THEORY OF DISTRIBUTED COMPUTING

Spanning Tree Construction
Computation in trees

A.A. 2023/24

Main References

"DESIGN AND ANALYSIS OF DISTRIBUTED ALGORITHMS"

Nicola Santoro
Wiley 2007 (available at the library)

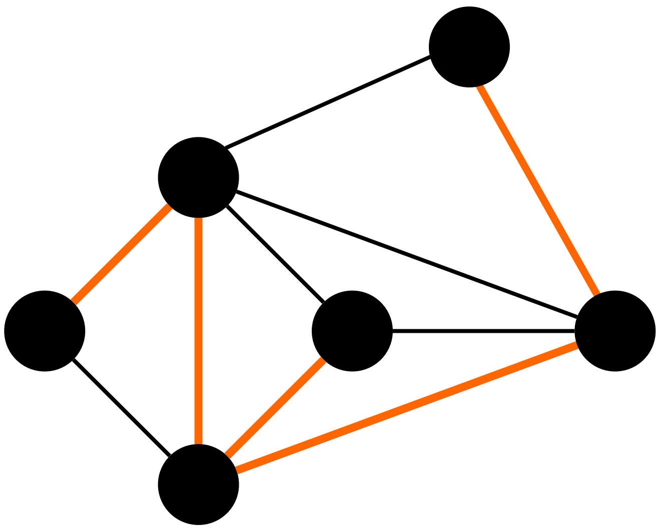
Original slides by Paola Flocchini, SITE, University of Ottawa, Canada (rearranged by Manuela Montangero)
SLIDES CAN NOT BE REDISTRIBUTED

Spanning Tree Construction

A spanning tree T of a graph $G = (V,E)$ is an acyclic subgraph of G such that $T=(V,E')$ and $E' \subseteq E$.

Restrictions:

Single initiator
Bidirectional links
Total reliability
G connected

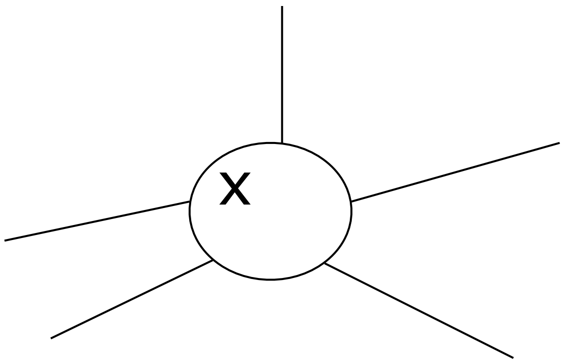


Spanning Tree: Protocol SHOUT

IDEAS?

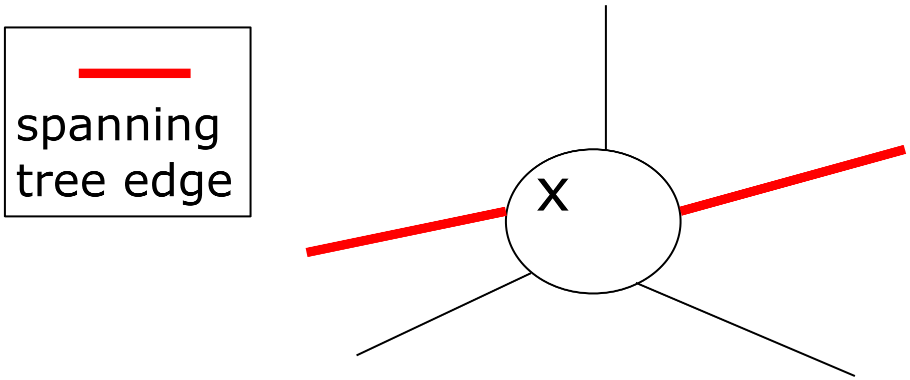
Initially:

∀ x, Tree-neighbors(x) = { }



At the end:

∀ x, Tree-neighbors(x) = {neighbours in the spanning tree }

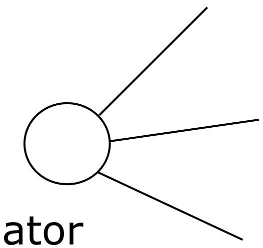


Observation

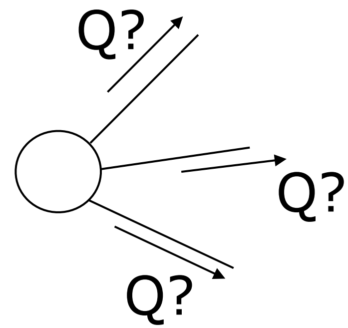
At the end entities do not know the entire spanning tree, but only those edges that connect them to the neighbours in the spanning tree

Protocol SHOUT

1.

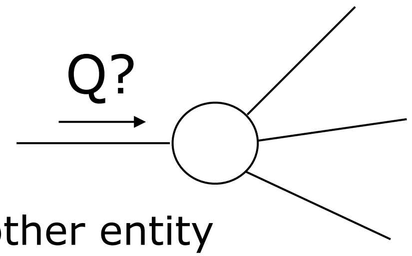


initiator



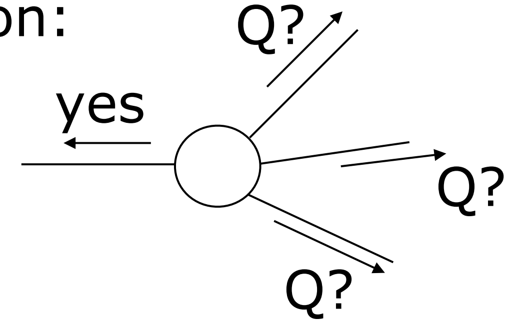
Q?
"Do you want to be my neighbour in the spanning tree ?"

2.

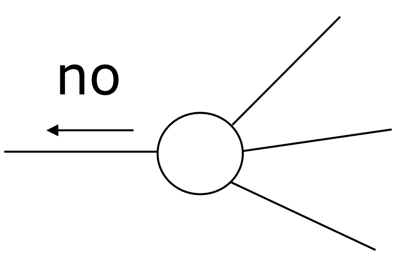


any other entity

If it is the first question:



If has already answered yes before:



Protocol SHOUT

State S = {INITIATOR, IDLE, ACTIVE, DONE}
Sinit = {INITIATOR, IDLE} (possible initial states)
Sterm = {DONE} (termination state)

Protocol per agent x

```
INITIATOR
spontaneously
    root = true
    Tree-neighbours(x)= {}
    send Q to N(x)
    counter = 0
    become(ACTIVE)
```

Counts the number of answers

```
IDLE
receiving(Q)
    root = false
    parent = sender
    Tree-neighbours(x)={sender}
    send YES to parent
    counter = 1
    if counter = |N(x)|
        then
            become(DONE)
        else
            send Q to N(x)-{sender}
            become(ACTIVE)
```

Protocol SHOUT

State S = {INITIATOR, IDLE, ACTIVE, DONE}
Sinit = {INITIATOR, IDLE} (possible initial states)
Sterm = {DONE} (termination state)

```
ACTIVE
receiving(Q)
    send NO to sender

receiving(YES)
    Tree-neighbours(x)=
        Tree-neighbours(x)∪{sender}
    counter = counter + 1
    if counter = |N(x)|
        then
          become(DONE)

receiving(NO)
    counter = counter + 1
    if counter = |N(x)|
        then
          become(DONE)
```

For any other pair (state,event) the corresponding action is nil

SHOUT: correctness and termination

- If x is in Tree-neighbours of y,
  then y is in Tree-neighbours of x

- If x sends `YES` to y,
  then y is in Tree-neighbours of x
  and
  is connected to the initiator by a chain of `YES`

- Every x (except the initiator) sends exactly one `YES`

$$\Downarrow$$

The spanning graph defined by the Tree-neighbours relation is a connected tree containing all entities

Note: local termination

SHOUT: message complexity

Observation

SHOUT = FLOODING + REPLY

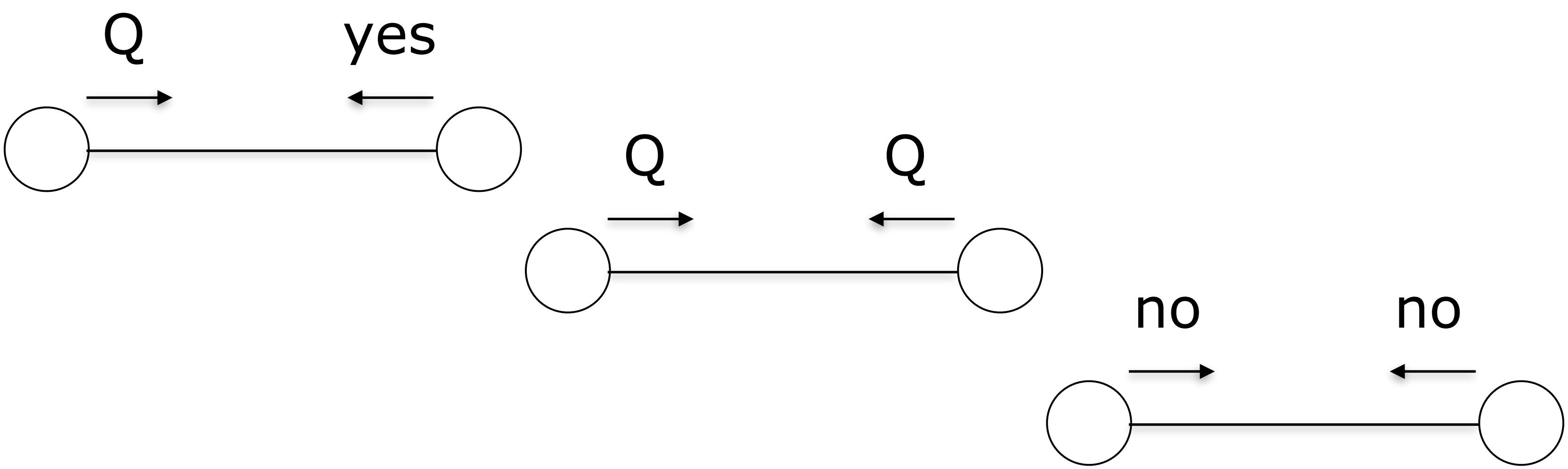
⇓

we expect

Message(SHOUT) = 2Message(FOOLDING)

SHOUT: message complexity

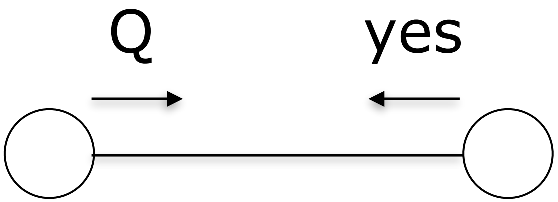
Possible situations



Impossible situations

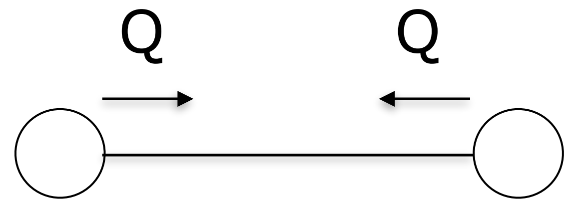
SHOUT: message complexity - worst case

Total number of Q



(n-1)

only one Q on the ST links

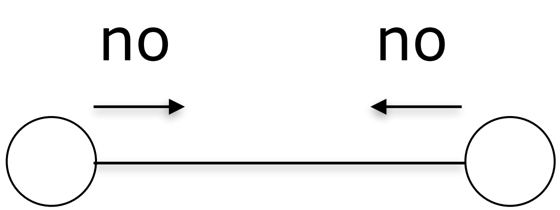


2[m -(n-1)]

on the other links

$$\text{Total} = (n-1) + 2[m-(n-1)] = 2m - n + 1$$

Total number of NO

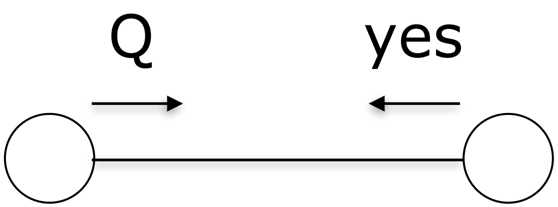


2[m - (n-1)]

as many as Q---Q

SHOUT: message complexity - worst case

Total number of YES



(n-1)

exactly one yes only on the ST links

Total number of messages (Q + NO + YES)

$$Message(SHOUT) = 2m - n + 1 + 2[m - (n-1)] + (n-1)$$
$$= 4m - 2n + 2$$
$$= 2\,(2m - n + 1) = 2Message(FLOODING)$$

Is it possible to reduce the number of messages?

Protocol SHOUT+: spanning tree construction without NO

State S = {`INITIATOR`, `IDLE`, `ACTIVE`, `DONE`}
Sinit = {`INITIATOR`, `IDLE`} (possible initial states)
Sterm = {`DONE`} (termination state)

Actions in reaction to events when in states

`INITIATOR` and `IDLE`

do not change

Protocol SHOUT+: spanning tree construction without NO

State S = {INITIATOR, IDLE, ACTIVE, DONE}
Sinit = {INITIATOR, IDLE} (possible initial states)
Sterm = {DONE} (termination state)

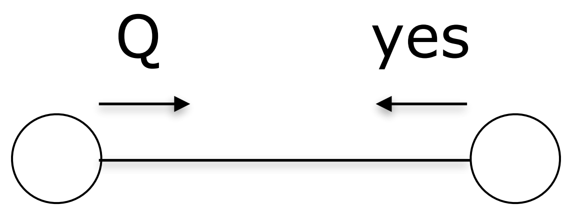
```
ACTIVE
receiving(Q)
    counter = counter + 1
    if counter = |N(x)|
        then
          become(DONE)

receiving(YES)
    Tree-neighbours(x)=
        Tree-neighbours(x)∪{sender}
    counter = counter + 1
    if counter = |N(x)|
        then
          become(DONE)
```

to be interpreted as no

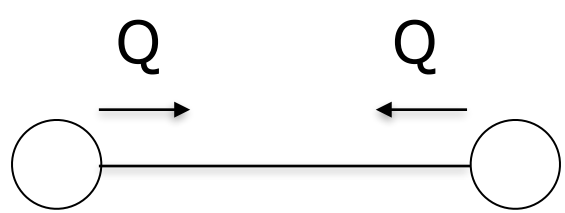
SHOUT+: message complexity

On each link there will be exactly two messages

either



or



Message(SHOUT+) = 2m

much better than

2 (2m - n + 1) = Message(SHOUT)

Spanning Tree Construction by Traversal

A spanning tree can be build
by traversing the graph

TRAVERSAL PROBLEM
Initially all entities are in the same
unvisited state
except for one that is *visited* and is the *initiator*
The goal is to make all entities
visited sequentially (one at the time)

Traversal protocol
Distributed algorithm that, starting from the initiator, uses a special message (token) and reaches every entity sequentially. Once an entity receives the token is considered visited.

A depth-first traversal of a graph builds a spanning tree of the graph

Spanning Tree Construction by Traversal

DEPTH-FIRST traversal

"The graph is traversed trying to forward (the token) as long as possible"

Restrictions

Single initiator

Bidirectional links

Connectivity

Total reliability

Spanning Tree Construction by Traversal

Traversal protocol

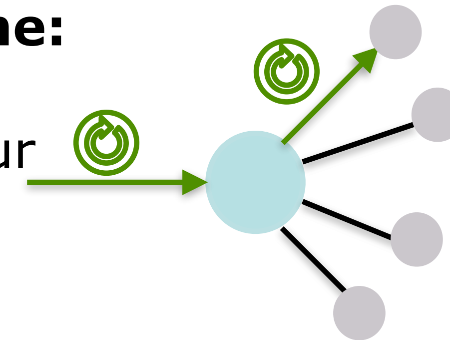
Visited node

Froward Token

Back-edge Token

Return Token

1. **When receiving the Forward Token the first time:**
   - remember who sent the token
   - send the Forward Token to ONE unvisited neighbour
   - wait for token to return

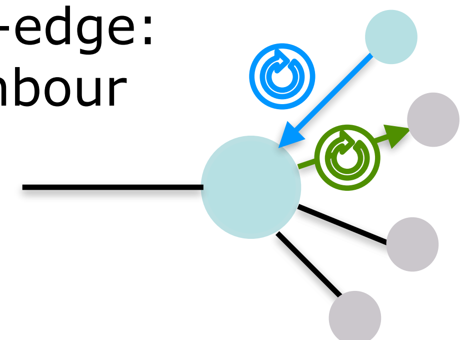


2. **When receiving the token again:**

   if is ReturnToken:

   if there still are unvisited neighbours with no back-edge:
   - send the Forward Token to ONE unvisited neighbour
   - wait for token to return

   otherwise



Spanning Tree Construction by Traversal

Traversal protocol

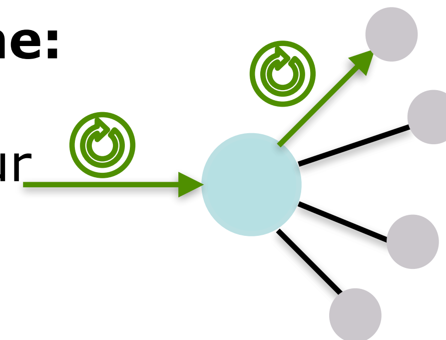
Visited node

Froward Token

Back-edge Token

Return Token

1. **When receiving the Forward Token the first time:**
   - remember who sent the token
   - send the Forward Token to ONE unvisited neighbour
   - wait for token to return



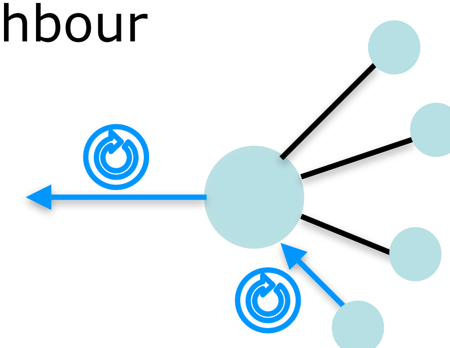
2. **When receiving the token again:**

   if is ReturnToken:

   if there still are unvisited neighbours with no back-edge:
   - send the Forward Token to ONE unvisited neighbour
   - wait for token to return

   otherwise

   send ReturnToken to the one from which it first received the Forward Token



Spanning Tree Construction by Traversal

Traversal protocol

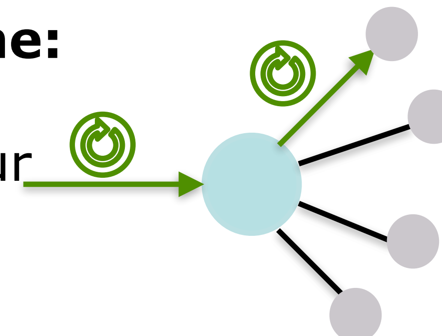
Visited node

Froward Token

Back-edge Token

Return Token

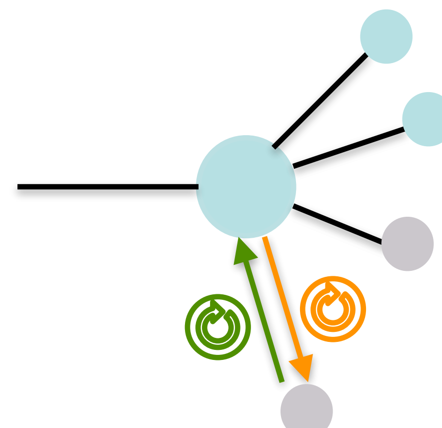
1. **When receiving the Forward Token the first time:**
   - remember who sent the token
   - send the Forward Token to ONE unvisited neighbour
   - wait for token to return



2. **When receiving the token again:**

   if is Forward Token:

   send back to sender Back-edge Token and classify the link as back-edge



Spanning Tree Construction by Traversal

Traversal protocol

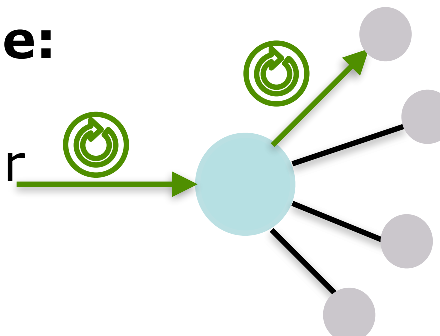
Visited node

Froward Token

Back-edge Token

Return Token

1. **When receiving the Forward Token the first time:**
   - remember who sent the token
   - send the Forward Token to ONE unvisited neighbour
   - wait for token to return



2. **When receiving the token again:**

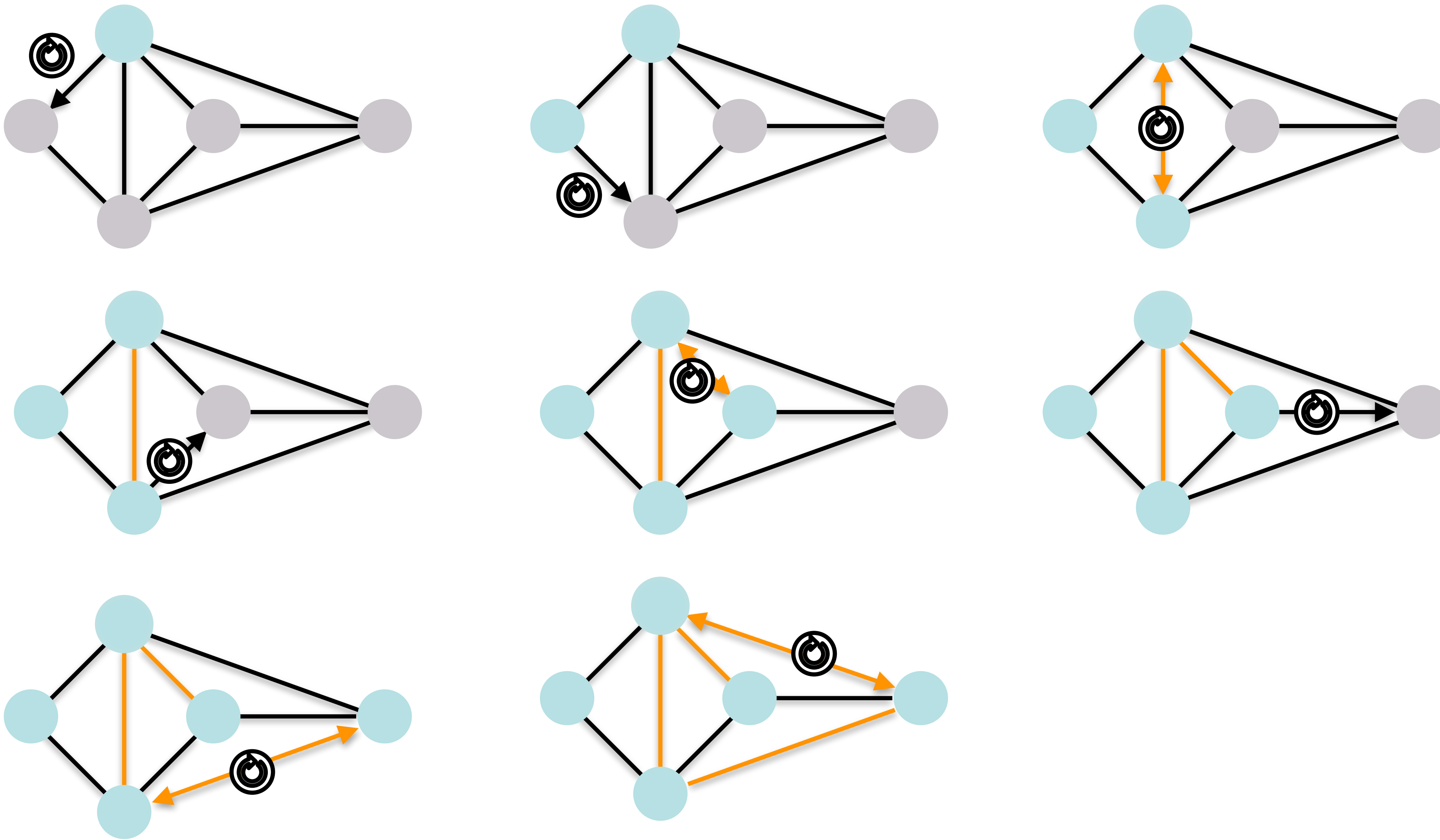
if is Forward Token:

send back to sender Back-edge Token and classify the link as back-edge

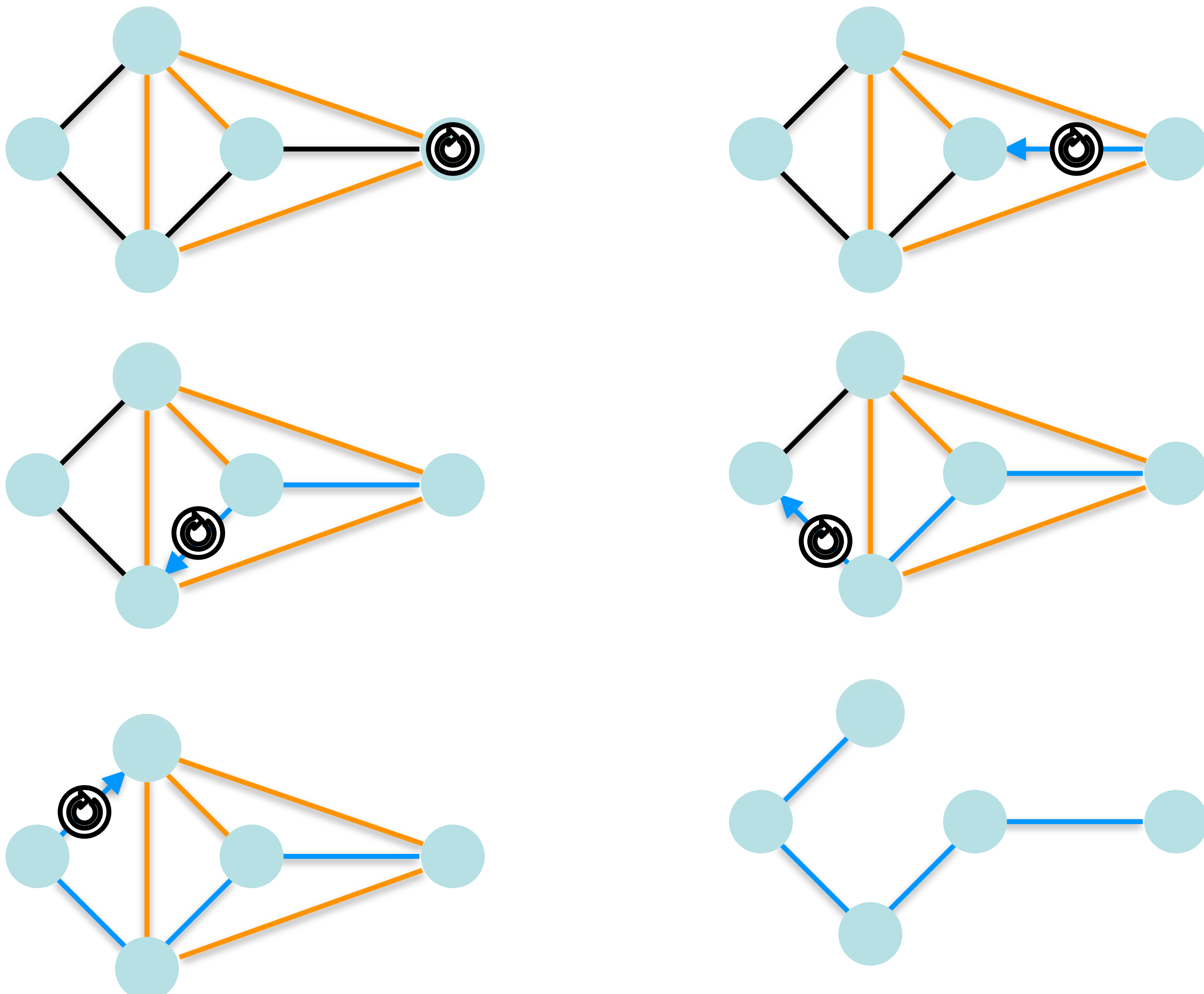
if is Back-edge Token:

proceed as with Return Token

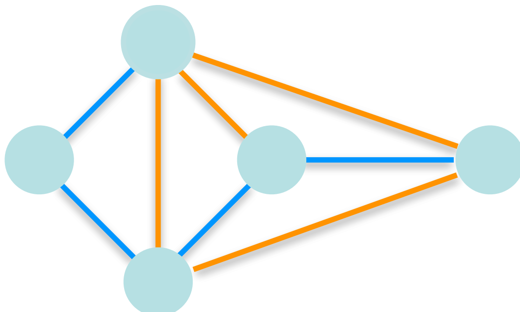
Depth-First Traversal Example



Depth-First Traversal Example



Spanning Tree Construction by Traversal



Removing back-edges
we have a spanning tree

Who is the root? — The root of the tree is the initiator

Who is entity x's parent? — The parent of entity x is the one from which it first received the token

Who are entity x's children? — The children of entity x are the neighbours that are not connected by a back-edge

Spanning Tree Construction by Traversal

State S = {INITIATOR, IDLE, VISITED, DONE}
Sinit = {INITIATOR, IDLE} (possible initial states)
Sterm = {DONE} (termination state)

Protocol per agent x

```
INITIATOR
spontaneously
   Unvisited := N(x)
   initiator := true
   VISIT
```

```
procedure VISIT
   if |Unvisited| > 0
     then
       next := pick(Unvisited)
       send Token to next
       become(VISITED)
     else
       if not(initiator)
           then
               send ReturnToken to entry
       become(DONE)
```

Picks one element in Unvisited and eliminates the element form the set

Spanning Tree Construction by Traversal

State S = {INITIATOR, IDLE, VISITED, DONE}
Sinit = {INITIATOR, IDLE} (possible initial states)
Sterm = {DONE} (termination state)

Protocol per agent x

```
IDLE
receiving(Token)
    entry := sender
    Unvisited := N(x)\{sender}
    initiator := false
    VISIT
```

```
procedure VISIT
    if |Unvisited| > 0
      then
        next := pick(Unvisited)
        send Token to next
        become(VISITED)
      else
        if not(initiator)
           then
              send ReturnToken to entry
        become(DONE)
```

Picks one element in Unvisited and eliminates the element form the set

UNIMORE

Spanning Tree Construction by Traversal

State S = {`INITIATOR`, `IDLE`, `VISITED`, `DONE`}
Sinit = {`INITIATOR`, `IDLE`} (possible initial states)
Sterm = {`DONE`} (termination state)

Protocol per agent x

```
VISITED
receiving(Token)
    Unvisited := Unvisited\{sender}
    send BackEdgeToken to sender

receiving(ReturnToken)
    VISIT

receiving(BackEdgeToken)
    VISIT
```

a descendant in the spanning tree is also a neighbour of x.
x will not visit it later on.

a neighbour terminated its visit,
x continues with next neighbour

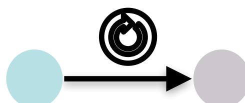
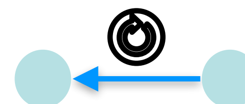
the neighbour has already received the token before,
x continues with next neighbour

When `DONE`, agent x will not receive messages any more

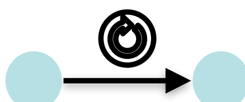
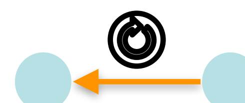
Spanning Tree Construction by Traversal

How many messages to perform depth-first traversal?

Given a link

 followed by 

XOR

 followed by 

MESSAGE COMPLEXITY = 2m

Is it possible to reduce message complexity?
No, *Message(DFT(G))* $\in \Omega(m)$
(the proof is analogous to the one given for broadcast)

Spanning Tree Construction by Traversal

How much time to perform depth-first traversal?

Since traversal is sequential, and 2m messages are sent sequentially…

TIME COMPLEXITY = 2m

Can we do better?

Time complexity lower bound
Time(DFT(G)) $\geq n - 1$

each node has to be visited sequentially

Spanning Tree Construction by Traversal

Improving time complexity

Each entity MUST receive the token at lest once
BUT
in our protocol each entity receives the token once from each neighbour

IDEA

Avoid sending the token on back-edges

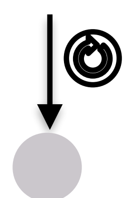
Spanning Tree Construction by Traversal

Improving time complexity

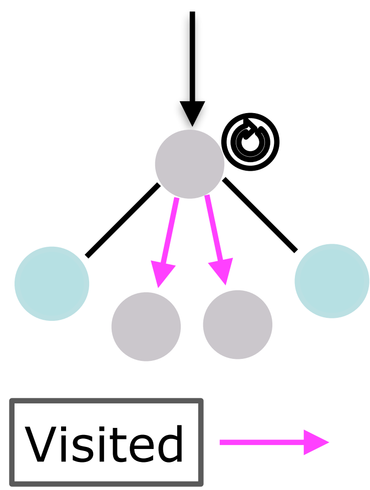
Avoid sending the token on back-edges

At any time, each entity has a set of visited neighbors (initially empty)

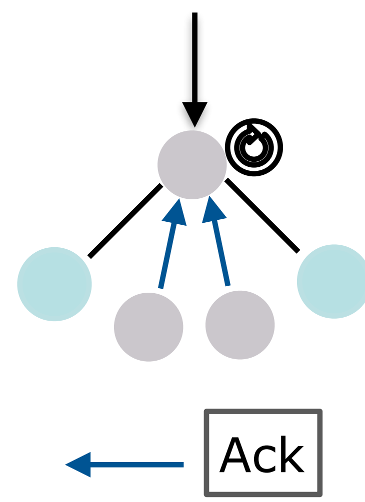
When receiving the token the first time…*



… send “Visited" msgs to non visited neighbors and…



… wait for “Acks" msgs…



* holds for initiator as well

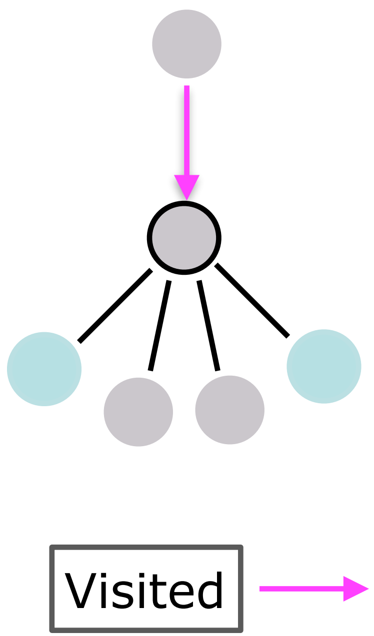
… then proceed as before considering only non visited neighbors.

Spanning Tree Construction by Traversal

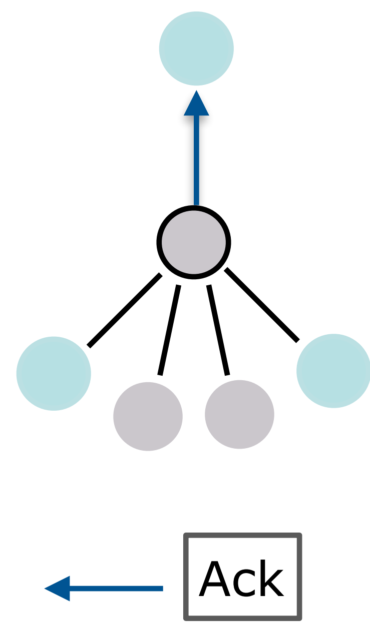
Improving time complexity

Avoid sending the token on back-edges

When receiving a "Visited" message...



... send "Ack" msg to sender and eliminate sender from the set of unvisited neighbors...



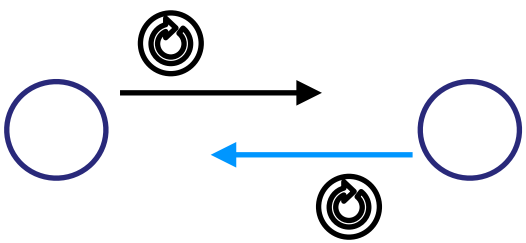
... then proceed as before, always considering only non visited neighbors.

Spanning Tree Construction by Traversal

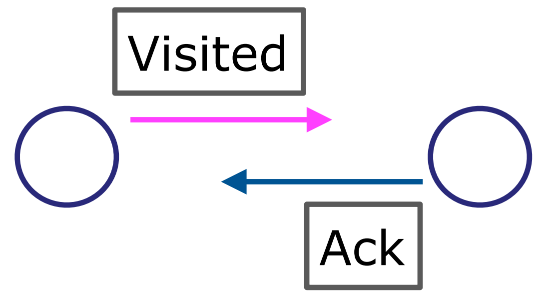
Improving time complexity

Avoid sending the token on back-edges

Messages through links



OR



Sequential:
2(n -1) message chain long

Concurrent to/from neighbors:
2 message chain long for each neigh.

longest message chain

$2n - 2 + 2n = 4n - 2$

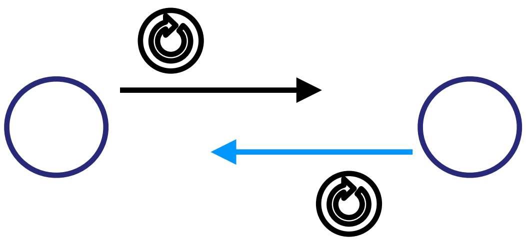
TIME COMPLEXITY $\in O(n)$

Spanning Tree Construction by Traversal

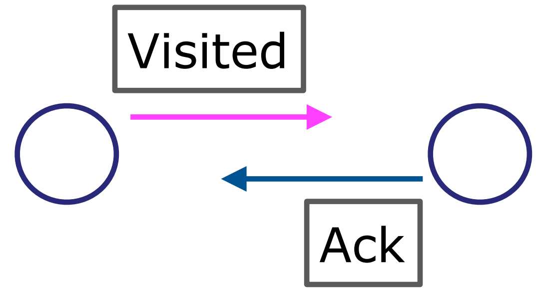
Avoid sending the token on back-edges

message complexity?

Messages through links



OR



For each entity (except initiator):
2 messages with token per link
in the tree

For each link: 2 messages
("visited" will not be sent twice on link)

number of messages

$2(n-1) + 2m$

MESSAGE COMPLEXITY $\in O(m)$

Spanning Tree Construction

Message(SHOUT+) = 2m

Which one?

Message(DFT) = 2m (+ 2(n-1))

Different techniques construct different spanning tree

The same protocol on the same graph might produce different spanning trees when executed at different times

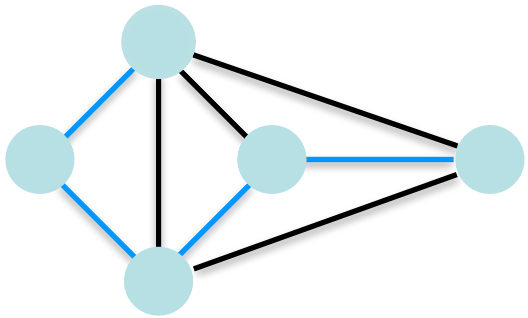
Using SHOUT, it is impossible to predict which spanning tree will be constructed

In general DFT constructs a tree with terrible diameter

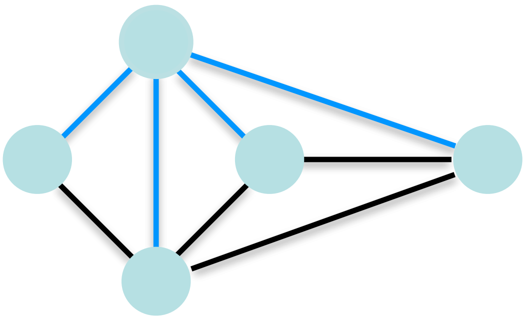
Spanning Tree Construction

Ideally it is desirable to have a spanning tree with SMALL diameter

WHY?



One possible spanning tree T
D(T) = 4



Another possible spanning tree T′
D(T′) = 2

Which one is more desirable for broadcasting?

Spanning Tree Construction

Ideally it is desirable to have a spanning tree with SMALL diameter

HOW?

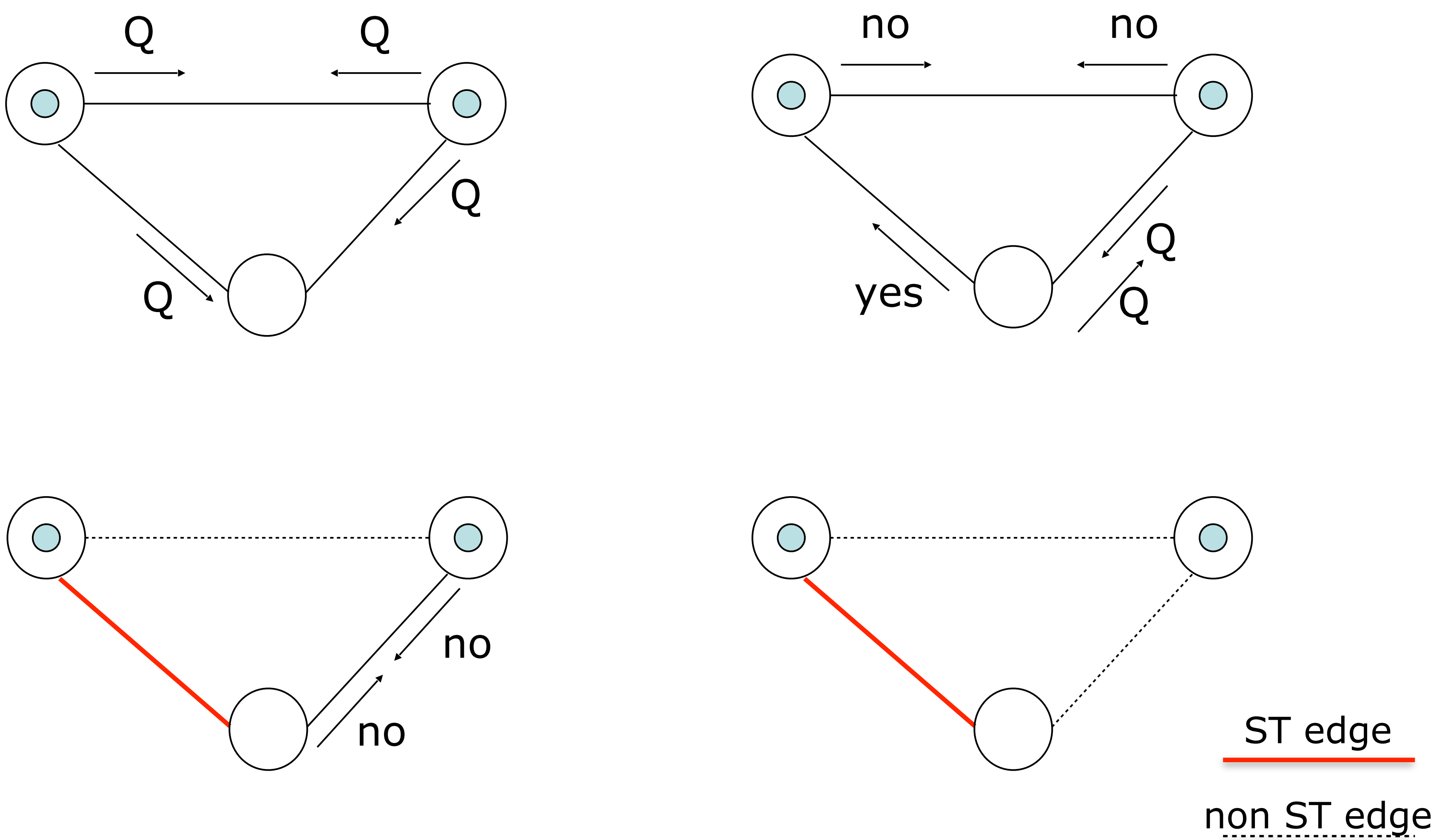
Broadcast-Tree Construction:

1. Determine a *center of G*
2. Construct a breadth-first spanning tree rooted in the center

1 and 2 are both expensive tasks

We will not go into further details

What happens with SHOUT if there are multiple initiators ?



Protocol SHOUT produces a forest

What happens if there are multiple initiators ?

In general an entity does not know if there are other initiators

⇩

- Devise a different protocol

impossible if deterministic and entities do not have unique identifiers

OR

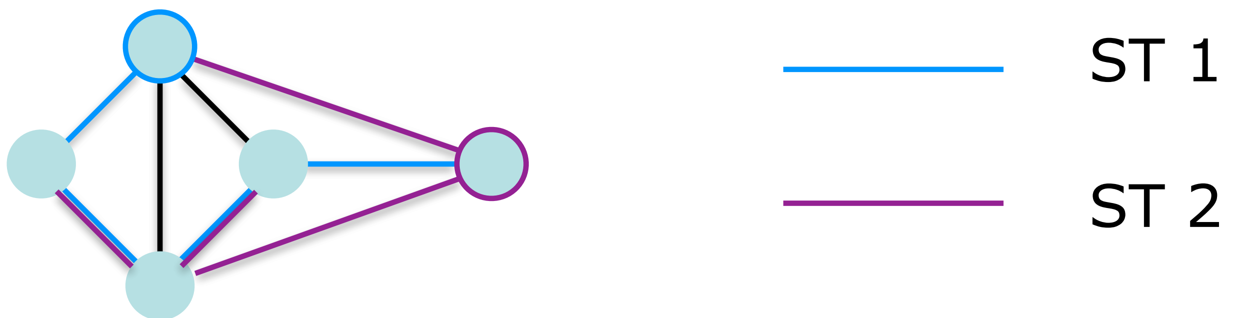
- <u>ELECT an entity (LEADER)</u> to be the unique initiator

Spanning Tree with multiple initiators

Additional restriction: UNIQUE IDs

IDEA 1: Multiple Spanning Tree

Each initiator constructs its own spanning tree with a single-initiator protocol and uses the IDs of the initiators to distinguish between the different constructions



Message cost depends on the number of initiators and used protocol.
In general is expensive.

Spanning Tree with multiple initiators

Additional restriction: UNIQUE IDs

IDEA 2: Selective construction

Each initiator starts the construction of its own spanning tree with a single-initiator protocol using their IDs to identify their spanning tree. Entities will eventually stop working for all but one constructions, keeping the spanning tree of the initiator with smaller ID.

Entities might re-execute the protocol several times

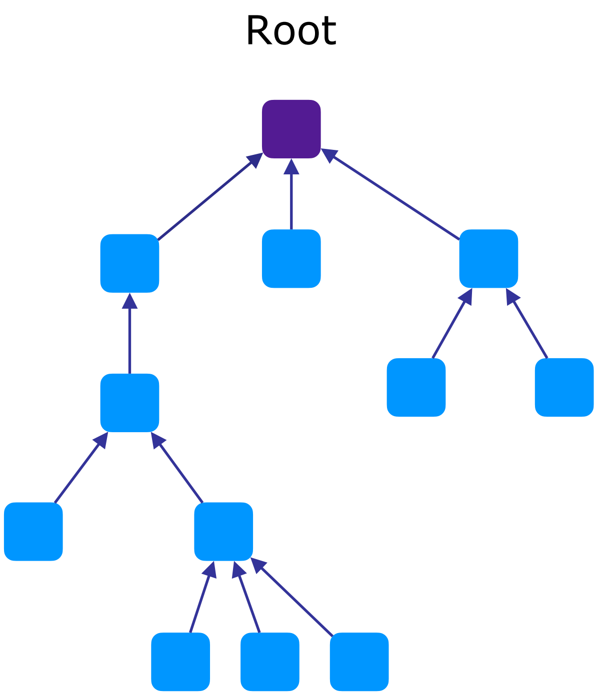
Need of a termination notification

Before talking about leader election…

Computation in Trees

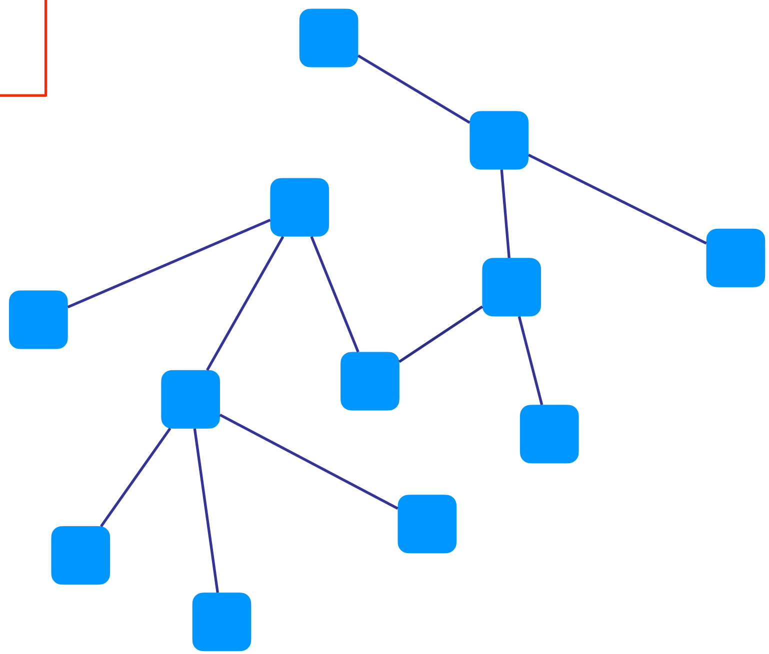
Entities are aware of belonging to a tree network

Acyclic graph
n entities
n-1 edges



Rooted Tree

Sense of direction: up-down



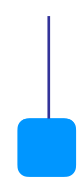
Unrooted Tree

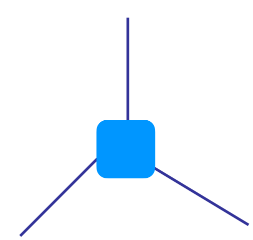
Computation in Unrooted Trees

Restrictions
Bidirectional links
Connectivity
FIFO messages
Full reliability
Knowledge of the topology

Each entity knows the network is a tree

Each entity knows if

it is a leaf:  Only one neighbour

or an internal node: 

Computation in Unrooted Trees
SATURATION TECHNIQUE

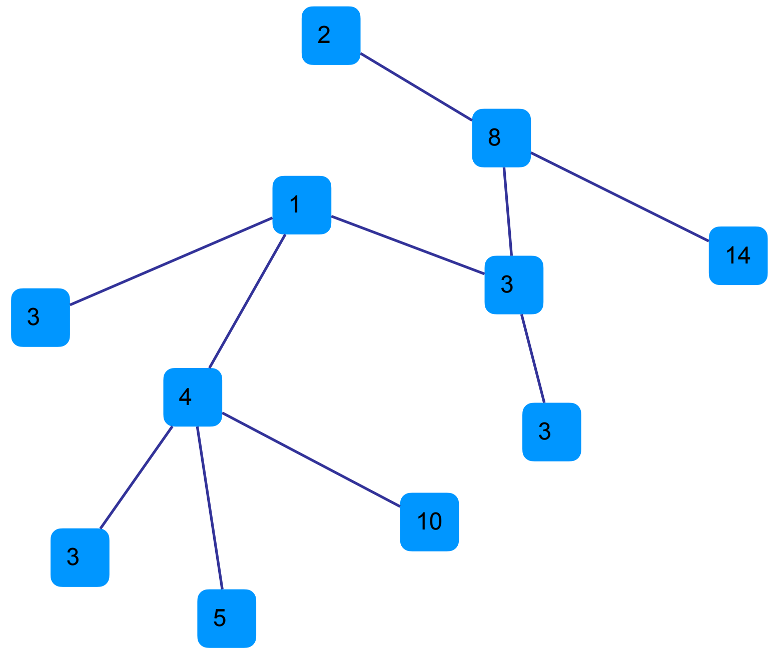
EXAMPLE: Minimum Finding

Every entity x has an input value v(x)
(not necessarily distinct)

At the end each entity must know if its
value is the smallest or not

Computation in Unrooted Trees
SATURATION TECHNIQUE

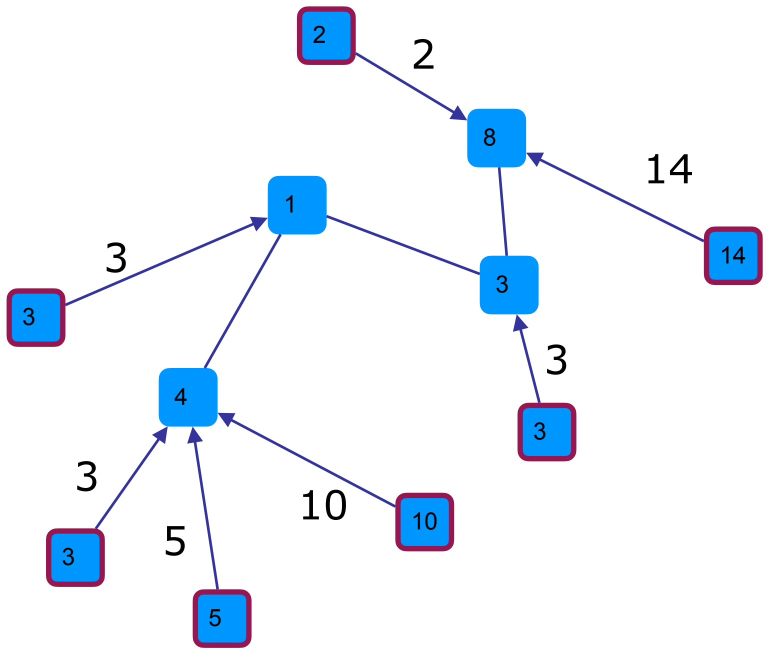
EXAMPLE: Minimum Finding



Leaves start the computation sending their value to their unique neighbor

Computation in Unrooted Trees
SATURATION TECHNIQUE

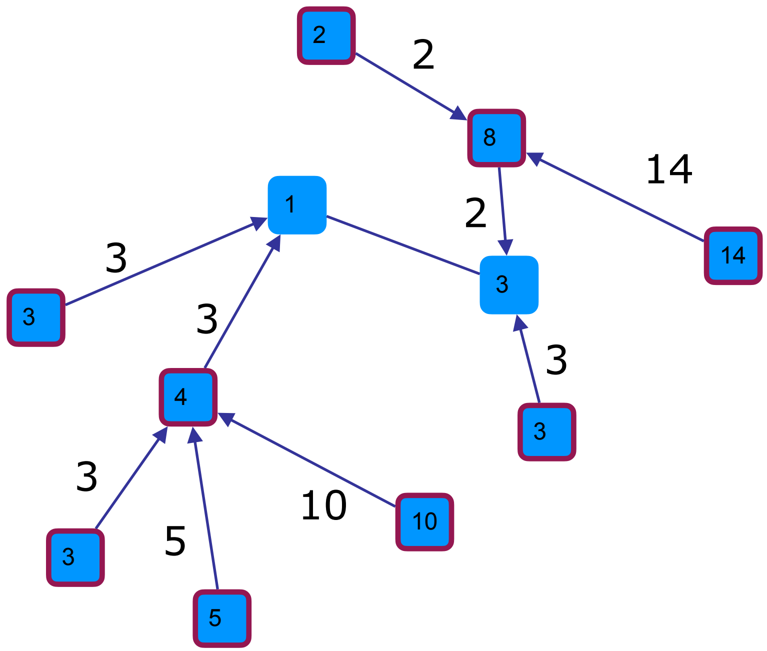
EXAMPLE: Minimum Finding



Internal entities wait for all but one neighbor to send a message,
then compute minimum and send to last neighbor

Computation in Unrooted Trees
SATURATION TECHNIQUE

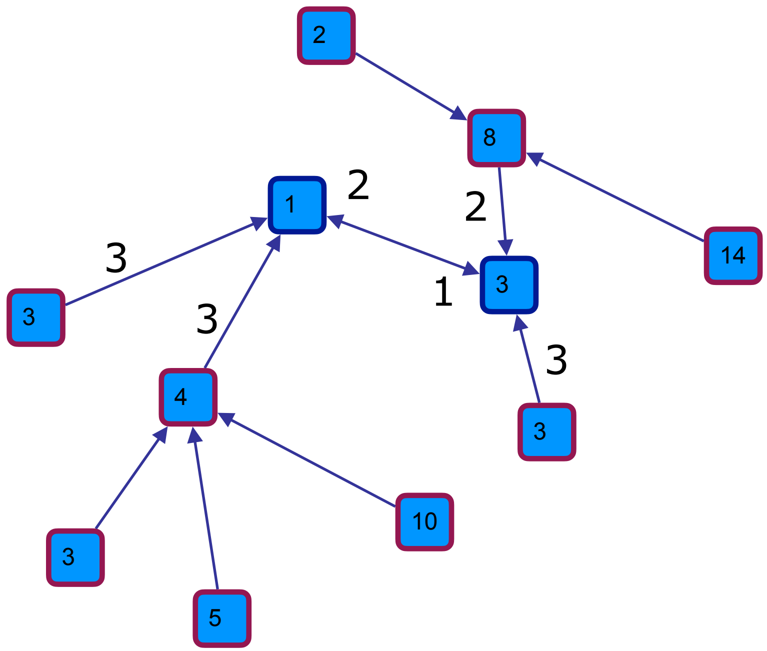
EXAMPLE: Minimum Finding



Internal entities wait for all but one neighbor to send a message,
then compute minimum and send to last neighbor

Computation in Unrooted Trees
SATURATION TECHNIQUE

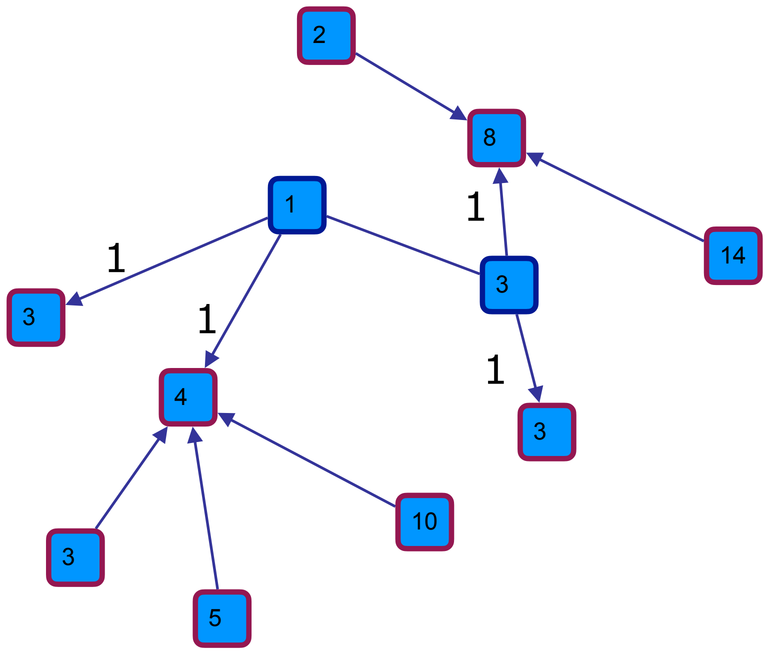
EXAMPLE: Minimum Finding



Two entities receive a message from all neighbors and send the minimum back

Computation in Unrooted Trees
SATURATION TECHNIQUE

EXAMPLE: Minimum Finding

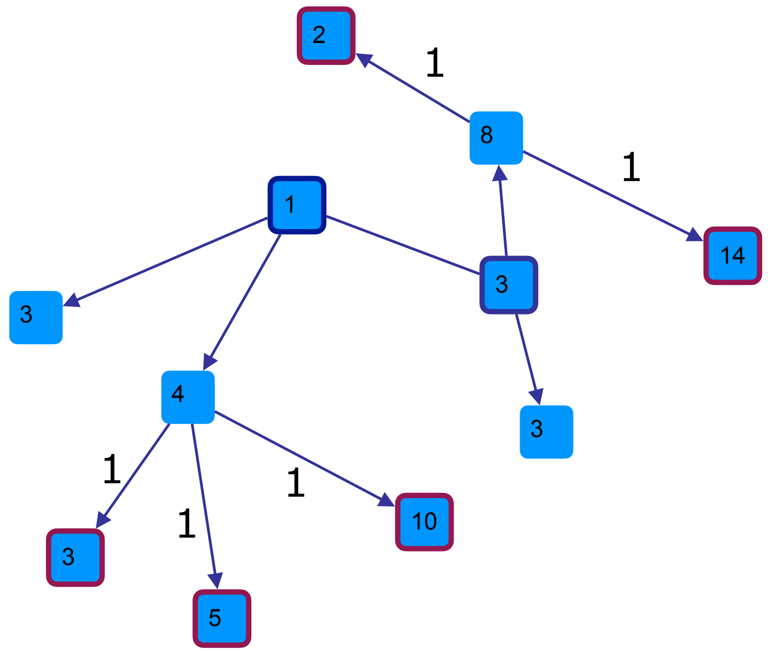


All internal entities send the message containing the minimum to the neighbors it first received messages from

UNIMORE

Computation in Unrooted Trees
SATURATION TECHNIQUE

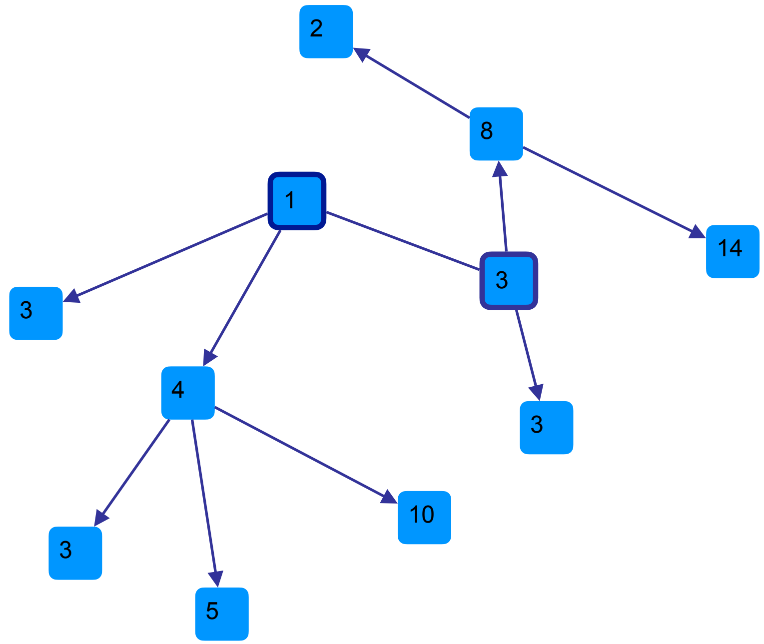
EXAMPLE: Minimum Finding



All internal entities send the message containing the minimum to the neighbors it first received messages from

Computation in Unrooted Trees
SATURATION TECHNIQUE

EXAMPLE: Minimum Finding



At the end, all entities know the minimum value and can decide (if they hold it or not)

Computation in Unrooted Trees
SATURATION TECHNIQUE

Full Saturation

Can be autonomously and independently started by any number of initiators

Activation stage:
Started by all initiators: all entities are activated

Saturation stage:
- Started by leaves
- At the end, one pair of neighbor entities is selected

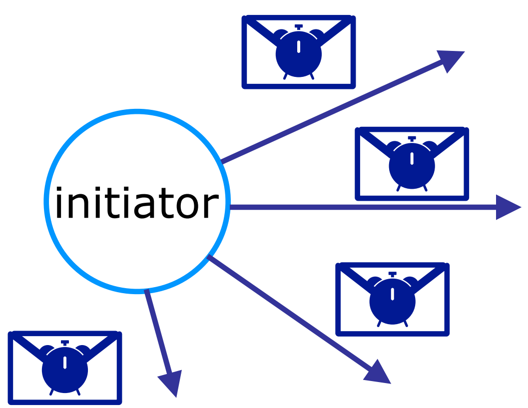
Resolution stage:
- Started by selected (saturated) entities

Computation in Unrooted Trees
SATURATION TECHNIQUE

Activation stage:
Started by all initiators: all entities are activated

Wake-up started by initiators

Within a finite time all entities become active



UNIMORE

Computation in Unrooted Trees
SATURATION TECHNIQUE

Saturation stage:
- Started by leaves
- At the end, one pair of neighbor entities is selected

Leaves send a saturation message to their only neighbor

Internal entities wait |N(.)|-1 saturation messages and then send a saturation message to the remaining neighbor

Computation in Unrooted Trees
SATURATION TECHNIQUE

Saturation stage:
- Started by leaves
- At the end, one pair of neighbor entities is selected

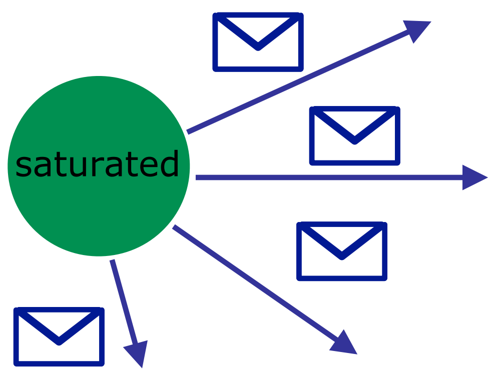
If a processing entity receives a saturation message it becomes saturated

Computation in Unrooted Trees
SATURATION TECHNIQUE

Resolution stage:
Started by selected (saturated) entities

Depends on the application

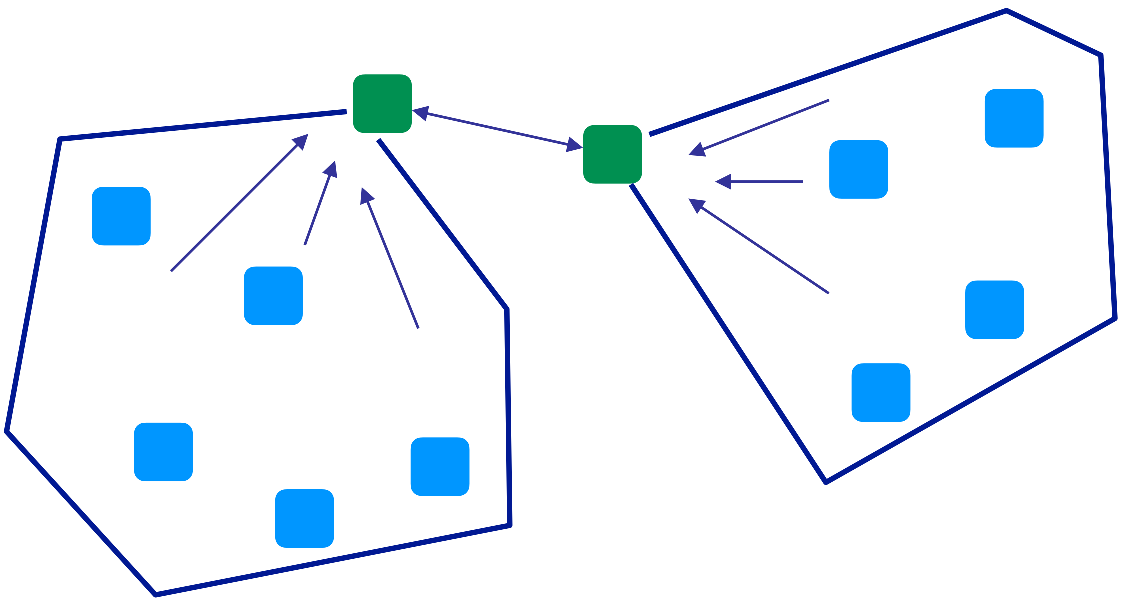
Usually is a notification



Computation in Unrooted Trees
SATURATION TECHNIQUE

LEMMA

Exactly two processing entities become saturated, and they are neighbors

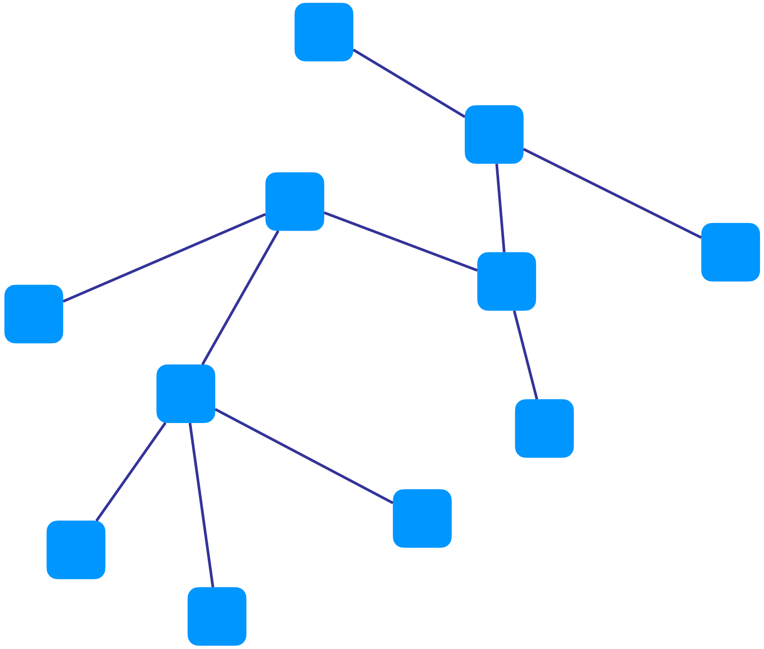


Different execution might result into a different pair of saturated entities, depending on communication delays

Computation in Unrooted Trees
SATURATION TECHNIQUE

MESSAGE COMPLEXITY

Activation:



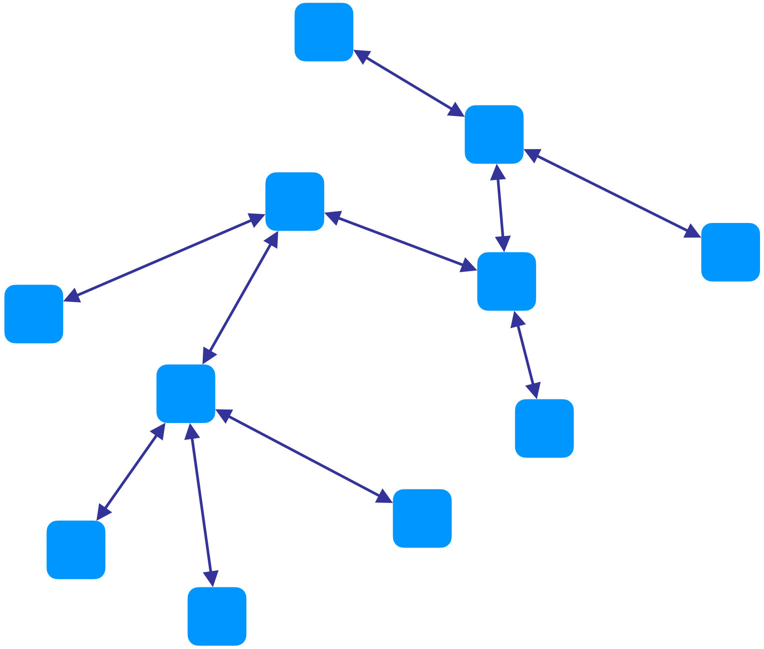
Saturation:

Resolution:

Computation in Unrooted Trees
SATURATION TECHNIQUE

MESSAGE COMPLEXITY

Activation: Worst Case
n initiators



Saturation:

Resolution:

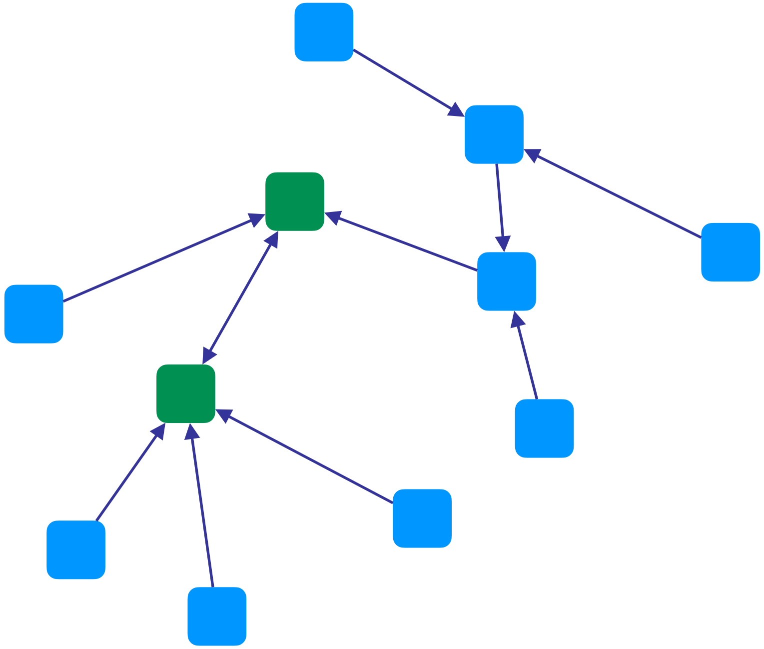
2 (n-1) messages

Computation in Unrooted Trees
SATURATION TECHNIQUE

MESSAGE COMPLEXITY

Activation: $\leq 2(n-1)$

Saturation:



Resolution:

$(n-1) + 1 = n$ messages

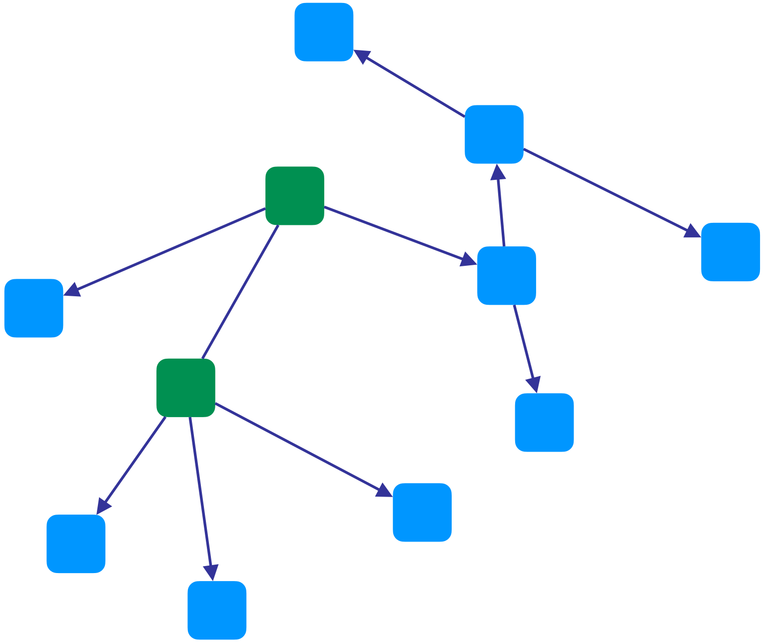
Computation in Unrooted Trees
SATURATION TECHNIQUE

MESSAGE COMPLEXITY

Activation: ≤ 2(n-1)

Saturation: n

Resolution:



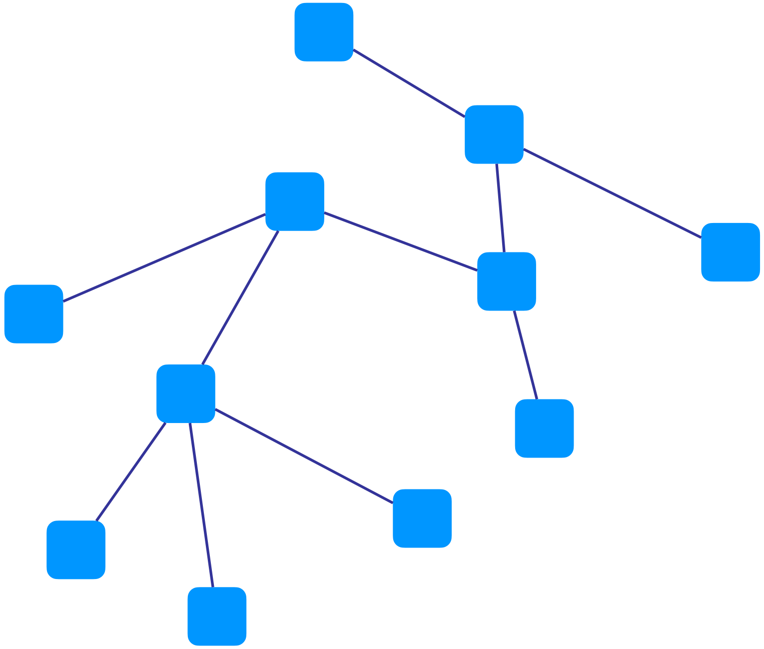
n-2 messages

UNIMORE

Computation in Unrooted Trees
SATURATION TECHNIQUE

MESSAGE COMPLEXITY

Activation: $\leq 2(n-1)$



Saturation: n

Resolution: $n-2$

Total $\leq 2(n-1) + n + n - 2 = 4n - 4$

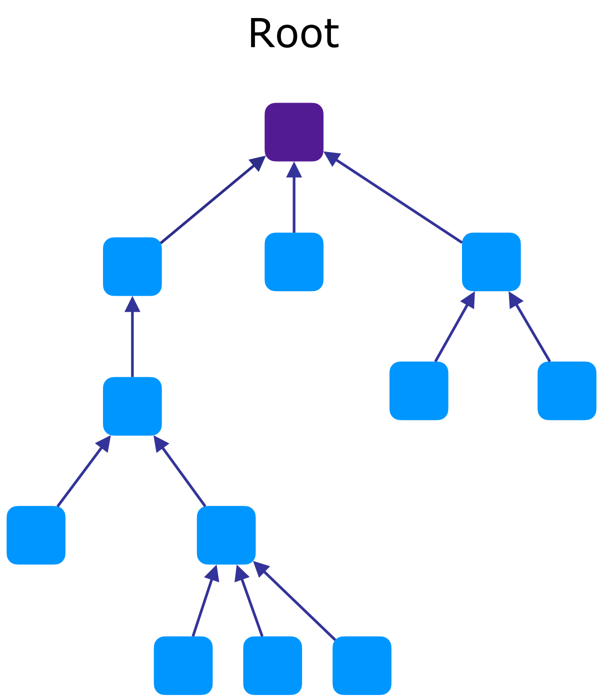
Computation in Unrooted Trees
SATURATION TECHNIQUE

Saturation can be used to solve a wide set of problems

Distributed Function Evaluation
compute a function whose arguments
are initially distributed among entities

Minimum Finding
Cardinal statistics
Find eccentricity
Center finding
Finding a median
Finding diametral path

Computation in Rooted Trees



Rooted Tree

Sense of direction
up-down

Each entity knows who are the children and who is the parent

There is a natural leader, the root

Protocols are started by the root with a broadcast

"Saturation" of the root is achieved by convergecast

Convergecast:
1. Leaves send msg to parent
2. internal entities send to parent after receiving from all children

Theorem
Without unique ID it is impossible to root an unrooted tree