(3, {2,4});
   P4: Process(4, {3.1});
   C12: Channel(1, 2);   C21: Channel(2,1);
   C23: Channel(2, 3);   C32: Channel(3,2);
   C32: Channel(3, 4);   C43: Channel(4,3);
   C41: Channel(4, 1);   C14: Channel(1,4)
```

Applications of BFS

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.

Proof Techniques

See Lynch, 8.5

Proofs apply to properties that apply to all states (invariants) or properties that apply to all traces (trace properties).

Definitions:
Execution: A sequence s0, a1, s1, a2, ... such that si are states and ai are actions and (si-1, ai, si) is a transition of the automaton.
Trace: An execution resticted to its external actions (inputs and outputs), i.e., its visible actions. No states, no internal actions.

Types of properties

Proving properties often proceeds by alternately trying to understand what the algorithm does, formulating a property, trying to prove it and finding it's not true or perhaps not as useful as hoped, and then going back to trying to understand the algorithm. The following traces some of the steps that I followed trying to prove important properties of the BFS algorithm.

Invariants

Properties of states that are true in all reachable states. Usually proof by induction.

Here are important invariants for BFS:
1. If q in p.sentMark, then q.parent ~= -1 or mark in C[p,q].queue
To prove: In start state, all initial values satisfy this. Suppose that in state si the above assertion holds. Then possible actions ai are:
init: doesn't affect either the antecedent or the consequent
send: adds a new node to sentMark, but also puts the mark in the channel
receive: takes the mark out of the channel, but if q.parent=-1 it sets q.parent to the sending node

2. There are no cycles in the set of parent edges defined by BFS.

Crudely speaking, if this is true at some stage of the algorithm, then the only way it can

be made false is by an action that sets a parent variable. This is a receive action. I claim that the receive sets the parent variable only during the first message that a node receives. Thus it has sent no messages itself, and no other node can have made it a parent.

The above is a safety property: a cycle is a bad thing that the algorithm avoids.

We also want to know that every node eventually is marked, and thus has selected a parent.
3. *Possible invariant:* After (2*)d*(n-1), or (2*)d*e, transitions, every non-root node within d of the start node has a parent node.
Clearly true for d=0, since the root is explicitly excluded. Let's say this is true for d=1, ie, after n-1 (resp e) transitions, Can we prove it for d=2, ie, does it hold for nodes at distance 2 from the root after another n-1 (resp e) transitions? How about assuming that it's true for d=i, can we prove it for d=i+1? THIS IS UGLY!!

4. *Better invariant*: After 2n transitions, at least n mark messages have been sent.

Possibilities after two transitions:
2 sent
1 sent, 1 received

Possibiliies after 2n transitions:
2n sent
2n-1 sent, 1 received
n sent, n received (the number received can't exceed the number sent)

4. How many messages does each channel transmit in the entire running of the algorithm?

Each channel transmits at most one message in each direction (remember, we can't prevent two processes from sending to each other at the same time).
Be sure you understand why this is so.
Thus the total number of messages sent (and received) is <= 2*|E|, where E is the set of edges.
An important consequence of this is that every execution is finite as long as the network is finite, ie, executions terminate.

Note that we can prove this by adding a counter to the Channel automaton, as discussed in class. Remember also that channels are one-way and each pair of processes has two channels between them – we ignored this in class, which gave the incorrect result.

[The following is a liveness property, not a state invariant.]
5. When a receive happens, the receiving node becomes marked if it hasn't been marked already.
So, can we guarantee that the system can always continue until a message travels over each edge, ie, there's a send and a receive for each edge?

The following reasoning requires a connected graph. Note that the algorithm will

construct a tree, but not a spanning tree (of the entire graph) if the graph is not connected.

Suppose an execution has proceeded for a while, but some nodes are still unmarked. We need an edge between a marked node and an unmarked node to make progress. Can we guarantee that there is one? Yes, because if there's not, the graph could be divided into marked and unmarked nodes, with no edges connecting them.

So, consider an edge that has a marked node at one end and an unmarked one at the other. The possibilities are that there is a message already in the channel and there is not. If not, the send is enabled on the process (need to discuss state variables here). If so, the receive is enabled on the channel.

The above is a liveness property, i.e., the execution continues until nothing more is enabled (this is the definition of fairness for finite traces – for infinite traces, it's more complicated). It requires assuming fair traces.

Exercise: Turn these observations into an induction proof.
Ask: What is the induction variable? Does the proof work for the initial case (or cases)? Does the induction step work?

False induction proof

All horses are the same color.

Induction base: Put one horse in a corral. All horses in the corral are the same color.
Induction hypothesis: All corrals with < n horses contain only one color of horse.
Induction step: Put n horses in a corral. Take one horse out and put it in a separate corral. Now all horses in each of the two corrals are the same color. But since we could have chosen any horse, by transitivity, all horses are the same color.

What's wrong?

Trace properties

Any property of the external behavior sequences of automata. Formally, a trace property P is an external signature together with a set of sequences of actions in the signature, i.e., the allowable sequences:
<sig(P), traces(P)>

An automaton A satisfies a trace property P if it has the same external signature and traces(A) $\subseteq$ traces(P) or maybe fairtraces(A) $\subseteq$ traces(P).

All of the problems we will consider in asynchronous systems can be formulated as trace properties. Also, we'll usually be concerned with fairness, i.e., we'll be making statements about trace properties that hold for fair traces.

Tasks for BFS:

tasks {{send(m,i,j)}, {receive(m,i,j)}}
We say that a class C in tasks(A), a class of actions of A, is *enabled* if any member of the class is enabled.

Formal definition of fairness:
An execution fragment α is fair if the following conditions hold for each class C in tasks(A):

1) If α is finite, then C is not enabled in the final state of α.
2) If α is infinite, then α contains either infinitely many events from C or infinitely many states in which C is not enabled.

For finite traces: No action is enabled at the end of the trace.
For infinite traces: If a member of {send(m,i,j)} occurs only finitely many times, there must be infinitely many states in which it is NOT enabled. It can still be enabled infinitely often, but after every enabled state there must be one in which it is not enabled.

Note that, in our proof of BFS, we actually assumed finite fair traces.

Now we've looked at two kinds of properties: safety and liveness.

Safety properties

A safety property says that bad things don't happen.
In other words, traces(P) is non-empty, prefix-closed, limit-closed:

1. λ is in P
2. if α is in P, all prefixes of α are in P
3. if all prefixes of α are in P, then α is in P

Examples:
1. At most one process declares itself as leader. (The traces are those including only one leader action.) Formally, define the property by:

$$sig(P) = \{ SEND(m,i,j), RECEIVE(m,i,j), leader(r) \mid m \in M, i,j \in \aleph, r \in \aleph \}$$
$$traces(P) = \{ \tau_1, \tau_2, ... \mid \forall i(\tau_i \in sig(P)) \wedge \neg\exists i, j, r, s\ (\tau_i = leader(r) \wedge \tau_j = leader(s)\ ) \}$$

2. DHCP: No two processes get the same IP address
3. ARP: No two processes respond to the same ARP message
4. HTTP: No response to a GET is returned before the response to an earlier GET.
5. RIP (or any other routing protocol): No loops in routing tables. The path taken by any packet is the shortest path.
6. Mutual exclusion: No two simultaneous grants of resources

How to prove a safety property:

1. Relate it to a state invariant
2. Prove the state invariant

No cycles in the set of parent edges defined by BFS algorithm. Consider doing an

induction: If the current set of parent edges has no cycles, then there is no action that will create one. What is the invariant for the proof?

Liveness properties

A liveness property says that good things do happen. traces(P) includes only those traces with the good things in them:
Every finite sequence over sig(P) has an extension in traces(P).

Examples
A leader is eventually elected

$$\text{traces}(P) = \{\ \tau_1, \tau_2, \ldots \mid \forall i(\tau_i \in \text{sig}(P)) \wedge \exists i, r\ (\tau_i = \text{leader}(r))\ \}$$

A process eventually gets an IP address
A browser eventually gets a page

Note: you will have to assume fairness to get these properties.
What we can say is that "Every *trace* of a correct leader election algorithm contains at most one leader action" and "Every *fair trace* of a correct leader election contains a leader action."

It is a liveness property that every node gets marked by BFS.
Invariant assertion:
Spanning tree example: powerpoint

Spanning Tree

See Peterson & Davie, 3.2.2

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

Here's a different spanning tree protocol.

What does the Cisco STP compute

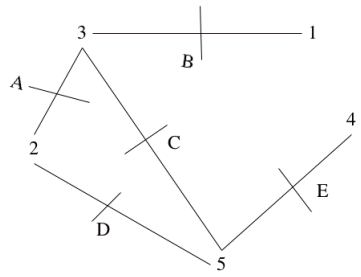
When the protocol stabilizes, the state should be as follows:

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

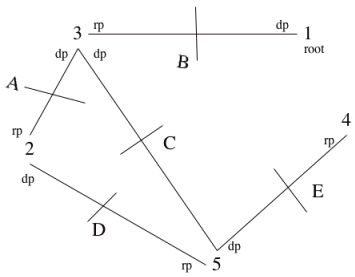
to the root. If there is a tie, it is on the bridge with the lowest MAC address. If the bridge selected by this rule has multiple ports on a network, it is the port with the lowest id.

Examples

Example 1

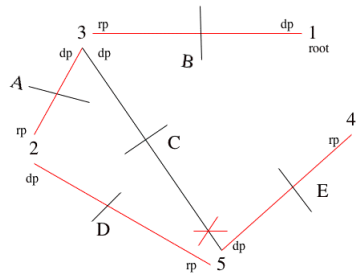


Example 1



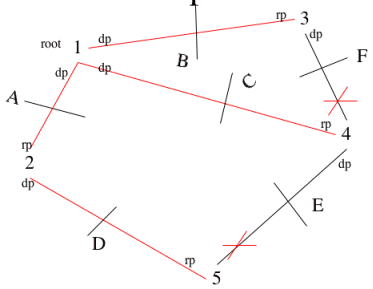
Assume ports are numbered clockwise starting at 12

Example 1



Assume ports are numbered clockwise starting at 12

Example 2



Assume ports are numbered clockwise starting at 12

Cisco Version

All processes send a BPDU (Basic Protocol Data Unit) at each round (actually, default is every 2 seconds). 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). The choice of root is essentially leader election.
Finally it designates the port to which the sending neighbor is connected as the root port. (Effectively choosing its parent) The desitnation of a root port 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. This ensures that messages on a network segment (or a LAN) connected to multiple switches will be forwarded.

Minimum Weight Spanning Tree

Minimize total weight of all edges in the tree. We will show a synchronous algorithm here, on the way to a much more complex asynchronous algorithm later.

Assume:
G=(V,E) undirected
weights are known by adjacent processes
UID's
n (size of graph) is known

Each node will decide which of its adjacent edges is or is not in the tree.

Theory

Spanning tree of G is a tree connecting all nodes, with edges selected from G.
Spanning forest is a collection of trees spanning all the nodes, with edges selected from G.

All MST algorithms are special cases of a general strategy.

1) Start with trivial spanning forest of n separate nodes.
2) Merge components along edges that connect components until all are connected (but no cycles).

The trick is to make sure that we merge only along edges that are minimum weight outgoing edges of some component.

Justification:
Lemma Let (V_i, E_i) $1 <= i <= k$ be a spanning forest, with $k>1$

Fix any i
Let e be an edge of smallest weight among the set of edges with exactly one endpoint in V_i.
Then there is a spanning tree for G that

1) Includes $\cup_j E_j$

2) Includes e
3) Has min weight among all spanning trees that include $\cup_j E_j$

Proof:
Suppose this is false, then there is some T that is a spanning tree for G and includes $\cup_j E_j$, doesn't contain e, and has weight strictly less than the weight of any spanning tree that includes both $\cup_j E_j$ and e.

We can construct T' by adding e to T. This contains a cycle, which must contain another edge outgoing from the same component (V_i,E_i). The weight of e' must be greater than or equal to the weight of e. Remove e' from T'. The result is a spanning tree that contains both $\cup_j E_j$ and e and has weight no greater than T'. This contradicts the choice of T.

Strategy for MST

Repeatedly:
Choose component i
Choose any least-weight outgoing edge from i and add (merging two components)

Sequential MST are special cases:
Dijkstra adds 1 more node on each iteration
Kruskal adds min weight edge globally

Distributed version:
We want to choose edges concurrently for multiple components – but (if multiple edges have the same weight) this can produce cycles.

Assume all edge weights are distinct (we can get the same effect by breaking ties with UID's.)

Lemma. If all edge weights are distinct, then there is a unique MST.

The concurrent strategy:
At each stage, suppose (inductively) that the forest produced so far is part of the unique MST.
Each component chooses a least-cost outgoing edge.
Each of these is in the (unique) MST, by Lemma 1.
So, all are ok – add them all.

Relationship to Cisco spanning tree

Is the Cisco spanning tree a min-weight tree?