Distributed and Parallel Computing

Lecture 14

Alan P. Sexton

University of Birmingham

Spring 2018

The Echo Algorithm

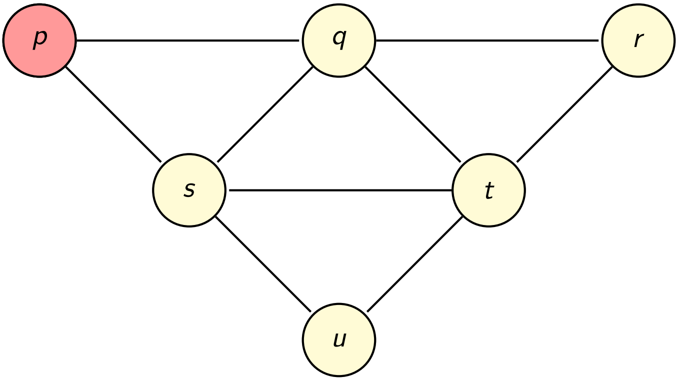
A wave, but not a traversal algorithm (so no *tokens* involved), *Echo* is a centralized algorithm (i.e. one initiator only) for undirected networks.

- 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

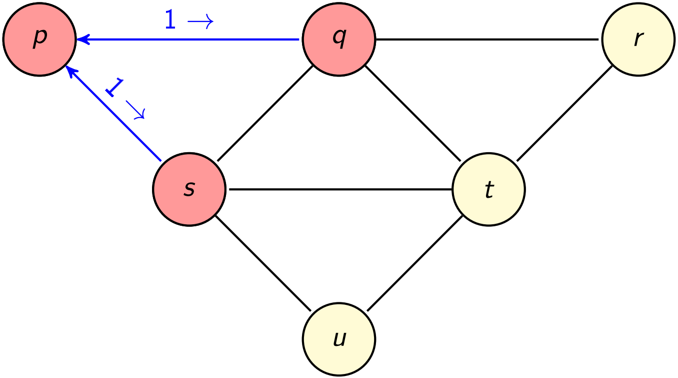
Example Execution of the Echo Algorithm

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.



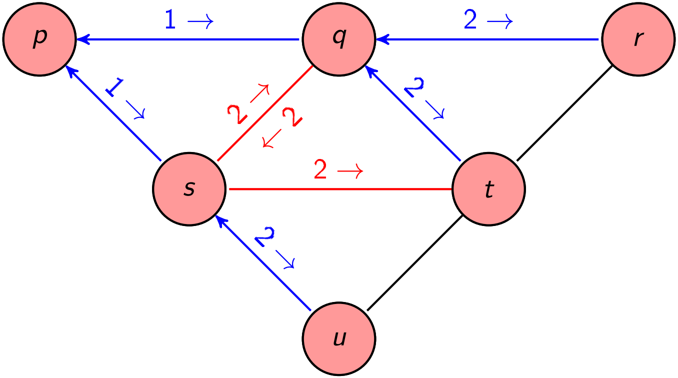
Example Execution of the Echo Algorithm

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.



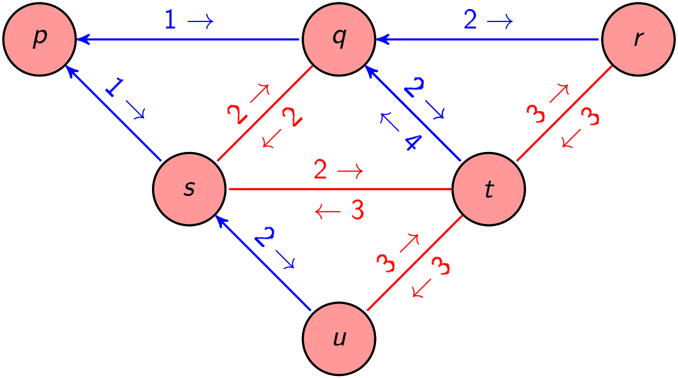
Example Execution of the Echo Algorithm

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.



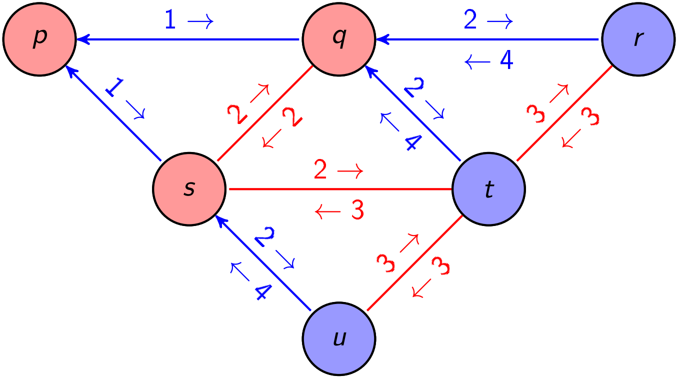
Example Execution of the Echo Algorithm

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.



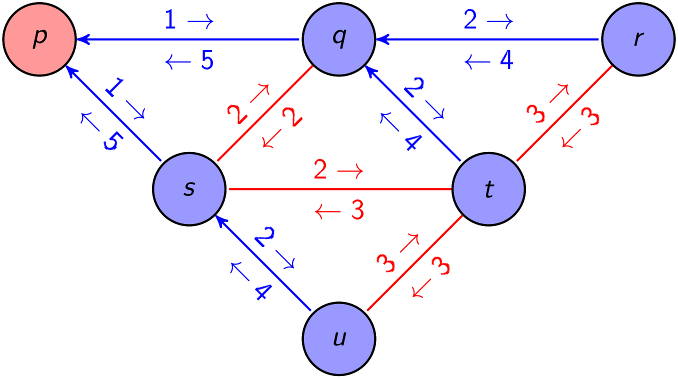
Example Execution of the Echo Algorithm

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.



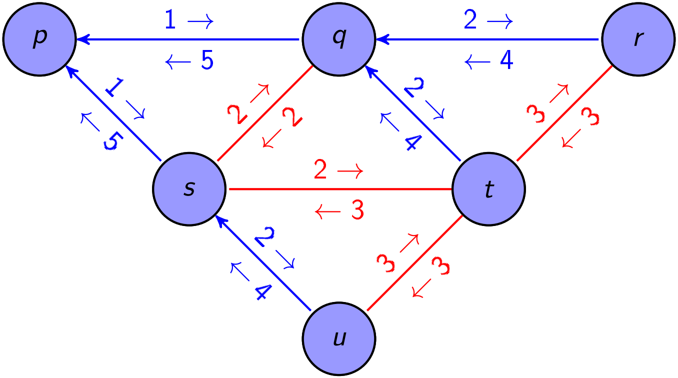
Example Execution of the Echo Algorithm

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.



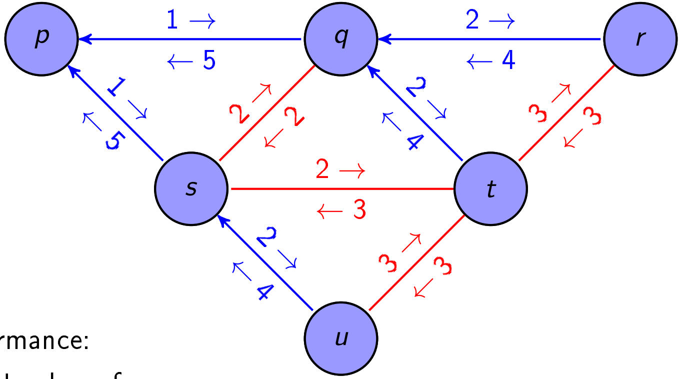
Example Execution of the Echo Algorithm

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.

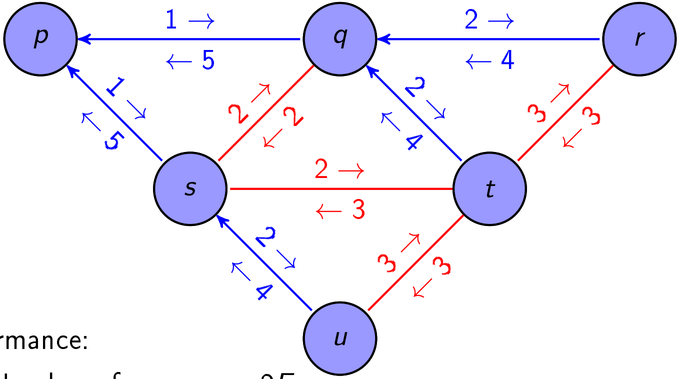


Performance:

- 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.

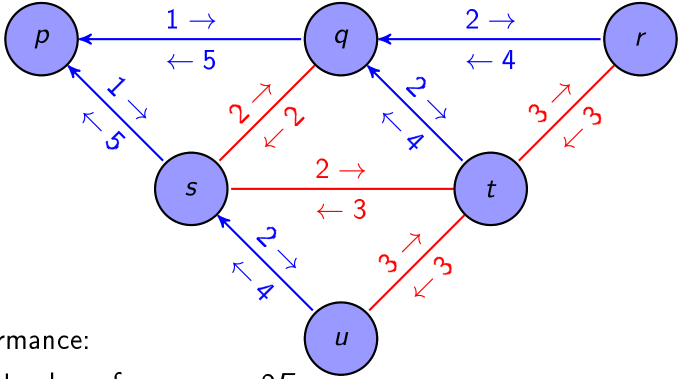


Performance:

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

Example Execution of the Echo Algorithm

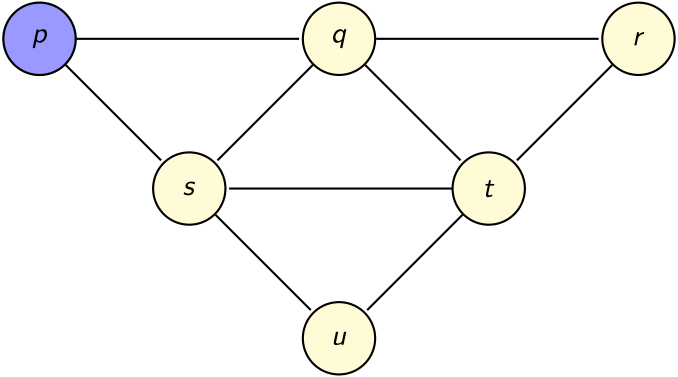
Here if messages are sent out immediately after each other without waiting to receive a message in between sends, they are given the same number. In reality, real messages are not sent simultaneously.



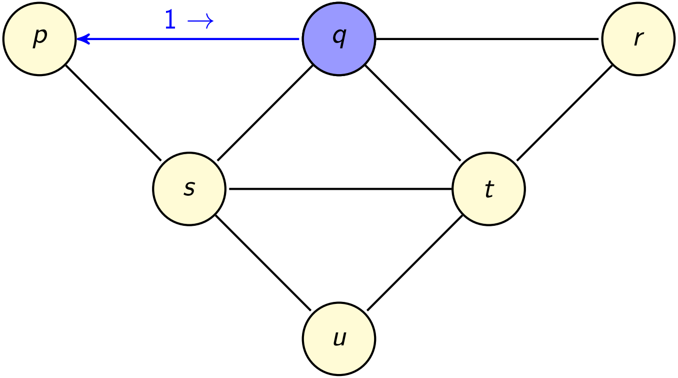
Performance:

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

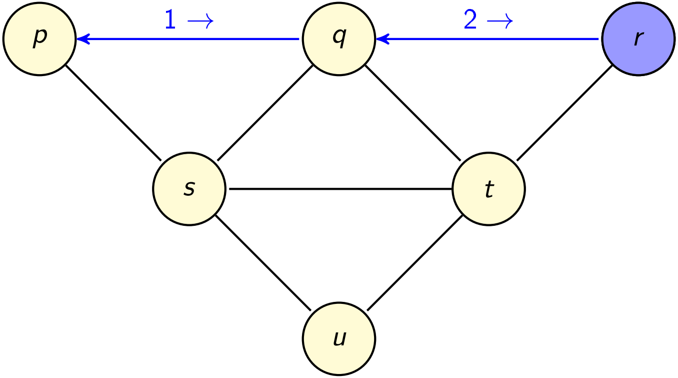
Tarry Algorithm considered as an Echo Algorithm



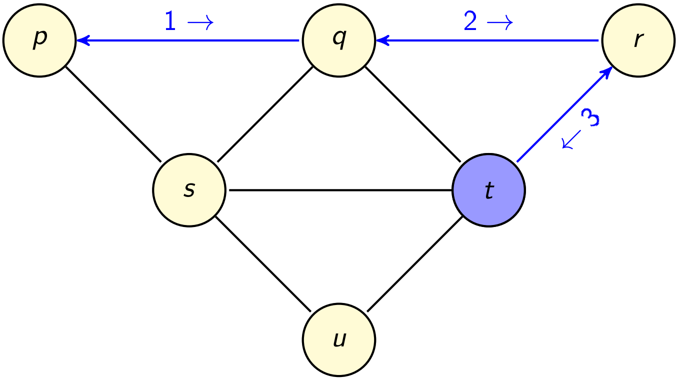
Tarry Algorithm considered as an Echo Algorithm



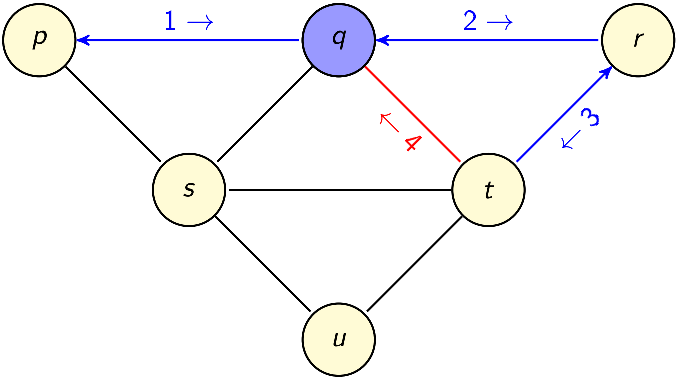
Tarry Algorithm considered as an Echo Algorithm



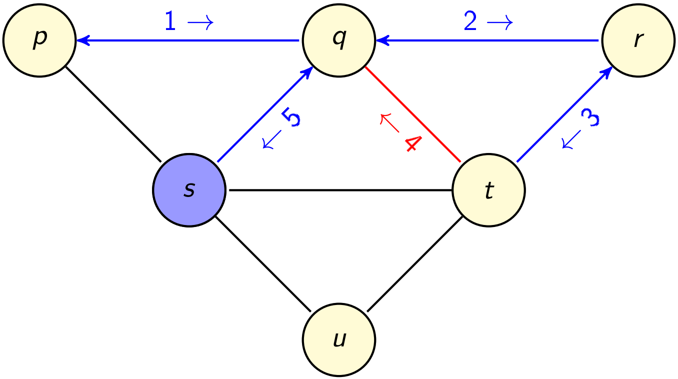
Tarry Algorithm considered as an Echo Algorithm



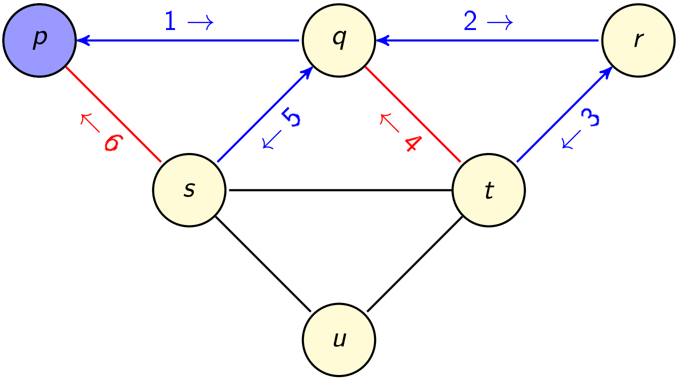
Tarry Algorithm considered as an Echo Algorithm



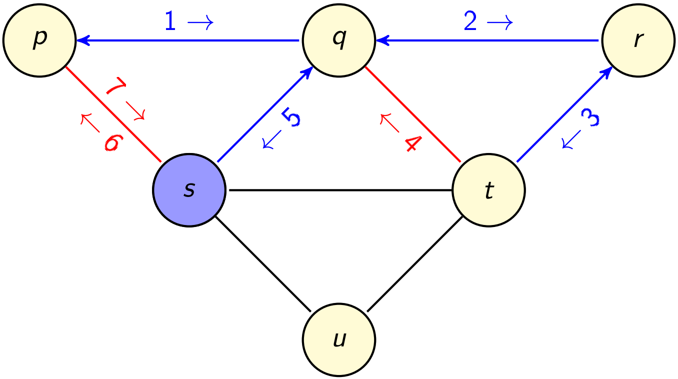
Tarry Algorithm considered as an Echo Algorithm



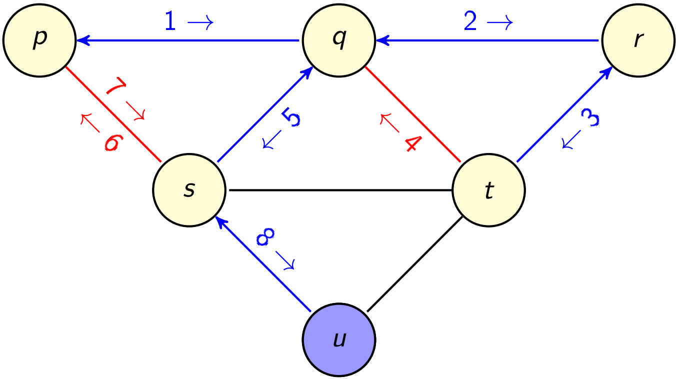
Tarry Algorithm considered as an Echo Algorithm



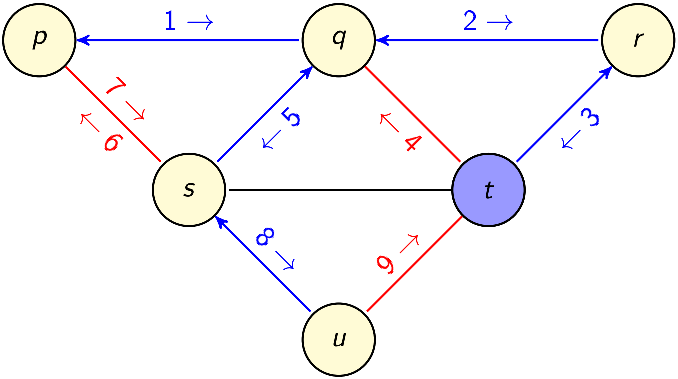
Tarry Algorithm considered as an Echo Algorithm



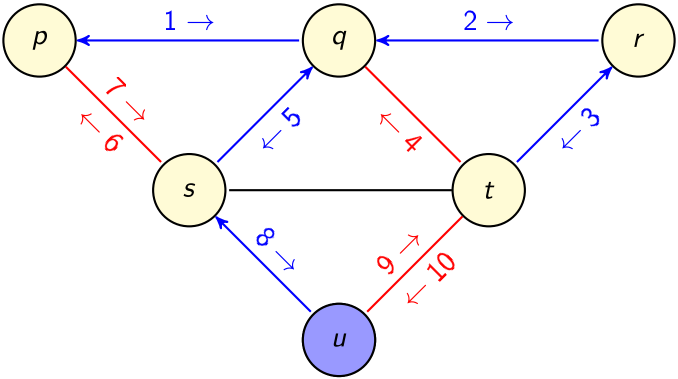
Tarry Algorithm considered as an Echo Algorithm



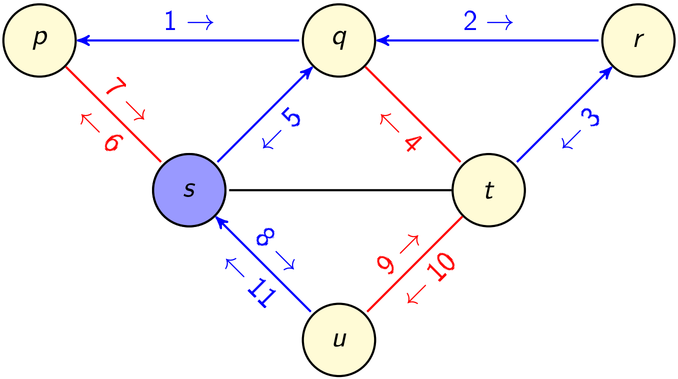
Tarry Algorithm considered as an Echo Algorithm



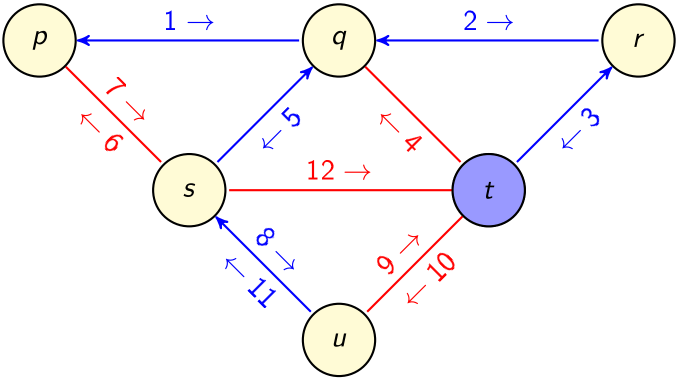
Tarry Algorithm considered as an Echo Algorithm



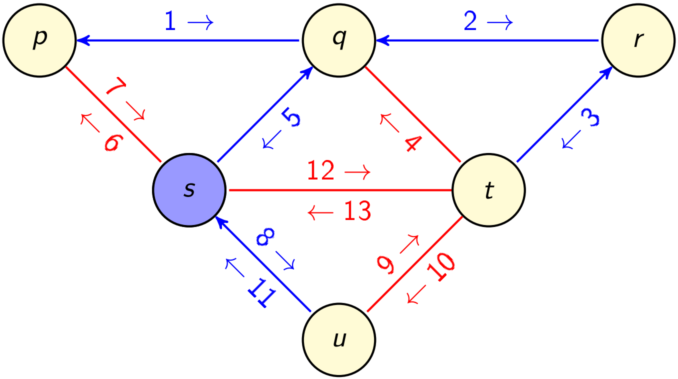
Tarry Algorithm considered as an Echo Algorithm



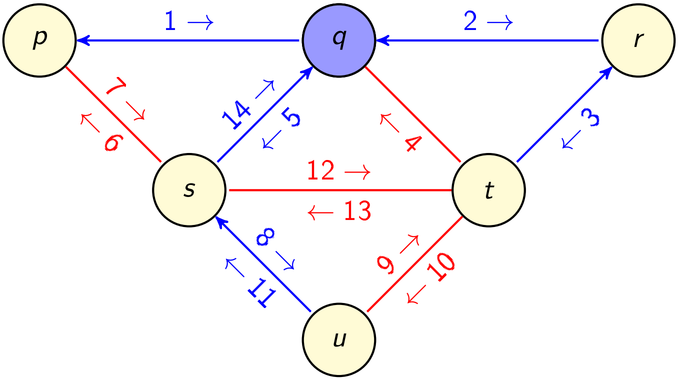
Tarry Algorithm considered as an Echo Algorithm

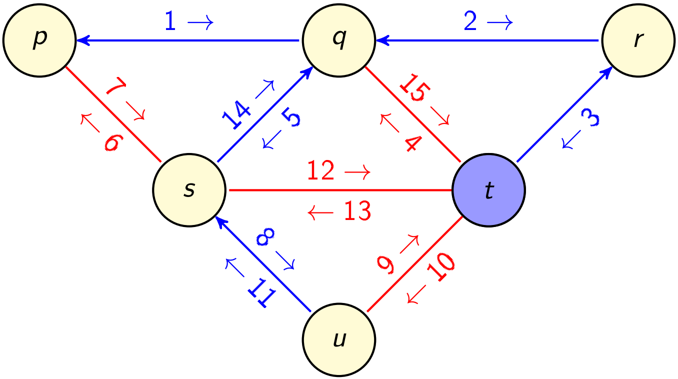


Tarry Algorithm considered as an Echo Algorithm

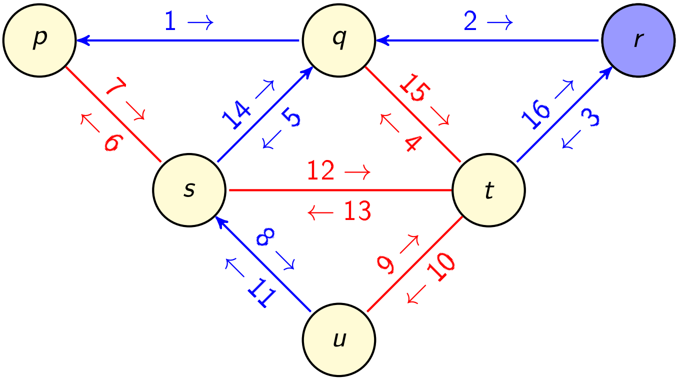


Tarry Algorithm considered as an Echo Algorithm

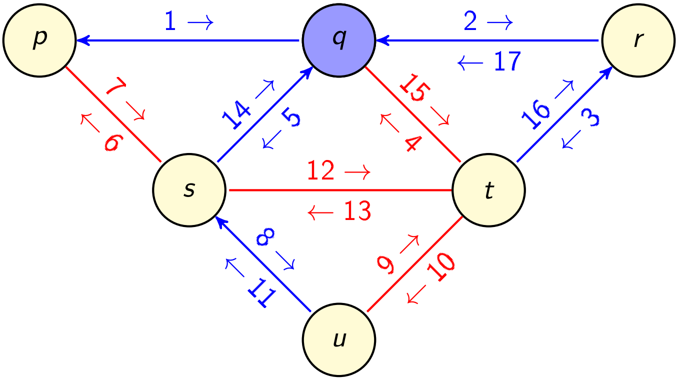


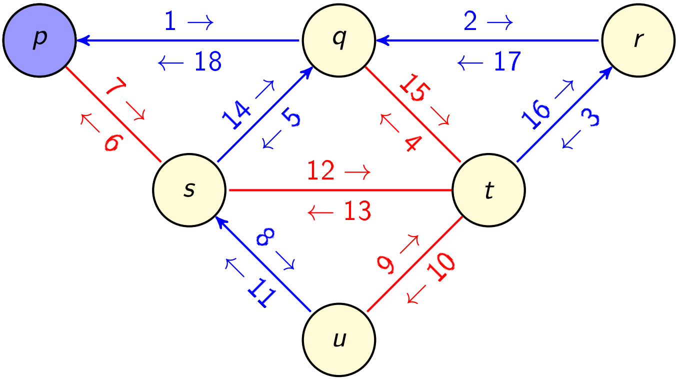

p
q
r
s
t
u
1 →
2 →
7 →
← 6
14 →
← 5
15 →
← 4
← 3
12 →
← 13
8 →
← 11
9 →
← 10

Tarry Algorithm considered as an Echo Algorithm

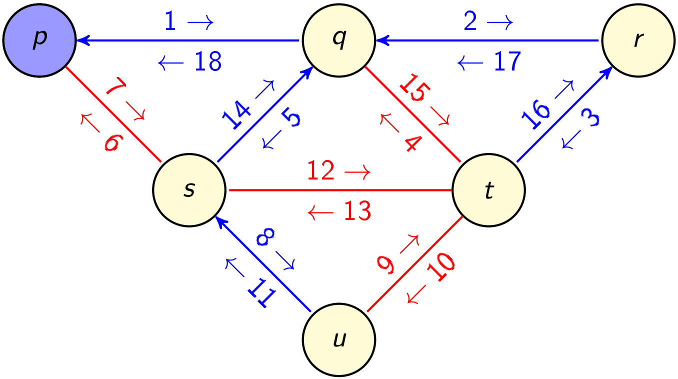


Tarry Algorithm considered as an Echo Algorithm



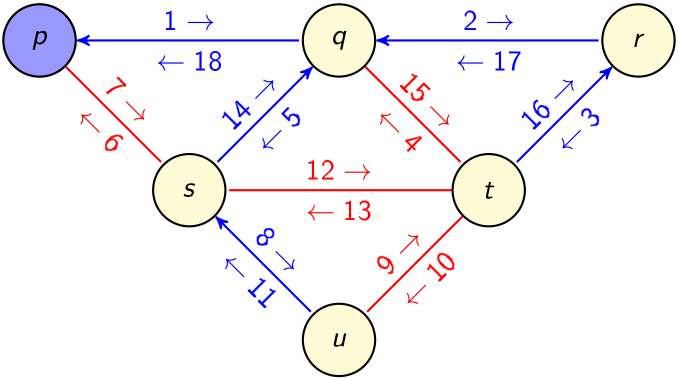

p
q
r
s
t
u
1 →
← 18
2 →
← 17
7 →
← 6
14 →
← 5
15 →
← 4
16 →
← 3
12 →
← 13
8 →
← 11
9 →
← 10

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

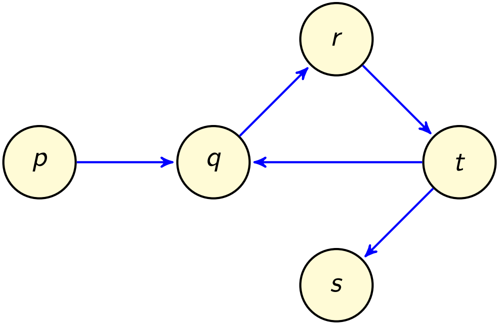
There are essentially 3 strategies for dealing with deadlocks:

- 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)*

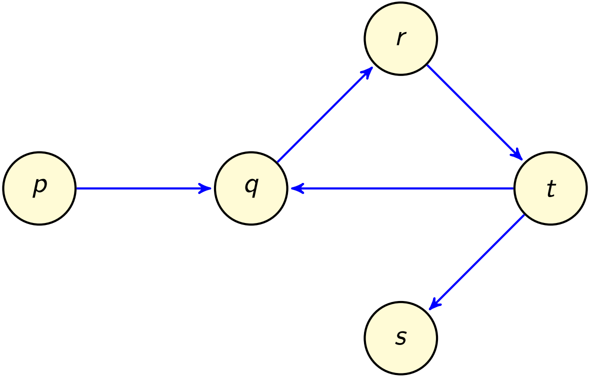
- Directed graph
- Nodes are processes
- Edge from p to q if p is blocked waiting for q to respond or release some resource
- In the simplest model, a cycle in the WFG $\Rightarrow$ deadlock



Deadlock Models

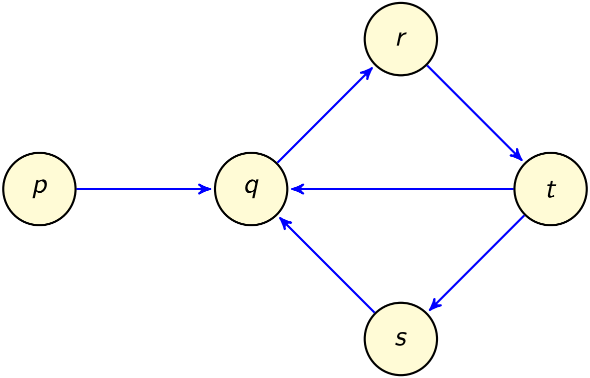
- *Single-resource* model:
  - A process can have at most one outstanding request for one (unit of a) resource
  - Cycle in WFG $\Rightarrow$ deadlock
  - Simplest model
- *AND* model
  - Each process can request multiple resources simultaneously and *all* requested resources must be supplied to unblock
  - 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