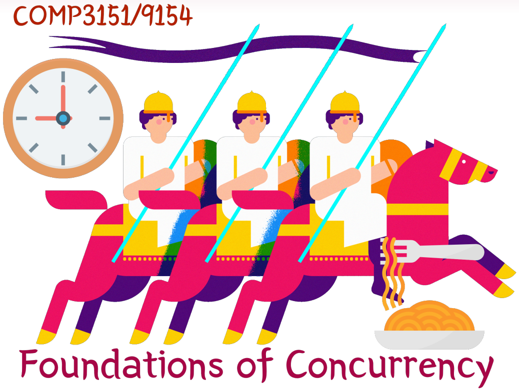


Global Properties

Johannes Åman Pohjola
UNSW

Term 2 2022

Where we're at

Last lecture, we discussed classic distributed algorithms for commitment and consensus.

Today, we'll discuss algorithms for establishing global properties about the system state.

Global Properties

We'll look at two global properties:

1. How to tell if a distributed computation has terminated.
2. How to get a snapshot of the current state of a distributed system.

Global Properties

We'll look at two global properties:

1. How to tell if a distributed computation has terminated.
2. How to get a snapshot of the current state of a distributed system.

Question

For single-machine systems, these are relatively easy. Why are they hard in a distributed setting?

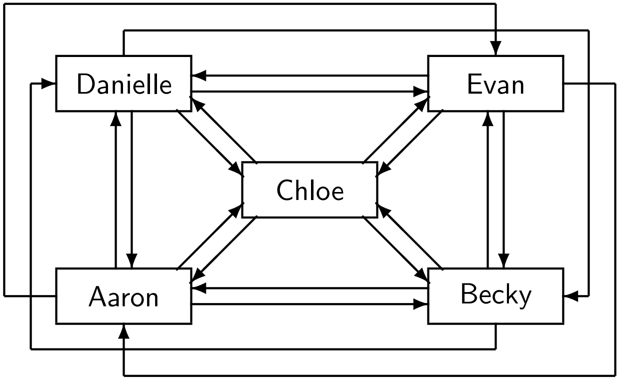
Setting

NB

The algorithms we present this lecture are not presented as stand-alone processes.

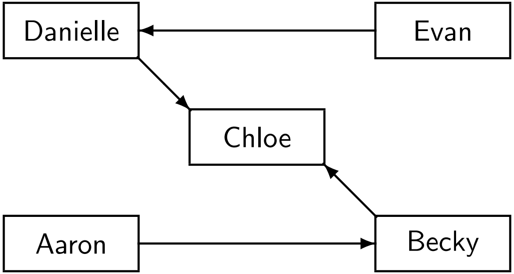
Rather, they are snippets of code meant to be integrated into some *underlying computation* the system is doing. For example, whenever the underlying computation wishes to send a message, the algorithm may require some additional local bookkeeping, and additional structure to the message.

Setting



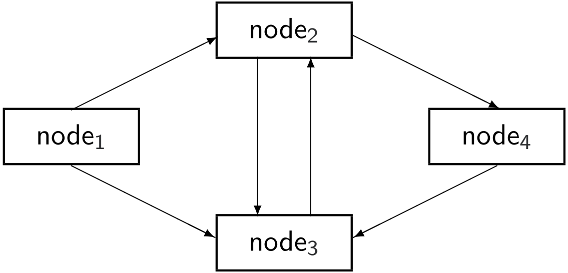
Last week, we assumed total connectivity: every node can communicate directly with every other node.

Setting



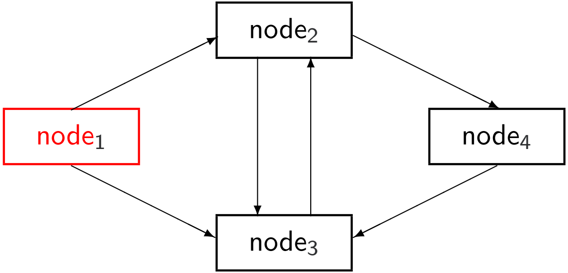
This week, nodes may be partially connected. Connectivity may be asymmetric. Multi-hop communication may be required to reach certain nodes, or there may be no path at all between two particular nodes.

Distributed System with an Environment Node



We'll assume an environment node with no incoming edges. Every other node must be reachable from it (perhaps via multiple hops).

Distributed System with an Environment Node



We'll assume an environment node with no incoming edges. Every other node must be reachable from it (perhaps via multiple hops).

node$_1$ is our environment node (and none of the others even qualify).

Distributed Termination

A distributed system has terminated when all processes at all nodes have terminated.

Problem

Design an algorithm that allows the environment node to announce termination, iff the system has indeed terminated.

Distributed Termination

A distributed system has terminated when all processes at all nodes have terminated.

Problem

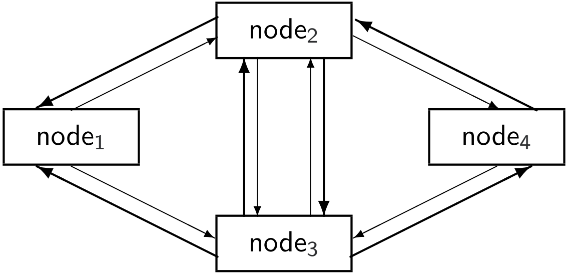
Design an algorithm that allows the environment node to announce termination, iff the system has indeed terminated.

node_1 is the environment node.

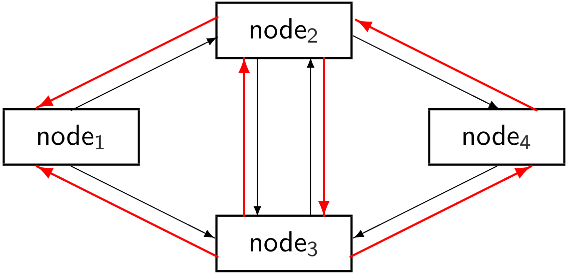
All nodes except node_1 are inactive intially. They can be awoken by incoming messages. Once messages arrive, nodes start their underlying computation, which may eventually terminate.

At each node, (local) termination of the underlying computation is detectable by a boolean function isTerminated.

Back Edges



Back Edges



We assume *back edges* to carry special messages called *signals*.

Dijkstra-Scholten algorithm (idea)

Use the *signal* channel to acknowledge receipt of every message along the other channels, while maintaining counters for message *deficits*. At node i we do the following.

- Every send increments $outDeficit_i$ — receiving a signal decrements it.
- Every receive along edge e increments $inDeficit_i[e]$ — sending a signal along e's back edge decrements $inDeficit_i[e]$.
- For convenience we also maintain the $\sum_e inDeficit_i[e]$ as $inDeficit_i$.

We start the computation by sending a message along every outgoing edge of node_1. node_1 announces termination when its $outDeficit$ returns to 0.

Algorithm 2.1: Dijkstra-Scholten algorithm (preliminary)

```
          integer array[incoming] inDeficit ← [0,. . . ,0]
          integer inDeficit ← 0, integer outDeficit ← 0
     send message
p1:  send(message, destination, myID)
p2:  increment outDeficit
     receive message
p3:  receive(message, source)
p4:  increment inDeficit[source] and inDeficit
     send signal
p5:  when inDeficit > 1 or
           (inDeficit = 1 and isTerminated and outDeficit = 0)
p6:     E ← some edge E with inDeficit[E] ≠ 0
p7:     send(signal, E, myID)
p8:     decrement inDeficit[E] and inDeficit
     receive signal
p9:  receive(signal, _)
p10: decrement outDeficit
```

Algorithm 2.2: Dijkstra-Scholten algorithm (env., preliminary)

integer outDeficit $\leftarrow$ 0

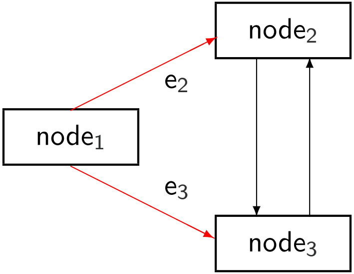
computation

```
p1: for all outgoing edges E
p2:     send(message, E, myID)
p3:     increment outDeficit
p4: await outDeficit = 0
p5: announce system termination
```

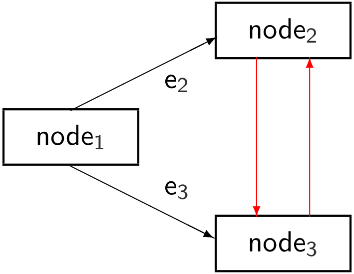
receive signal

```
p6: receive(signal, source)
p7: decrement outDeficit
```

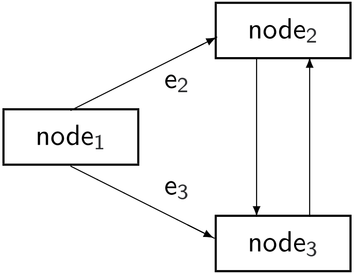
The Preliminary DS Algorithm is Unsafe



The Preliminary DS Algorithm is Unsafe



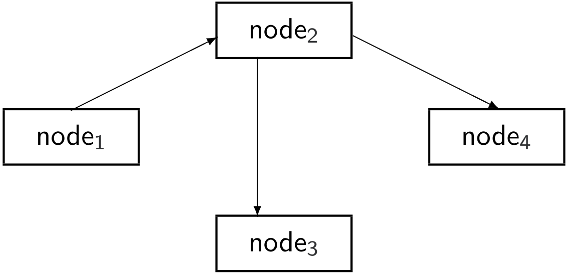
The Preliminary DS Algorithm is Unsafe



For $i \in \{2, 3\}$ we have node$_i$'s *inDeficit* $= 2$ and *inDeficit*[e$_i$] $= 1$ so both can signal back to node$_1$, who is now fooled into announcing termination.

Spanning Tree

The unsafe example cannot be reconstructed if the channel graph is a tree (with the environment node as root).



Algorithm 2.3: Dijkstra-Scholten algorithm

```
          integer array[incoming] inDeficit ← [0,...,0]
          integer inDeficit ← 0
          integer outDeficit ← 0
          integer parent ← −1
    send message
p1: when parent ≠ −1          // Only active nodes send messages
p2:    send(message, destination, myID)
p3:    increment outDeficit
    receive message
p4: receive(message,source)
p5: if parent = −1
p6:    parent ← source
p7: increment inDeficit[source] and inDeficit
```

Algorithm 2.3: Dijkstra-Scholten algorithm (continued)
send signal

```
p8:  when inDeficit > 1
p9:      E ← some edge E for which
             (inDeficit[E] > 1) or (inDeficit[E] = 1 and E ≠ parent)
p10:     send(signal, E, myID)
p11:     decrement inDeficit[E] and inDeficit
p12: or when inDeficit = 1 and isTerminated and outDeficit = 0
p13:     send(signal, parent, myID)
p14:     inDeficit[parent] ← 0
p15:     inDeficit ← 0
p16:     parent ← −1
```

receive signal

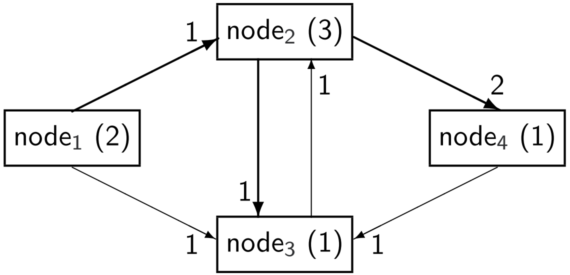
```
p17: receive(signal, _)
p18: decrement outDeficit
```

Partial Scenario for DS Algorithm

Action	$node_1$	$node_2$	$node_3$	$node_4$
$1 \Rightarrow 2$	(-1,[ ],0)	(-1,[0,0],0)	(-1,[0,0,0],0)	(-1,[0],0)
$2 \Rightarrow 4$	(-1,[ ],1)	(1,[1,0],0)	(-1,[0,0,0],0)	(-1,[0],0)
$2 \Rightarrow 3$	(-1,[ ],1)	(1,[1,0],1)	(-1,[0,0,0],0)	(2,[1],0)
$2 \Rightarrow 4$	(-1,[ ],1)	(1,[1,0],2)	(2,[0,1,0],0)	(2,[1],0)
$1 \Rightarrow 3$	(-1,[ ],1)	(1,[1,0],3)	(2,[0,1,0],0)	(2,[2],0)
$3 \Rightarrow 2$	(-1,[ ],2)	(1,[1,0],3)	(2,[1,1,0],0)	(2,[2],0)
$4 \Rightarrow 3$	(-1,[ ],2)	(1,[1,1],3)	(2,[1,1,0],1)	(2,[2],0)
	(-1,[ ],2)	(1,[1,1],3)	(2,[1,1,1],1)	(2,[2],1)

$i \Rightarrow k$ means "$node_i$ sends to $node_k$"; node state notation: (parent,inDeficit[E],outDeficit)

Data Structures After Completion of Partial Scenario



(outDeficit in parentheses, inDeficits on edges)

Dijkstra-Scholten

Dijkstra-Scholten so far has two shortcomings:

Traffic overhead For huge computations, we need to send as many signals as messages

Unbounded deficits For huge computations, deficits may grow to the point where they no longer fit in memory.

Dijkstra-Scholten

Dijkstra-Scholten so far has two shortcomings:

Traffic overhead For huge computations, we need to send as many signals as messages

Unbounded deficits For huge computations, deficits may grow to the point where they no longer fit in memory.

Unbounded deficitis is mainly a theoretical problem. Deficits are only ever stored locally, not sent, so messages are fixed size. Just use arbitrary-precision arithmetic to represent them locally.

Dijkstra-Scholten

One way to reduce traffic overhead in Dijkstra-Scholten is to reduce the deficit as much as possible in a signal. That is, let signals carry a number:

send(signal, E, myID, N)

Where N is the amount deficit we're discharging. (Note the tradeoff: less message quantity, more message complexity)

Credit-recovery algorithms

In a credit-recovery algorithm, the environment node initially holds 1 credit. Every other node holds 0 credits.

Credit-recovery algorithms

In a credit-recovery algorithm, the environment node initially holds 1 credit. Every other node holds 0 credits.

The credit is a divisible token. Every active node needs to hold > 0 credit.

Credit-recovery algorithms

In a credit-recovery algorithm, the environment node initially holds 1 credit. Every other node holds 0 credits.

The credit is a divisible token. Every active node needs to hold > 0 credit.

When you message somebody, give them some credit.

Credit-recovery algorithms

In a credit-recovery algorithm, the environment node initially holds 1 credit. Every other node holds 0 credits.

The credit is a divisible token. Every active node needs to hold > 0 credit.

When you message somebody, give them some credit.

When a node terminates, it gives all credit back to the environment node.

Algorithm 2.4: Credit-recovery algorithm (environment node)	
float weight $\leftarrow$ 1.0	
	computation
p1:	**for** all outgoing edges E
p2:	weight $\leftarrow$ weight / 2.0
p3:	send(message, E, myID, weight)
p4:	**await** weight = 1.0
p5:	announce system termination
	receive signal
p6:	receive(signal, w)
p7:	weight $\leftarrow$ weight + w

Algorithm 2.5: Credit-recovery algorithm (non-environment node)

```
            constant integer parent ← 0 // Environment node
            boolean active ← false
            float weight ← 0.0
     send message
p1:  if active
p2:     weight ← weight / 2.0
p3:     send(message, destination, myID, weight)
     receive message
p4:  receive(message, source, w)
p5:  active ← true
p6:  weight ← weight + w
     send signal
p7:  when terminated
p8:     send(signal,parent,weight)
p9:     weight ← 0.0
p10:    active ← false
```

Credit-recovery algorithms

Credit-recovery algorithms reduce message overhead in two ways:

1. You can discharge all your credit in a single message.
2. Credit goes straight back to the source. No need for multiple hops up the tree.

Credit-recovery algorithms

Credit-recovery algorithms reduce message overhead in two ways:

1. You can discharge all your credit in a single message.
2. Credit goes straight back to the source. No need for multiple hops up the tree.

Question

What are some (potential) drawbacks of credit-recovery algorithms?

Snapshots

A *global snapshot* is a recording of the states of all nodes and channels in the system.
This recording is (necessarily) distributed.
Node states record

- values of local variables
- sequences of messages sent and received

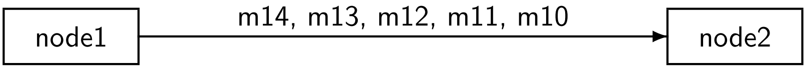
Channel states record

- sequences of messages still in transit

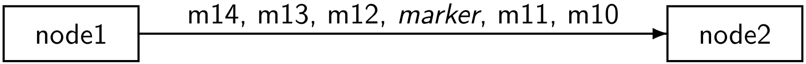
Definition

A global snapshot is *consistent* iff every sent message is either in transit or received.

Messages on a Channel



Sending a Marker



Algorithm 2.6: Chandy-Lamport algorithm for global snapshots	
	integer array[outgoing] lastSent $\leftarrow$ [0, . . . , 0]
	integer array[incoming] lastReceived $\leftarrow$ [0, . . . , 0]
	integer array[outgoing] stateAtRecord $\leftarrow$ [-1, . . . , -1]
	integer array[incoming] messageAtRecord $\leftarrow$ [-1, . . . , -1]
	integer array[incoming] messageAtMarker $\leftarrow$ [-1, . . . , -1]
	send message
p1:	send(message, destination, myID)
p2:	lastSent[destination] $\leftarrow$ message
	receive message
p3:	receive(message,source)
p4:	lastReceived[source] $\leftarrow$ message

where message $\neq$ marker

NB

This algorithm assumes all channels are FIFO.

Algorithm 2.6: Chandy-Lamport algorithm for global snapshots (continued)

```
    receive marker
receive(marker, source)
messageAtMarker[source] ← lastReceived[source]
if stateAtRecord = [−1,...,−1]      // Not yet recorded
    stateAtRecord ← lastSent
    messageAtRecord ← lastReceived
    for all outgoing edges E
        send(marker, E, myID)
    record state
await markers received on all incoming edges
recordState
```

The final state

When all marker messages have been received, the final state consists of the following:

- stateAtRecord[E]: the last message sent on each outgoing edge E.

The final state

When all marker messages have been received, the final state consists of the following:

- stateAtRecord[E]: the last message sent on each outgoing edge E.
- messageAtRecord[E]: the last message received on each incoming edge E.

The final state

When all marker messages have been received, the final state consists of the following:

- stateAtRecord[E]: the last message sent on each outgoing edge E.
- messageAtRecord[E]: the last message received on each incoming edge E.
- The messages in transit on the edge E are:
  - none, if messageAtMarker[E] and messageAtRecord[E] are equal;
  - the messages from messageAtRecord[E]+1 to messageAtMarker[E], otherwise.

The final state

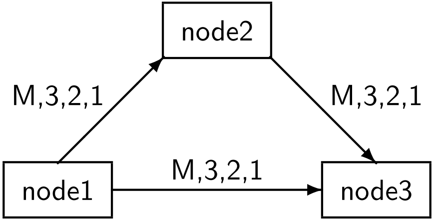
When all marker messages have been received, the final state consists of the following:

- stateAtRecord[E]: the last message sent on each outgoing edge E.
- messageAtRecord[E]: the last message received on each incoming edge E.
- The messages in transit on the edge E are:
  - none, if messageAtMarker[E] and messageAtRecord[E] are equal;
  - the messages from messageAtRecord[E]+1 to messageAtMarker[E], otherwise.

NB

Here we've assumed messages are numbered in the order they were sent, to simplify the presentation. Adaptations to record message content are straightforward.

Messages and Markers for a Scenario



Scenario for CL Algorithm (1)

Here, the three message from node1 to node2 have been received. Thre three messages from node1 to node3, and from node2 to node3, have all been sent but not received.

Action	node1					node2				
	ls	lr	st	rc	mk	ls	lr	st	rc	mk
	[3,3]					[3]	[3]			
1M⇒2	[3,3]		[3,3]			[3]	[3]			
1M⇒3	[3,3]		[3,3]			[3]	[3]			
2⇐1M	[3,3]		[3,3]			[3]	[3]			
2M⇒3	[3,3]		[3,3]			[3]	[3]	[3]	[3]	[3]

Scenario for CL Algorithm (2)

Action	node3				
	ls	lr	st	rc	mk
$3 \Leftarrow 2$					
$3 \Leftarrow 2$		[0,1]			
$3 \Leftarrow 2$		[0,2]			
$3 \Leftarrow 2$M		[0,3]			
$3 \Leftarrow 1$		[0,3]		[0,3]	[0,3]
$3 \Leftarrow 1$		[1,3]		[0,3]	[0,3]
$3 \Leftarrow 1$		[2,3]		[0,3]	[0,3]
$3 \Leftarrow 1$M		[3,3]		[0,3]	[0,3]
		[3,3]		[0,3]	[3,3]

Chandy-Lamport summary

Once consistent local snapshots are taken, they can be collected through a procedure similar to Dijkstra-Scholten.

The snapshot may be a state that never actually occurred in the system execution. But it's guaranteed to be a state that *could* have occurred in a different interleaving.

What now?

We're on the home stretch! Next week is recap and examp prep.

In the meantime, please fill in the myExperience survey to evaluate the course. Your feedback is tremendously helpful.