DoC 437 - 2009

Distributed Algorithms

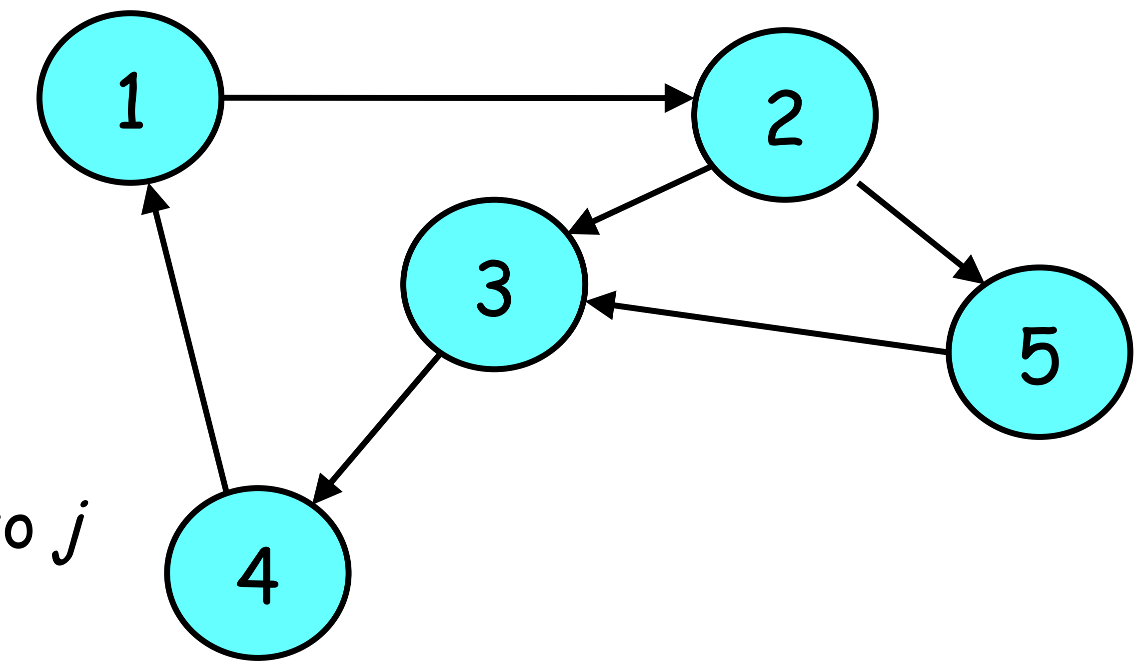
Part 2: Synchronous Algorithms

Synchronous algorithms

- Synchronous network model
- Leader election in a synchronous ring
  - impossibility result for identical processes
  - LCR algorithm
- Leader election in a general network
- Breadth-first search in a general network

Synchronous network model

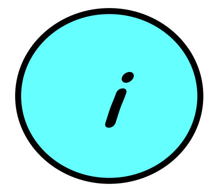
- Directed graph $G = (V, E)$
    - nodes V are *processes*
    - edges E are *channels* or *links*
- Some topologic properties
    - distance(i,j)
        - shortest path from i to j in G
    - diameter
        - maximum distance(i,j) over all pairs (i,j)



Processes

- Each process is modelled by a state machine

  (*states*, *start*, *msg*, *trans*)

  states: set of states

  start: initial states

  non-empty subset of states

  msg: message generation function

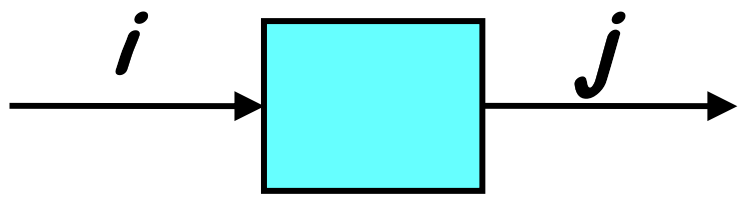
  maps current state to output messages

  trans: transition function

  maps current state and received messages to new state

Channels

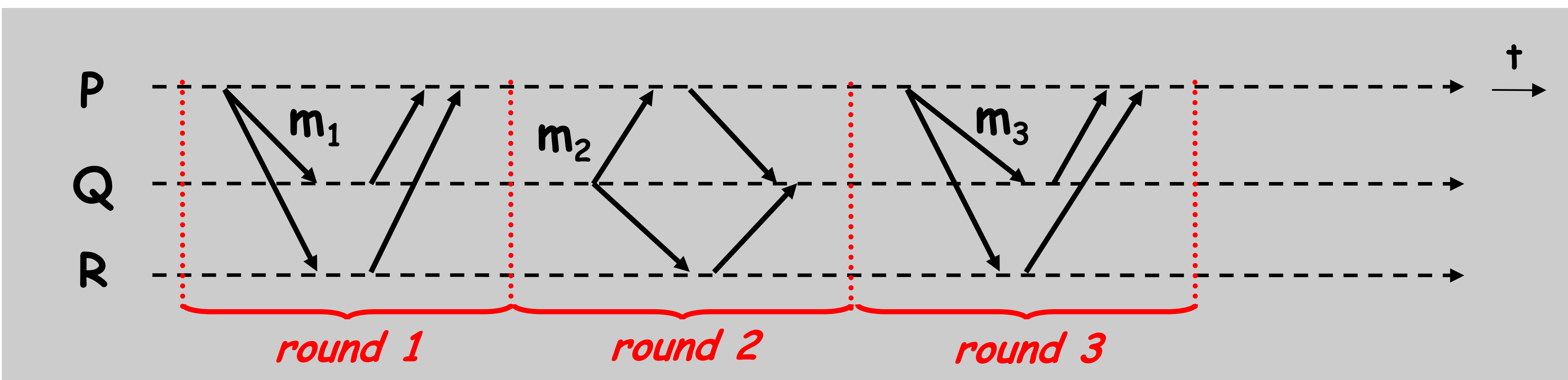
- Channel (associated with each edge in *G*) is a place holder for a single message from some fixed alphabet of messages *M* or *null*



- *null* indicates the absence of a message

Execution in the synchronous model

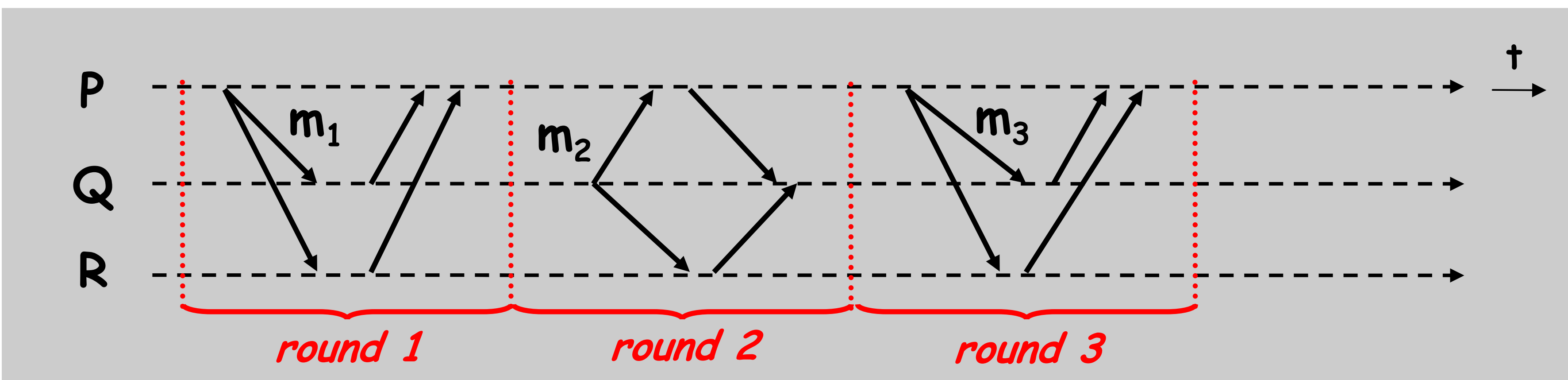
- Execution proceeds in *rounds*
- Initially all processes are in arbitrary start states and all channels are empty
- Processes then execute rounds in lock step



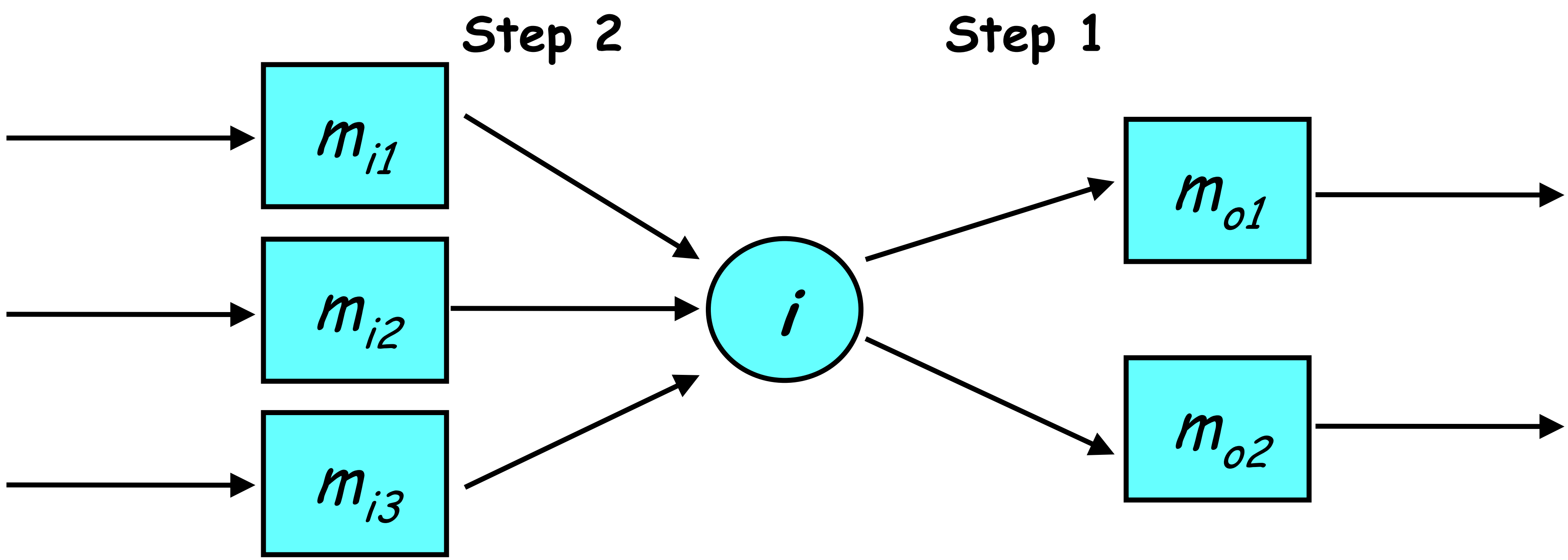
Rounds consist of two steps

Step 1 - For each process, apply *msg* to generate outgoing messages; place in appropriate channels

Step 2 - Apply *trans* to current state and incoming messages to obtain new state; remove messages from incoming channels



Execution at a process



Why is the computation of output messages the first step?

How could we use the null message to model failure?

Failures

- Process failures

  Crash: halt before or after Step 1 and Step 2, or anywhere in Step 1

  Byzantine: generate messages and next states inconsistent with *msg* and *trans*

- Communication failures

  General omission: modelled as null messages placed in channels

Synchronous execution, formally

- Infinite sequence (trace)

  $C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots,$

- C_r: state assignment at round r

  i.e., just after r rounds have occurred

  C_0 is the initial state

- M_r: messages (not including *null*) sent at round r
- N_r: messages received at round r

- if $M_r \neq N_r$ then messages lost

Synchronous leader election in a ring

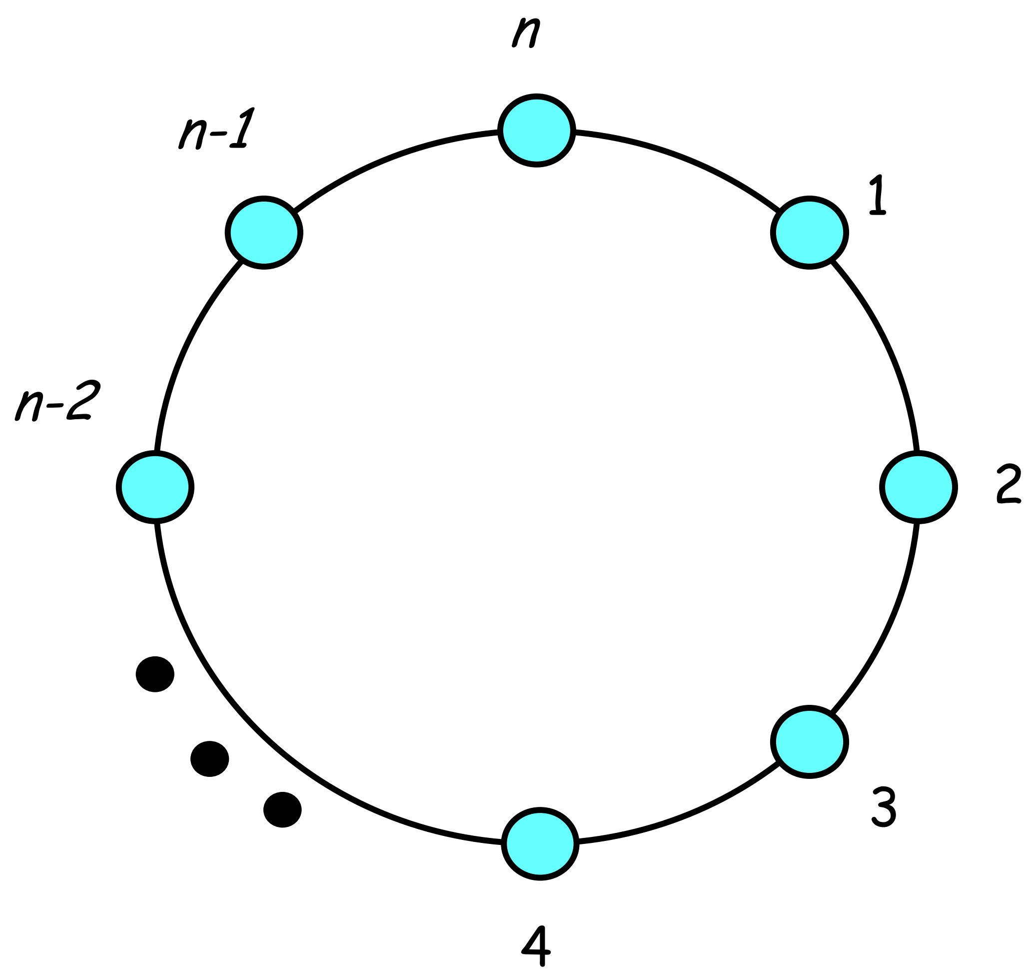
- The basic model

  processes connected in a ring topology

  process unaware of its "index" in the ring

- The desired outcome

  exactly one process makes the decision that it is the "leader"



Additional factors

- Unidirectional vs. bidirectional channels
- Known vs. unknown number of processes
- Simplifying assumption: no failures
- Identical vs. distinguished processes

    distinguished by unique identifiers (UIDs)

    UIDs not necessarily ordinal or consecutive

    UIDs support only comparison operator

Impossibility: identical processes

Theorem: Let *A* be a network of *n* processes, $n>1$, arranged in a bidirectional ring. If all the processes in *A* are identical, then *A* has no solution to the leader-election problem.

Proof (sketch): Assume *A* solves the problem and each process has one start state. *A* must therefore have exactly one execution sequence. By induction on *r*, after *r* rounds all processes are in identical states. If any process reaches a state in which it decides it is the leader, all other processes in *A* reach the same state at the same time. This violates the uniqueness requirement. ■

LCR algorithm

- LCR model for leader election in a ring
  - unidirectional communication
  - processes unaware of ring size
  - each process assigned a UID from large ordinal set

- Outline of algorithm
  - each process forwards its UID to neighbor
  - if received UID < own UID, discard received
  - if received UID > own, forward to neighbor
  - if received UID = own, declare self as leader

LCR algorithm, formally

M: message alphabet is the set U of UIDs

For each process i

states: tuple of three values (u, snd, $stat$)

$u \in U$, initially UID of i

$snd \in U + null$, initially UID of i

$stat \in \{leader, unknown\}$, initially *unknown*

msg: *place* value of snd on output channel

trans: $snd = null$; receive $v \in U$ on input channel

if $v = null$ or else $v < u$ then exit

if $v > u$ then $snd := v$

if $v = u$ then $stat := leader$

Correctness proof for LCR

First show that (1):

process i_{max} is elected leader by the end of round n

where by definition i_{max} is the process initialized with $u = u_{max}$ the maximum UID

Proof: Since the algorithm does not modify u, this is equivalent to showing that:

after n rounds, stat of i_{max} = leader

Correctness proof for LCR

Start by showing that:

for $0 \leq r \leq n\text{-}1$*, after r rounds, snd of* $i_{max+r} = u_{max}$

by induction on r (notice use of modulo arithmetic, where $n \approx 0$, $n + 1 \approx 1$, etc.)

So, for $r = n - 1$: *snd* of $i_{max+n-1} = u_{max}$ and therefore at round n, v of $i_{max} = u_{max}$ resulting in *stat* of i_{max} = *leader*

Correctness proof for LCR

Now we must show that (2):

no process other than i_{max} is elected leader

Proof (sketch): We must show that all other processes have *stat* = *unknown*; this is true because no message will get forwarded by process i_{max} so only i_{max} can receive its own UID in a message and hence become leader

Therefore, LCR solves the leader-election problem

Does the algorithm terminate?

- In the LCR algorithm, only the leader knows that the algorithm has finished computing the desired result

- To terminate other processes, the leader can simply send a "halt" message (which can contain the identity of the leader)

- A process terminates after forwarding this message

What is the complexity of LCR?

- Time complexity

  n rounds until a leader is "discovered"

  $2n$ rounds until the algorithm terminates

- Communication complexity

  $O(n^2)$ messages

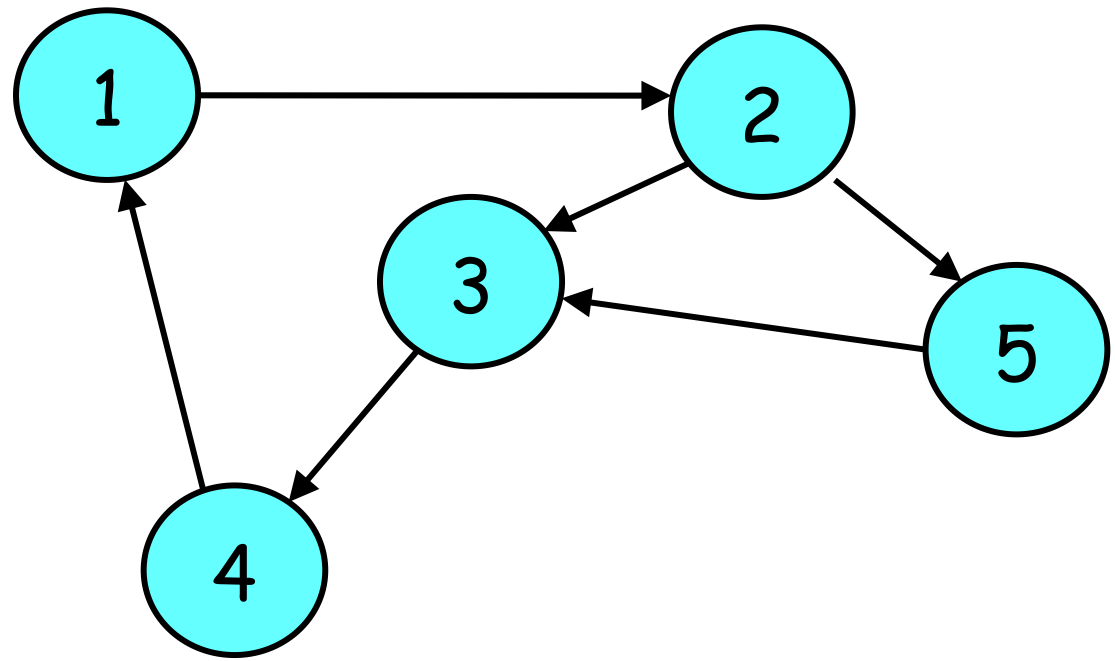
  termination adds n more messages, but still $O(n^2)$

Leader election in a general network

- The basic model

    strongly connected digraph – forall (*i*,*j*) a path with finite distance(*i*,*j*) exists

    processes have UIDs from some totally ordered set



- The desired outcome

    exactly one process elected as the leader

FloodMax algorithm

- Assumes that processes know the diameter of the network (or can make a good guess)
- Outline of algorithm

  every process maintains a record of the maximum UID it has seen so far (initially its own)

  at each round, each process propagates this maximum on all its outgoing channels

  after *diameter* rounds, if the maximum value seen is the processes own UID then it elects itself leader

FloodMax algorithm, formally

M: message alphabet is the set *U* of UIDs

For each process *i*

states: tuple of four values (*u*, *max-u*, *stat*, *rnds*)

u, *max-u* $\in$ *U*, initially UID of *i*

rnds $\in \aleph$, initially 0

stat $\in$ {*leader*, *unknown*, *follower*}, initially *unknown*

FloodMax algorithm, formally

msg:

  if $rnds < diameter$ then

    place $max\text{-}u$ on all output channels

trans:

  $rnds := rnds + 1$

  receive $v \in 2^U$ on input channels

  $max\text{-}u := \max(max\text{-}u + v)$

  if $rnds = diameter$ then

    if $max\text{-}u = u$ then $stat := leader$

      else $stat := follower$

Analysis of FloodMax

- Correctness (sketch)

  follows from the definition of diameter

  after *diameter* rounds, every process will know the maximum UID

- Time complexity

  diameter rounds

- Communication complexity

  number of messages = *diameter* x $|E|$

Can we do better?

- FloodMaxOpt

  after the first round, a process only sends *max-u* if it received a new value in the previous round

states: tuple of five values (*u*, *max-u*, *stat*, *rnds*, *new*)

  . . . *new* $\in$ {true, false}, initially true . . .

msg:

  if *rnds* < *diameter* and *new* then

    place *max-u* on all output channels

trans:

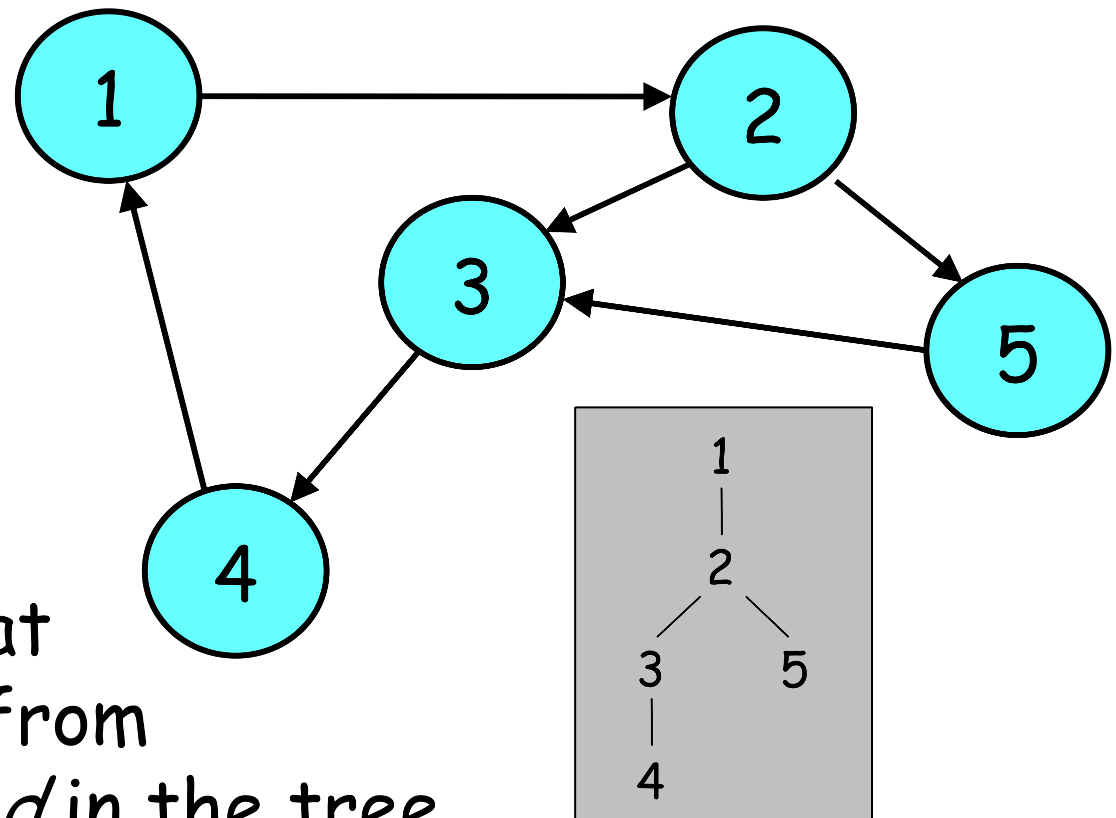
  . . . *new* := max(*v*) > *max-u* . . .

Breadth-first search

- The basic model

  processes connected in a general topology

  processes unaware of network size or diameter



- The desired outcome

  spanning tree, such that process at distance *d* from root appears at depth *d* in the tree

SyncBFS algorithm

- Outline of algorithm
  - initially the root process i_0 is "marked"
  - at round 1, i_0 sends a probe message to each of its "outgoing neighbors"
  - in any round, if an unmarked process receives a probe, it marks itself and sets one of the processes from which it received the probe as its parent
  - in the next round, newly marked processes send a probe to each of their outgoing neighbors

SyncBFS algorithm, formally

M: message alphabet is {probe:*i*}

For each process *i*

states: tuple of three values (*marked*, *parent*, *sent*)

marked $\in$ {true, false}, initially ($i = i_0$)

parent $\in$ 0..*n*, initially 0 (i.e., undefined)

sent $\in$ {true, false}, initially false

SyncBFS algorithm, formally

msg:

if *marked* and not *sent*, then

send probe:i to all outgoing neighbors

trans:

sent := *marked*

receive probe or null messages

if received probe and not *marked*, then

marked := true

parent := one of senders

Complexity analysis of SyncBFS

- Time complexity

  $O(\textit{diameter})$ rounds

- Communication complexity

  number of messages = $O(|E|)$

Termination in SyncBFS

- Problem

  initiating process does not know when spanning tree construction is complete

  additionally, each process knows its parent, but not its children

- Solution

  allow back-channel *reply* messages

  two kinds: *not child* and *child*

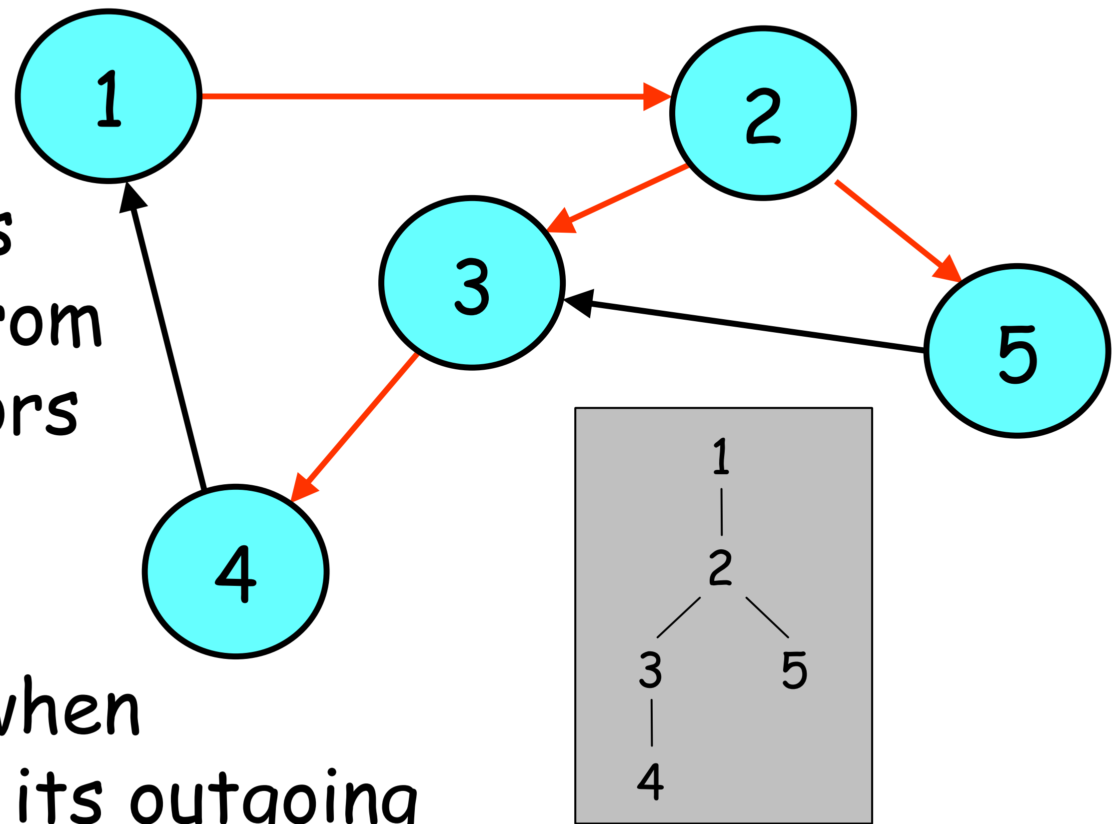
Termination algorithm for SyncBFS

- Outline of algorithm ("convergecast")

  when a marked process receives a probe message, it replies *not child* in the next round

  when a marked process gets reply messages from all its outgoing neighbors it replies *child* to its parent

  algorithm terminates when i_0 gets replies from all its outgoing neighbors



Additional notes on SyncBFS

- The spanning tree constructed by SyncBFS can be used to perform an efficient *broadcast*

- The terminating SyncBFS algorithm can be used to compute the *diameter* of a network

- If channels are unidirectional and a reply channel is not available, each process can use a SyncBFS broadcast to communicate with its parent

    cost: increased communication