Distributed and Parallel Computing

Lecture 11

Alan P. Sexton

University of Birmingham

Spring 2018

Distributed Computing

By *Distributed Computing* we mean:

- Computing on multiple *nodes* (i.e. *processes*)
- Each node has a unique *identifier*
- Nodes are connected via a *network* of *channels* (i.e. *edges*)
- Nodes do not share memory or a global clock
- There can be 0 or 1 channels between any two nodes
- The network is assumed to be *strongly connected*: there is a (possibly multi-hop) path between every two nodes
- The network may or may not be *complete*: there is an undirected channel between every pair of nodes
    - Note difference from *strongly connected*
- Channels can be *directed* (messages travel one way only) or *undirected* (messages travel in either direction)
- Communication is via passing *messages* over *channels*
- Channels are not necessarily *First-In-First-Out (FIFO)*: messages sent on the same channel may overtake each other

More Definitions

- A process knows:
  - its own state
  - the messages it sends and receives
  - its direct neighbours in the network
- Underlying communication protocol is reliable
  - No messages corrupted, duplicated or lost
- Communication is *asynchronous*: there is an arbitrary, non-deterministic but finite delay between sending and receiving a message
- Parameters:
  - N: the number of nodes
  - E: the number of edges
  - D: the diameter of the network: the longest "shortest path"
    - the number of edges in the longest path chosen from the set of shortest paths between every pair of nodes in the graph

Failures in a Distributed System

Failure Rate (FR): The number of failures per unit time
Mean Time Before Failure (MTBF): $\frac{1}{\mathrm{FR}}$
Example: Distributed system with 1000 nodes, all of which are critical. MTBF of each node of 10,000 hours $\approx$ 60 weeks. What is the MTBF of the system?

Failures in a Distributed System

Failure Rate (FR): The number of failures per unit time
Mean Time Before Failure (MTBF): $\frac{1}{\mathrm{FR}}$
Example: Distributed system with 1000 nodes, all of which are critical. MTBF of each node of 10,000 hours $\approx$ 60 weeks. What is the MTBF of the system?

- Given a system for which all sub-systems are critical:

$$\mathrm{FR}_{\mathrm{sys}} = \sum_{s \in \mathrm{sys}} \mathrm{FR}_s$$

Failures in a Distributed System

Failure Rate (FR): The number of failures per unit time
Mean Time Before Failure (MTBF): $\frac{1}{\mathrm{FR}}$
Example: Distributed system with 1000 nodes, all of which are critical. MTBF of each node of 10,000 hours $\approx$ 60 weeks. What is the MTBF of the system?

- Given a system for which all sub-systems are critical:

$$\mathrm{FR}_{\mathrm{sys}} = \sum_{s \in \mathrm{sys}} \mathrm{FR}_s$$

Hence:

$$\begin{aligned}
\mathrm{FR}_{\mathrm{node}} &= \frac{1}{10000} \text{ failures per hour} \\
\mathrm{FR}_{\mathrm{sys}} &= \sum_{\mathrm{node} \in \mathrm{sys}} \frac{1}{10000} \text{ failures per hour} \\
&= 1000/10000 = 0.1 \text{ failure per hour} \\
\mathrm{MTBF}_{\mathrm{sys}} &= 10 \text{ hours}
\end{aligned}$$

Distributed Computing Issues

- Parallel GPU computing is all about efficiency
- Distributed computing is all about uncertainty: Without explicit communication and computation, a node does not know
  - What time the other nodes think it is
  - What state the other nodes are in
  - Whether the other nodes have finished their computations
  - Whether it is safe to access a shared resource
  - Whether any of the nodes have failed (crashed)
  - Whether all the nodes are mutually waiting on each other (deadlock)
  - Whether it is safe to delete/destroy a shared resource when this node no longer needs it

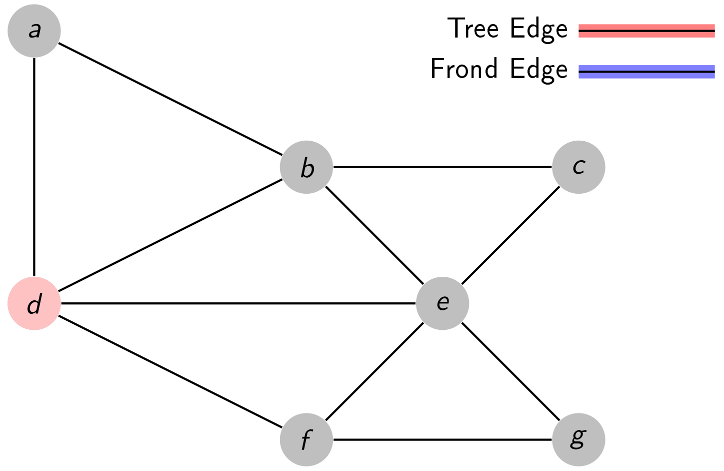
Spanning Tree of a Network

A *Spanning Tree*

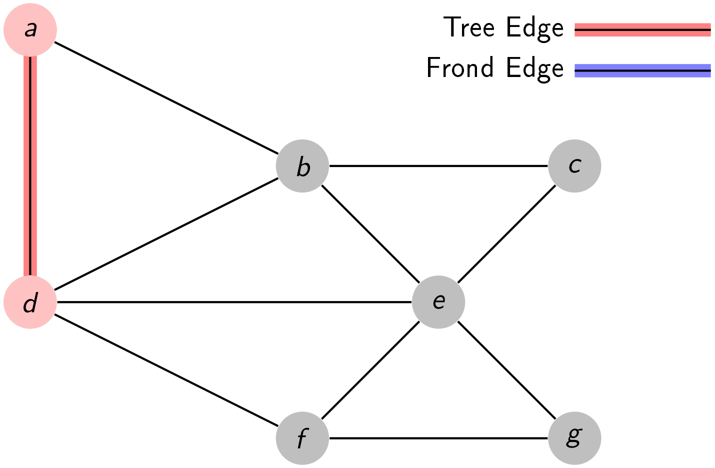
- Contains all the nodes of a network
- Its edges are a subset of the network edges
- Has no cycles
- Is undirected

Tree Edges: edges in the spanning tree
Frond Edges: edges in the network but not in the spanning tree
Sink Tree: a tree made by making all the edges of a spanning tree directed from *child* nodes to *parent* nodes terminating at the *root* node

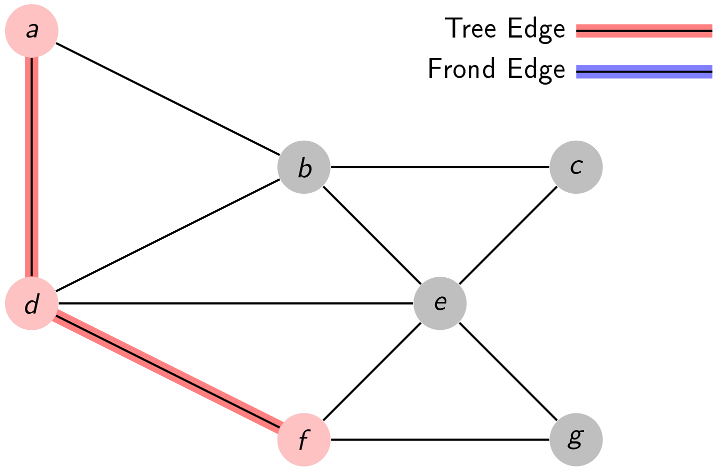
Spanning Tree Example



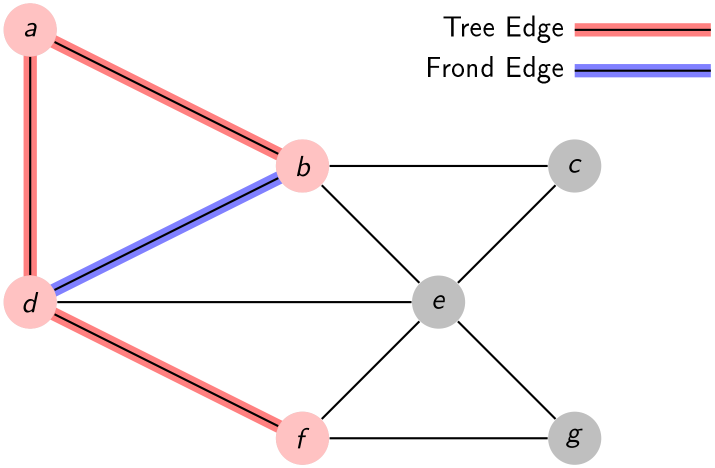
Spanning Tree Example



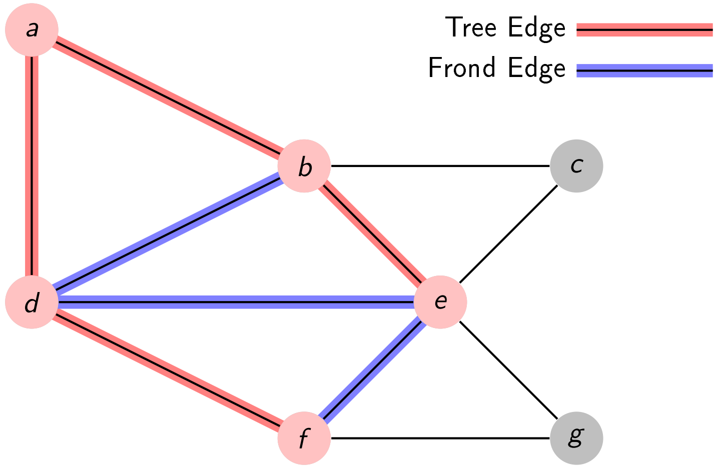
Spanning Tree Example



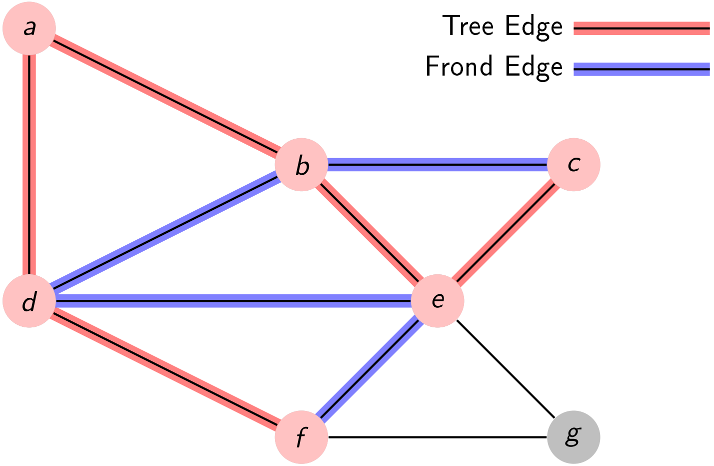
Spanning Tree Example



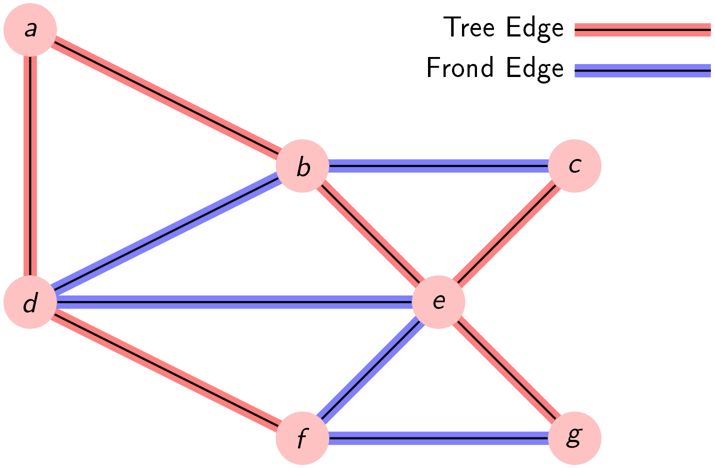
Spanning Tree Example



Spanning Tree Example



Spanning Tree Example



Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
    - Send to all neighbours recursively
        - Never terminates

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively
    - Never terminates
  - Send to all neighbours recursively, stop if already received

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
  - Send to all neighbours recursively
    - Never terminates
  - Send to all neighbours recursively, stop if already received
    - Wasteful: every node receives message multiple times

Why are Spanning Trees Important?

- Nodes only see immediate neighbours, not the whole topology
- How can a node distribute a message to the whole network?
    - Send to all neighbours recursively
        - Never terminates
    - Send to all neighbours recursively, stop if already received
        - Wasteful: every node receives message multiple times
    - Send to spanning tree root, which sends to all tree children recursively
- Actual algorithms are more sophisticated, but depend on spanning trees for efficiency and guarantees of correctness

Transition Systems

The behaviour of a *distributed algorithm* is given by a *transition system*:

- A set of *configurations* ($\lambda, \delta \in \mathcal{C}$): each configuration is a *global* state of a distributed algorithm. $\mathcal{C}$ is the set of all possible configurations of an algorithm.
- A binary *transition* relation on $\mathcal{C}$. Each transition $\lambda \to \delta$ is a step that changes the global state from one configuration to another.
- A set $\mathcal{I}$ of *initial* configurations representing the possible starting configurations for the system.

Transition System Definitions

A number of definitions apply to a transition system:

- A configuration is called *terminal* if there are no transitions out of that configuration.
- An *execution* of the distributed algorithm is a sequence of configurations beginning with an initial configuration, connected via transitions, and which is either infinite or ends in a terminal configuration.
- A configuration is *reachable* if there is a possible execution that includes that configuration.

States and Events

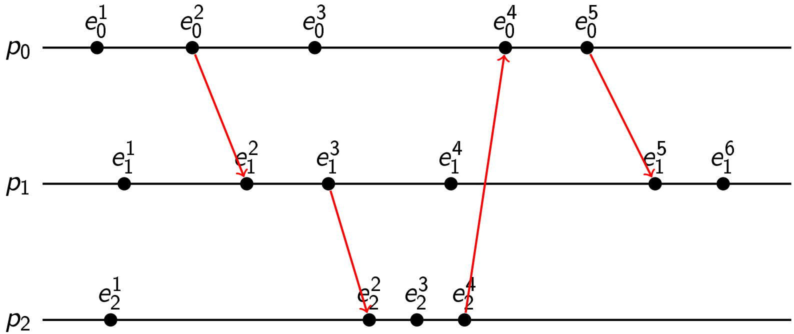
A configuration is composed of the set of local states of each node and the messages in transit between the nodes.
Transitions are connected with events:

- *internal*: some event internal to an individual process such as reading or writing a variable. Internal events affect only the local state of the process involved
- *send*: A message is sent from one process to another (causing a future *receive* event at the other process)
- *receive*: A message is received from another process

We will restrict ourselves to asynchronous systems: events never occur exactly simultaneously.

- A process is an *initiator* if its first event is an *internal* or *send* event.
- A *centralized* algorithm has precisely one initiator
- A *decentralized* algorithm can have multiple initiators

Diagram of an Execution



Assertions

Assertions are predicates on configurations of a distributed algorithm

- *Safety Property*: an assertion that is required to be true in every *reachable* configuration of every execution
  - Typically used to assert that nothing bad will happen
  - If the assertion fails, there will a *finite witness*: if something bad happens on an infinite run, it already happens on a finite prefix.
  - e.g. all results produced are correct
- *Liveness Property*: an assertion that is required to be true in some configuration in every execution after some point
  - Typically used to assert that something good will eventually happen
  - If the assertion fails, there will **NOT** be a *finite witness*: no matter what happens along a finite run, something good could still happen later.
  - e.g. a result *will* be produced

Partial vs Total Order

A *Total Order on A* defines a relationship $\leq$ on a set A such that for all elements a, b, c in A:

- $a \leq a$ (reflexivity)
- $a \leq b$ and $b \leq a \rightarrow a = b$ (antisymmetry)
- $a \leq b$ and $b \leq c \rightarrow a \leq c$ (transitivity)
- $a \leq b$ or $b \leq a$ (totality) note: totality implies reflexivity

For example:

- The usual order on integers: $1 \leq 2$
- Lexicographic (dictionary) order on strings: `"Jones"` $\leq$ `"Smith"`

A *Partial Order* has the first 3 properties above but does not have totality. For example:

- Prefix order on strings: `"abc"` $\leq$ `"abcd"`, `"abc"` $\not\leq$ `"xyz"`, `"xyz"` $\not\leq$ `"abc"`,
- Subset order on sets: $\{1,3\} \subseteq \{1,2,3\}, \{1,3\} \not\subseteq \{1,2\}, \{1,2\} \not\subseteq \{1,3\}$

Order

In each configuration, the possible events that can occur on different processes can occur in any order.

Causal Order ($\prec$) on events: $a \prec b$ if and only if a must occur before b in any execution:

- If a and b are events in the same process and a occurs before b, then $a \prec b$
- If a is a send and b is the corresponding receive, then $a \prec b$
- If $a \prec b$ and $b \prec c$ then $a \prec c$

Order

In each configuration, the possible events that can occur on different processes can occur in any order.

Causal Order ($\prec$) on events: $a \prec b$ if and only if a must occur before b in any execution:

- If a and b are events in the same process and a occurs before b, then $a \prec b$
- If a is a send and b is the corresponding receive, then $a \prec b$
- If $a \prec b$ and $b \prec c$ then $a \prec c$

Concurrent events: distinct events in an execution that are not causally related.

- With the above definition, concurrent events may not have occurred at the same physical time. However, a distributed system cannot tell the difference, so we can treat them as if they have.

Computations

If a sequence of configurations starting with λ and ending with δ in an execution is triggered by concurrent events, then changing the order of those events will change the intermediate configurations, but the new sequence will still start with λ and end with δ.

For example, given nodes a, b, c, d:

a sends to *b*	*a* sends to *b*
c sends to *d*	*b* receives from *a*
b receives from *a*	*c* sends to *d*
d receives from *c*	*d* receives from *c*

Executions are too specific: They distinguish between insignificant differences

We will be more concerned with sets of executions which differ only by reorderings of concurrent events

Computation: a set of executions equivalent up to permutations of sequences of concurrent events.

Logical Clocks

It is difficult and expensive to simulate a shared global real time clock in a distributed system. A logical clock is easy and efficient to maintain and is sufficient for many purposes that a global real time clock is used for.

A *logical clock* ($C(a)$) maps events to a partially ordered set (usually integers) such that

$$a \prec b \Rightarrow C(a) < C(b)$$

Implementation requires each process maintains data structures to support:

- A *local logical clock*: measures this process's own progress
- A *global logical clock*: this process's view of the logical global time, used to update the local logical clock to a globally consistent logical time.

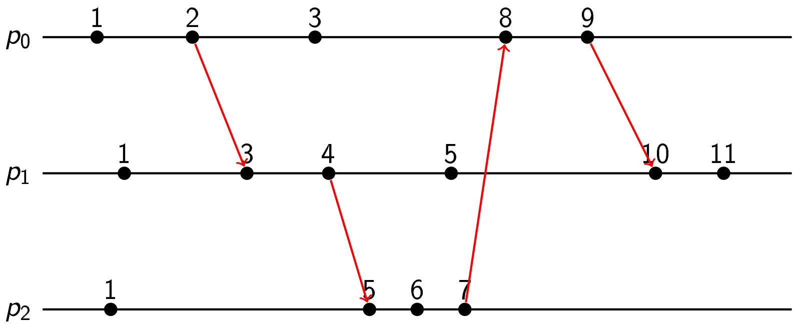
Lamport's Clock (1978)

Lamport's Clock, $\mathrm{LC}(a)$, is a distributed algorithm used to assign numbers to all events so that the natural order of the numbers respects the causal order of those events. It combines both the local and global logical clocks into a single integer variable.

Each process executes the following:

- Let a be an event and let k be the clock value of the previous event ($k = 0$ if there was no previous event)
  - If a is an internal or send event, $\mathrm{LC}(a) = k + 1$
  - If a is a receive event, and b the send event corresponding to a, then $\mathrm{LC}(a) = \max(k, \mathrm{LC}(b)) + 1$
    (The message carries the value $\mathrm{LC}(b)$ with it)

Example of Lamport's Clock



Properties of Lamport's Clock

Thus $\mathrm{LC}(a)$ assigns to a the length of a longest causality chain (there may be more than one) in the computation to the occurrence of a.
If $a \prec b$ then there is a causality chain from a to b and therefore $\mathrm{LC}(a) < \mathrm{LC}(b)$, so Lamport's clock *IS* a logical clock.

- Events with the same logical time are not causally related.
  - p_0^3 and p_1^2 are not causally related but both have the same LC time of 3.
- If needed we can impose a total ordering by breaking ties with the process identifier
  - Useful for liveness properties: e.g. serve requests according to total order

Properties of Lamport's Clock

- Lamport's Clock is consistent with causality but not *strongly consistent*:
  - $\mathrm{LC}(a) < \mathrm{LC}(b) \not\Rightarrow a \prec b$
  - Events which are not causally related may be ordered in LC time.
    - $\mathrm{LC}(p_0^3) = 3 < \mathrm{LC}(p_1^4) = 5 < \mathrm{LC}(p_2^3) = 6$, but all three events are concurrent.
  - This is because it uses one variable to store two different pieces of information: the local time and the global time. $\mathrm{LC}(p_2^2) = 5$ but now p_2 has no record that the latest time it has seen at p_1 is 4 or at p_0 is 2.
- Sometimes, we want a strongly consistent logical clock

Vector Clock

Each process i maintains a vector $v_i[0, ..., N-1]$ of integers of the same length as the number of processes in the system

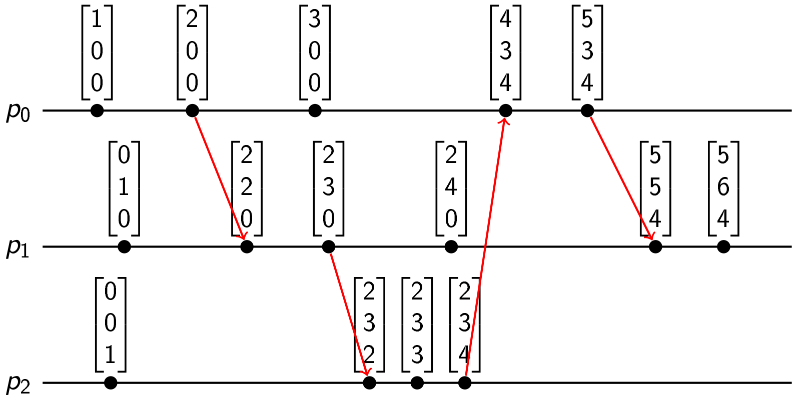
- $v_i[i]$ is the local logical clock of process i
- $v_i[j]$, where $i \neq j$, is process i's most recent knowledge of process j's local time

Each process i executes the following:

- Initialize all clocks to the 0 vector
- Let a be an event
    - If a is an internal or send event, $v_i[i] = v_i[i] + 1$
    - If a is a receive event, and m is the vector clock sent with the message, then

```
for (int j = 0 ; j < N ; j++)
    $v_i$[j] = max($v_i$[j], m[j])
$v_i$[i] += 1
```

Vector Clock Example



p_0
p_1
p_2
$\begin{bmatrix}1\\0\\0\end{bmatrix}$
$\begin{bmatrix}2\\0\\0\end{bmatrix}$
$\begin{bmatrix}3\\0\\0\end{bmatrix}$
$\begin{bmatrix}4\\3\\4\end{bmatrix}$
$\begin{bmatrix}5\\3\\4\end{bmatrix}$
$\begin{bmatrix}0\\1\\0\end{bmatrix}$
$\begin{bmatrix}2\\2\\0\end{bmatrix}$
$\begin{bmatrix}2\\3\\0\end{bmatrix}$
$\begin{bmatrix}2\\4\\0\end{bmatrix}$
$\begin{bmatrix}5\\5\\4\end{bmatrix}$
$\begin{bmatrix}5\\6\\4\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\end{bmatrix}$
$\begin{bmatrix}2\\3\\2\end{bmatrix}$
$\begin{bmatrix}2\\3\\3\end{bmatrix}$
$\begin{bmatrix}2\\3\\4\end{bmatrix}$

Properties of Vector Clocks

Define the order on vector clocks to be:

$$\begin{aligned} u = v &\iff \forall i \,.\, u[i] = v[i] \\ u \leq v &\iff \forall i \,.\, u[i] \leq v[i] \\ u < v &\iff u \leq v \wedge (\exists i \,.\, u[i] < v[i]) \\ u \| v &\iff u \not\leq v \wedge v \not\leq u \end{aligned}$$

Vector clocks are *strongly consistent* with causality:

$$a \prec b \iff \mathrm{VC}(a) < \mathrm{VC}(b)$$

Because they track causal dependencies exactly, vector clocks have many applications where Lamport clocks are insufficient:

- Distributed debugging
- Global breakpointing
- Consistency of checkpoints in optimistic recovery
- Implementations of causal distributed shared memory
- etc.

Centralised Mutual Exclusion in a Distributed System

Sometimes there are scarce resources that can only be used by one node at a time. In order to manage access to the resource, the nodes in a distributed system need to cooperate on their access. If we elect one node to be the coordinator, a solution is simple:

- To gain access to the resource, nodes must request access from the coordinator by sending a *request* message.
- If no other node is currently using the resource, the coordinator grants access by responding with a *grant* message, upon receipt of which the requestor can use the resource.
- If another node is currently using the resource, the coordinator queues the request and does not respond (yet), thus causing the requestor to wait.
- When a node finishes with the resource, it stops using it and sends a *release* message to the coordinator to tell it so.
- When the coordinator receives that release message, it takes the first request message off its queue (if any) and sends a *grant* message to the sender

Centralised Mutual Exclusion

Advantages:

- It works!
- Easy to implement
- Fair: access granted in order of requests
- No starvation: no node waits forever (assuming requesting nodes don't hold on to the resource forever)
- Only 3 messages per use of the resource

Disadvantages:

- Coordinator is a single point of failure
- In a large system, the coordinator can become a bottleneck

Distributed Mutual Exclusion (Ricart-Agrawala)

To request a resource, a node sends a message to all nodes requesting the resource and including its node number and the local logical clock value.
On receipt of a request, the recieving nodes reacts according to the state it is in:

- Not using and does not want to use the resource: it sends an OK message back to the sender
- Already has access: it does not reply but queues the request
- Wants the resource but does not have it yet: It compares the logical clock value of the message with that of its own request message. If the incoming message has the lower clock value, it sends back an OK message. Otherwise it queues the other message and sends nothing.

After sending out a request message, the sender waits for an OK from everyone else. When it has them, it can access the resource. When finished, it pops all requests off its queue and sends OK to each of them.

Distributed Mutual Exclusion (Ricart-Agrawala)

Advantages:

- It Works
- Fair
- No starvation

Disadvantages:

- $2(N-1)$ messages
- N points of failure
- Every node needs to keep track of all the nodes in the system

Distributed and Parallel Computing

Lecture 12

Alan P. Sexton

University of Birmingham

Spring 2018

Snapshots

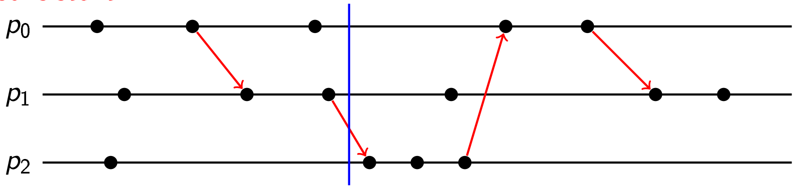
In a sequential program it is not unusual to save snapshots of the program state:

- Allows exiting and restarting to continue later on
- Guards against losing all the work after a system crash
- Allows returning to a previous good state if something bad happens (e.g. a poor choice is made, a character in a game dies, etc.)

Since the program is sequential, it is easy to access the state of the program and save a consistent snapshot of that state.

Distributed Snapshots

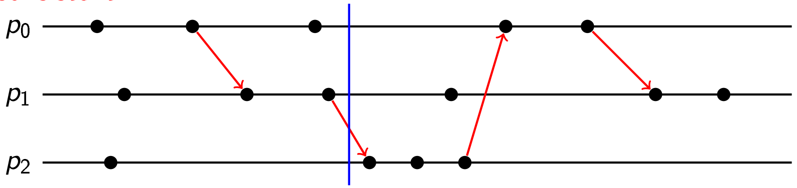
In a distributed program a snapshot must record the state of all the nodes and the state of all the channels (i.e. what messages are in transit on each channel) in such a way that the results are *consistent*:



Try: agree on a time in the future when all nodes and channels save their local states

Distributed Snapshots

In a distributed program a snapshot must record the state of all the nodes and the state of all the channels (i.e. what messages are in transit on each channel) in such a way that the results are *consistent*:

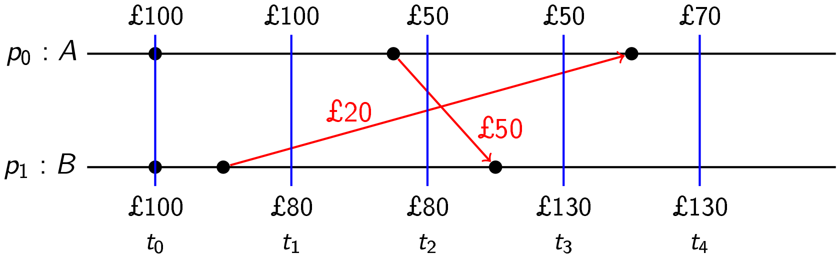


Try: agree on a time in the future when all nodes and channels save their local states

- **NOT POSSIBLE**: no global real time clock

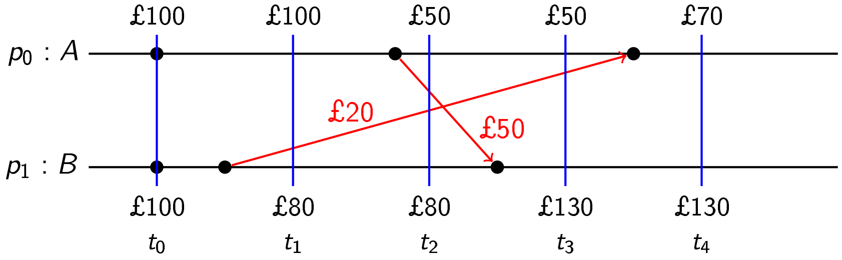
Consistency Example in Distributed Snapshots

A system has two bank nodes maintaining accounts A and B. Initially both accounts contain £100. Two transfers: £50 from A along channel c_{01} to B and £20 from B to A along channel c_{10}:



Consistency Example in Distributed Snapshots

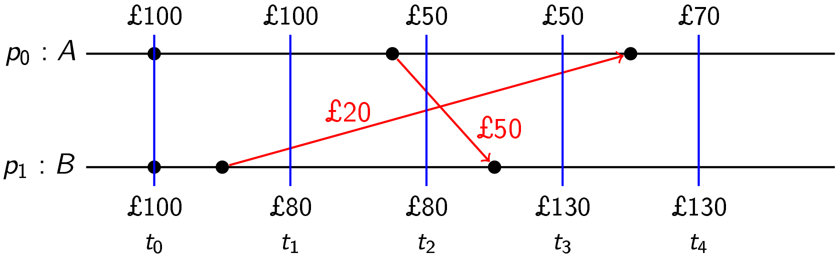
A system has two bank nodes maintaining accounts A and B. Initially both accounts contain £100. Two transfers: £50 from A along channel c_{01} to B and £20 from B to A along channel c_{10}:



- Snapshot taken on p_0 at time t_1, and on c_{01}, c_{10} and p_1 at time t_3 giving values £100, £0, £20 and £130 respectively $\Rightarrow$ £250 in the system — **NOT CONSISTENT**

Consistency Example in Distributed Snapshots

A system has two bank nodes maintaining accounts A and B. Initially both accounts contain £100. Two transfers: £50 from A along channel c_{01} to B and £20 from B to A along channel c_{10}:



- Snapshot taken on p_0 at time t_1, and on c_{01}, c_{10} and p_1 at time t_3 giving values £100, £0, £20 and £130 respectively $\Rightarrow$ £250 in the system — **NOT CONSISTENT**
- Snapshot taken on p_0 at time t_1, and on c_{01}, c_{10} and p_1 at time t_2 giving values £100, £50, £20 and £80 respectively $\Rightarrow$ £250 in the system — **NOT CONSISTENT**

Inconsistency in Distributed Snapshots

An inconsistent snapshot does not correspond to any possible configuration. That is:

- There is no possible execution of the system such that an inconsistent snapshot captures the true global state at any point in that execution

Recording of local snapshots must be coordinated correctly to ensure a consistent global snapshot.

If each process takes a local snapshot, then

- An event is *pre-snapshot* if it occurs in a process *before* the *local* snapshot in that process is taken
- Otherwise it is *post-snapshot*

Since the global snapshot includes all the local snapshots, all pre-snapshot events will be included in the global snapshot and all post-snapshot events will not be.

Consistency in Distributed Snapshots

A global distributed snapshot is *consistent* if:

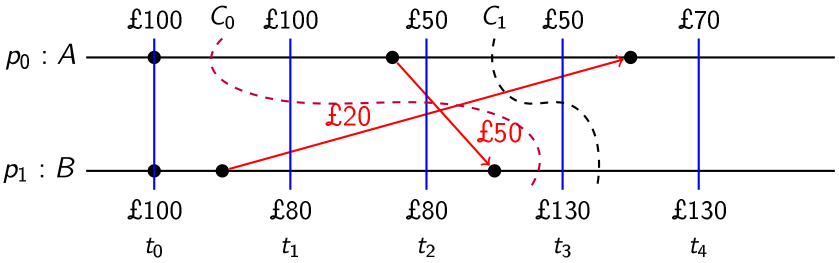
1. When a is pre-snapshot, $x \prec a \Rightarrow x$ is pre-snapshot
    - This ensures that the snapshot captures a point in a *computation*: i.e. a viable permutation of the concurrent events of the execution.
    - Note that x could be in a different process to a
2. A message is included in the channel state if and only if its sending is pre-snapshot and its reception is post-snapshot
    - This ensures that no message can be recorded as received in the snapshot if it is not also recorded as sent in the snapshot, and no message recorded as sent can be lost.

Thus, for message m with send and receive events s and r:

- if s is pre-snapshot, then
    - r is post-snapshot $\Rightarrow$ m is recorded in the channel state
    - r is pre-snapshot $\Rightarrow$ the effects of the message are recorded in the local node snapshots
- if s is post-snapshot $\Rightarrow$ r is also post-snapshot and m is not part of the global snapshot

Cuts

Visualise pre- and post-snapshots with *cuts*: a continuous line that cuts through each process line once:

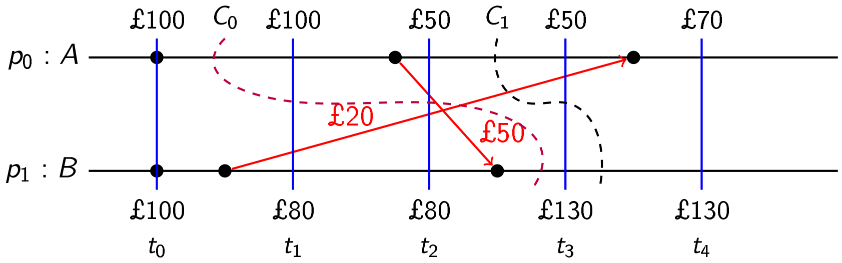


A consistent global state corresponds to a cut where every message received in the logical past of that cut was also sent in the logical past of that cut:

- C_0:

Cuts

Visualise pre- and post-snapshots with *cuts*: a continuous line that cuts through each process line once:

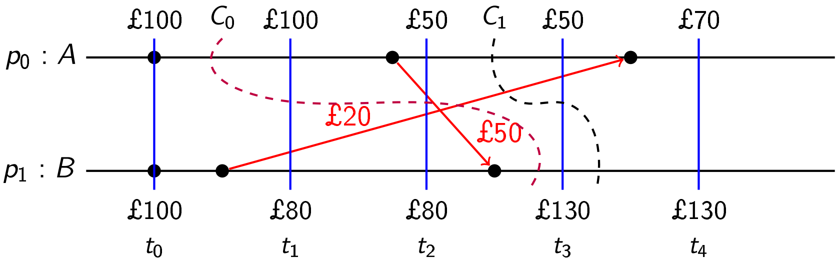


A consistent global state corresponds to a cut where every message received in the logical past of that cut was also sent in the logical past of that cut:

- C_0: The message from A to B was received in the logical past of the cut but sent in the logical future: **NOT CONSISTENT**
- C_1:

Cuts

Visualise pre- and post-snapshots with *cuts*: a continuous line that cuts through each process line once:



A consistent global state corresponds to a cut where every message received in the logical past of that cut was also sent in the logical past of that cut:

- C_0: The message from A to B was received in the logical past of the cut but sent in the logical future: **NOT CONSISTENT**
- C_1: All messages received in the logical past of the cut were sent in the logical past of the cut: **CONSISTENT**

Distributed Snapshot Algorithms

A simple distributed snapshot algorithm could come to a quiescent state, then, when all processes are quiescent, take all the local snapshots.

- This is inefficient: the whole system has to come to a stop to take a snapshot
- The system has to be designed so that quiescent states are possible — not always feasible

A better solution is to allow snapshots to be taken without interfering with the underlying processing.

- *Basic Algorithm*: the underlying distributed algorithm that we are taking a snapshot of. This algorithm uses *Basic Messages*
- *Control Algorithm*: the higher level distributed algorithm, in this case the global snapshot algorithm, implemented using *Control Messages*

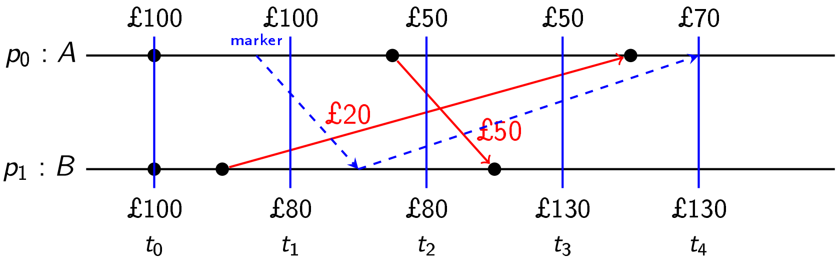
Chandy-Lamport Algorithm

NOTE: Applies to FIFO channel systems only

Idea: send control messages called *markers* along channels to separate pre- and post-snapshot events and trigger local snapshots.

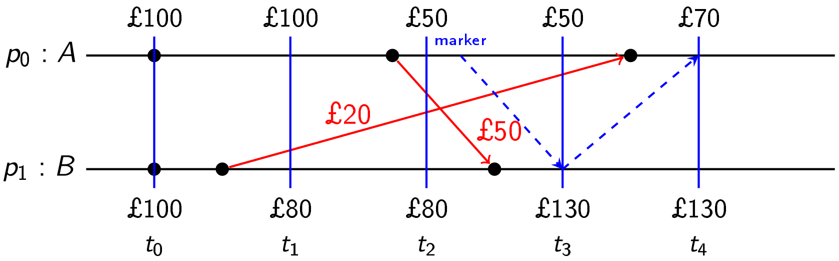
- Initiator takes a local snapshot and sends a marker through all outgoing channels
- On process p_j receiving a marker along channel c_{ij}
  - if p_j has not yet saved its local state
    - p_j saves its local state
    - p_j sets channel state c_{ij} to $\{\}$
    - p_j sends a marker through all outgoing channels
  - else
    - p_j records the state of channel c_{ij} as the set of all basic messages received on c_{ij} after it saved its local state and before it received the marker message from p_i
- Algorithm terminates at each process when it has received a marker along every incoming channel
- Note: a marker message is sent once along every channel in every allowed direction

Chandy-Lamport Example 1



- p_0 is the initiator, records local state $A = £100$, and sends a marker to p_1
- FIFO channels $\Rightarrow$ p_1 receives marker before £50 message
- p_1 records local state $B = £80$, sets channel state $c_{01} = \{\}$ and sends a marker to p_0
- FIFO channels $\Rightarrow$ p_0 receives marker after £20 message
- p_0 records channel state $c_{10} = \{£20\}$

Chandy-Lamport Example 2



- p_0 is the initiator, records local state $A = £50$ and sends a marker to p_1
- FIFO channels $\Rightarrow$ p_1 receives marker after £50 message
- p_1 records local state $B = £130$, sets channel state $c_{01} = \{\}$ and sends a marker to p_0
- FIFO channels $\Rightarrow$ p_0 receives marker after £20 message
- p_0 records channel state $c_{10} = \{£20\}$

Correctness of Chandy-Lamport, part 1

To prove: If $a \prec b$ and b is pre-snapshot, then a is pre-snapshot

- If a and b are in the same process, then trivially true
- Suppose a is a send and b is the corresponding receive in processes p and q respectively.
  - Since b is pre-snapshot, q has not yet received a marker when b occurs.
  - Since channels are FIFO, this means p has not yet sent a marker to q when a occurs.
  - Hence a is pre-snapshot.
- This argument extends transitively to causal chains of internal, send and receive events

Correctness of Chandy-Lamport, part 2

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Rightarrow$ direction: Assume m is in the channel state for c_{pq}:

- Since m is in the channel state of c_{pq}, the receive of m must occur after saving the local state of q, hence it is post-snapshot
- q must receive the m first and then the marker through c_{pq}
- Since channels are FIFO, p must send m first and then the marker through c_{pq}
- Since sending a marker is always immediately preceded by saving local state, the send of m must be pre-snapshot

Correctness of Chandy-Lamport, part 3

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Leftarrow$ direction: Assume the send of m is pre-snapshot and the receive of m is post-snapshot

- The send of m is pre-snapshot $\Rightarrow$ the local state of p is saved after the send of m
- Hence a marker is sent from p to q after the send of m
- Channels are FIFO $\Rightarrow$ the receive of the marker occurs after the receive of m
- The receive of m is post-snapshot $\Rightarrow$ it occurs after the local state of q is saved
- The receive of m is between the saving of the local state of q and the receive of the marker $\Rightarrow$ m is in the channel state of c_{pq}

Lai-Yang-Mattern Algorithm

NOTE: works on NON-FIFO channels

- Here, a marker message sent before/after a basic message may arrive after/before that same basic message respectively.
- Idea: rather than having separate marker messages, piggyback the markers on basic messages, and keep track of state by marking processes and messages with boolean flags (traditionally described as the colours white and red).
- The original Lai-Yang algorithm had no control messages at all, but required keeping a full history of all messages sent and received at each node
- The Mattern algorithm uses logical clocks directly and forces processes to wait until an agreed logical time.
- The Mattern paper describes a modification of Lai-Yang that replaces the need for a full history of messages with a single control message. This algorithm is what we will study.

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)
- A white process can save its local state at any time, but must save it no later than on receiving a red message and **BEFORE** processing that message

Lai-Yang-Mattern Algorithm

- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)
- A white process can save its local state at any time, but must save it no later than on receiving a red message and **BEFORE** processing that message
- Each process, on receiving the control message, immediately saves its local state if it hasn't done so already, and, knowing how many white messages it has received to date on each input channel, knows how many white messages to wait for along the corresponding channels.

Lai-Yang-Mattern Algorithm

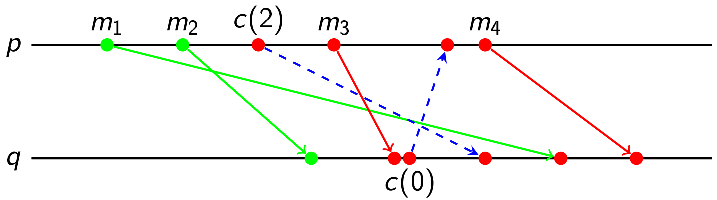
- Every process is initialised to white.
- When a process saves its local state, it turns red and sends a *control* message on all outgoing channels to say how many white messages in total it has sent down that channel.
- Every *basic* message is the same colour as the process that sends it (a white message is sent pre-snapshot, a red is sent post-snapshot)
- A white process can save its local state at any time, but must save it no later than on receiving a red message and **BEFORE** processing that message
- Each process, on receiving the control message, immediately saves its local state if it hasn't done so already, and, knowing how many white messages it has received to date on each input channel, knows how many white messages to wait for along the corresponding channels.
- After waiting for outstanding white messages, each process q computes channel state c_{pq} as the set of white messages it receives after it has saved its local state.

Lai-Yang-Mattern Algorithm details

The control message element to the algorithm is necessary for two reasons:

1. After saving its local state, a process may not have any basic messages to send for a long time (or ever!). The control message ensures that the snapshot algorithm proceeds
2. Without the control message, processes have no idea how long to wait for messages in transit, or how many such messages there might be.

Lai-Yang-Mattern Example



For clarity, white messages and nodes are drawn in green, control messages as dashed blue

1. p sends white m_1 and m_2, then saves local snapshot and sends control message to q with count of 2 white messages sent to q, then sends red m_3
2. q receives white m_2, then receives red m_3, saves local snapshot, sends control message to p with count of 0 white messages sent to p
3. q receives control message and waits to receive 1 white message (2 sent - 1 previously received), then sets channel c_{pq} state to $\{m_1\}$.
4. p receives control message, doesn't have to wait and sets channel c_{qp} state to $\{\}$

Correctness of Lai-Yang-Mattern, part 1

To prove: If $a \prec b$ and b is pre-snapshot, then a is pre-snapshot

- If a and b are in the same process, then trivially true
- Suppose a is a send and b is the corresponding receive in processes p and q respectively.
    - Since b is pre-snapshot, b is white.
    - Hence a must be white
    - Therefore a is pre-snapshot
- This argument extends transitively to causal chains of internal, send and receive events

Correctness of Lai-Yang-Mattern, part 2

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Rightarrow$ direction: Assume m is in the channel state for c_{pq}:

- Since m is in the channel state of c_{pq}, m must be white, hence the send of m is pre-snapshot
- Since m is in the channel state of c_{pq}, the receive of m must occur after the save of the local snapshot at q, hence the receive of m is post-snapshot

Correctness of Lai-Yang-Mattern, part 3

To prove: Basic message m from p to q is in the channel state of c_{pq} if and only if the send at p is pre-snapshot and the receive at q is post-snapshot

$\Leftarrow$ direction: Assume the send of m is pre-snapshot and the receive of m is post-snapshot

- The send of m is pre-snapshot $\Rightarrow$ the local state of p is saved after the send of m, hence m is white.
- The receive of m is post-snapshot $\Rightarrow$ the local state of q is saved before the receive of m
- Hence m is a white message received after the local state of q is saved hence will be in the channel state of c_{pq}

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
  - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
  - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
  - All messages sent after then will be red, but should be considered to be white to the next global snapshot

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
  - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
  - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
  - All messages sent after then will be red, but should be considered to be white to the next global snapshot
- Solution: instead of red/white flags, use counter k:
  - Initially processes and messages have $k = 0$
  - Instead of turning processes and messages red on saving local snapshot, increment k
  - Now, for first global snapshot, $k = 0 \Rightarrow$ white, $k = 1 \Rightarrow$ red
  - For second global snapshot, $k = 1 \Rightarrow$ white, $k = 2 \Rightarrow$ red

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
    - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
    - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
    - All messages sent after then will be red, but should be considered to be white to the next global snapshot
- Solution: instead of red/white flags, use counter k:
    - Initially processes and messages have $k = 0$
    - Instead of turning processes and messages red on saving local snapshot, increment k
    - Now, for first global snapshot, $k = 0 \Rightarrow$ white, $k = 1 \Rightarrow$ red
    - For second global snapshot, $k = 1 \Rightarrow$ white, $k = 2 \Rightarrow$ red
- What if two different processes start a global snapshot concurrently?

Lai-Yang-Mattern Multiple Snapshots

As described previously, after the first global snapshot, all messages are red. How can we handle a sequence of global snapshots?

- On saving local snapshot in a process:
  - All white messages received before then will be captured in the local snapshot and not relevant to the next global snapshot
  - All white messages received after then will be captured in the channel state and not relevant to the next global snapshot
  - All messages sent after then will be red, but should be considered to be white to the next global snapshot
- Solution: instead of red/white flags, use counter k:
  - Initially processes and messages have $k = 0$
  - Instead of turning processes and messages red on saving local snapshot, increment k
  - Now, for first global snapshot, $k = 0 \Rightarrow$ white, $k = 1 \Rightarrow$ red
  - For second global snapshot, $k = 1 \Rightarrow$ white, $k = 2 \Rightarrow$ red
- What if two different processes start a global snapshot concurrently?
  - Both processes increment k to the same value so both global snapshots will be the same single global snapshot

Distributed and Parallel Computing

Lecture 13

Alan P. Sexton

University of Birmingham

Spring 2018

Wave Algorithms

A *wave* algorithm sends requests through the whole network to gather information.

Applications include:

- Termination detection
- Routing
- Leader election
- Transaction commit voting in the case of network partitions

To be a wave algorithm, it must meed 3 conditions:

- It must be finite
- It contains one or more *decide* events
- For each decide event a, and process p, $\exists$ event b in p . $b \prec a$

Wave Algorithms

A *wave* algorithm sends requests through the whole network to gather information.

Applications include:

- Termination detection
- Routing
- Leader election
- Transaction commit voting in the case of network partitions

To be a wave algorithm, it must meed 3 conditions:

- It must be finite
- It contains one or more *decide* events
- For each decide event a, and process p, $\exists$ event b in p . $b \prec a$
  - i.e. every process must participate in each decide event

Traversal Algorithms

A *traversal* algorithm is a type of *wave* algorithm

- An *initiator* sends a *token* to visit each process in the network
- The *token* may collect and/or distribute information on the way
- The *token* eventually returns to the *initiator* with the accumulated information
- The *initiator* makes the *decision*.
- Note that a *token* can only be at one process at any one time

Traversal algorithms can be used to build a *spanning tree* of the network

- The initiator becomes the *root* of the spanning tree
- For every other node, its *parent* is the node from which it received the token for the first time

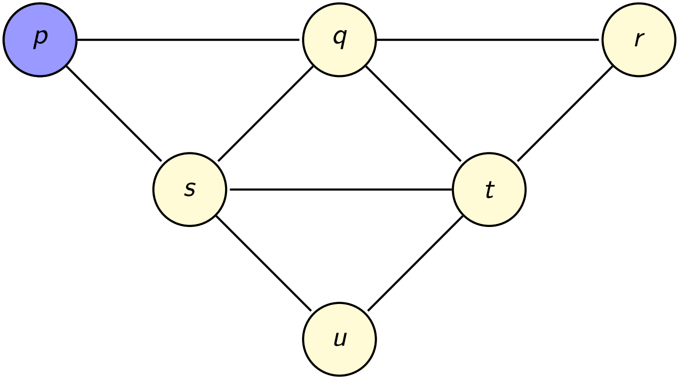
Tarry's Algorithm

Tarry's algorithm is a traversal algorithm for undirected networks. It is based on two rules:

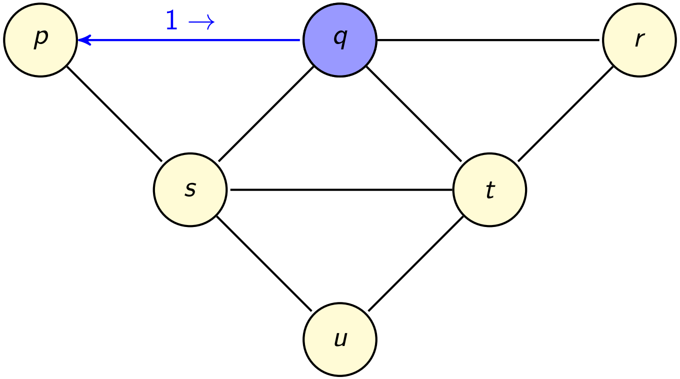
1. A process never forwards the token through the same channel twice
2. A process only forwards the token to its parent when there is no other option

These rules ensure that the token travels through every channel exactly twice, once in each direction, and ends up back at the initiator

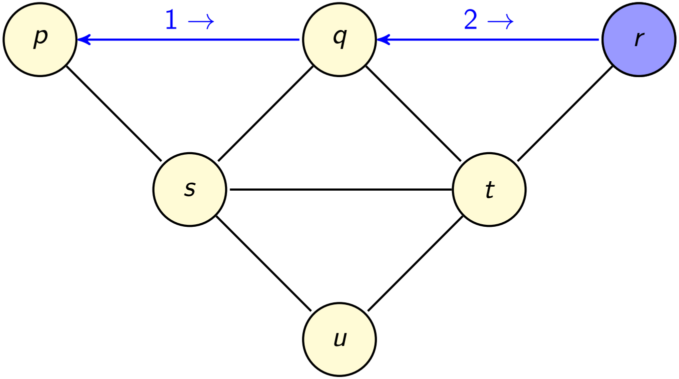
Example Execution of Tarry's Algorithm



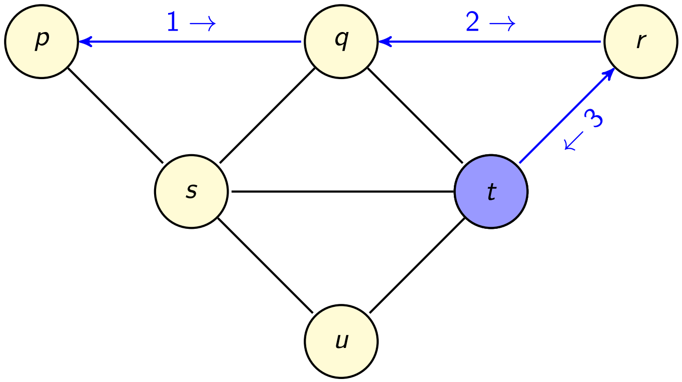
Example Execution of Tarry's Algorithm



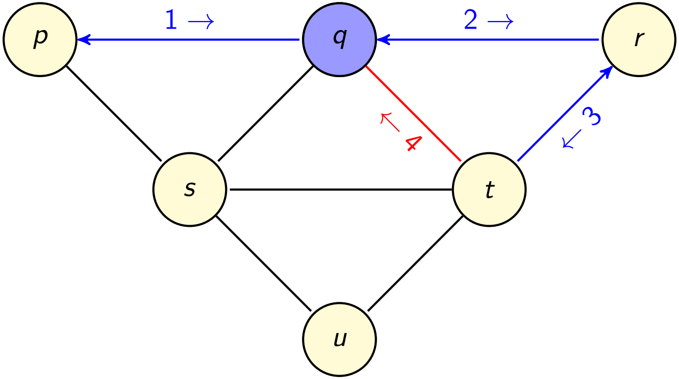
Example Execution of Tarry's Algorithm



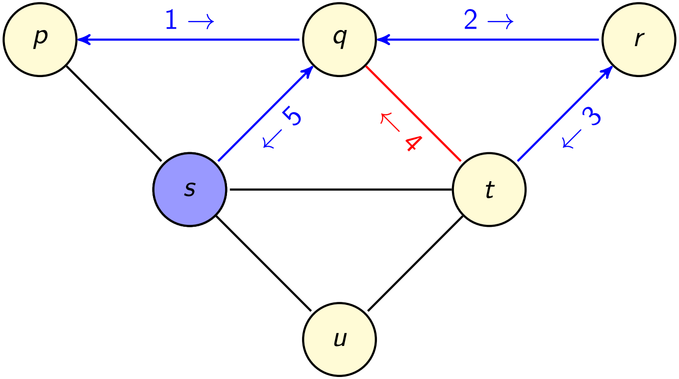
Example Execution of Tarry's Algorithm



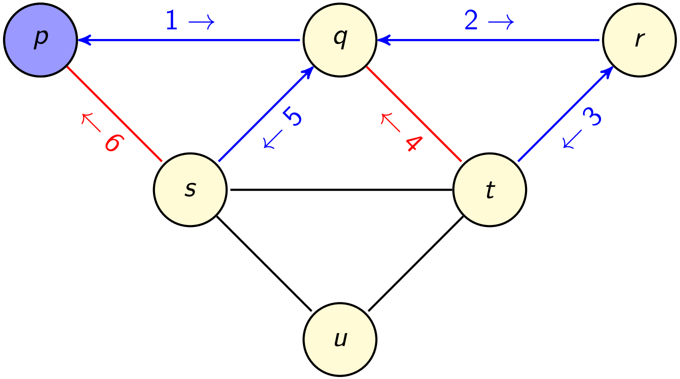
Example Execution of Tarry's Algorithm



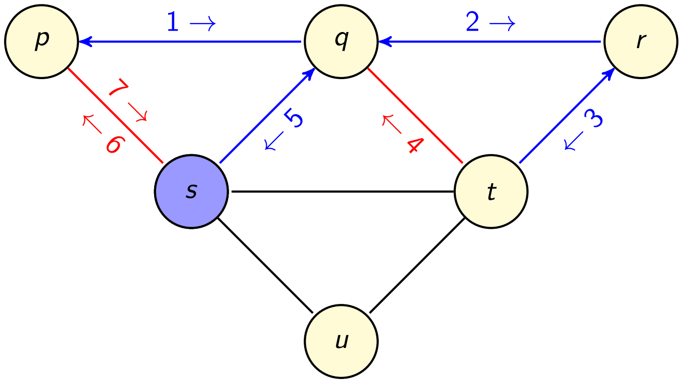
Example Execution of Tarry's Algorithm



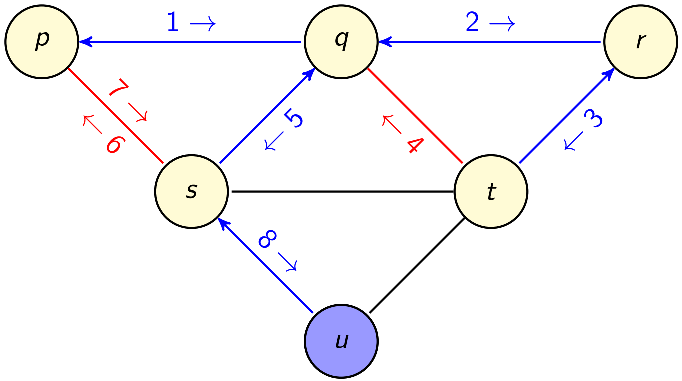
Example Execution of Tarry's Algorithm



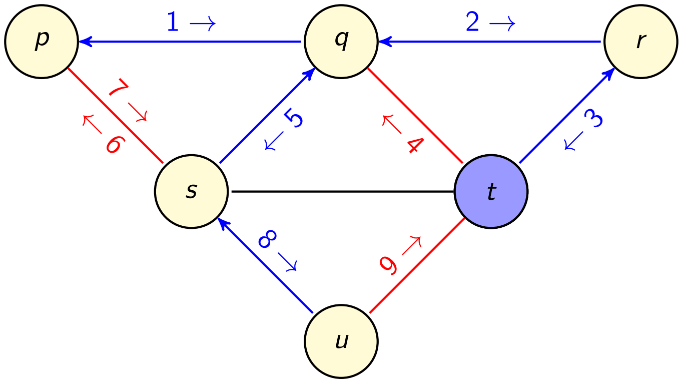
Example Execution of Tarry's Algorithm



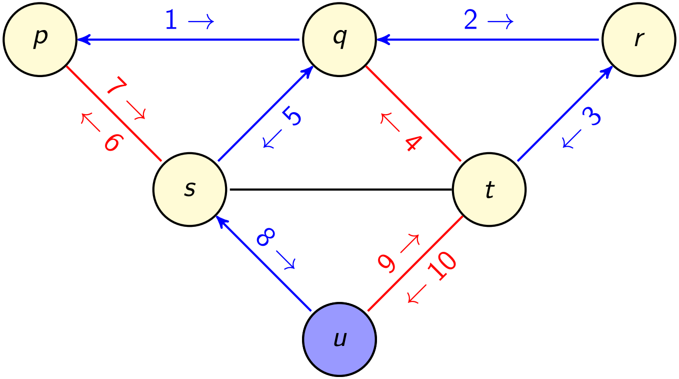
Example Execution of Tarry's Algorithm



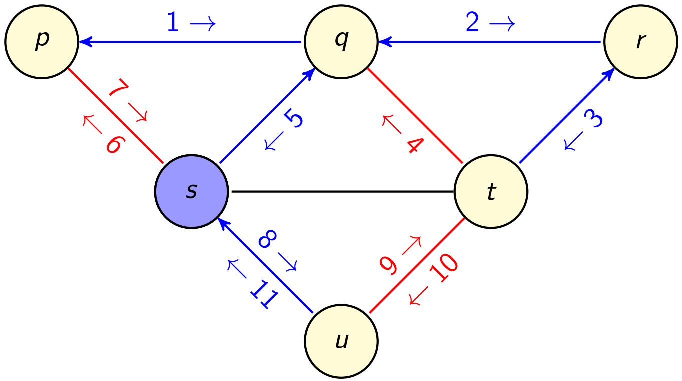
Example Execution of Tarry's Algorithm



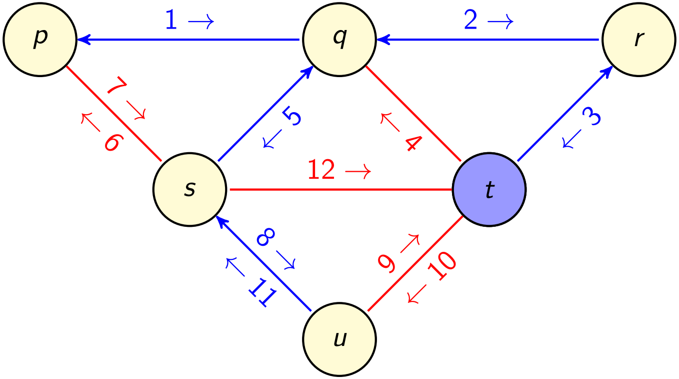
Example Execution of Tarry's Algorithm



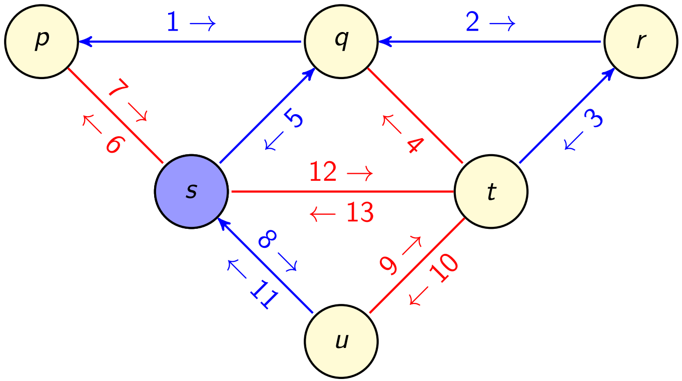
Example Execution of Tarry's Algorithm



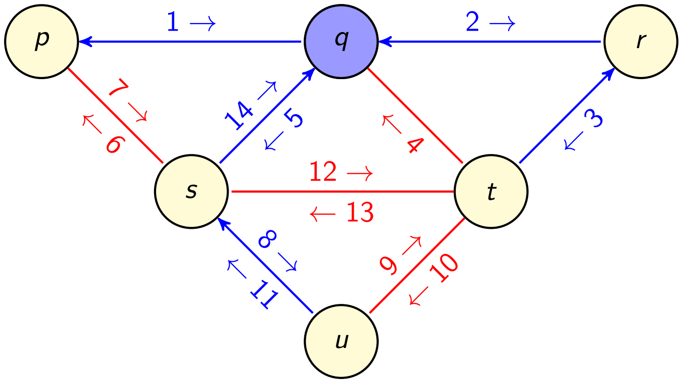
Example Execution of Tarry's Algorithm



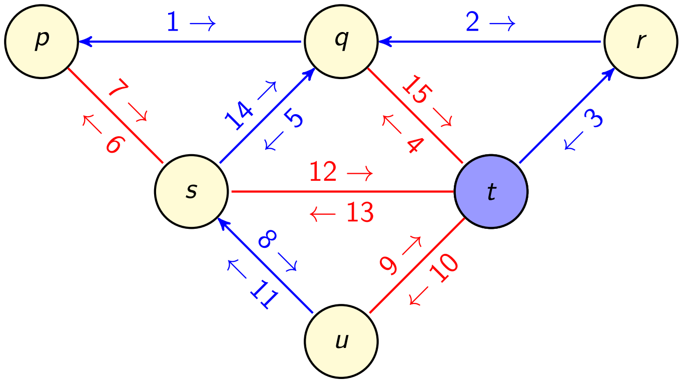
Example Execution of Tarry's Algorithm



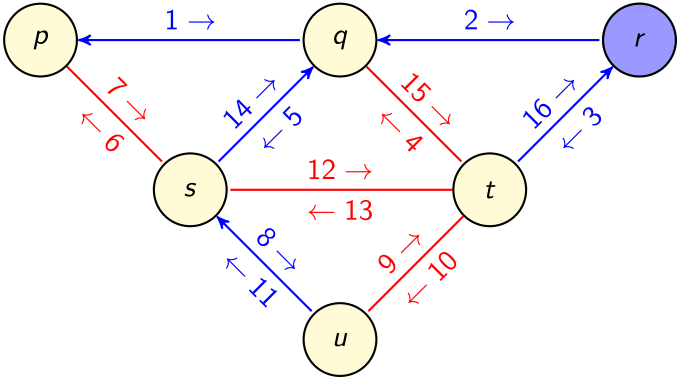
Example Execution of Tarry's Algorithm



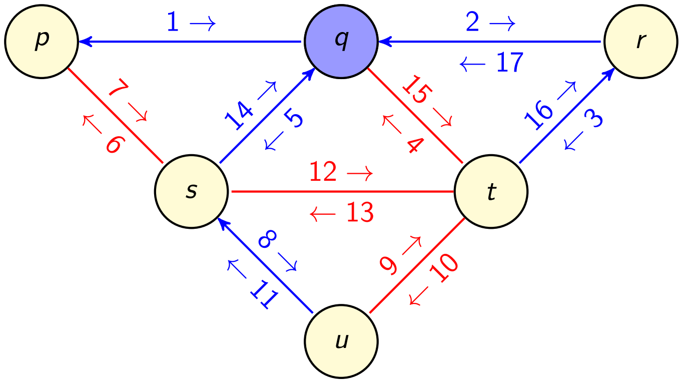
Example Execution of Tarry's Algorithm



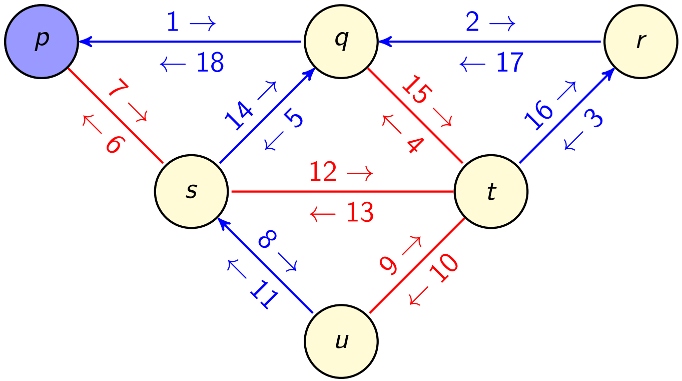
Example Execution of Tarry's Algorithm



Example Execution of Tarry's Algorithm



Example Execution of Tarry's Algorithm



Correctness of Tarry's Algorithm, part 2

To prove:

1. The token travels through each channel twice, once in each direction
2. The token ends up at the initiator

Correctness of Tarry's Algorithm, part 2

To prove:

1. The token travels through each channel twice, once in each direction
2. The token ends up at the initiator

Proof that the token ends up at the initiator:

- Rule 1 $\Rightarrow$ the token is never sent through the same channel in the same direction twice

Correctness of Tarry's Algorithm, part 2

To prove:

1. The token travels through each channel twice, once in each direction
2. The token ends up at the initiator

Proof that the token ends up at the initiator:

- Rule 1 $\Rightarrow$ the token is never sent through the same channel in the same direction twice
- Each time a non-initiator holds the token, it has received it one more time than it has sent it to at least one neighbour

Correctness of Tarry's Algorithm, part 2

To prove:

1. The token travels through each channel twice, once in each direction
2. The token ends up at the initiator

Proof that the token ends up at the initiator:

- Rule 1 $\Rightarrow$ the token is never sent through the same channel in the same direction twice
- Each time a non-initiator holds the token, it has received it one more time than it has sent it to at least one neighbour
- Hence there is still a channel into which it has not yet sent the token

Correctness of Tarry's Algorithm, part 2

To prove:

1. The token travels through each channel twice, once in each direction
2. The token ends up at the initiator

Proof that the token ends up at the initiator:

- Rule 1 $\Rightarrow$ the token is never sent through the same channel in the same direction twice
- Each time a non-initiator holds the token, it has received it one more time than it has sent it to at least one neighbour
- Hence there is still a channel into which it has not yet sent the token
- Therefore, by rule 1, it can send the token into this channel

Correctness of Tarry's Algorithm, part 2

To prove:

1. The token travels through each channel twice, once in each direction
2. The token ends up at the initiator

Proof that the token ends up at the initiator:

- Rule 1 $\Rightarrow$ the token is never sent through the same channel in the same direction twice
- Each time a non-initiator holds the token, it has received it one more time than it has sent it to at least one neighbour
- Hence there is still a channel into which it has not yet sent the token
- Therefore, by rule 1, it can send the token into this channel
- Hence, at the end of the algorithm, no non-initiator can be holding the token

Correctness of Tarry's Algorithm, part 2

To prove:

1. The token travels through each channel twice, once in each direction
2. The token ends up at the initiator

Proof that the token ends up at the initiator:

- Rule 1 $\Rightarrow$ the token is never sent through the same channel in the same direction twice
- Each time a non-initiator holds the token, it has received it one more time than it has sent it to at least one neighbour
- Hence there is still a channel into which it has not yet sent the token
- Therefore, by rule 1, it can send the token into this channel
- Hence, at the end of the algorithm, no non-initiator can be holding the token
- Hence, at the end of the algorithm, the token is with the initiator

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.
- Let p be the earliest visited (non-initiator) process for which such a channel exists and s be the parent of p

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.
- Let p be the earliest visited (non-initiator) process for which such a channel exists and s be the parent of p
- Therefore s was visited before p

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.
- Let p be the earliest visited (non-initiator) process for which such a channel exists and s be the parent of p
- Therefore s was visited before p
- Hence all channels of s were traversed in both directions

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.
- Let p be the earliest visited (non-initiator) process for which such a channel exists and s be the parent of p
- Therefore s was visited before p
- Hence all channels of s were traversed in both directions
- Hence p must have sent the token to its parent s

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.
- Let p be the earliest visited (non-initiator) process for which such a channel exists and s be the parent of p
- Therefore s was visited before p
- Hence all channels of s were traversed in both directions
- Hence p must have sent the token to its parent s
- But, by rule 2, p must have sent the token into all its other channels before sending it to its parent

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.
- Let p be the earliest visited (non-initiator) process for which such a channel exists and s be the parent of p
- Therefore s was visited before p
- Hence all channels of s were traversed in both directions
- Hence p must have sent the token to its parent s
- But, by rule 2, p must have sent the token into all its other channels before sending it to its parent
- Since p has sent and received the token an even number of times, it must have received it back through all its channels

Correctness of Tarry's Algorithm, part 2

Proof that the token travels through each channel twice, once in each direction:

- Assume that, at termination, some channel C_{pq} has not been traversed by the token in each direction.
- Let p be the earliest visited (non-initiator) process for which such a channel exists and s be the parent of p
- Therefore s was visited before p
- Hence all channels of s were traversed in both directions
- Hence p must have sent the token to its parent s
- But, by rule 2, p must have sent the token into all its other channels before sending it to its parent
- Since p has sent and received the token an even number of times, it must have received it back through all its channels
- Contradiction

More on Tarry's Algorithm

Performance:

- Number of messages:
- Time to complete:

More on Tarry's Algorithm

Performance:

- Number of messages: $2E$
- Time to complete:

More on Tarry's Algorithm

Performance:

- Number of messages: $2E$
- Time to complete: $2E$ time units

More on Tarry's Algorithm

Performance:

- Number of messages: $2E$
- Time to complete: $2E$ time units
- Note that this is a serial algorithm: There is only one token and only one process can send the token down one channel at a time — think of football players passing the ball

Tarry's algorithm and Depth First Search

In a depth first search, the token is forwarded to a process that has not yet held the token in preference to one that has.

A depth first spanning tree is one that could have been created by a depth-first search.

A depth first spanning tree will have its frond edges connecting nodes only to their ancestors or descendents in the spanning tree.

- Edges are frond edges if they connect a node to a node that has already been visited.
- In a depth first search, all nodes in a subtree are searched before any nodes in a sibling subtree
- Hence in a depth first search, frond edges will only connect ancestor-descendent pairs

Tarry's algorithm and Depth First Search

We can make Tarry's algorithm generate a depth-first spanning tree by adding an extra rule:

- When a process receives the token, it immediately sends it back through the same channel if allowed by rules 1 and 2.

Note: this does **NOT** make the search in Tarry's algorithm depth-first, but when it diverges off depth-first, it puts it back onto the depth-first track before any more parent-child edges are added to the spanning tree.

There is no extra cost to this change.

Benefits of Depth-First Search

If a depth-first spanning tree is created by the modified Tarry's algorithm:

- We can optimise the algorithm by letting the token carry the information of all processes that have held it:
- Avoid sending the token along frond edges at the cost of extra memory required in the token
- Messages only travel along spanning tree edges, so $2E$ messages reduced to $2N-2$
- Time complexity reduced from $2E$ to $2N-2$ time units.
- Bit complexity increases from $O(1)$ to $O(N \log N)$, where $O(\log N)$ bits are needed to represent the process identifiers.

Distributed and Parallel Computing

Lecture 14

Alan P. Sexton

University of Birmingham

Spring 2018

The Echo Algorithm

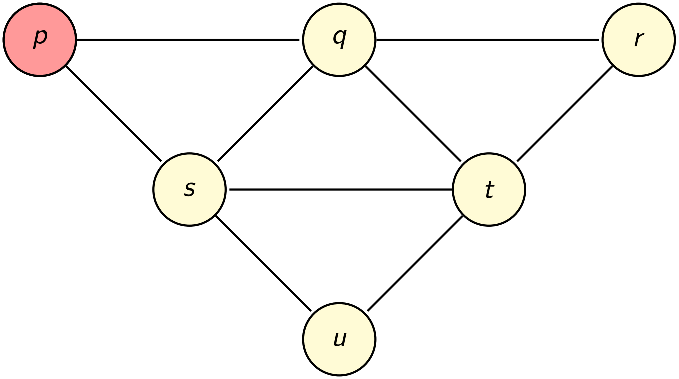
A wave, but not a traversal algorithm (so no *tokens* involved), *Echo* is a centralized algorithm (i.e. one initiator only) for undirected networks.

- Initiator sends message to all neighbours
- When a non-initiator *first* receives a message
  - It makes the sender its parent
  - It sends a message to all neighbours except its parent
- When a non-initiator has received messages from *all* its neigbours
  - It sends a message to its parent
- When the initiator has received messages from all its neighbours, it *decides* and the algorithm terminates

This algorithm builds a spanning tree

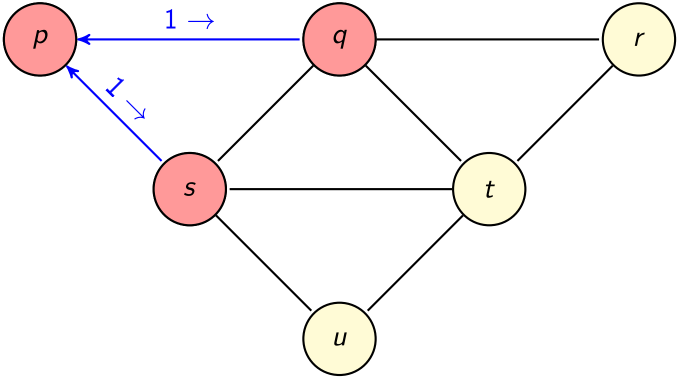
Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



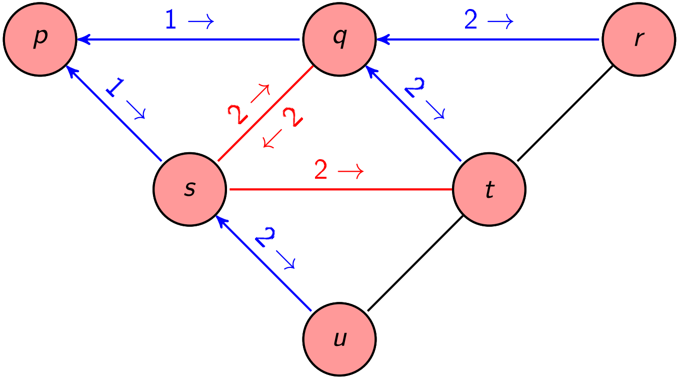
Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



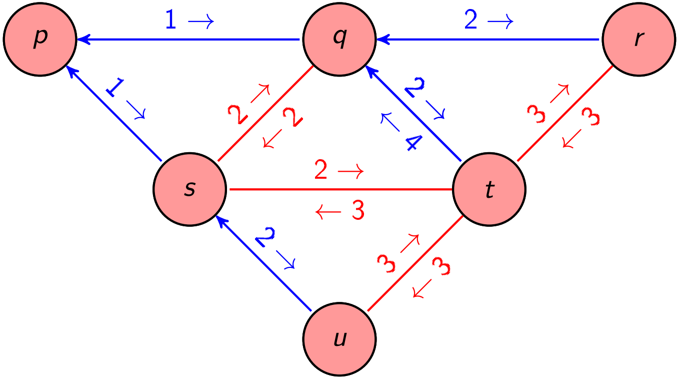
Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



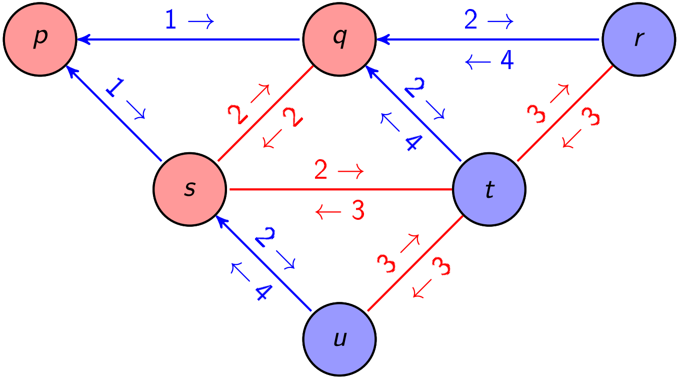
Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



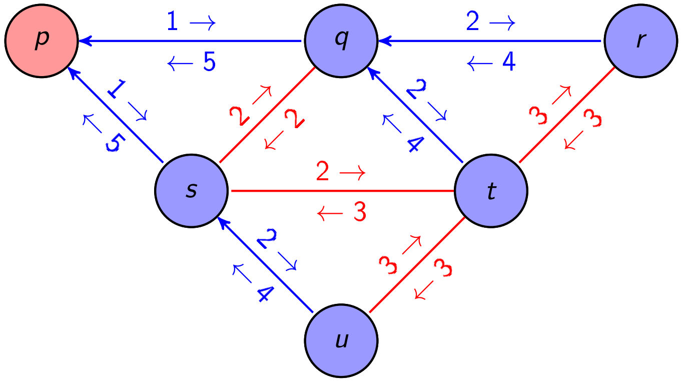
Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



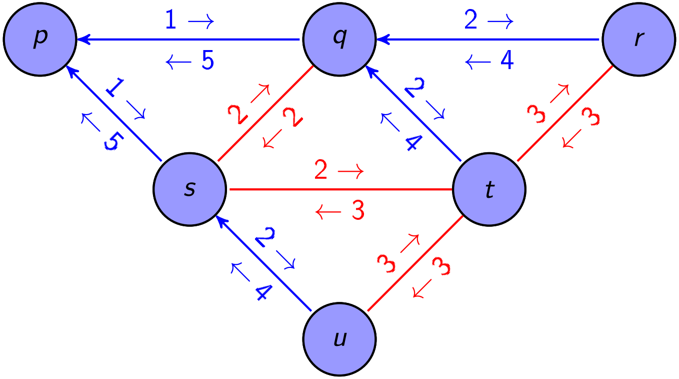
Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



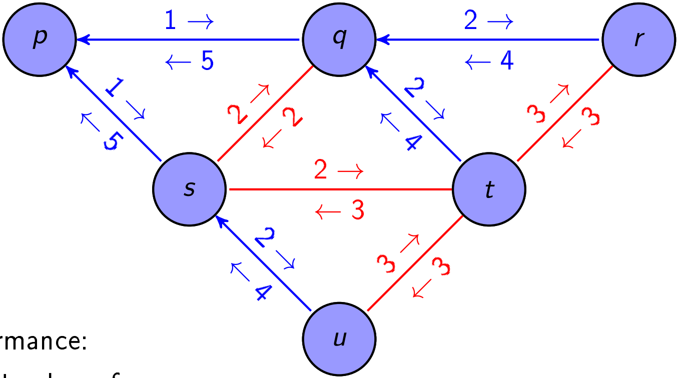
Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.

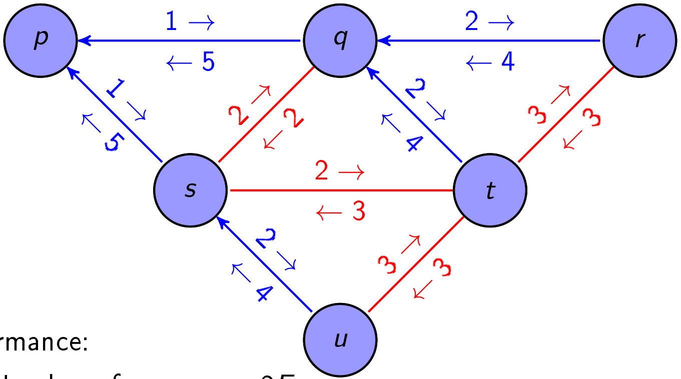


Performance:

- Number of messages:
- Worst case time to complete:

Example Execution of the Echo Algorithm

Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.

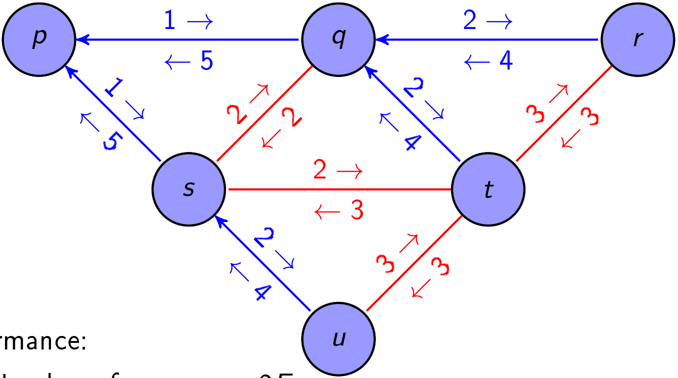


Performance:

- Number of messages: $2E$
- Worst case time to complete:

Example Execution of the Echo Algorithm

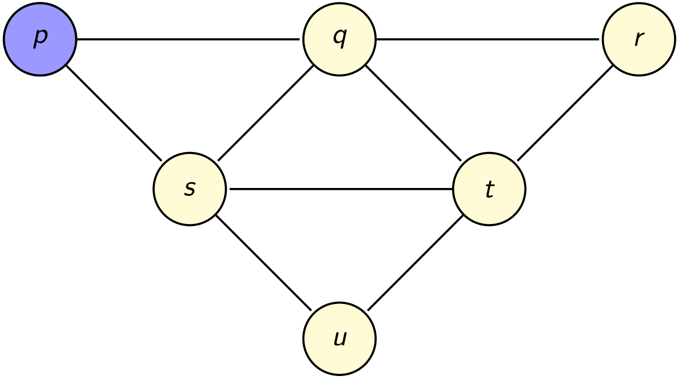
Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



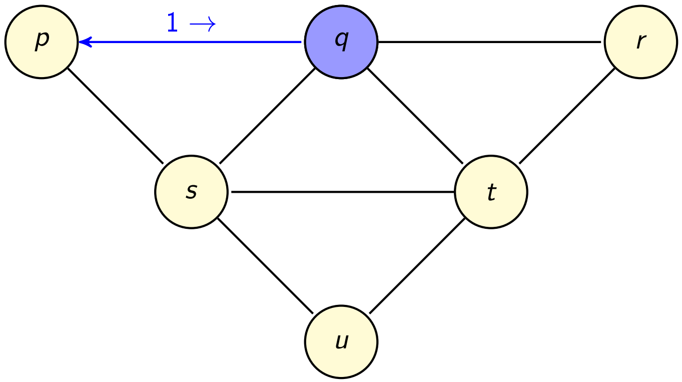
Performance:

- Number of messages: $2E$
- Worst case time to complete: $2N - 2$ time units

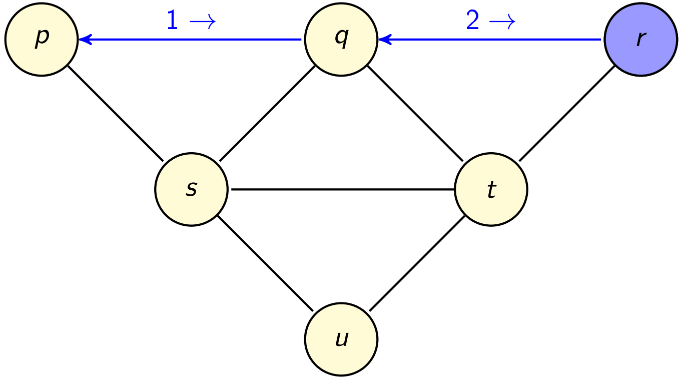
Tarry Algorithm considered as an Echo Algorithm



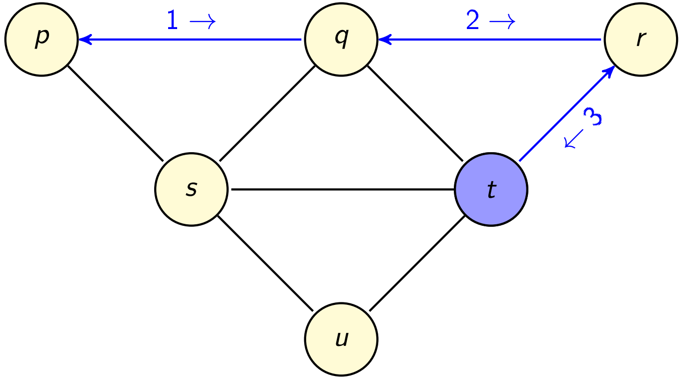
Tarry Algorithm considered as an Echo Algorithm



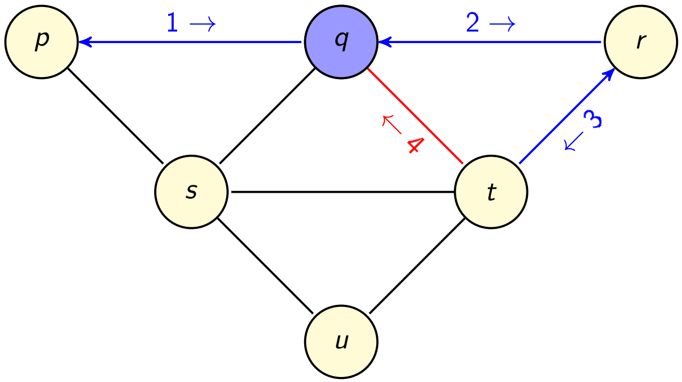
Tarry Algorithm considered as an Echo Algorithm



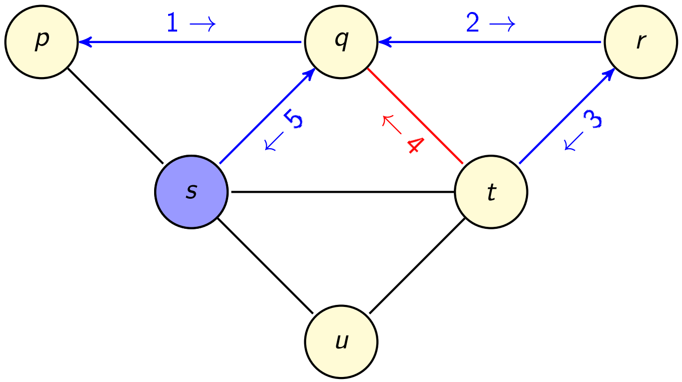
Tarry Algorithm considered as an Echo Algorithm



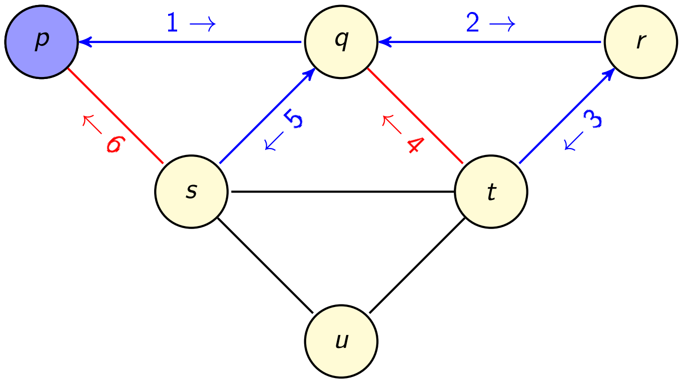
Tarry Algorithm considered as an Echo Algorithm



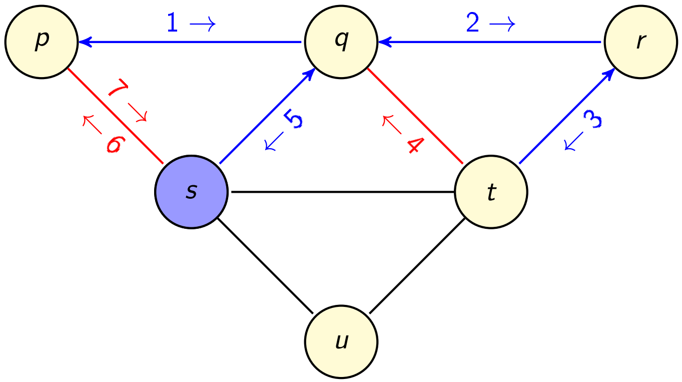
Tarry Algorithm considered as an Echo Algorithm



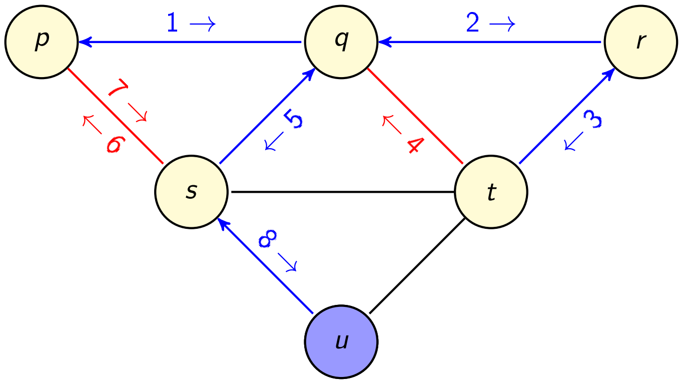
Tarry Algorithm considered as an Echo Algorithm



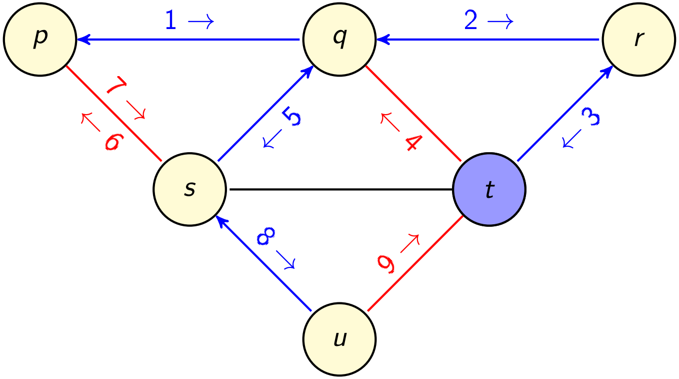
Tarry Algorithm considered as an Echo Algorithm



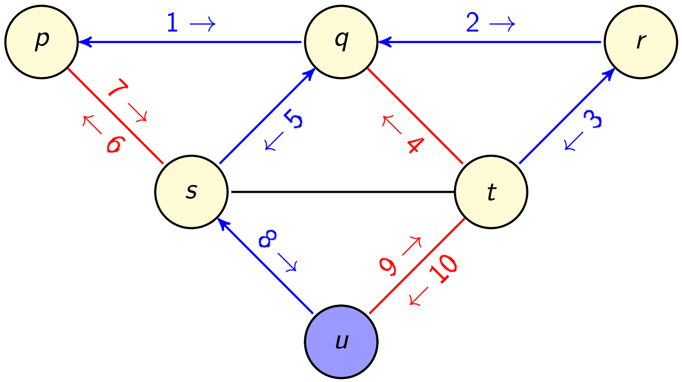
Tarry Algorithm considered as an Echo Algorithm



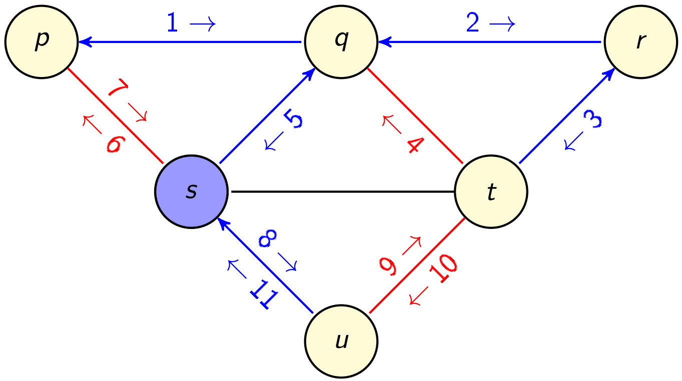
Tarry Algorithm considered as an Echo Algorithm



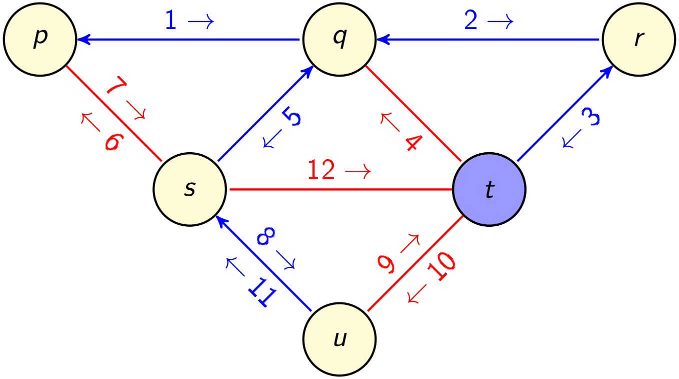
Tarry Algorithm considered as an Echo Algorithm



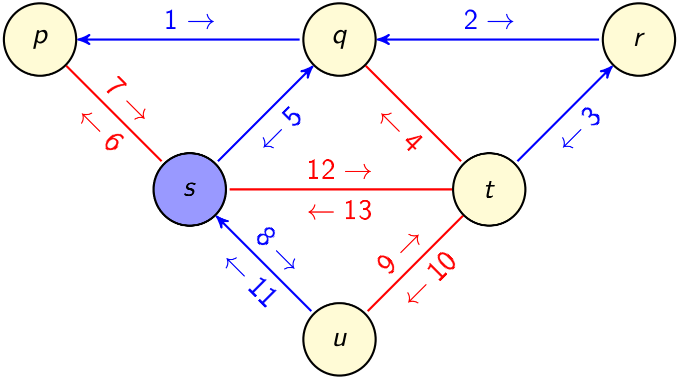
Tarry Algorithm considered as an Echo Algorithm



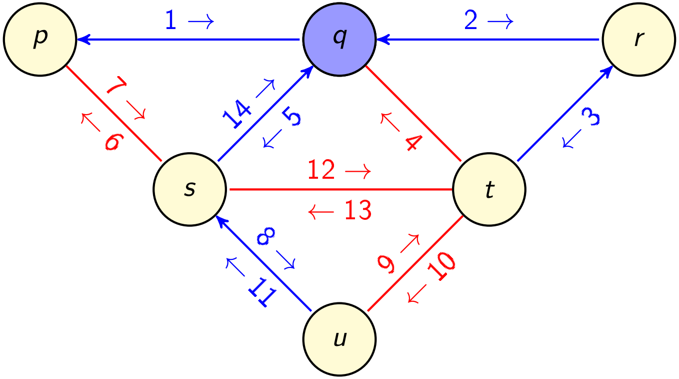
Tarry Algorithm considered as an Echo Algorithm



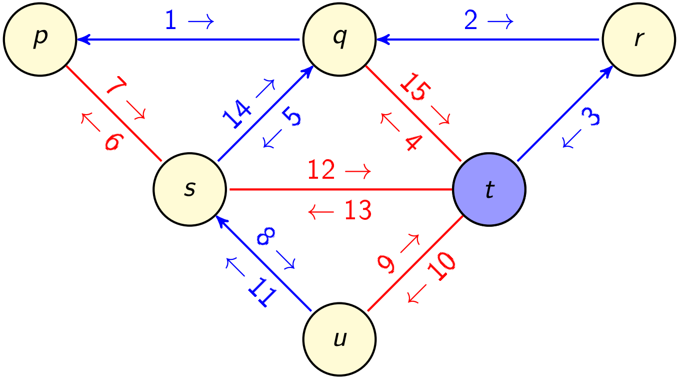
Tarry Algorithm considered as an Echo Algorithm



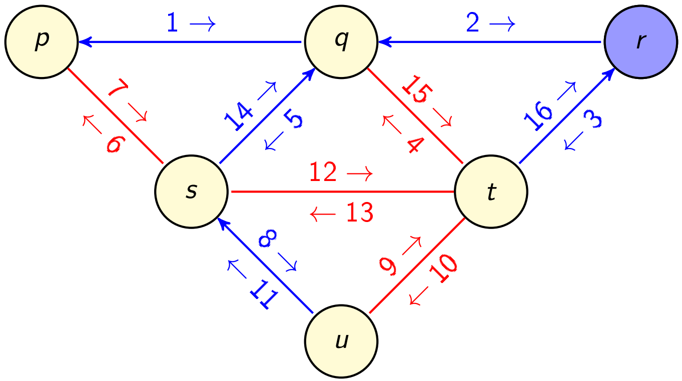
Tarry Algorithm considered as an Echo Algorithm



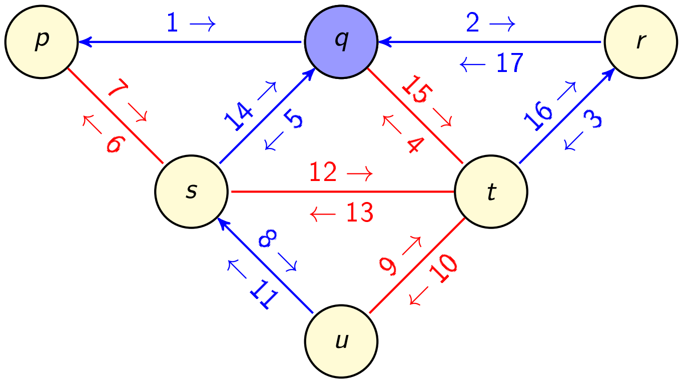
Tarry Algorithm considered as an Echo Algorithm



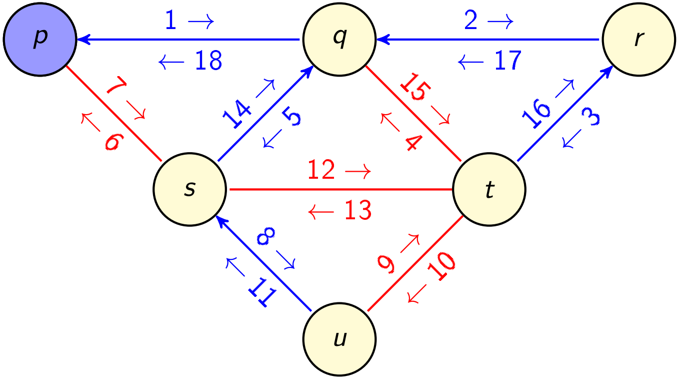
Tarry Algorithm considered as an Echo Algorithm



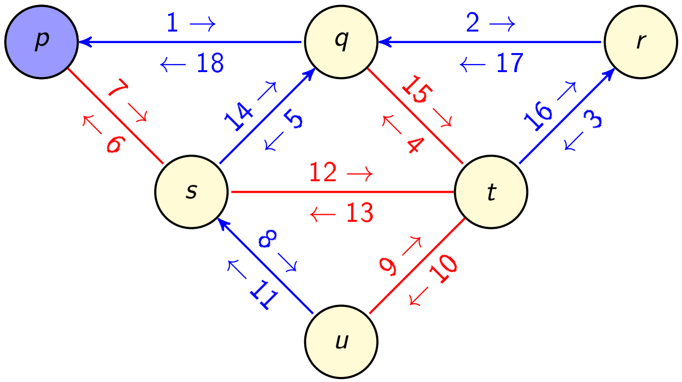
Tarry Algorithm considered as an Echo Algorithm



Tarry Algorithm considered as an Echo Algorithm

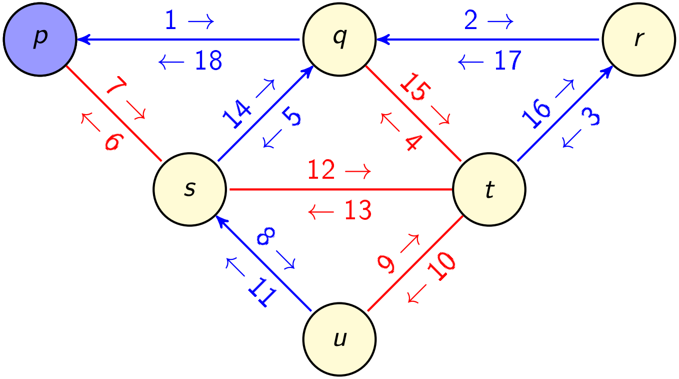


Tarry Algorithm considered as an Echo Algorithm



- Every message trace of the execution of a Tarry algorithm is a possible message trace of the execution of an Echo algorithm

Tarry Algorithm considered as an Echo Algorithm



- Every message trace of the execution of a Tarry algorithm is a possible message trace of the execution of an Echo algorithm
- Exercise: Find a message trace of an Echo algorithm that is not the message trace of a Tarry algorithm

Deadlocks

A *deadlock* is what the situation is called if a process is stuck in an infinite wait.

- A *communication deadlock* is where there is a cycle of processes each waiting for the next process in the cycle to send it a message
  - Process p will not send any message until it receives one from q, which will only send it after it receives a message from r, which will only send it after it receives a message from p.
- A *resouce deadlock* is where there is a cycle of processes each waiting for a resource held by another process in the cycle
  - Process p wants to transfer money from account A to account B, has obtained a lock on A and is waiting to obtain a lock on B
  - Process q wants to transfer money from account B to account A, has obtained a lock on B and is waiting to obtain a lock on A

Dealing with Deadlocks

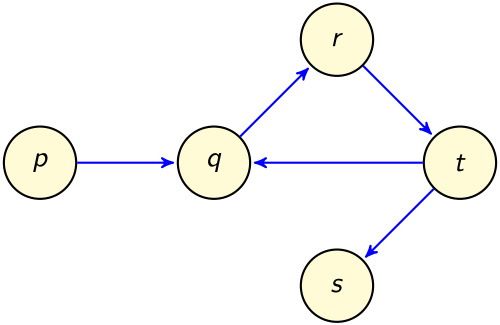
There are essentially 3 strategies for dealing with deadlocks:

- Make deadlocks impossible: Define protocols to ensure that a deadlock can never happen
  - e.g. require a process to obtain all necessary resources simultaneously before proceeding (if any cannot be obtained, release all held resources and try again)
  - Usually impractical and inefficient in distributed systems
- Avoid deadlocks
  - Only obtain resources if the global state ensures it is safe
  - Usually impractical in distributed systems
- Detect deadlocks
  - Detect deadlocks when they occur and break the chain by forcing one or more processes to fail, release their resources and recover

Waits-For Graph (WFG)

Model the deadlock state with a *Waits-For Graph (WFG)*

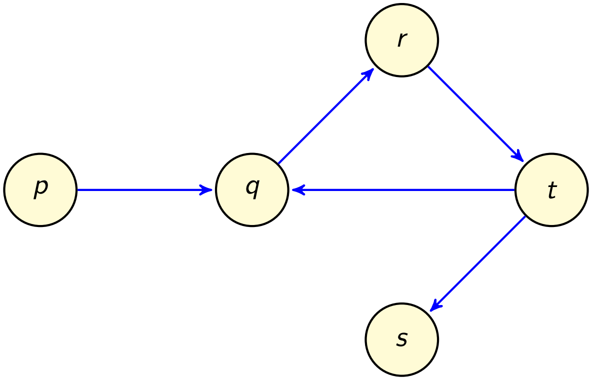
- Directed graph
- Nodes are processes
- Edge from p to q if p is blocked waiting for q to respond or release some resource
- In the simplest model, a cycle in the WFG $\Rightarrow$ deadlock



Deadlock Models

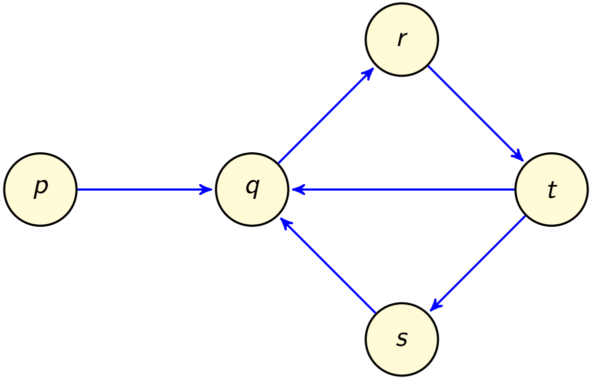
- *Single-resource* model:
  - A process can have at most one outstanding request for one (unit of a) resource
  - Cycle in WFG $\Rightarrow$ deadlock
  - Simplest model
- *AND* model
  - Each process can request multiple resources simultaneously and *all* requested resources must be supplied to unblock
  - Each node is called an *AND node*
  - Cycle in WFG $\Rightarrow$ deadlock
- *OR* model
  - Each process can request multiple resources simultaneously and *one* requested resource must be supplied to unblock
  - Each node is called an *OR node*
  - Cycle in WFG $\not\Rightarrow$ deadlock
  - *Knot* in WFG $\Rightarrow$ deadlock
  - a *knot* is a set of vertices such that every vertex u reachable from a knot vertex v can also reach v

OR Model Example



Cycle but no knot $\Rightarrow$ no deadlock for OR-model

OR Model Example



Knot $\{q, r, s, t\} \Rightarrow$ deadlock for OR-model

More Deadlock Models

- *AND-OR* model
  - Generalisation of both *AND* and *OR* models
  - Each process can request any combination of *AND* and *OR* requests simultaneously and *a set satisfying the requested condition* of requested resources must be supplied to unblock e.g. x and (y or z)
  - No simple graph structure whose presence identifies deadlock
- $\binom{p}{q}$ or q-out-of-p model
  - Equivalent to *AND-OR* model
  - Each process can request from p resources simultaneously and q from these resources must be supplied to unblock
  - An *AND* node is equivalent to a p-out-of-p node and an *OR* node is equivalent ot a 1-out-of-p node.
- *Unrestricted* model
  - No assumptions other than stability of deadlock (once it occurs, it does not release without action to break the deadlock being taken)
  - Only of theoretical interest because of high overhead

Deadlock Detection

Two problems need to be solved to implement deadlock detection in a distributed system:

1. How to maintain the WFG?
2. How to find cycles or knots in the WFG?

To be correct, a deadlock detection algorithm must guarantee:

1. *Progress:* All existing deadlocks must be found in finite time
    - Once all wait-for edges of a deadlock have formed in the WFG, the algorithm should be able to detect the deadlock without having to wait further
2. *Safety:* The algorithm should not report deadlocks that do not exist (*phantom deadlocks*)
    - No global memory or shared clocks $\Rightarrow$ processes have only partial knowledge of global state
    - $\Rightarrow$ processes may detect cycles in the WFG that never truly existed but which are composed of different parts that did exist in the system at different times
    - Main source of errors in published papers on deadlock detection

Distributed and Parallel Computing

Lecture 15

Alan P. Sexton

University of Birmingham

Spring 2018

Operations on a WFG

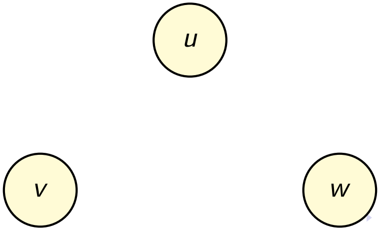
Conceptually, a WFG works as follows:

- There is one node v for each process v in the network
- A node v can be *active* or *blocked*
- An *active* node can make n-out-of-m requests of other nodes (and then becomes blocked) or grant requests to other nodes
- A *blocked* node can not make or grant requests but can become *active* if a sufficient number of its outstanding requests are granted
- When a *blocked* node with an outstanding n-out-of-m request has received n grants, it *purges* the remaining $m - n$ outstanding requests by informing the nodes involved that it no longer needs the resource requested

Mutually Exclusive Resources and the WFG

Mutually exclusive use of a resource requires a particular pattern on a WFG:

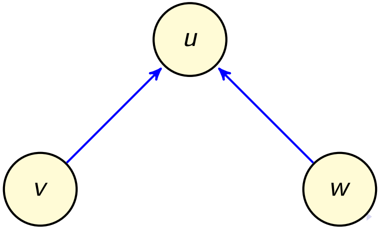
- u manages a mutually exclusive resource for v and w:



Mutually Exclusive Resources and the WFG

Mutually exclusive use of a resource requires a particular pattern on a WFG:

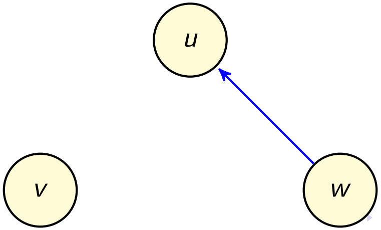
- u manages a mutually exclusive resource for v and w:
- v and w request the resource from u, so edges $v \to u$ and $w \to u$ are added to the WFG



Mutually Exclusive Resources and the WFG

Mutually exclusive use of a resource requires a particular pattern on a WFG:

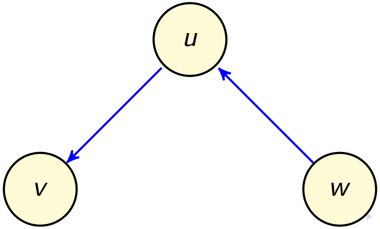
- u manages a mutually exclusive resource for v and w:
- v and w request the resource from u, so edges $v \to u$ and $w \to u$ are added to the WFG
- Let u grants v's request first: then the edge $v \to u$ is removed
  - Problem: v holds the resource but nothing stops u granting w's request immediately



Mutually Exclusive Resources and the WFG

Mutually exclusive use of a resource requires a particular pattern on a WFG:

- u manages a mutually exclusive resource for v and w:
- v and w request the resource from u, so edges $v \to u$ and $w \to u$ are added to the WFG
- Let u grants v's request first: then the edge $v \to u$ is removed
  - Problem: v holds the resource but nothing stops u granting w's request immediately
- Solution: granting v's request introduces a *new* dependency of u on v, which is modelled by adding an edge $u \to v$ for u to get back the resource from v



Representation of a Distributed WFG

We do not wish to centralise deadlock detection by getting the full global WFG onto a single node. Instead:

- Each node retains information about its local part of the WFG
- Distributed deadlock detection algorithm invoked by initiator:
    - Detects whether this node is deadlocked
    - Triggered after timeout when this node suspects it might be deadlocked

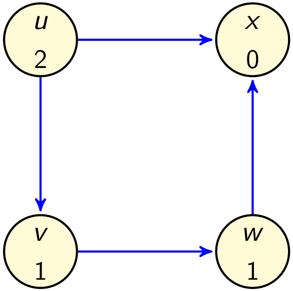
At each node u, have a number of variables:

- $\mathtt{OUT}_u$: The set of nodes u has sent a request to that are not yet granted or purged
- $\mathtt{IN}_u$: The set of nodes u has received a request from that are not yet granted or purged
- n_u: The number of grants that u currently needs to receive until it becomes unblocked. Note that $0 \le n_u \le |\mathtt{OUT}_u|$ and $n_u = 0 \Leftrightarrow \mathtt{OUT}_u = \{\}$

Detecting deadlock idea

Rather than searching for cycles or knots in the WFG (NP-hard), we simulate granting of grantable requests in the WFG until no more requests can be granted and see if the initiator node is unblocked

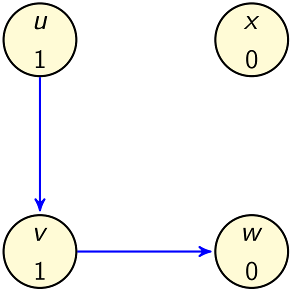
- Initiator is u



Detecting deadlock idea

Rather than searching for cycles or knots in the WFG (NP-hard), we simulate granting of grantable requests in the WFG until no more requests can be granted and see if the initiator node is unblocked

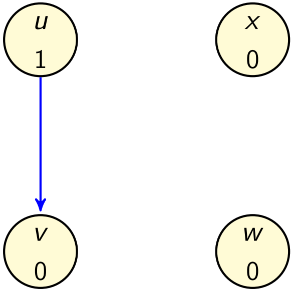
- Initiator is u
- x grants requests from u and w



Detecting deadlock idea

Rather than searching for cycles or knots in the WFG (NP-hard), we simulate granting of grantable requests in the WFG until no more requests can be granted and see if the initiator node is unblocked

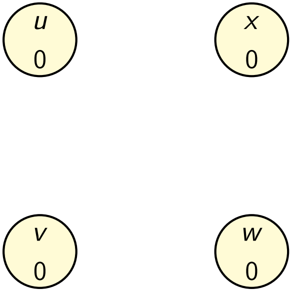
- Initiator is u
- x grants requests from u and w
- w grants request from v



Detecting deadlock idea

Rather than searching for cycles or knots in the WFG (NP-hard), we simulate granting of grantable requests in the WFG until no more requests can be granted and see if the initiator node is unblocked

- Initiator is u
- x grants requests from u and w
- w grants request from v
- v grants request from u



Bracha-Toueg Deadlock Detection Algorithm

Bracha-Toueg [1984] presented 3 variants of an algorithm for distributed deadlock detection:

1. On a network with instant messages where the base algorithm is static during deadlock detection
   - i.e. no requests, grants or purges occuring in parallel with deadlock detection
2. On a network with time delays in message delivery where the base algorithm is static
3. On a network with time delays in message delivery and the base algorithm is dynamic

Bracha-Toueg Deadlock Detection Algorithm

- Variation 1 requires that the $\mathtt{IN}_u$, $\mathtt{OUT}_u$ and n_u on each node u be pre-calculated from the local state and channel states of a globally consistent snapshot
- Variation 2 relaxes the need for the channel states to be used
- Variation 3 relaxes the need for a global snapshot to be pre-calculated
  - i.e. it integrates taking the snapshot with the deadlock detection

We will consider only the first variation, where we first apply a global snapshot algorithm which calculates $\mathtt{IN}_u$, $\mathtt{OUT}_u$ and n_u on each node u from the local and channel states.

Bracha-Toueg idea

Starting with the globally consistent $\mathtt{IN}_u$, $\mathtt{OUT}_u$ and n_u on each node u, execute 2 nested *Echo* algorithms to (virtually) construct a spanning tree of spanning trees.

- The first spanning tree is rooted at the initiator and traversed using Notify/Done messages
- The nested spanning trees are rooted at each active node, traversed using Grant/Ack messages and propagate all grants through the WFG

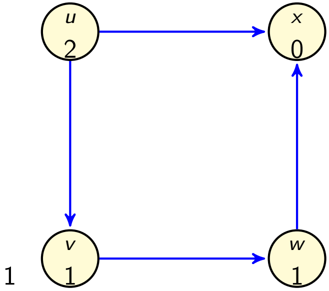
The Bracha-Toueg Algorithm

Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls Notify()

```
Notify_u():
    notified_u ← True
    for all w ∈ OUT_u, send ⟨NOTIFY⟩ to w
    if n_u = 0, then Grant_u()
    for all w ∈ OUT_u, await ⟨DONE⟩ from w
Grant_u():
    free_u ← True
    for all w ∈ IN_u send ⟨GRANT⟩ to w
    for all w ∈ IN_u await ⟨ACK⟩ from w
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    u sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If n_u > 0, then
        n_u ← n_u − 1
        if n_u = 0, then Grant_u()
    u sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is True at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.



The Bracha-Toueg Algorithm

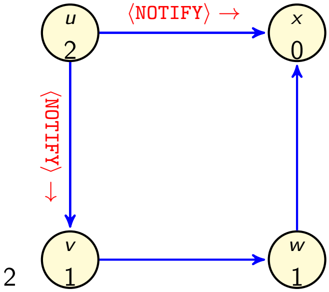
Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls $\texttt{Notify}()$

```
Notify_u():
    notified_u ← True
    for all $w \in \texttt{OUT}_u$, send ⟨NOTIFY⟩ to $w$
    if $n_u = 0$, then Grant_u()
    for all $w \in \texttt{OUT}_u$, await ⟨DONE⟩ from $w$
Grant_u():
    free_u ← True
    for all $w \in \texttt{IN}_u$ send ⟨GRANT⟩ to $w$
    for all $w \in \texttt{IN}_u$ await ⟨ACK⟩ from $w$
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    $u$ sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If $n_u > 0$, then
        $n_u \leftarrow n_u - 1$
        if $n_u = 0$, then Grant_u()
    $u$ sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is $\texttt{True}$ at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.

await ⟨DONE⟩
from v, x



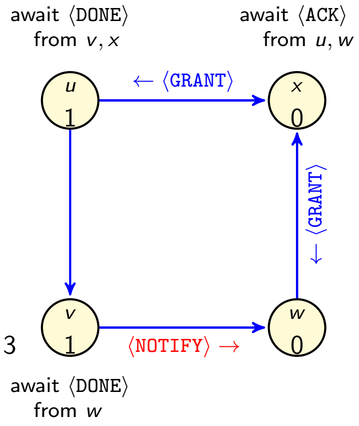
The Bracha-Toueg Algorithm

Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls $\texttt{Notify}()$

```
Notify_u():
    notified_u ← True
    for all $w \in \texttt{OUT}_u$, send ⟨NOTIFY⟩ to $w$
    if $n_u = 0$, then Grant_u()
    for all $w \in \texttt{OUT}_u$, await ⟨DONE⟩ from $w$
Grant_u():
    free_u ← True
    for all $w \in \texttt{IN}_u$ send ⟨GRANT⟩ to $w$
    for all $w \in \texttt{IN}_u$ await ⟨ACK⟩ from $w$
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    $u$ sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If $n_u > 0$, then
        $n_u \leftarrow n_u - 1$
        if $n_u = 0$, then Grant_u()
    $u$ sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is True at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.



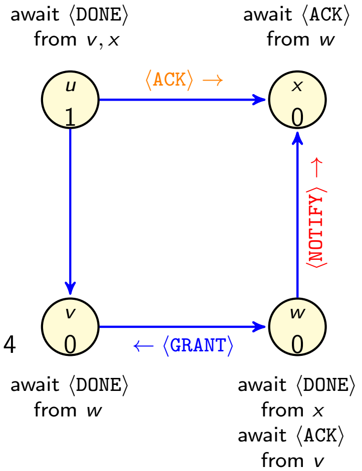
The Bracha-Toueg Algorithm

Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls Notify()

```
Notify_u():
    notified_u ← True
    for all $w \in \texttt{OUT}_u$, send ⟨NOTIFY⟩ to $w$
    if $n_u = 0$, then Grant_u()
    for all $w \in \texttt{OUT}_u$, await ⟨DONE⟩ from $w$
Grant_u():
    free_u ← True
    for all $w \in \texttt{IN}_u$ send ⟨GRANT⟩ to $w$
    for all $w \in \texttt{IN}_u$ await ⟨ACK⟩ from $w$
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    $u$ sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If $n_u > 0$, then
        $n_u \leftarrow n_u - 1$
        if $n_u = 0$, then Grant_u()
    $u$ sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is True at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.



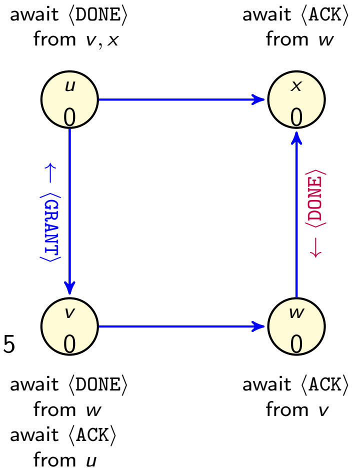
The Bracha-Toueg Algorithm

Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls Notify()

```
Notify_u():
    notified_u ← True
    for all $w \in \texttt{OUT}_u$, send ⟨NOTIFY⟩ to $w$
    if $n_u = 0$, then Grant_u()
    for all $w \in \texttt{OUT}_u$, await ⟨DONE⟩ from $w$
Grant_u():
    free_u ← True
    for all $w \in \texttt{IN}_u$ send ⟨GRANT⟩ to $w$
    for all $w \in \texttt{IN}_u$ await ⟨ACK⟩ from $w$
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    $u$ sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If $n_u > 0$, then
        $n_u \leftarrow n_u - 1$
        if $n_u = 0$, then Grant_u()
    $u$ sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is True at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.



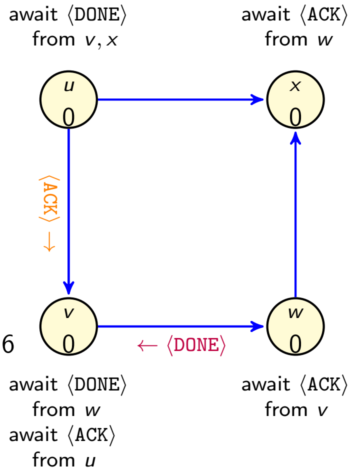
The Bracha-Toueg Algorithm

Initially: $\forall u, \mathtt{notified}_u = \mathtt{free}_u = \mathtt{False}$. Initiator calls Notify()

```
Notify_u():
    notified_u ← True
    for all w ∈ OUT_u, send ⟨NOTIFY⟩ to w
    if n_u = 0, then Grant_u()
    for all w ∈ OUT_u, await ⟨DONE⟩ from w
Grant_u():
    free_u ← True
    for all w ∈ IN_u send ⟨GRANT⟩ to w
    for all w ∈ IN_u await ⟨ACK⟩ from w
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    u sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If n_u > 0, then
        n_u ← n_u − 1
        if n_u = 0, then Grant_u()
    u sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\mathtt{free}_u$ is True at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.



The Bracha-Toueg Algorithm

Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls $\texttt{Notify}()$

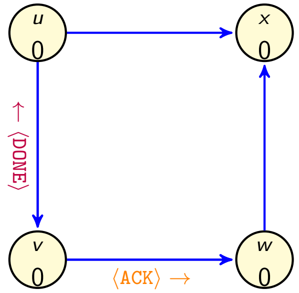
```
Notify_u():
    notified_u ← True
    for all w ∈ OUT_u, send ⟨NOTIFY⟩ to w
    if n_u = 0, then Grant_u()
    for all w ∈ OUT_u, await ⟨DONE⟩ from w
Grant_u():
    free_u ← True
    for all w ∈ IN_u send ⟨GRANT⟩ to w
    for all w ∈ IN_u await ⟨ACK⟩ from w
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    u sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If n_u > 0, then
        n_u ← n_u − 1
        if n_u = 0, then Grant_u()
    u sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is $\texttt{True}$ at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.

await ⟨DONE⟩ from x	await ⟨ACK⟩ from w

7



The Bracha-Toueg Algorithm

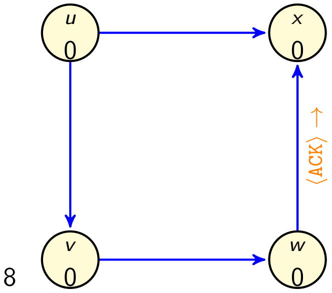
Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls $\texttt{Notify}()$

```
Notify_u():
    notified_u ← True
    for all w ∈ OUT_u, send ⟨NOTIFY⟩ to w
    if n_u = 0, then Grant_u()
    for all w ∈ OUT_u, await ⟨DONE⟩ from w
Grant_u():
    free_u ← True
    for all w ∈ IN_u send ⟨GRANT⟩ to w
    for all w ∈ IN_u await ⟨ACK⟩ from w
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    u sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If n_u > 0, then
        n_u ← n_u − 1
        if n_u = 0, then Grant_u()
    u sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is $\texttt{True}$ at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.

await ⟨DONE⟩
  from x



The Bracha-Toueg Algorithm

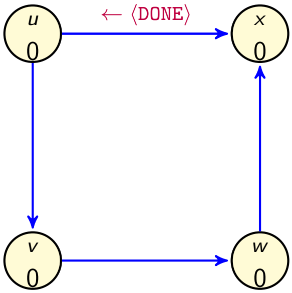
Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls $\texttt{Notify}()$

```
$\texttt{Notify}_u()$:
    $\texttt{notified}_u \leftarrow \texttt{True}$
    for all $w \in \texttt{OUT}_u$, send $\langle\texttt{NOTIFY}\rangle$ to $w$
    if $n_u = 0$, then $\texttt{Grant}_u()$
    for all $w \in \texttt{OUT}_u$, await $\langle\texttt{DONE}\rangle$ from $w$
$\texttt{Grant}_u()$:
    $\texttt{free}_u \leftarrow \texttt{True}$
    for all $w \in \texttt{IN}_u$ send $\langle\texttt{GRANT}\rangle$ to $w$
    for all $w \in \texttt{IN}_u$ await $\langle\texttt{ACK}\rangle$ from $w$
On receive $\langle\texttt{NOTIFY}\rangle$:
    If not $\texttt{notified}_u$, then $\texttt{Notify}_u()$
    $u$ sends back $\langle\texttt{DONE}\rangle$
On receive $\langle\texttt{GRANT}\rangle$:
    If $n_u > 0$, then
        $n_u \leftarrow n_u - 1$
        if $n_u = 0$, then $\texttt{Grant}_u()$
    $u$ sends back $\langle\texttt{ACK}\rangle$
```

Initiator u is not deadlocked if $\texttt{free}_u$ is $\texttt{True}$ at the end

A node awaiting $\langle\texttt{DONE}\rangle$ or $\langle\texttt{ACK}\rangle$ can process incoming $\langle\texttt{NOTIFY}\rangle$ and $\langle\texttt{GRANT}\rangle$ messages.

9



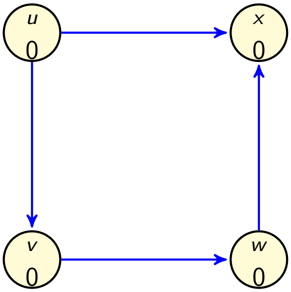
The Bracha-Toueg Algorithm

Initially: $\forall u, \texttt{notified}_u = \texttt{free}_u = \texttt{False}$. Initiator calls `Notify()`

```
Notify_u():
    notified_u ← True
    for all $w \in \texttt{OUT}_u$, send ⟨NOTIFY⟩ to $w$
    if $n_u = 0$, then Grant_u()
    for all $w \in \texttt{OUT}_u$, await ⟨DONE⟩ from $w$
Grant_u():
    free_u ← True
    for all $w \in \texttt{IN}_u$ send ⟨GRANT⟩ to $w$
    for all $w \in \texttt{IN}_u$ await ⟨ACK⟩ from $w$
On receive ⟨NOTIFY⟩:
    If not notified_u, then Notify_u()
    $u$ sends back ⟨DONE⟩
On receive ⟨GRANT⟩:
    If $n_u > 0$, then
        $n_u \leftarrow n_u - 1$
        if $n_u = 0$, then Grant_u()
    $u$ sends back ⟨ACK⟩
```

Initiator u is not deadlocked if $\texttt{free}_u$ is `True` at the end

A node awaiting ⟨DONE⟩ or ⟨ACK⟩ can process incoming ⟨NOTIFY⟩ and ⟨GRANT⟩ messages.



What about the Mutual Exclusion Pattern?

Bracha-Toueg doesn't guarantee that a deadlock won't occur in the future: only whether the initiator is deadlocked or not.

- The global snapshot will capture the state either before the resource is handed off to the first requestor, or after.
- The WFG will be different in the two cases
- Bracha-Toueg will give the correct answer for the particular case of the WFG that appears in the global state, even if the next operation would necessarily put the node into deadlock