The Model & Basic Computations

Chapter 1 and 2

The Model

Broadcast

Spanning Tree Construction

Traversal

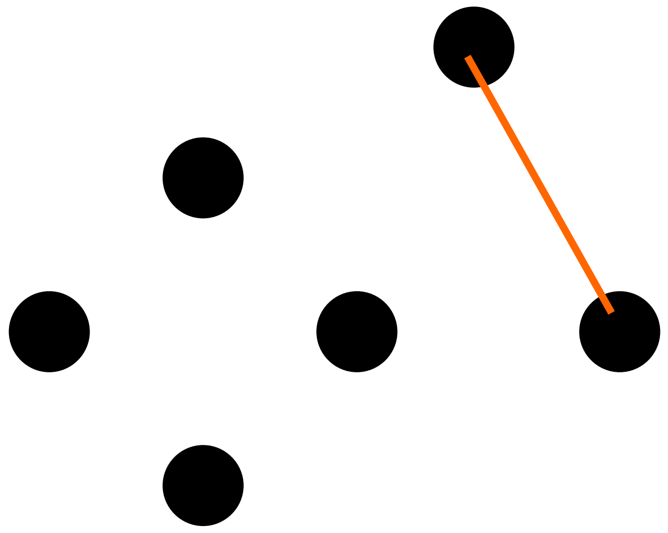
Wake-up

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 $E' \subseteq E$.

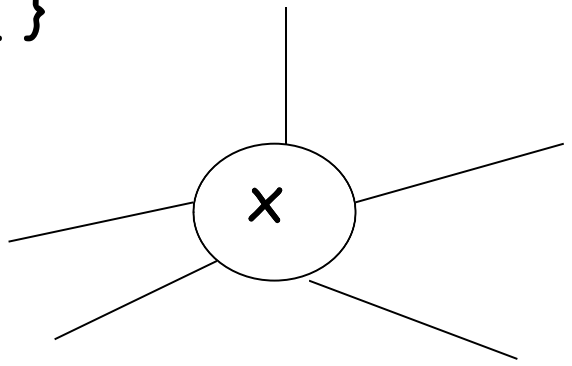
Assumptions:

single initiator
bidirectional links
total reliability
G connected



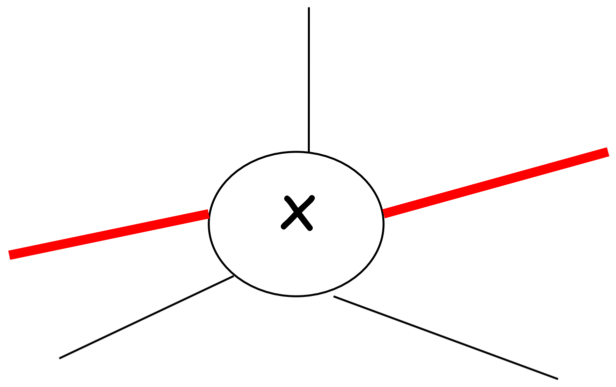
Protocol SHOUT

Initially: ∀ x, Tree-neighbors(x) = { }



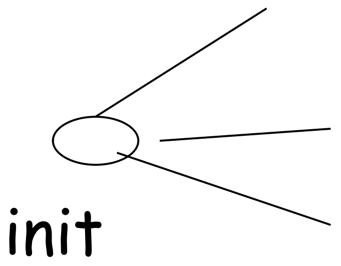
At the end:

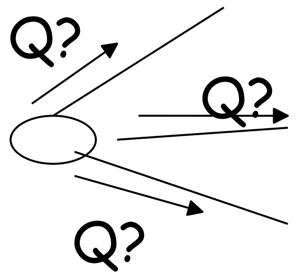
∀ x, Tree-neighbors(x) = {links that belong to the spanning tree }



Tree-neighbors(x) is a subset of N(x)

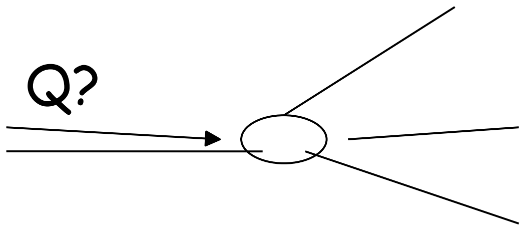
1.



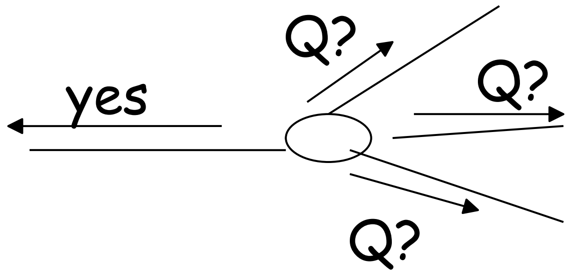


Q? = do you want to be my neighbour in the spanning tree ?

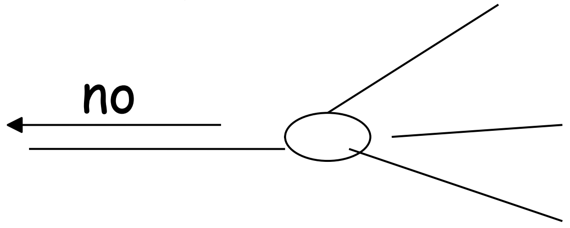
2.



If it is the first time:



If I have already answered yes to someone else:



States S={INITIATOR, IDLE, ACTIVE, DONE}
Sinit = {INITIATOR, IDLE}
Sterm = {DONE}
Restrictions = **R**;UI   **R**={BL, TR, CN}

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

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

```
ACTIVE

receiving(Q)
        send(no) to sender

receiving(yes)
        Tree-neighbours:=
                Tree-neighbours U sender
        counter := counter +1
        if counter = |N(x)|
                become DONE

receiving(no)
        counter := counter +1
        if counter = |N(x)|
                become DONE
```

Note:

SHOUT = FLOOD + REPLY

Correctness and Termination

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

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

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

$$\Rightarrow$$

The spanning graph defined by the Tree-neighbour relation is connected, acyclic and contains all the entities

Note: local termination

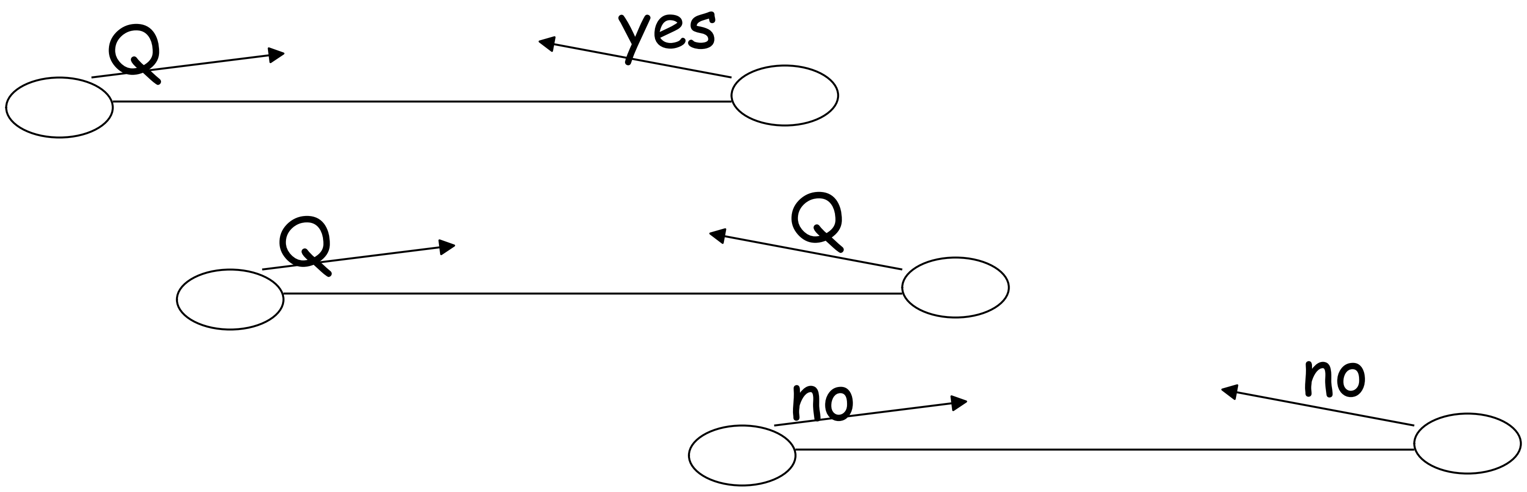
Message Complexity

SHOUT = FLOOD + REPLY

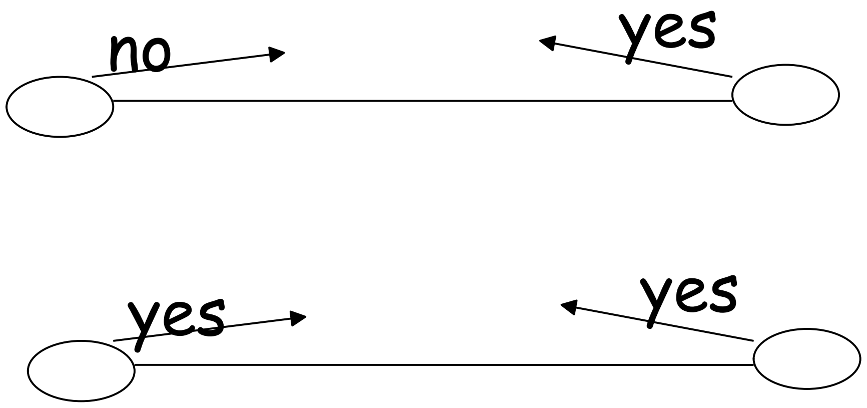
$\Rightarrow$

Messages(SHOUT) = 2 M(FLOOD)

Possible situations

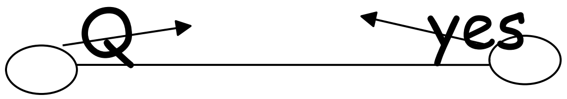


Impossible situations



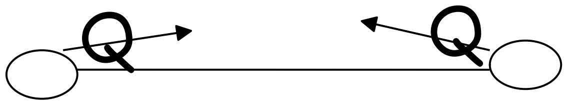
Message Complexity - worst case

Total n. of Q:



(n-1)

only one Q on the ST links



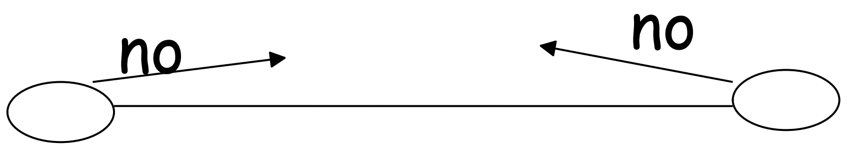
m -(n-1)

on the other links

Total: $2(m - (n-1)) + (n-1)$
$= 2m - n + 1$

Message Complexity - worst case

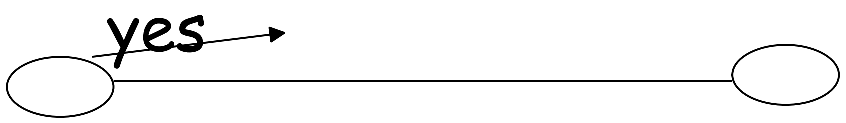
Total n. of NO:



as many as Q---Q

2(m - (n-1))

Total n. of YES:



Exactly: (n-1)

Message Complexity - worst case

$2m - n + 1 + 2(m - (n-1)) + n-1$
$= 2m - n + 1 + 2m - 2n + 2 + n - 1$
$= 4m - 2n + 2$

Messages(SHOUT) = $4m - 2n + 2$

In fact: M(SHOUT) = 2 M(FLOOD) = $2(2m-n+1)$

$\Omega(m)$ is a lower bound also in this case
therefore SHOUT is optimal asympotically

Spanning Tree Construction

Without “NO”

Protocol SHOUT+

States S={INITIATOR, IDLE, ACTIVE, DONE}
Sinit = {INITIATOR, IDLE}
Sterm = {DONE}

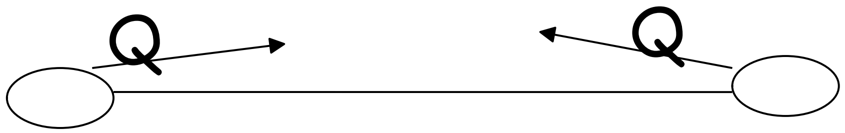
```
INITIATOR
Spontaneusly
        root:= true
        Tree-neighbours := { }
        send(Q) to N(x)
        counter:=0
        become ACTIVE
```

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

ACTIVE

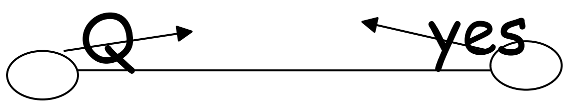
```
receiving(Q) (to be interpreted as NO)

        counter := counter +1
        if counter = |N(x)|
                become DONE

receiving(yes)
        Tree-neighbours:=
                Tree-neighbours U {sender}
        counter := counter +1
        if counter = |N(x)|
                become DONE
```

On each link there will be exactly 2 messages:



either

or



Messages(SHOUT+) = 2m

Much better than:

Messages(SHOUT) = 4m -2n + 2

Spanning Tree Construction

With Notification

States S={INITIATOR, IDLE, ACTIVE, DONE}
Sinit = {INITIATOR, IDLE}
Sterm = {DONE}

```
INITIATOR
Spontaneusly

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

```
IDLE
receiving(Q)

        root:= false
        parent := sender
        Tree-neighbours := {sender}
        send(yes) to sender
        counter := 1
        ack-counter:= 0
        if counter = |N(x)| then
                CHECK
        else
                send(Q) to N(x) – {sender}
        become ACTIVE
```

ACTIVE

```
receiving(Q)
        counter := counter +1
        if counter = |N(x)| and not root then
                CHECK

receiving(yes)
        Tree-neighbours :=  Tree-neighbours U {sender}
        counter := counter +1
        if counter = |N(x)| and not root then
                CHECK
```

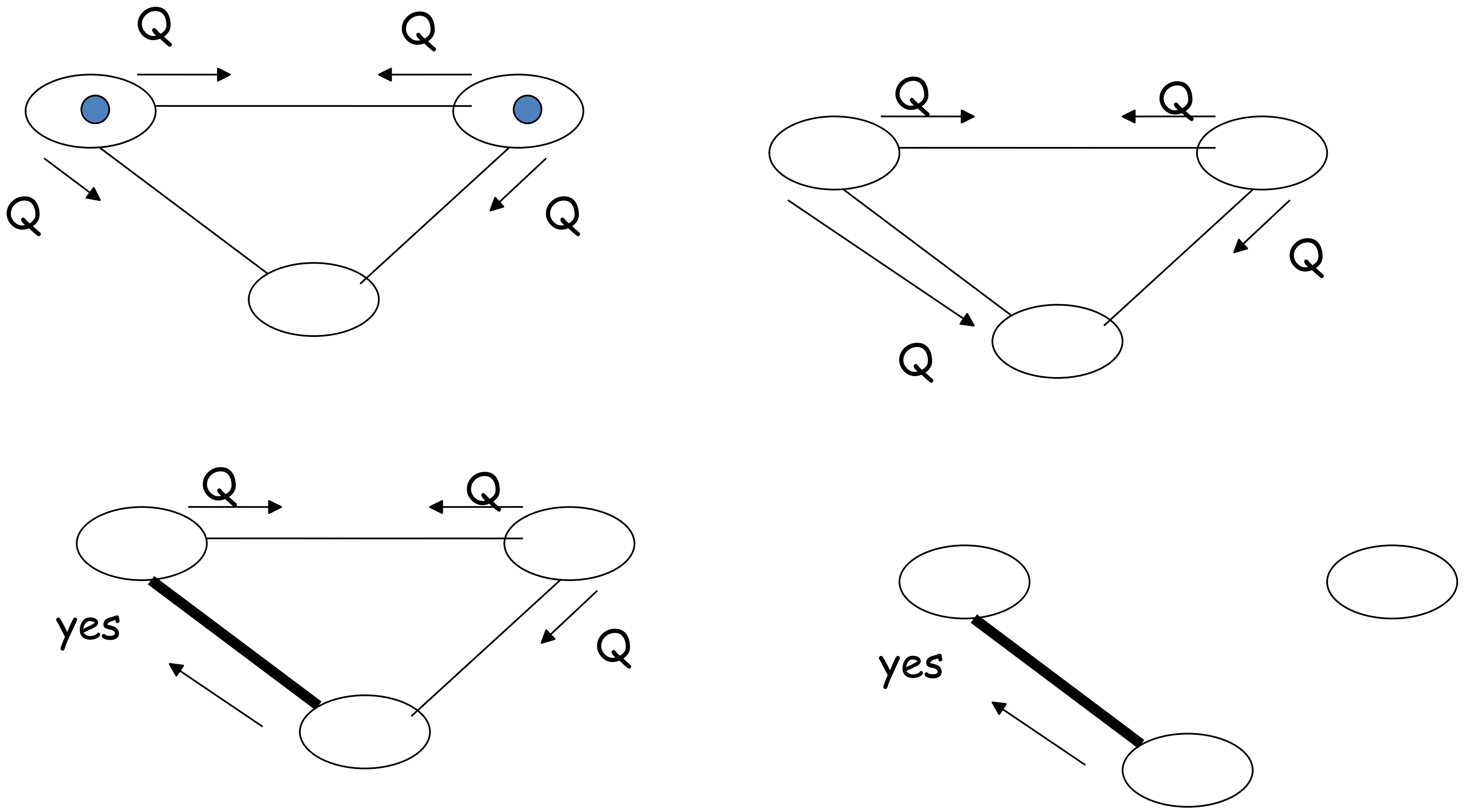
ACTIVE (cont)

```
receiving(Ack)
        ack-counter:= ack-counter +1
        if counter = |N(x)| // indicate tree-neighbors is done
                if root then
                    if ack-counter = |Tree-neighbours|
                        send(Terminate) to Tree-neighbours
                        become DONE
                else if ack-counter = |Tree-neighbours| - 1
                        send(Ack) to parent

receiving(Terminate)
        send(Terminate) to Children
        become DONE

        // Children is Tree-neighbours – {parent}
```

CHECK

```
If  I am a leaf
            (* that is:  Children:= Tree-neighbours – {parent}
                         if Children = emptyset   *)
  send(Ack) to parent
```

What happens if there are multiple initiators ?



Impossibility Result

Theorem
The SPT problem is deterministically unsolvable under **R**

no deterministic protocol that always correctly terminates within finite time

The idea:

- **all initiator entities start in the same state and maintain it for the whole execution**
- **a correct solution requires entities end up with different states**

An election is needed to have a unique initiator.

or

Another protocol has to be devised.

NOTE: Election is impossible if the nodes do not have distinct IDs

Traversal
Depth First Search

Assumptions

Single initiator
Bidirectional links
No faults
G connected

S = {INITIATOR, VISITED, DONE}

One version

1) When first visited, remember who sent,
   forward the token to one of the unvisited neighbours
   wait for its reply

2) When neighbour receives,
   if already visited, it will return the token saying it is a
   back edge
   otherwise, will forward it (sequentially)
   to all its unvisited neighbour before returning it

3) If there are no more unvisited neighbours, return the token (reply)
   to the node from which it first received the token

4) Upon reception of reply, forward the token to another
   unvisited neighbour

States S={INITIATOR, IDLE, VISITED, DONE}
Sinit = {INITIATOR, IDLE}
Sterm = {DONE}

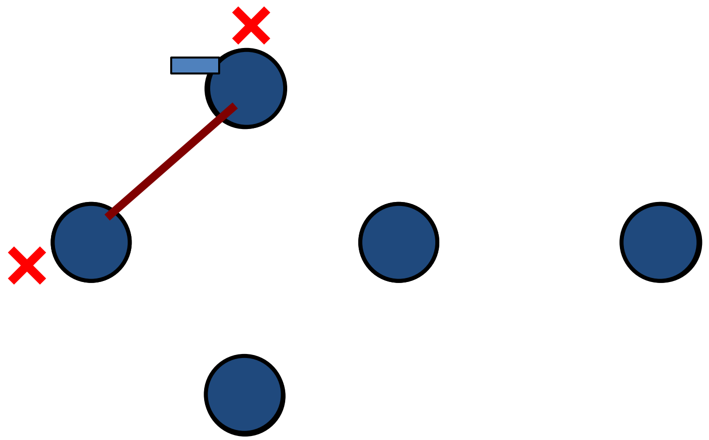
```
INITIATOR
Spontaneusly

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

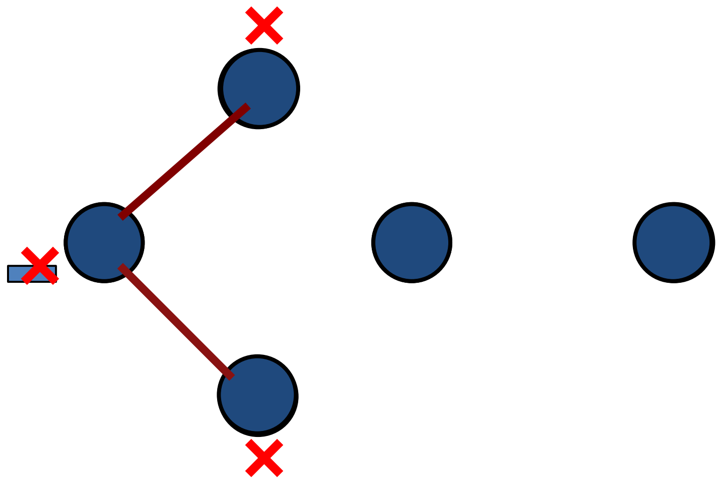
```
IDLE
receiving(T)    // T token message

        entry := sender
        Unvisited := N(x) - sender
        initiator := false
        VISIT
```

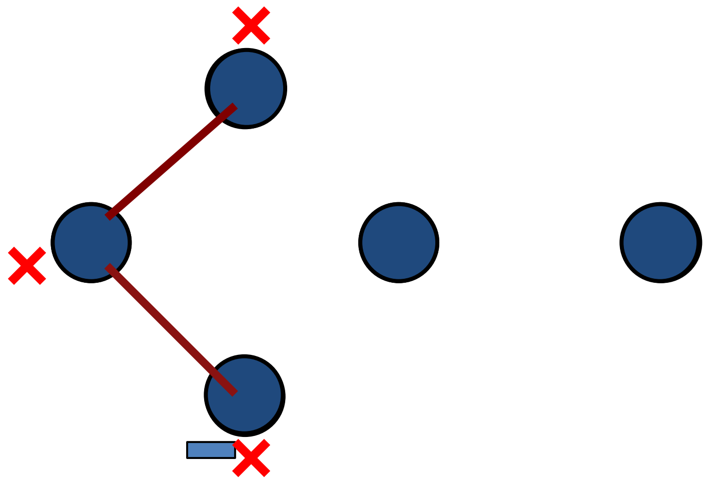
States S={INITIATOR, IDLE, VISITED, DONE}
Sinit = {INITIATOR, IDLE}
Sterm = {DONE}

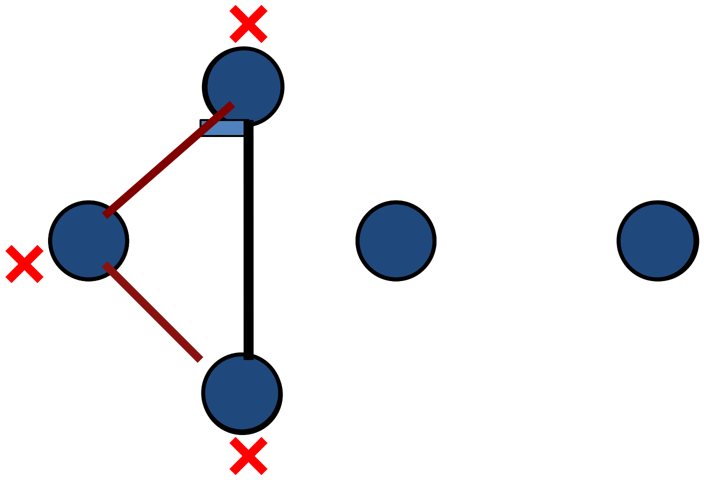
```
VISIT

    if Unvisited ≠Ø
            next <= Unvisited
            send(T) to next
            become VISITED
    else
            if not initiator
                send(RET) to entry
            become DONE
```

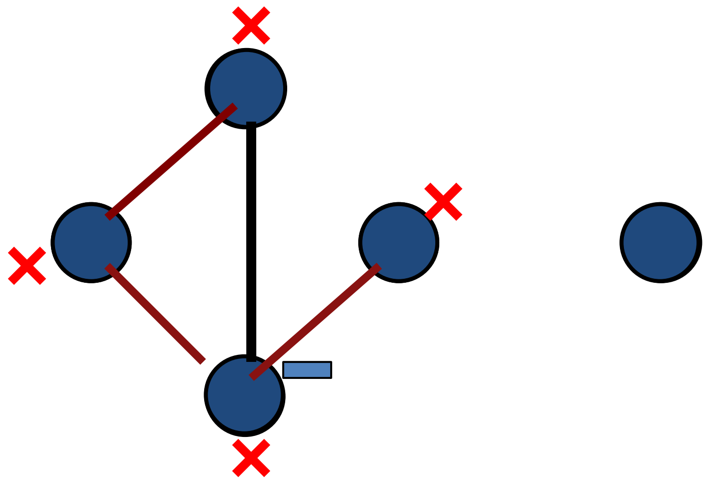
```
VISITED
receiving(T)    // T token message

    Unvisited := Unvisited - sender
    send(BEGDE) to sender

receiving(RET)
    VISIT

receiving(BEDGE)
    VISIT
```

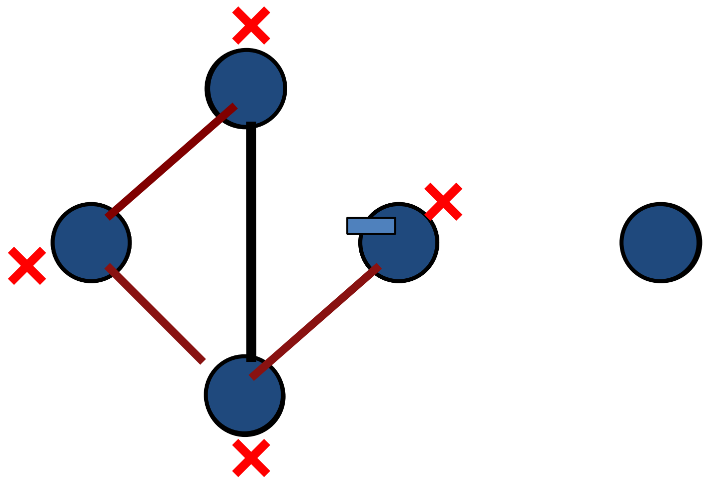


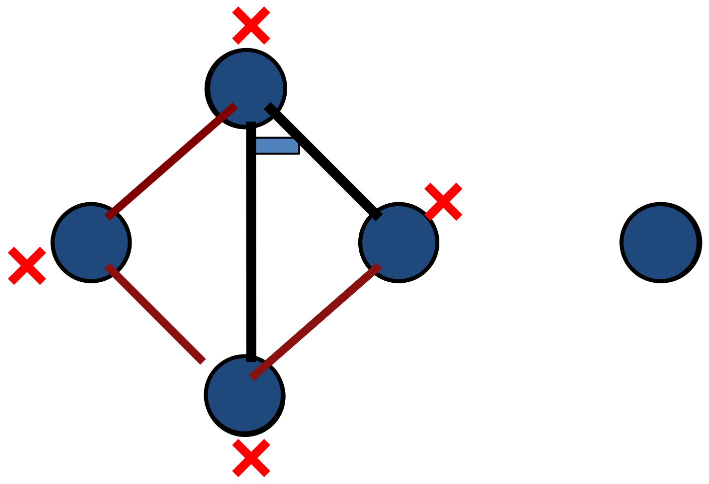


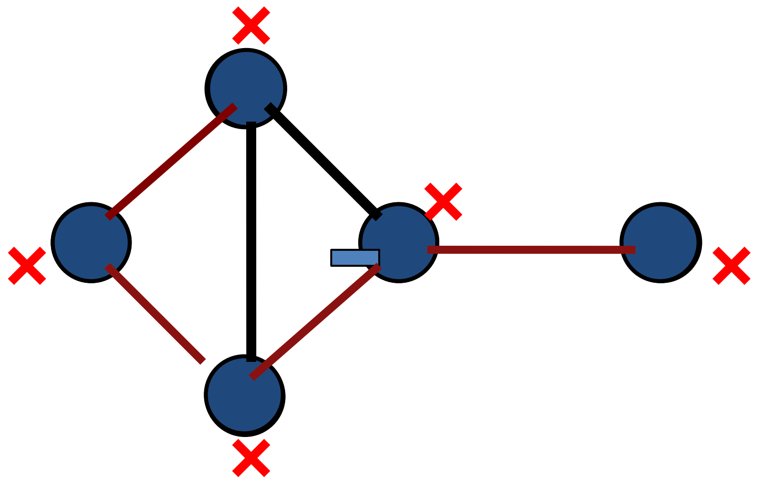


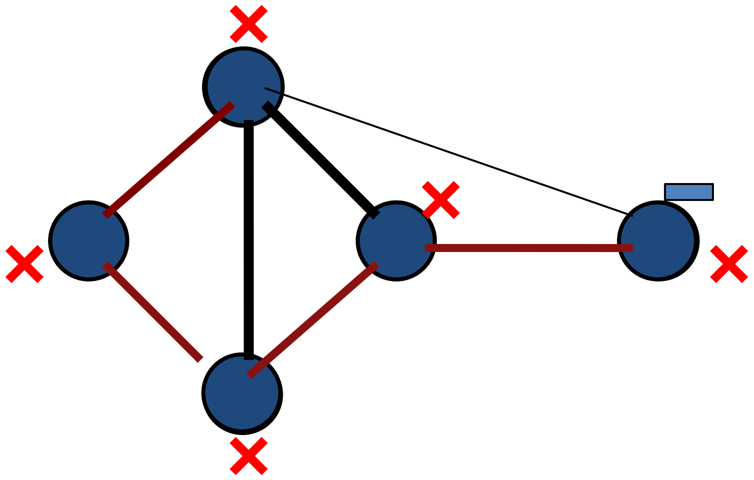


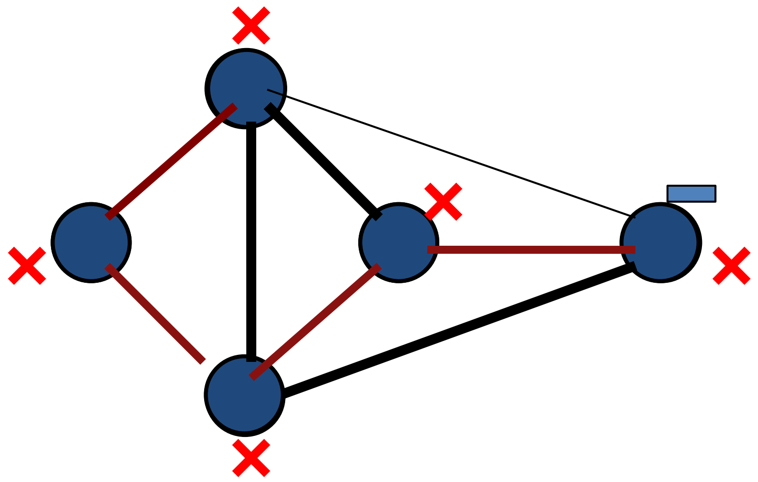


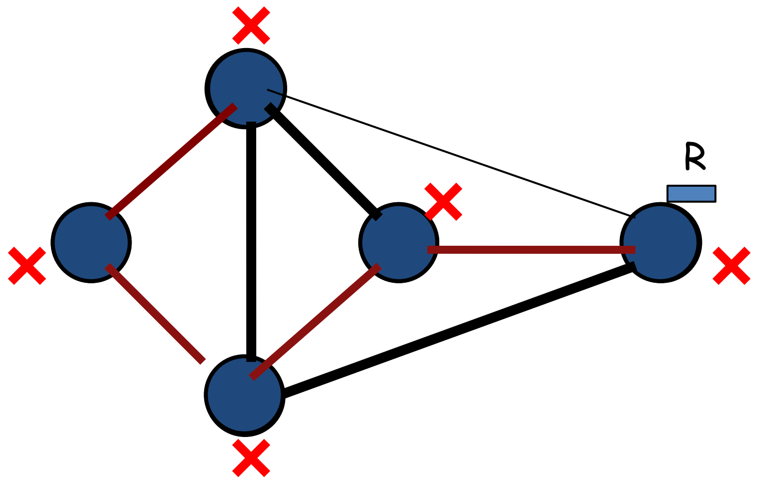


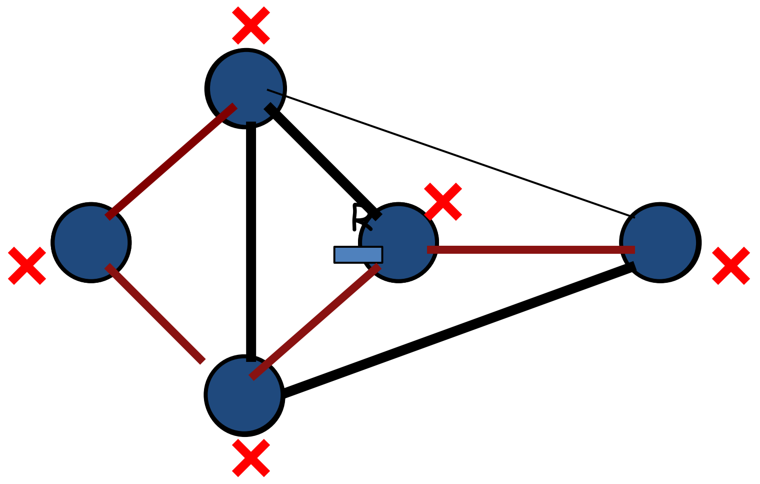








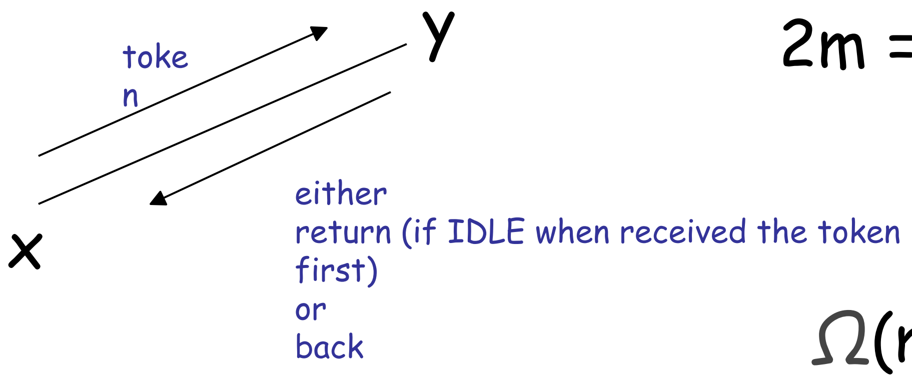

R


R

Complexity

Message Complexity:

Type of messages: `token, back, return`



$2m = O(m)$

$\Omega(m)$ is also a lower bound

Time Complexity:
(ideal time)

$2m = O(m)$

Totally sequential

$\Omega(n)$ is also a lower bound

Note:
most messages are on Back Edges

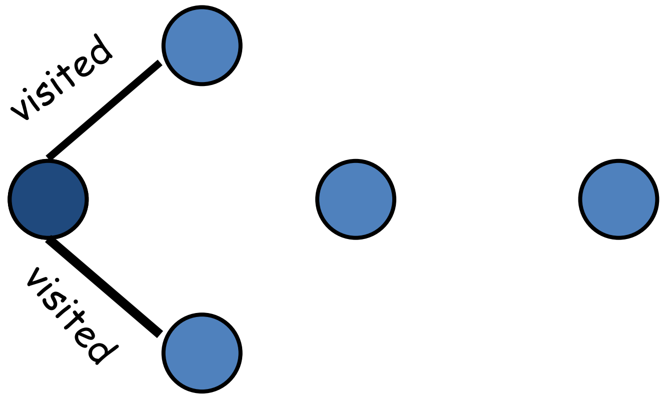
---> most time is spent on Back Edges

Idea: avoid sending messages on back edges

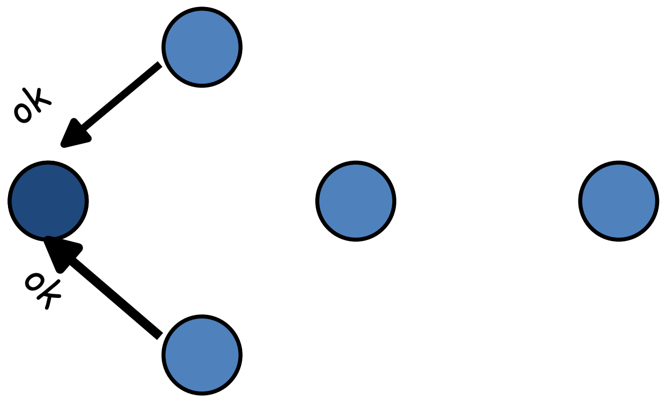
How ?

Using notification and ACK messages

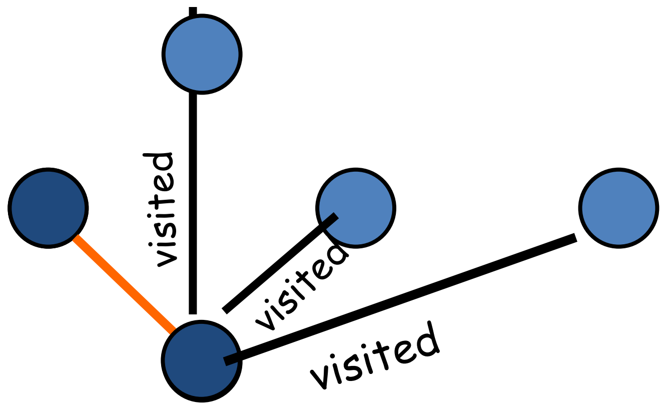
DF+ Improving Time



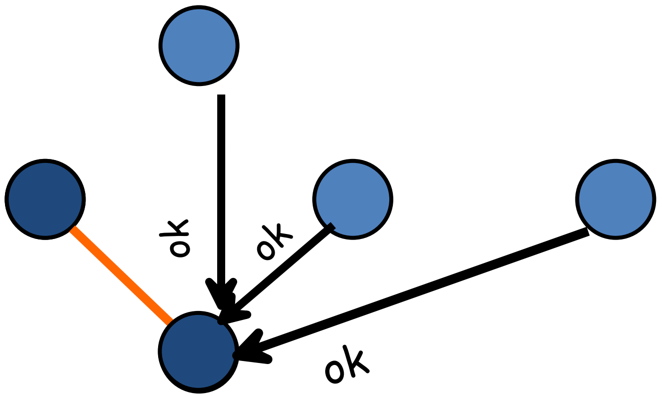
DF+ Improving Time



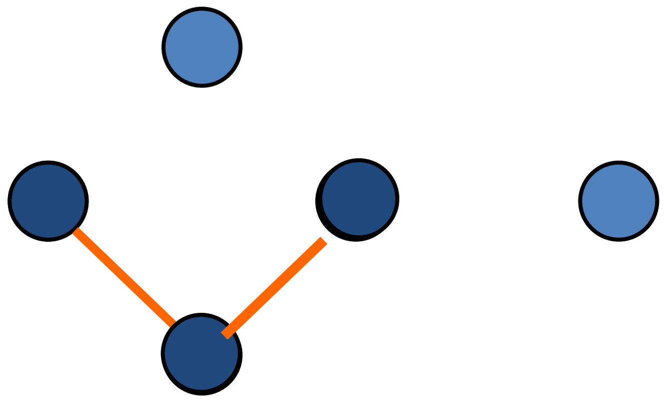
DF+ Improving Time



DF+ Improving Time



DF+ Improving Time



DF+ Complexity Message

Messages: **Token**, **Return**, **Visited**, **Ack (ok)**

Each entity (except init): receives 1 **Token**, sends 1 **Return**:

2(n-1)

Each entity:

1 **Visited** to all neighbours except the sender

Let s be the initiator

$$\Sigma |N(s)| + \Sigma_{x \neq s} (|N(x)|-1)$$

$$= \quad 2m - (n-1)$$

(same for **Ack**)

TOT: 4m

DF+ Complexity

Time (ideal time)

Token and Return are sent sequentially: 2(n-1)

Visited and Ack are done in parallel: 2n

TOT: 4n -2

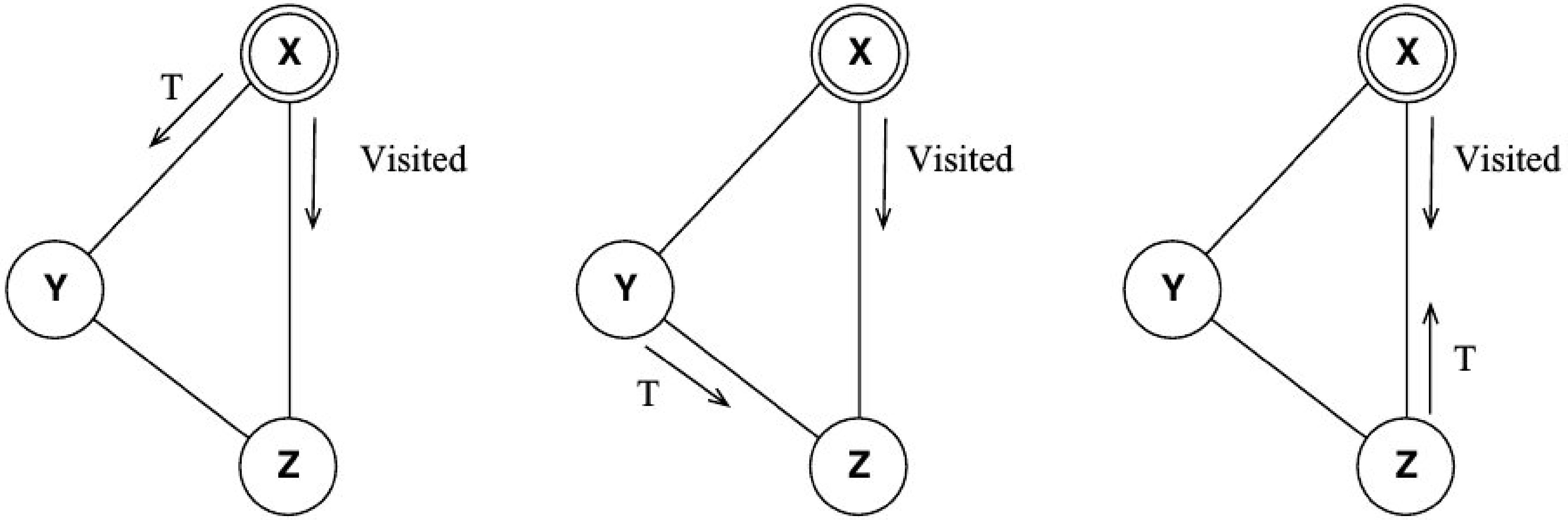
Summarizing:

DF Traversal

	Messages	Ideal Time
DF:	2m	2m
DF+:	4m	4n -2

DF++

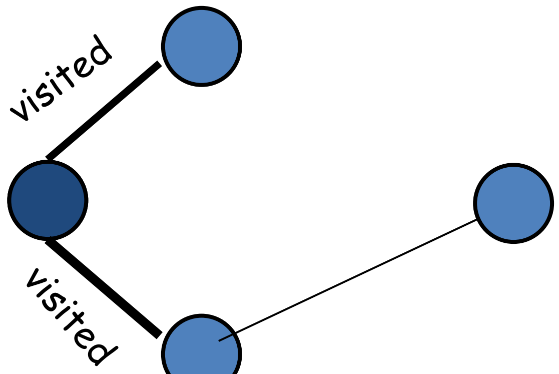
Do not send the `Ack`
What happens ?



DF++

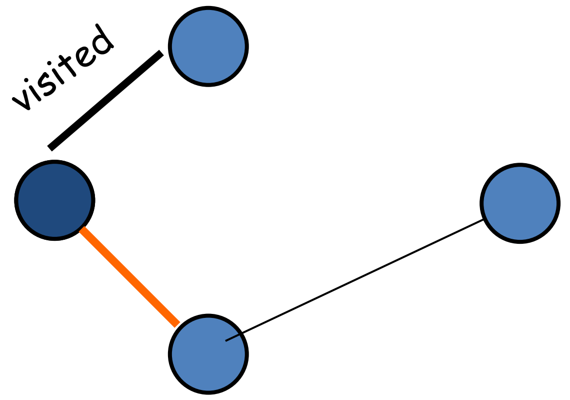
Do not send the `Ack`

What happens ?



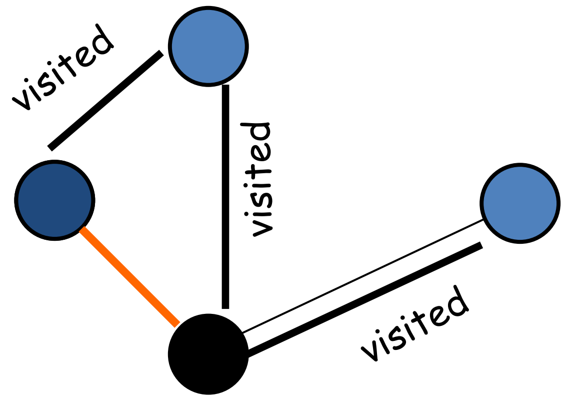
DF++

Do not send the `Ack`
What happens ?



DF++

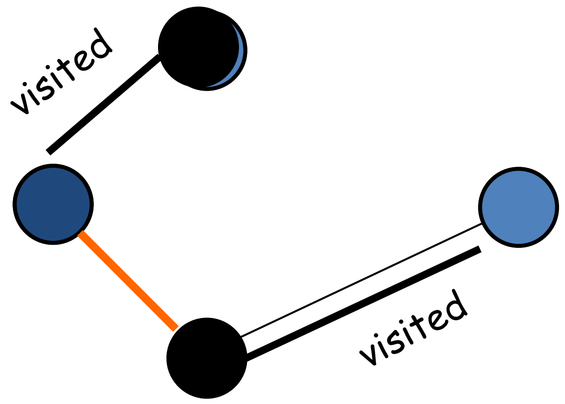
Do not send the `Ack`
What happens ?



DF++

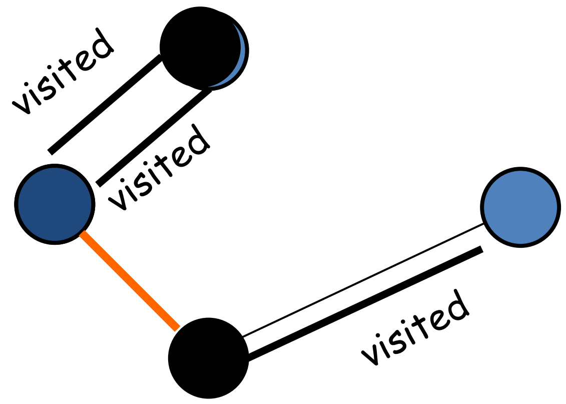
Do not send the `Ack`

What happens ?



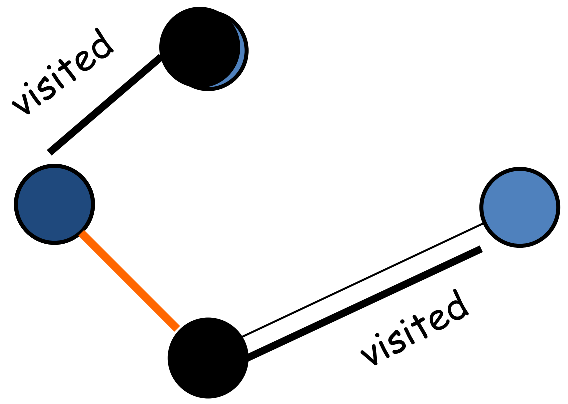
DF++

Do not send the `Ack`
What happens ?



DF++

Do not send the `Ack`
What happens ?



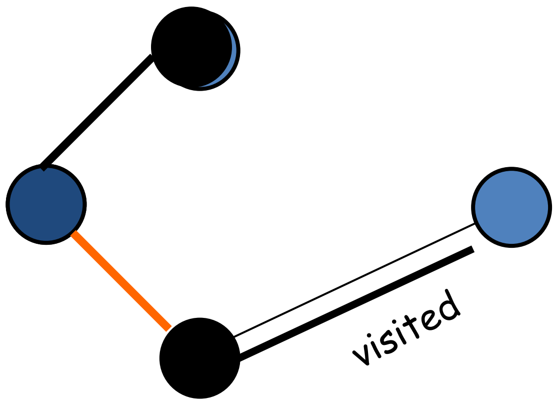
A token is sent to an already visited node (= back edge)

Both nodes will eventually understand the "mistake"

DF++

Do not send the `Ack`

What happens ?



A token is sent to an already visited node (= back edge)

Both nodes will eventually understand the "mistake"

and pretend nothing happened

DF++ Complexity

In the worst case there is a “mistake” on each link except for the tree links

T & Return = 2n - 2

Visited = 2m - n + 1

"Mistakes" = 2(m - n + 1)

Messages = 4m -(n-1)

BUT when we measure ideal time:

“mistakes” will not happen

Time = 2(n-1)

Summary

	Messages	Ideal Time
DF:	2m	2m
DF+:	4m	4n -2
DF++	4m-n+1	2n-1

Observations

An application:
    access permission problems, e.g., Mutual Exclusion

Any Traversal does a Broadcast (not very efficient)
The reverse is not true.

Another Traversal: Smart Traversal

1- Build a Spanning Tree with SHOUT+

Messages = 2m

2- Perform DF Traversal

Messages = 2(n-1)

Total Messages = 2(m+n-1)

Another Traversal: Smart Traversal

1- Build a Spanning Tree with SHOUT+

Time $\le$ d+1 d: diameter

2- Perform DF Traversal

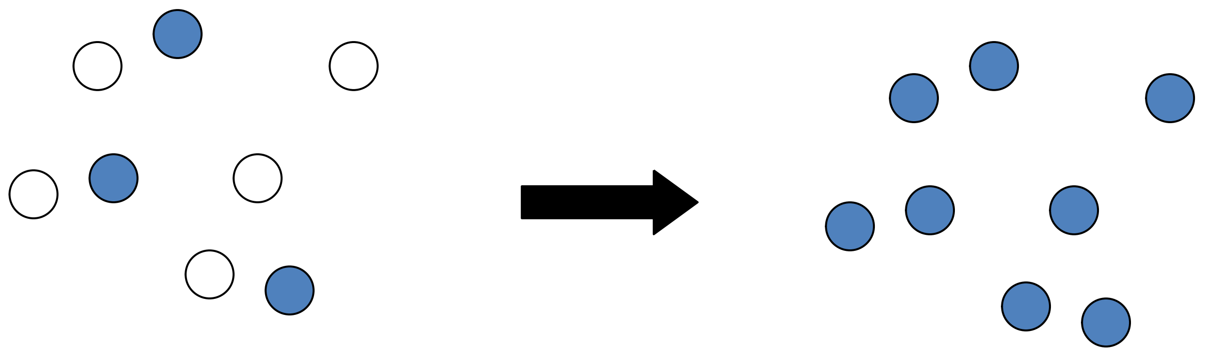
Time = 2(n-1)

Total Time $\le$ 2n+d-1

Summary

	Messages	Ideal Time
DF:	2m	2m
DF+:	4m	4n -2
DF++	4m-n+1	2n-1
Smart	2m+2n-2	2n+d-1

Computations with Multiple initiator: WAKE-UP



FLOOD solves the problem.

General FLOOD algorithm: $O(m)$

More precisely: $2m - n + k^*$

$\searrow$

n. of initiators

1 init = broadcast = $2m - n + 1$

All init = $2m$

States S={ASLEEP, AWAKE}
Sinit = {ASLEEP}
Sterm = {AWAKE}
Restrictions = **R**

```
ASLEEP
Spontaneously

    send(W) to N(x)
    become AWAKE

receiving(W)
    send(W) to N(x) - sender
    become AWAKE
```

Computations with Multiple initiator: WAKE-UP

In special topologies ?

TREE

Flood is optimal

$n + k^* - 2$

Computations with Multiple initiator: WAKE-UP

COMPLETE GRAPH

Broadcast		Wakeup	
Flood	Specific	Flood	Specific
$O(n^2)$	$O(n)$	$\Omega(n^2)$	

Need additional assumptions to reduce the complexity

HYPERCUBE

Broadcast		Wakeup	
Flood	Specific	Flood	Specific
$O(n \log n)$	$O(n)$	$\Omega(n \log n)$	