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 the same time

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

https://managementfromscratch.wordpress.com/2016/04/01/introduction-to-gossip/
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: \( \text{invariant} \) (\( claim \rightarrow \text{termination condition} \))
  - Progress: \( \text{termination condition} \rightarrow claim \)
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

\[
\text{invariant } ((p.\text{parent} = \text{null}) \equiv (p \notin \text{tree}))
\]
\[
\text{invariant } ((p.\text{parent} = v) \equiv \\
(\text{parent}(p) = v) \land (v \in \text{tree}))
\]
\[
\text{invariant } (\forall \ p :: \\
(p \notin \text{tree}) \lor (p \in \text{tree} \land \text{tree.root} = I) \land \\
(p.\text{state} = \text{active} \Rightarrow p \in \text{tree}) \land \\
(\text{msg}(p, q) \Rightarrow p \in \text{tree}))
\]
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 \( \neq \) 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

Program \textit{DetectTermination} (for \( p \))

\[
\begin{align*}
p\text{.state} &= \text{active} \land p\text{.parent} \neq \text{null} \\
& \rightarrow \text{send}(p, q) || \text{msg}(p, q) += 1 \\
\end{align*}
\]

\[
\begin{align*}
\text{recv}(p, q) \land p\text{.state} &= \text{idle} \\
& \rightarrow p\text{.state} = \text{active} || p\text{.parent} := q || \\
& \text{msg}(q, p) -= 1 \\
\end{align*}
\]

\[
\begin{align*}
p\text{.state} &= \text{active} \\
& \rightarrow p\text{.state} = \text{idle} || p\text{.parent} = \text{null}
\end{align*}
\]
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 $q.parent := p$
  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 \land p.parent \neq null
\rightarrow send(p, q) \parallel msg(p, q) += 1 \parallel
\quad ack(p, q) += 1
\]
\[
\quad p.state = idle \land (\forall q :: ack(p, q) = 0)
\rightarrow p.parent = null \parallel send_\text{ack}(p.parent)
\]
\[
\quad recv(p, q) \land p.state = idle
\rightarrow p.state = active \parallel p.parent := q \parallel
\quad msg(p, q) -= 1
\]
\[
\quad recv(p, q) \land p.state = active
\rightarrow send_\text{ack}(p, q)
\]
\[
\quad recv_\text{ack}(q) \rightarrow ack(p, q) -= 1
\]
\[
\quad p.state = active\rightarrow p.state = idle
```
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, \( q.\text{parent} = p \) 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 \( \text{DetectTermination} (p) \)

\[
\begin{align*}
\text{recv}(p, q) & \land p.\text{state} = \text{active} \rightarrow p.\text{state} = \text{idle} \\
\text{send}(p, q) & \lor \text{msg}(p, q) += 1 \\
\text{ack}(p, q) & += 1 \\
\text{recv}(p, q) & \land p.\text{state} = \text{active} \rightarrow \text{send\_ack}(p, q) \\
\text{send\_ack}(q) & \rightarrow \text{ack}(p, q) -= 1 \\
\text{send}(p, q) & \lor \text{msg}(p, q) += 1 \\
\text{ack}(p, q) & += 1 \\
\text{send\_ack}(p, q) & \rightarrow \text{p.state} = \text{idle} \\
\text{recv\_ack}(q) & \rightarrow \text{p.state} = \text{idle}
\end{align*}
\]
Termination Detection: Sketch of Proof (by Construction)

Specification (claim = initiator variable, “claim terminated”)
  • Safety: invariant (claim \implies (\forall p \neq I :: p\text{.state} = idle))
  • Progress: (\forall p \neq I :: p\text{.state} = idle) \rightsquigarrow claim

Fixed point: (\forall p,q :: p\text{.state} = idle \land 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) \implies number of nodes must decrease

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

(\forall m :: P \land M = m \rightsquigarrow (P \land M < m) \lor Q ) \implies P \rightsquigarrow Q

invariant ((p.parent = null) \equiv (p \notin tree))

invariant ((p.parent = v) \equiv (parent(p) = v) \land (v \in tree))

invariant (\forall p ::

((p \notin tree) \lor (p \in tree \land tree\text{.root} = I))
\land (p\text{.state} = active \implies p \in tree)
\land (\exists q :: msg(p, q) \implies p \in tree))

Program DetectTermination (p)

p\text{.state} = active \land p\text{.parent} \neq null
\implies send(p, q) \parallel msg(p, q) += 1 \parallel ack(p, q) += 1

\parallel p\text{.state} = idle \land (\forall q :: ack(p, q) = 0)
\implies p\text{.parent} = null \parallel send\_ack(p\text{.parent})

\parallel recv(p, q) \land p\text{.state} = idle
\implies p\text{.state} = active \parallel p\text{.parent} := q \parallel msg(p, q) -= 1

\parallel recv(p, q) \land p\text{.state} = active
\implies send\_ack(p, q)

\parallel recv\_ack(q) \implies ack(p, q) -= 1

\parallel p\text{.state} = active
\implies p\text{.state} = idle
Barrier Synchronization: Operational View

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

**Progress:** \( (\forall v : v \text{ nbr } I : \text{msg}(I,v)) \sim \text{done} \)
Barrier Synchronization: Algorithm

Define:
- $T_1$ = graph with vertices = set of active or complete nodes + I, edges = (u, u.parent) for all nodes
- $T_2$ = graph with vertices = set of active nodes + initiator, edges = (u, u.parent) for all nodes

Add’l useful invariants:
- $\text{msg}(u, v) \Rightarrow u \in T_1$
- $u.\text{complete} \Rightarrow (\forall v : <v, u> \in T_1 : v.\text{complete})$

Invariants (to show):
- $T_1$ is an (expanding) tree
- $T_2$ is an (expanding and contracting) tree

Meaning of process states:
- idle: waiting to hear
- active: rec’d gossip ($\Rightarrow$ can repeat to nbr)
- complete: ack(s) rec’d ($\Rightarrow$ neighbor knows)

Program Gossip $u$

var

\begin{align*}
\text{parent}_u & : \text{process}, \\
\text{state}_u & : \{\text{idle, active, complete}\}, \\
\text{msg}(a, b) & : \text{channel from } a \text{ to } b, \\
\text{initially} & \quad \text{idle} \\
\land & \quad (\forall v : u \text{ nbr } v : \neg \text{msg}(u, v)) \\
\text{assign} & \\
\quad (\mid v : v \text{ nbr } u : \text{idle} \land \text{msg}(v, u) \rightarrow \\
\quad \quad \text{parent}_u := v \\
\quad \mid (\mid w : w \text{ nbr } u \land w \neq v : \text{msg}(u, w) := \text{true} ) \\
\quad \mid \text{state}_u := \text{active} ) \\
\mid \text{active} \land (\forall v : v \text{ nbr } u \land v \neq \text{parent}_u : \text{msg}(v, u) ) \rightarrow \\
\quad \text{msg}(u, \text{parent}_u) := \text{true} \\
\mid \text{state}_u := \text{complete}
\end{align*}
Barrier Synchronization: Proof Structure

Metric: \( m = (\#\text{complete}, \#\text{active}) = (|T_1| - |T_2|, |T_2|) \)
- Use lexicographic ordering: \((a_1, b_1) \leq (a_2, b_2) \equiv a_1 \leq a_2 \lor (a_1 = a_2 \land b_1 \leq b_2)\)
- Metric is bounded above since graph is bounded
- Guaranteed not to decrease since complete nodes can only go up

Need to show \( (\exists v :: \neg \text{complete} . v) \land M = m \implies M > m \)
- 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 \( T_1 \)
- For that node, action A1 (previous slide) can eventually get selected \( \implies M \) 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 $\implies$ termination condition)
- Progress: termination condition $\leadsto$ claim

Example algorithm (synchronization)

```
initially idle
\land (\forall v : u \text{ nbr } v : \neg msg(u, v))
assign 
    ( \| v : v \text{ nbr } u : idle \land msg(v, u) \implies 
        parent_u := v 
    
    \| ( \| w : w \text{ nbr } u \land w \neq v : msg(u, w) := \text{true} ) 
    
    \| state_u := \text{active} ) 
\| active \land (\forall v : v \text{ nbr } u \land v \neq parent_u : msg(v, u) ) \implies 
    msg(u, parent_u) := \text{true} 
\| state_u := \text{complete}
```

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