Caltech

CS/IDS 142: Lecture 4.2
Diffusing Computations (Gossip) and Synchronization

Richard M. Murray

23 October 2019

Goals:

- Describe computations in which information spreads randomly but occasionally needs to be "synchronized"
- Introduce the idea of designing the algorithm to match the proof

Reading:

- P. Sivilotti, *Introduction to Distributed Algorithms*, Chapter 6

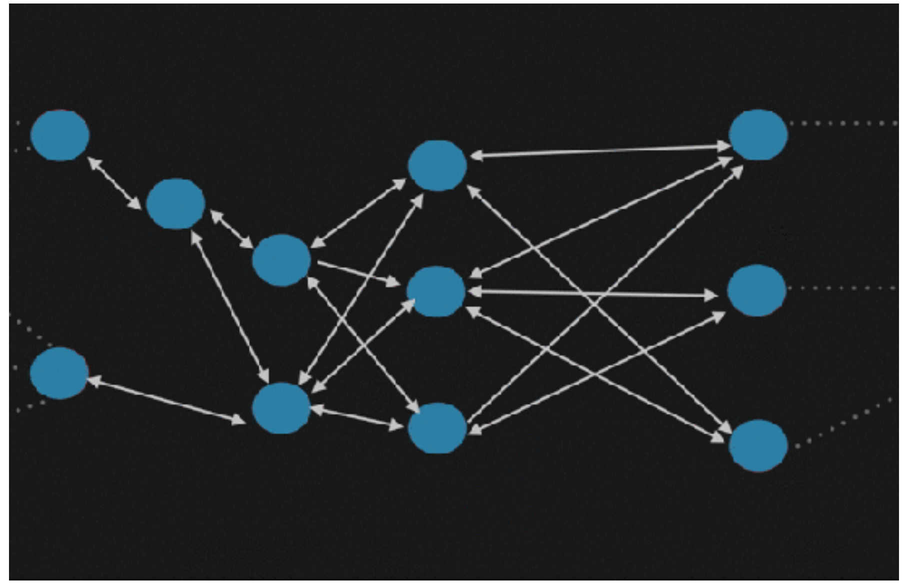
Diffusing Computations

Gossip algorithms

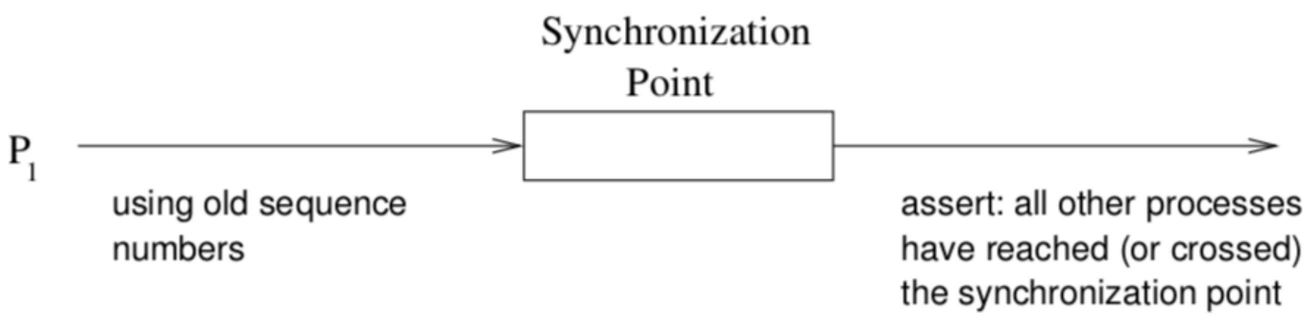
- Spread information around a network using random messages between agents
- Agents that receive the information spread it further in the graph until everyone knows

Some examples of gossip algorithms

- Bitcoin: transactions are broadcast to the network and aggregators add to their block chain, then spread the information
- Barrier synchronization: roll over seq numbers to account for finite bit counters: make sure that everyone rolls over at same time



https://managementfromscratch.wordpress.com/2016/04/01/introduction-to-gossip/



Properties we would like to prove

- Safety properties: once gossip is complete, it remains complete (nobody forgets) = FP
- Progress properties: everyone eventually gets the message

Termination Detection: Specification

Problem description

- Distribute a computing task amongst a set of agents (al a MapReduce)
- Subnodes may farm out computing further
- Computation terminates when all nodes finished
- Initiator node needs to keep track of when everyone is done with their computation

Initial state

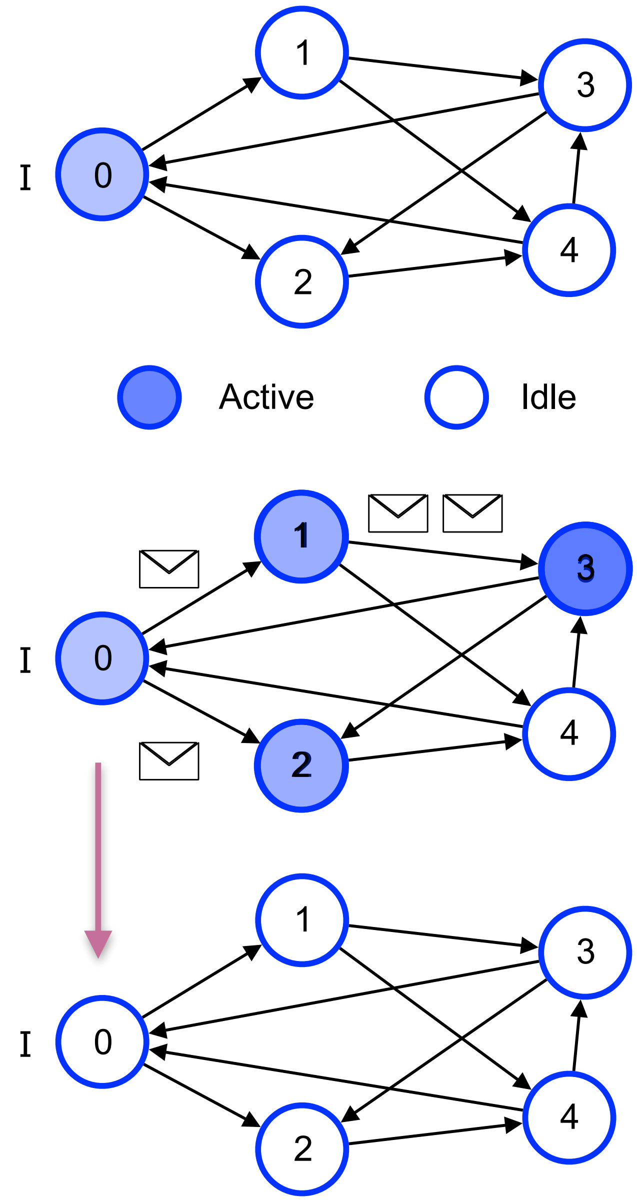
- All agents other than initiator are idle. All channels are empty. The initiator is active.

Termination condition (fixed point)

- All agents (including initiator) are idle; all channels are empty

Problem statement: Devise an algorithm by which the initiator detects that the computation has terminated. Initiator has a variable *claim* (for claim terminated) where

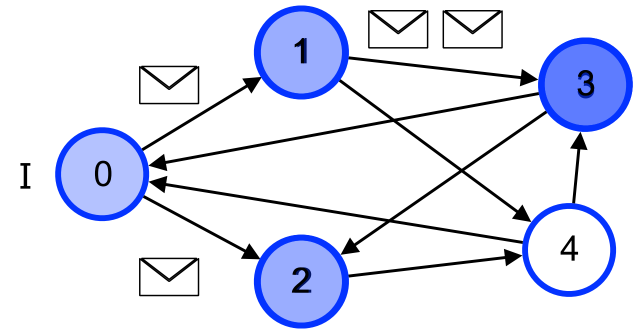
- Safety: **invariant** ($claim \Longrightarrow$ termination condition)
- Progress: termination condition $\rightsquigarrow$ *claim*



Termination Detection: Data Structure for Proof

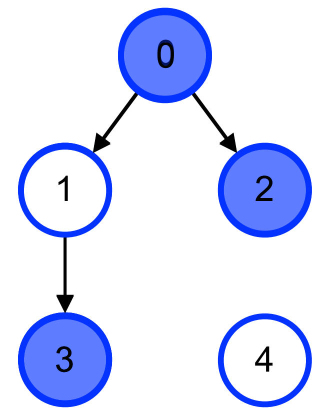
Approach: determine structure to represent system state

- Find a distributed data structure such that:
  - All active processes and the initiator are in the structure, and if a message is in transit from u to v then u or v (or both) are in the structure
  - Termination condition leads to the structure becoming empty.
- Q: what type of data structure should we use?



Rooted tree structure

- Variables: for each process *p*, keep track of *p.parent*
- Invariant: *p.parent* = null if and only if *p* is not on the tree
- Invariant: *p.parent* = *v* (for some *v*) if and only if *p*'s parent is *v* where *v* is on the tree.
- Invariant:
  - Every vertex is either
    - Unconnected (i.e., has no parent) or
    - Part of a tree rooted at initiator (i.e., it has a unique directed path along parent edges to the initiator)
  - Process *q* is active implies q is on the tree
  - Message from *p* to *q* implies *p* is on the tree



invariant ((p.parent = null) ≡ (p ∉ tree))

invariant ((p.parent = v) ≡
(parent(p) = v) ∧ (v ∈ tree))

invariant (∀ p ::
(p ∉ tree) ∨ (p ∈ tree ∧ tree.root = I)
∧ (p.state = active ⇒ p ∈ tree)
∧ (msg(p, q) ⇒ p ∈ tree))

Termination Detection: Create Algorithm

Idea: construct the algorithm to match the proof

- Look at the invariants that we want to impose based on the proof structure
- Create guarded commands that enforce the invariants
- When done, check to make sure that the progress condition is also satisfied (not enough to just be safe, have to actually do something useful)

Action: send message

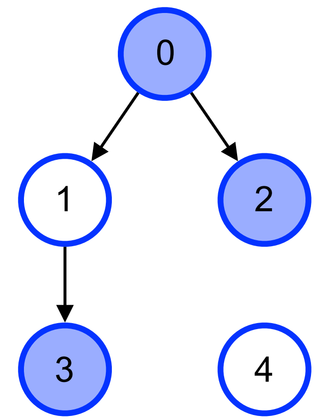
- Given: Only active processes can send messages
- From the invariant it follows that a process *p* sends a message only if *p* is on the tree (p.parent ≠ null)

Action: receiving messages

- When a process receives a message it becomes active
- To maintain the invariant, this process must join the tree if it is not already part of the tree.
- To maintain the invariant, a process *q* that is not on the tree and that receives a message from a process *p* and thus becomes active does what: ________________

Action*:* compute

- change state from active to idle



invariant ((p.parent = null) ≡ (p ∉ tree))

invariant ((p.parent = v) ≡
(parent(p) = v) ∧ (v ∈ tree))

invariant (∀ p ::
((p ∉ tree) ∨ (p ∈ tree ∧ tree.root = I))
∧ (p.state = active ⇒ p ∈ tree)
∧ (∃q :: msg(p, q)) ⇒ p ∈ tree))

Program *DetectTermination* (for p)

p.state = active ∧ p.parent ≠ null
→ send(p, q) || msg(p, q) += 1

[] recv(p, q) ∧ p.state = idle
→ p.state = active || p.parent := q ||
msg(q, p) -= 1

[] p.state = active
→ p.state = idle || p.parent = null

Termination Detection: Create Algorithm

Action: process becomes idle

- When *p* becomes idle it must remain on the tree if:
  - There exists message from *p* in a channel to an agent
  - Process *p* is the root of a subtree
- Solution: associate an *ack* (counter) with each process
  - An idle process on the tree does not change its parent while it has an outstanding ack

Action: process is idle and no outstanding acks

- Remove *p* from the tree and send ack to parent

Action: receiving messages

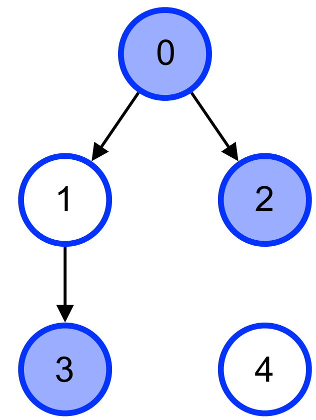
- When an agent *q* that is not on the tree becomes active because it got a message from an agent *p*, then it sets

  $$q\text{.parent} := p$$

  and *holds on to the ack for that message*
  - Invariant: *q*.parent = *p* only if *p* is active or has at least one outstanding ack
- Send acks for all other messages immediately
  - p is active and already on tree

Action: receive ack

- Decrease ack counter



Program *DetectTermination (p)*

```
   p.state = active ∧ p.parent ≠ null
     → send(p, q) || msg(p, q) += 1 ||
         ack(p, q) += 1
[] p.state = idle ∧ (∀q :: ack(p, q) = 0)
     →p.parent = null || send_ack(p.parent)
[] recv(p, q) ∧ p.state = idle
     → p.state = active || p.parent := q ||
         msg(p, q) -= 1
[] recv(p, q) ∧ p.state = active
     → send_ack(p, q)
[] recv_ack(q) → ack(p, q) -= 1
[] p.state = active
     → p.state = idle
```

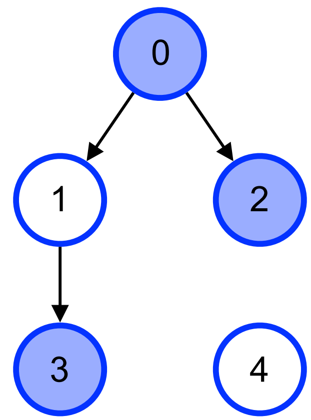
Termination Detection: Progress Property

Progress

- After computation terminates the tree must shrink to including just the initiator.
- Variant function: size (number of nodes on the tree).
- Assume computation has terminated. How do we ensure that the tree shrinks?
- From the invariant, all active processes are on the tree so only idle processes can drop off
- To maintain the rooted tree, only leaves of the tree can drop off
- Therefore, the invariant tells us that only idle leaves can drop off the tree

Determining whether a process is a leaf

- From invariant, *q*.parent = *p* only if "q has received more messages from *p* than it has ack'd to *p*", it follows that:
  - Invariant: *p* has no children if *p* has received acks for all the messages it has sent.
- So, p can drop off the tree if *p* has received acks for all the messages it has sent and *p* is idle



Program *DetectTermination (p)*

```
   p.state = active ∧ p.parent ≠ null
     → send(p, q) || msg(p, q) += 1 ||
          ack(p, q) += 1
[] p.state = idle ∧ (∀q :: ack(p, q) = 0)
     →p.parent = null || send_ack(p.parent)
[] recv(p, q) ∧ p.state = idle
     → p.state = active || p.parent := q ||
          msg(p, q) -= 1
[] recv(p, q) ∧ p.state = active
     → send_ack(p, q)
[] recv_ack(q) → ack(p, q) -= 1
[] p.state = active
     → p.state = idle
```

Termination Detection: Sketch of Proof (by Construction)

Specification (*claim* = initiator variable, "claim terminated")

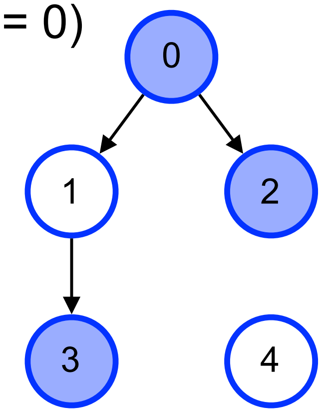
- Safety: **invariant** (*claim* $\Longrightarrow$ ($\forall$p ≠ I :: p.state = idle))
- Progress: ($\forall$p ≠ I :: p.state = idle) $\rightsquigarrow$ *claim*

Fixed point: ($\forall$p,q :: p.state = idle $\wedge$ msg(p, q) = 0)

Invariants: (right)

Metric: number of nodes in rooted tree

- Assume computation has terminated
- Size of the tree = m
- The only actions that are allowed at this point are for leaf nodes to remove themselves by setting p.parent = null (see program)
- This action must eventually take place (fairness) => number of nodes must decrease



Theorem 10 (Induction for $\rightsquigarrow$). *For a metric M,*

$$(\forall m :: \; P \wedge M = m \rightsquigarrow (P \wedge M < m) \vee Q \,) \quad \Rightarrow \quad P \rightsquigarrow Q$$

invariant ((p.parent = null) ≡ (p $\notin$ tree))

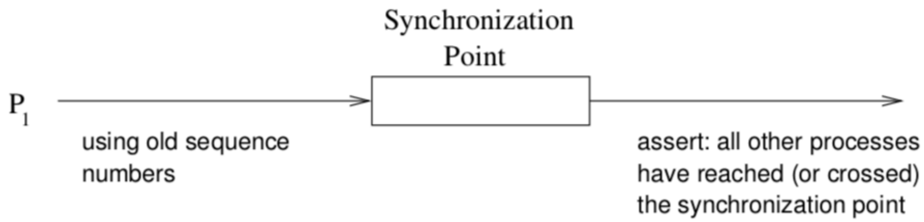
invariant ((p.parent = v) ≡
(parent(p) = v) $\wedge$ (v $\in$ tree))

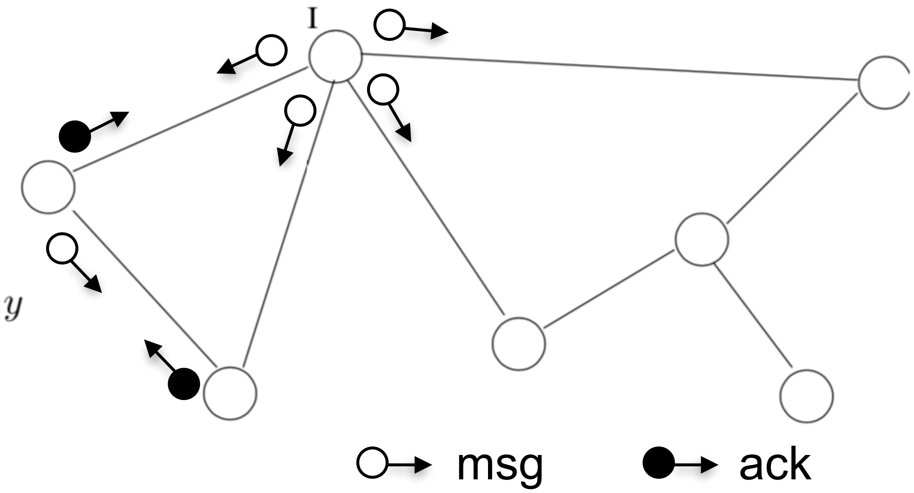
invariant ($\forall$ p ::
((p $\notin$ tree) $\vee$ (p $\in$ tree $\wedge$ tree.root = I))
$\wedge$ (p.state = active $\Rightarrow$ p $\in$ tree)
$\wedge$ ($\exists$q :: msg(p, q)) $\Rightarrow$ p $\in$ tree))

Program *DetectTermination (p)*

```
   p.state = active ∧ p.parent ≠ null
     → send(p, q) || msg(p, q) += 1 ||
         ack(p, q) += 1
[] p.state = idle ∧ (∀q :: ack(p, q) = 0)
     →p.parent = null || send_ack(p.parent)
[] recv(p,  q) ∧ p.state = idle
     → p.state = active || p.parent := q ||
        msg(p, q) -= 1
[] recv(p,  q) ∧ p.state = active
     → send_ack(p, q)
[] recv_ack(q) → ack(p, q) -= 1
[] p.state = active
     → p.state = idle
```

Barrier Synchronization: Operational View





$$x \ nbr \ y \quad \equiv \quad x \text{ and } y \text{ are neighbors}$$

$$msg(x, y) \quad \equiv \quad \text{there's a message in the channel from } x \text{ to } y$$

$$done \quad \equiv \quad (\forall v \ : \ v \ nbr \ I \ : \ msg(v, I) \ )$$

Phase I: diffusion

- Initiator sends synchronization message
- Each agent receiving message passes on to “children”
- Eventually, all agents will receive a message (possibly more than once)

Phase II: termination (everyone needs to know that everyone else is at the barrier)

- Basic idea: send back and acknowledgement once all children have responded
- Problem: graph may not be a tree => could get multiple copies; how to ack?
- Solution: send *message* back as ack => can stop as soon as “child” responds
- Challenge: prove that this works properly in all cases

Safety: $\textbf{invariant}.(done \ \Rightarrow \ (\forall u \ :: \ u \text{ has completed gossip }))$

Progress: $(\forall v \ : \ v \ nbr \ I \ : \ msg(I, v) \ ) \leadsto done$

Barrier Synchronization: Algorithm

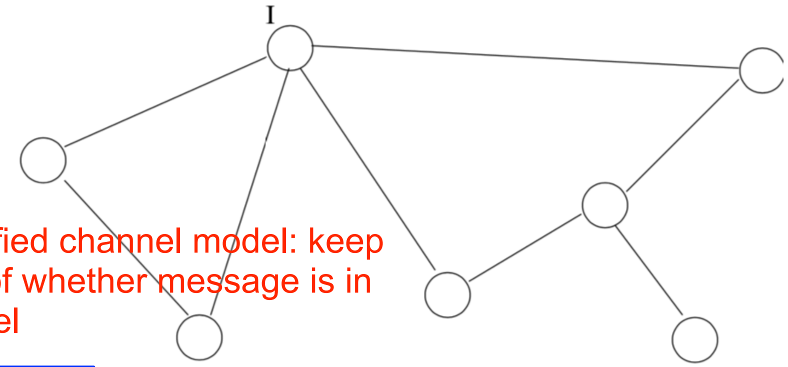
$$
\begin{array}{ll}
\textbf{Program} & \text{Gossip } u \\
\textbf{var} & parent_u : \text{process}, \\
 & state_u : \{idle, active, complete\}, \\
 & msg(a,b) : \text{channel from } a \text{ to } b\,, \\
\textbf{initially} & idle \\
\quad\quad\wedge & (\forall v \,:\, u \; nbr \; v \,:\, \neg msg(u,v)\;) \\
\textbf{assign} &
\end{array}
$$

$$
\boxed{
\begin{array}{l}
(\;[\!]\; v \,:\, v \; nbr \; u \,:\, idle \wedge msg(v,u) \longrightarrow \\
\quad\quad parent_u := v \\
\quad \| \quad (\,\|\; w \,:\, w \; nbr \; u \wedge w \neq v \,:\, msg(u,w) := \mathbf{true}\;) \\
\quad \| \quad state_u := active\;)
\end{array}
}
$$

A1

$$
[\!]\;\boxed{
\begin{array}{l}
active \wedge (\forall v \,:\, v \; nbr \; u \wedge v \neq parent_u \,:\, msg(v,u)\;) \longrightarrow \\
\quad\quad msg(u, parent_u) := \mathbf{true} \\
\quad \| \quad state_u := complete
\end{array}
}
$$

A2



Simplified channel model: keep track of whether message is in channel

Meaning of process states:

- idle: waiting to hear
- active: rec'd gossip (=> can repeat to nbr)
- complete: ack(s) rec'd (=> neighbor knows)

Define:

- T1 = graph with vertices = set of *active* or *complete* nodes + I, edges = (u, u.parent) for all nodes
- T2 = graph with vertices = set of *active* nodes + initiator, edges = (u, u.parent) for all nodes

Invariants (to show)

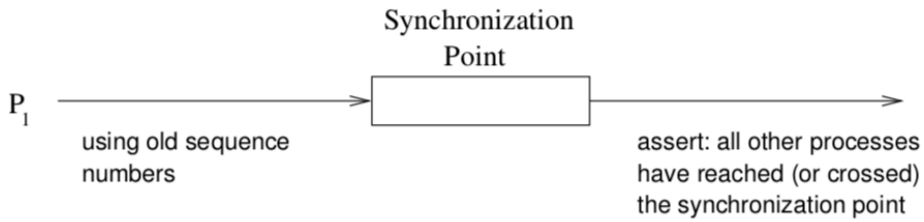
- T1 is an (expanding) tree
- T2 is an (expanding and contracting) tree

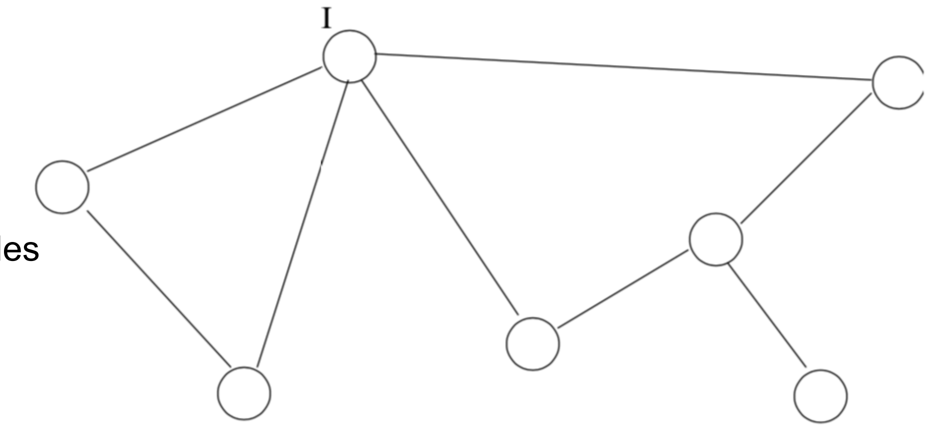
Add'l useful invariants:

$$msg(u,v) \quad \Rightarrow \quad u \in T_1$$

$$u.complete \quad \Rightarrow \quad (\forall v \,:\, <v,u> \in T_1 \,:\, v.complete\;)$$

Barrier Synchronization: Proof Structure





- idle: waiting to hear
- active: rec'd gossip (=> can repeat to nbr)
- complete: ack(s) rec'd (=> neighbor knows)

- T1 = active and complete nodes
- T2 = active nodes

Metric: m = (#complete, #active) = (|T1| - |T2|, |T2|)

- Use lexigraphic ordering: $(a1, b1) \le (a2, b2) \equiv a1 \le a2 \vee (a1 = a2 \wedge b1 \le b2)$
- Metric is bounded above since graph is bounded
- Guaranteed not to decrease since complete nodes can only go up

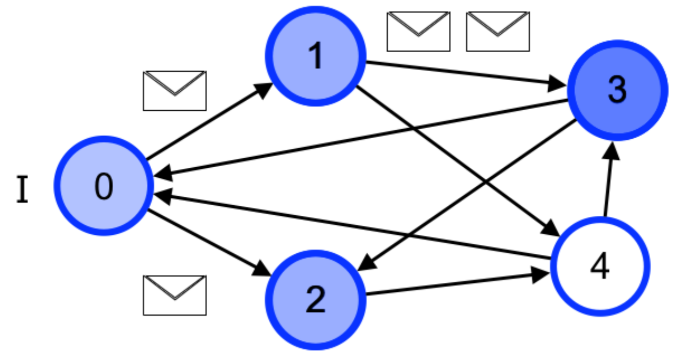
Need to show $(\exists v :: \neg \text{complete}.v) \wedge M{=}m \rightsquigarrow M > m$

- If node v is *idle* then it hasn't received the gossip yet
- There must exist path back to root and some node along this path is *idle* with parent in T1
- For that node, action A1 (previous slide) can eventually get selected => M will increase
- If node v is *active* but not *complete* then action A2 implies that node can send a message to neighbor who hasn't yet heard gossip and change state to complete (increasing m) or there is no such node and we still change state to complete (increasing m)

Summary: Diffusing Computations (Gossip)

Basic idea: distribute information to all nodes

- Key problem is understanding when the algorithm has terminated (all nodes idle, no information in channels)
- Make use of a tree structure to propagate information



Properties

- Safety: **invariant** (*claim* $\Longrightarrow$ termination condition)
- Progress: termination condition $\rightsquigarrow$ *claim*

Example algorithm (synchronization)

$$
\begin{array}{ll}
\textbf{initially} & idle \\
& \wedge \quad (\forall v : u \; nbr \; v : \neg msg(u,v)) \\
\textbf{assign} & \\
& (\, [\!] \; v : v \; nbr \; u : idle \wedge msg(v,u) \longrightarrow \\
& \qquad parent_u := v \\
& \quad \| \quad (\| \; w : w \; nbr \; u \wedge w \neq v : msg(u,w) := \textbf{true} \,) \\
& \quad \| \quad state_u := active \,) \\
& [\!] \; active \wedge (\forall v : v \; nbr \; u \wedge v \neq parent_u : msg(v,u)) \longrightarrow \\
& \qquad msg(u, parent_u) := \textbf{true} \\
& \quad \| \quad state_u := complete
\end{array}
$$

$parent_u$: process,
$state_u$: $\{idle, active, complete\}$,
$msg(a,b)$: channel from a to b,

Simplified channel model

- Keep track of whether message is in channel
- Abstracts away the details of what is in the channel