Chapter 5: Terminology and Basic Algorithms

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Topology Abstraction and Overlays

- System: undirected (weighted) graph (N, L), where $n = |N|$, $l = |L|$
- Physical topology
  - Nodes: network nodes, routers, all end hosts (whether participating or not)
  - Edges: all LAN, WAN links, direct edges between end hosts
  - E.g., Fig. 5.1(a) topology + all routers and links in WANs
- Logical topology (application context)
  - Nodes: end hosts where application executes
  - Edges: logical channels among these nodes

  All-to-all fully connected (e.g., Fig 5.1(b))
  or any subgraph thereof, e.g., neighborhood view, (Fig 5.1(a)) - partial system view, needs multi-hop paths, easy to maintain
- Superimposed topology (a.k.a. topology overlay):
  - superimposed on logical topology
  - Goal: efficient information gathering, distribution, or search (as in P2P overlays)
  - e.g., ring, tree, mesh, hypercube

Topology Abstractions

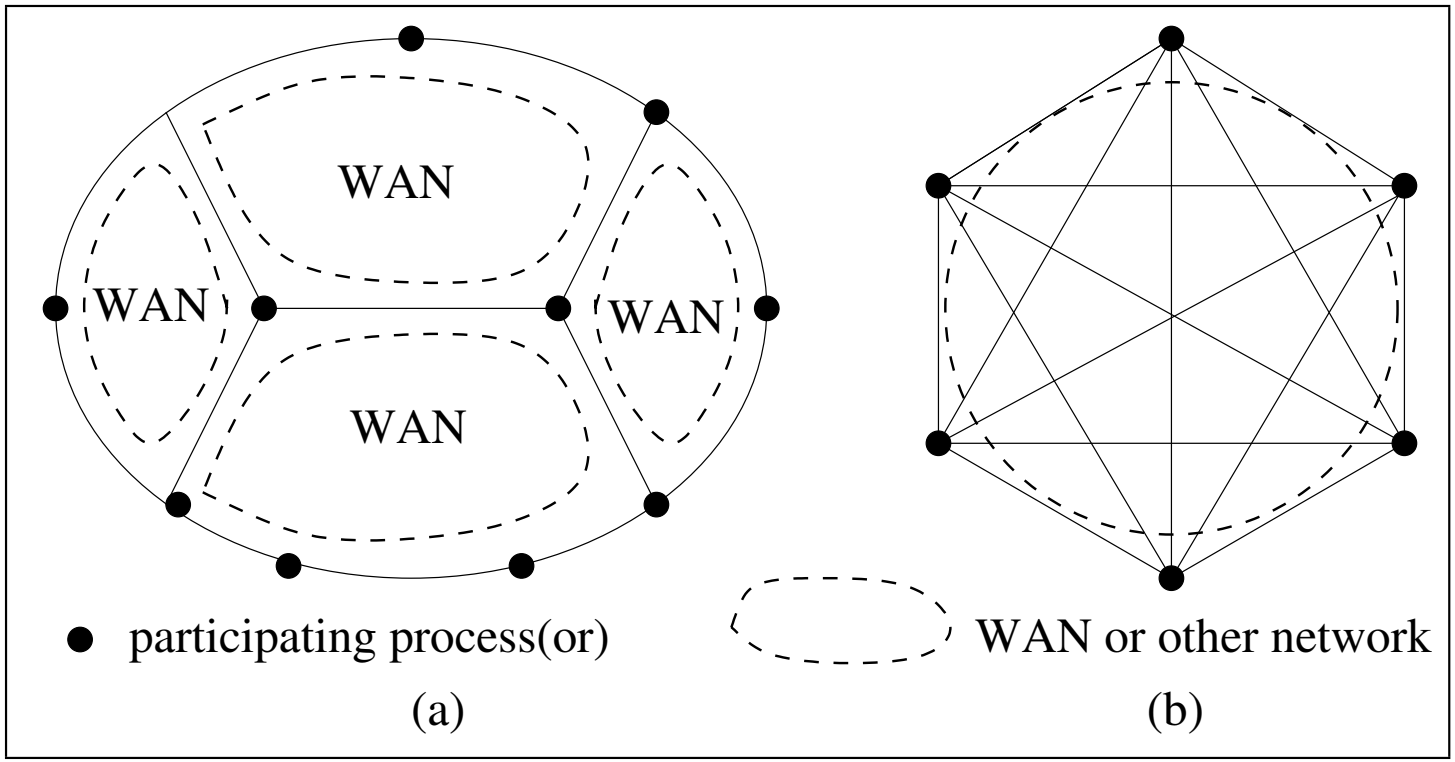

Figure 5.1: Example topology views at different levels of abstraction

Classifications and Basic Concepts (1)

- Application execution vs. control algorithm execution, each with own events
  - Control algorithm:
    - ⋆ for monitoring and auxiliary functions, e.g., creating a ST, MIS, CDS, reaching consensus, global state detection (deadlock, termination etc.), checkpointing
    - ⋆ superimposed on application execution, but does not interfere
    - ⋆ its send, receive, internal events are transparent to application execution
    - ⋆ a.k.a. *protocol*
- Centralized and distributed algorithms
  - Centralized: asymmetric roles; client-server configuration; processing and bandwidth bottleneck; point of failure
  - Distributed: more balanced roles of nodes, difficult to design perfectly distributed algorithms (e.g., snapshot algorithms, tree-based algorithms)
- Symmetric and asymmetric algorithms

Classifications and Basic Concepts (2)

- Anonymous algorithm: process ids or processor ids are not used to make any execution (run-time) decisions
  - Structurally elegant but hard to design, or impossible, e.g., anonymous leader election is impossible
- Uniform algorithm: Cannot use n, the number of processes, as a parameter in the code
  - Allows scalability; process leave/join is easy and only neighbors need to be aware of logical topology changes
- Adaptive algorithm: Let k ($\leq n$) be the number of processes participating in the context of a problem X when X is being executed. Complexity should be expressible as a function of k, not n.
  - E.g., mutual exclusion: critical section contention overhead expressible in terms of number of processes contending at this time (k)

Classifications and Basic Concepts (3)

- Deterministic vs. nondeterministic executions
  - Nondeterministic execution: contains at least 1 nondeterministic *receive*; deterministic execution has no nondeterministic receive
    - ⋆ Nondeterministic receive: can receive a message from any source
    - ⋆ Deterministic receive: source is specified

    Difficult to reason with
  - Asynchronous system: re-execution of deterministic program will produce same partial order on events ((used in debugging, unstable predicate detection etc.)
  - Asynchronous system: re-execution of nondeterministic program may produce different partial order (unbounded delivery times and unpredictable congestion, variable local CPU scheduling delays)

Classification and Basic Concepts (4)

- Execution inhibition (a.k.a. freezing)
  - Protocols that require suspension of normal execution until some stipulated operations occur are inhibitory
  - Concept: Different from blocking vs. nonblocking primitives
  - Analyze inhibitory impact of control algo on underlying execution
  - Classification 1:
    - ⋆ Non-inhibitory protocol: no event is disabled in any execution
    - ⋆ Locally inhibitory protocol: in any execution, any delayed event is a locally delayed event, i.e., inhibition under local control, *not* dependent on any receive event
    - ⋆ Globally inhibitory: in some execution, some delayed event is not locally delayed
  - Classification 2: send inhibitory/ receive inhibitory/ internal event inhibitory

Classifications and Basic Concepts (5)

- Synchronous vs. asynchronous systems
  - Synchronous:
    - ⋆ upper bound on message delay
    - ⋆ known bounded drift rate of clock wrt. real time
    - ⋆ known upper bound for process to execute a logical step
  - Asynchronous: above criteria not satisfied
    spectrum of models in which some combo of criteria satisfied

  Algorithm to solve a problem depends greatly on this model
  Distributed systems inherently asynchronous
- On-line vs. off-line (control) algorithms
  - On-line: Executes as data is being generated
    Clear advantages for debugging, scheduling, etc.
  - Off-line: Requires all (trace) data before execution begins

Classification and Basic Concepts (6)

- Wait-free algorithms (for synchronization operations)
  - resilient to $n-1$ process failures, i.e., ops of any process must complete in bounded number of steps, irrespective of other processes
  - very robust, but expensive
  - possible to design for mutual exclusion
  - may not always be possible to design, e.g., producer-consumer problem
- Communication channels
  - point-to-point: FIFO, non-FIFO
    At application layer, FIFO usually provided by network stack

Classifications and Basic Concepts (7)

- Process failures (sync + async systems) in order of increasing severity
  - Fail-stop: Properly functioning process stops execution. Other processes learn about the failed process (thru some mechanism)
  - Crash: Properly functioning process stops execution. Other processes do not learn about the failed process
  - Receive omission: Properly functioning process fails by receiving only some of the messages that have been sent to it, or by crashing.
  - Send omission: Properly functioning process fails by sending only some of the messages it is supposed to send, or by crashing. Incomparable with receive omission model.
  - General omission: Send omission + receive omission
  - Byzantine (or malicious) failure, with authentication: Process may (mis) behave anyhow, including sending fake messages.
    Authentication facility $\Longrightarrow$ If a faulty process claims to have received a message from a correct process, that is verifiable.
  - Byzantine (or malicious) failure, no authentication

  The non-malicious failure models are "benign"

Classifications and Basic Concepts (8)

- Process failures (contd.) $\rightarrow$ Timing failures (sync systems):
  - General omission failures, or clocks violating specified drift rates, or process violating bounds on time to execute a step
  - More severe than general omission failures

  Failure models influence design of algorithms
- Link failures
  - Crash failure: Properly functioning link stops carrying messages
  - Omission failure: Link carries only some of the messages sent on it, not others
  - Byzantine failure: Link exhibits arbitrary behavior, including creating fake messages and altering messages sent on it
- Link failures $\rightarrow$ Timing failures (sync systems): messages delivered faster/slower than specified behavior

Complexity Measures and Metrics

- Each metric specified using lower bound (Ω), upper bound (O), exact bound (θ)
- Metrics
  - Space complexity per node
  - System-wide space complexity ($\neq n \cdot$ space complexity per node). E.g., worst case may never occur at all nodes simultaneously!
  - Time complexity per node
  - System-wide time complexity. Do nodes execute fully concurrently?
  - Message complexity
    - Number of messages (affects space complexity of message ovhd)
    - Size of messages (affects space complexity of message ovhd + time component via increased transmission time)
    - Message time complexity: depends on number of messages, size of messages, concurrency in sending and receiving messages
  - : Other metrics: # send and # receive events; # multicasts, and how implemented?
  - (Shared memory systems): size of shared memory; # synchronization operations

Program Structure

- Communicating Sequential Processes (CSP) like:

$$*[\, G_1 \longrightarrow CL_1 \,||\, G_2 \longrightarrow CL_2 \,||\, \cdots \,||\, G_k \longrightarrow CL_k \,]$$

- The *repetitive* command "*" denotes an infinite loop.
- Inside it, the *alternative* command "||" is over *guarded* commands.
  Specifies execution of exactly one of its constituent guarded commands.
- *Guarded* command syntax: "$G \longrightarrow CL$"
  guard G is boolean expression,
  CL is list of commands to be executed if G is true.
  Guard may check for message arrival from another process.
- Alternative command fails if all the guards fail; if > 1 guard is true, one is nondeterministically chosen for execution.
- $G_m \longrightarrow CL_m$: CL_m and G_m atomically executed.

Basic Distributed Graph Algorithms: Listing

- Sync 1-initiator ST (flooding)
- Async 1-initiator ST (flooding)
- Async conc-initiator ST (flooding)
- Async DFS ST
- Broadcast & convergecast on tree
- Sync 1-source shortest path
- Distance Vector Routing
- Async 1-source shortest path
- All sources shortest path: Floyd-Warshall
- Sync, async constrained flooding
- MST, sync
- MST, async
- Synchronizers: simple, α, β, γ
- MIS, async, randomized
- CDS
- Compact routing tables
- Leader election: LCR algorithm
- Dynamic object replication

Sync 1-initiator ST (flooding)

```
(local variables)
int visited, depth ⟵ 0
int parent ⟵ ⊥
set of int Neighbors ⟵ set of neighbors
(message types)
QUERY

(1) if i = root then
(2)       visited ⟵ 1;
(3)       depth ⟵ 0;
(4)       send QUERY to Neighbors;
(5) for round = 1 to diameter do
(6)       if visited = 0 then
(7)             if any QUERY messages arrive then
(8)                   parent ⟵ randomly select a node from which QUERY was received;
(9)                   visited ⟵ 1;
(10)                  depth ⟵ round;
(11)                  send QUERY to Neighbors \ {senders of QUERYs received in this round};
(12)      delete any QUERY messages that arrived in this round.
```

Synchronous 1-init Spanning Tree: Example

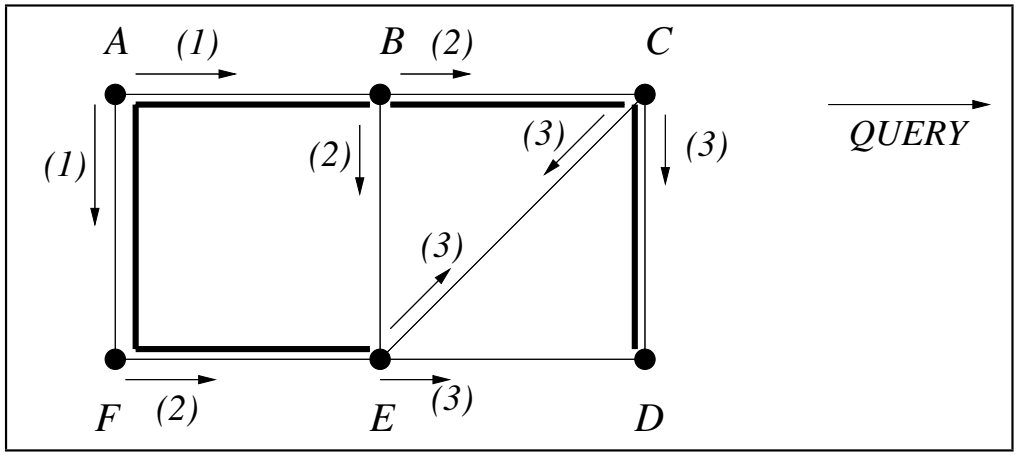

Figure 5.2: Tree in boldface; round numbers of QUERY are labeled

- Designated root. Node A in example.
- Each node identifies parent
- How to identify child nodes?

Synchronous 1-init Spanning Tree: Complexity

Termination: after *diameter* rounds.
How can a process terminate after setting its *parent*?

Complexity:

- Local space: $O(degree)$
- Global space: $O(\sum \text{local space})$
- Local time: $O(degree + diameter)$
- Message time complexity: d rounds or message hops
- Message complexity: $\geq 1, \leq 2$ messages/edge. Thus, $[l, 2l]$

Spanning tree: analogous to breadth-first search

Asynchronous 1-init Spanning Tree: Code

(local variables)
int *parent* $\longleftarrow \perp$
set of int *Children*, *Unrelated* $\longleftarrow \emptyset$
set of int *Neighbors* $\longleftarrow$ set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1) When the predesignated root node wants to initiate the algorithm:
(1a) **if** ($i = root$ **and** $parent = \perp$) **then**
(1b) **send** QUERY to all neighbors;
(1c) $parent \longleftarrow i$.

(2) When QUERY arrives from j:
(2a) **if** $parent = \perp$ **then**
(2b) $parent \longleftarrow j$;
(2c) **send** ACCEPT to j;
(2d) **send** QUERY to all neighbors except j;
(2e) **if** $(Children \cup Unrelated) = (Neighbors \setminus \{parent\})$ **then**
(2f) **terminate**.
(2g) **else send** REJECT to j.

(3) When ACCEPT arrives from j:
(3a) $Children \longleftarrow Children \cup \{j\}$;
(3b) **if** $(Children \cup Unrelated) = (Neighbors \setminus \{parent\})$ **then**
(3c) **terminate**.

(4) When REJECT arrives from j:
(4a) $Unrelated \longleftarrow Unrelated \cup \{j\}$;
(4b) **if** $(Children \cup Unrelated) = (Neighbors \setminus \{parent\})$ **then**
(4c) **terminate**.

Async 1-init Spanning Tree: Operation

- root initiates flooding of QUERY to identify tree edges
- *parent*: 1st node from which QUERY received
  - ACCEPT (+ rsp) sent in response; QUERY sent to other nbhs
  - Termination: when ACCEPT or REJECT (- rsp) received from non-parent nbhs. Why?
- QUERY from non-parent replied to by REJECT
- Necessary to track neighbors? to determine children and when to terminate?
- Why is REJECT message type required?
- Can use of REJECT messages be eliminated? How? What impact?

Asynchronous 1-init Spanning Tree: Complexity

Local termination: after receiving ACCEPT or REJECT from non-parent nbhs.
Complexity:

- Local space: $O(degree)$
- Global space: $O(\sum \text{local space})$
- Local time: $O(degree)$
- Message complexity: $\geq 2, \leq 4$ messages/edge. Thus, $[2l, 4l]$
- Message time complexity: $d + 1$ message hops.

Spanning tree: no claim can be made. Worst case height $n - 1$

Asynchronous 1-init Spanning Tree: Example

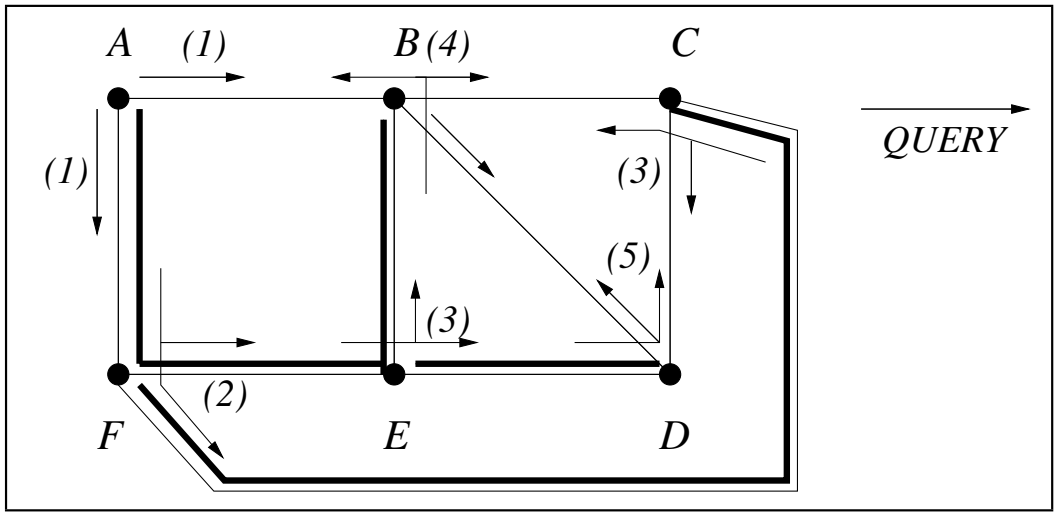

Figure 5.3: Tree in boldface; Number indicates approximate order in which QUERY get sent

- Designated root. Node A in example.
- tree edges: QUERY + ACCEPT msgs
- cross-edges and back-edges: 2(QUERY + REJECT) msgs

Asynchronous Spanning Tree: Concurrent Initiators

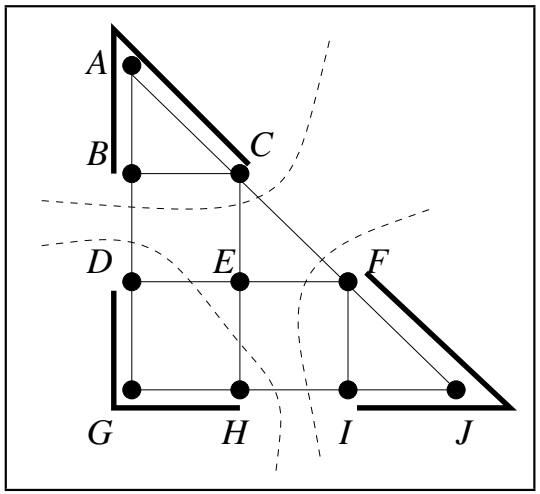

Figure 5.4: Concurrent initiators A,G,J

No pre-designated root:

- Option 1: Merge partial STs. Difficult based on local knowledge, can lead to cycles
- Option 2: Allow one ST computation instance to proceed; supress others.
  - Used by algorithm; selects root with higher process id to continue
  - 3 cases: *newroot* $<=>$ *myroot*

Algorithm:

- A node may spontaneously initiate algorithm and become "root".
- Each "root" initiates variant of 1-init algorithm; lower priorities suppressed at intermediate nodes
- Termination: Only root detects termination. Needs to send extra messages to inform others.
- Time complexity: $O(l)$
- Message complexity: $O(nl)$

Asynchronous Spanning Tree: Code (1/2)

(local variables)
int *parent*, *myroot* $\longleftarrow \perp$
set of int *Children*, *Unrelated* $\longleftarrow \emptyset$
set of int *Neighbors* $\longleftarrow$ set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1) When the node wants to initiate the algorithm as a root:
(1a) **if** (*parent* $= \perp$) **then**
(1b) **send** QUERY(i) to all neighbors;
(1c) *parent*, *myroot* $\longleftarrow$ i.

(2) When QUERY(*newroot*) arrives from j:
(2a) **if** *myroot* $<$ *newroot* **then** // discard earlier partial execution due to its lower priority
(2b) *parent* $\longleftarrow$ j; *myroot* $\longleftarrow$ *newroot*; *Children*, *Unrelated* $\longleftarrow \emptyset$;
(2c) **send** QUERY(*newroot*) to all neighbors except j;
(2d) **if** *Neighbors* $= \{j\}$ **then**
(2e) **send** ACCEPT(*myroot*) to j; **terminate.** // leaf node
(2f) **else send** REJECT(*newroot*) to j. // if *newroot* $=$ *myroot* then *parent* is already identified.
// if *newroot* $<$ *myroot* ignore the QUERY. j will update its root when it receives QUERY(*myroot*).

Asynchronous Spanning Tree: Code (2/2)

(3) When ACCEPT(*newroot*) arrives from j:
(3a) **if** $newroot = myroot$ **then**
(3b) $\quad Children \longleftarrow Children \cup \{j\}$;
(3c) $\quad$ **if** $(Children \cup Unrelated) = (Neighbors \setminus \{parent\})$ **then**
(3d) $\quad\quad$ **if** $i = myroot$ **then**
(3e) $\quad\quad\quad$ **terminate**.
(3f) $\quad\quad$ **else send** ACCEPT(*myroot*) to *parent*.
//if $newroot < myroot$ then ignore the message. $newroot > myroot$ will never occur.

(4) When REJECT(*newroot*) arrives from j:
(4a) **if** $newroot = myroot$ **then**
(4b) $\quad Unrelated \longleftarrow Unrelated \cup \{j\}$;
(4c) $\quad$ **if** $(Children \cup Unrelated) = (Neighbors \setminus \{parent\})$ **then**
(4d) $\quad\quad$ **if** $i = myroot$ **then**
(4e) $\quad\quad\quad$ **terminate**.
(4f) $\quad\quad$ **else send** ACCEPT(*myroot*) to *parent*.
//if $newroot < myroot$ then ignore the message. $newroot > myroot$ will never occur.

Asynchronous DFS Spanning Tree

- Handle concurrent initiators just as for the non-DFS algorithm, just examined
- When QUERY, ACCEPT, or REJECT arrives: actions depend on whether $myroot <= newroot$
- Termination: only successful root detects termination. Informs others using ST edges.
- Time complexity: $O(l)$
- Message complexity: $O(nl)$

Asynchronous DFS Spanning Tree: Code

(local variables)
int *parent*, *myroot* ←− ⊥
set of int *Children* ←− ∅
set of int *Neighbors*, *Unknown* ←− set of neighbors
(message types)
QUERY, ACCEPT, REJECT

(1) When the node wants to initiate the algorithm as a root:
(1a) **if** (*parent* = ⊥) **then**
(1b) **send** QUERY(*i*) to *i* (itself).

(2) When QUERY(*newroot*) arrives from *j*:
(2a) **if** *myroot* < *newroot* **then**
(2b) *parent* ←− *j*; *myroot* ←− *newroot*; *Unknown* ←− set of neighbours;
(2c) *Unknown* ← *Unknown* \ {*j*};
(2d) **if** *Unknown* ≠ ∅ **then**
(2e) **delete** some *x* from *Unknown*;
(2f) **send** QUERY(*myroot*) to *x*;
(2g) **else send** ACCEPT(*myroot*) to *j*;
(2h) **else if** *myroot* = *newroot* **then**
(2i) **send** REJECT to *j*. // if *newroot* < *myroot* ignore the query.
// *j* will update its root to a higher root identifier when it receives its QUERY.

(3) When ACCEPT(*newroot*) or REJECT(*newroot*) arrives from *j*:
(3a) **if** *newroot* = *myroot* **then**
(3b) **if** ACCEPT message arrived **then**
(3c) *Children* ←− *Children* ∪ {*j*};
(3d) **if** *Unknown* = ∅ **then**
(3e) **if** *parent* ≠ *i* **then**
(3f) **send** ACCEPT(*myroot*) to *parent*;
(3g) **else** set *i* as the root; **terminate**.
(3h) **else**
(3i) **delete** some *x* from *Unknown*;
(3j) **send** QUERY(*myroot*) to *x*.
// if *newroot* < *myroot* ignore the query. Since sending QUERY to *j*, *i* has updated its *myroot*.
// *j* will update its *myroot* to a higher root identifier when it receives a QUERY initiated by it. *newroot* > *myroot* will never occur.

Broadcast and Convergecast on a Tree (1)

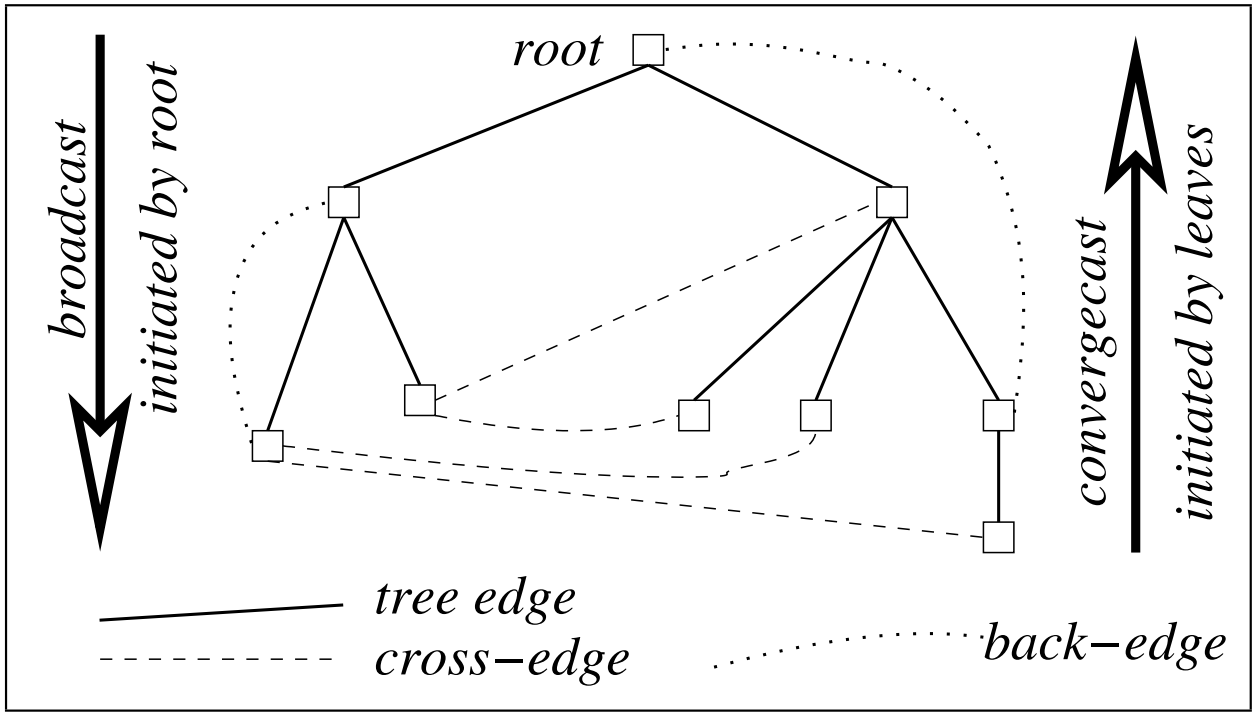

Figure 5.5: Tree structure for broadcast and convergecast

Question:

- how to perform BC and CC on a ring? on a mesh?
- Costs?

Broadcast and Convergecast on a Tree (2)

Broadcast: distribute information

- BC1. Root sends info to be broadcast to all its children. Terminate.
- BC2. When a (nonroot) node receives info from its parent, it copies it and forwards it to its children. Terminate.

Convergecast: collect info at root, to compute a global function

- CVC1. Leaf node sends its report to its parent. Terminate.
- CVC2. At a non-leaf node that is not the root: When a report is received from all the child nodes, the collective report is sent to the parent. Terminate.
- CVC3. At root: When a report is received from all the child nodes, the global function is evaluated using the reports. Terminate.

Uses: compute min, max, leader election, compute global state functions
Time complexity: $O(h)$; Message complexity: $n-1$ messages for BC or CC

Single Source Shortest Path: Sync Bellman-Ford

- Weighted graph, no cycles with negative weight
- No node has global view; only local topology
- Assumption: node knows n; needed for termination
- After k rounds: *length* at any node has length of shortest path having k hops
- After k rounds: *length* of all nodes up to k hops away in final MST has stabilized
- Termination: $n-1$ rounds
- Time Complexity: $n-1$ rounds
- Message complexity: $(n-1) \cdot l$ messages

Sync Distributed Bellman-Ford: Code

(local variables)
int $length \longleftarrow \infty$
int $parent \longleftarrow \perp$
set of int $Neighbors \longleftarrow$ set of neighbors
set of int $\{weight_{i,j}, weight_{j,i} \mid j \in Neighbors\} \longleftarrow$ the known values of the weights of incident links

(message types)
UPDATE

(1) **if** $i = i_0$ **then** $length \longleftarrow 0$;
(2) **for** $round = 1$ **to** $n - 1$ **do**
(3) **send** UPDATE(i, $length$) to all neighbors;
(4) **await** UPDATE(j, $length_j$) from each $j \in Neighbors$;
(5) **for** each $j \in Neighbors$ **do**
(6) **if** $(length > (length_j + weight_{j,i})$ **then**
(7) $length \longleftarrow length_j + weight_{j,i}$; $parent \longleftarrow j$.

Distance Vector Routing

- Used in Internet routing (popular upto to mid-1980s), having dynamically changing graph, where link weights model delay/ load
- Variant of sync Bellman-Ford; outer **for** loop is infinite
- Track shortest path to *every* destination
- *length* replaced by $LENGTH[1..n]$; *parent* replaced by $PARENT[1..n]$
- kth component denotes best-known length to $LENGTH[k]$
- In each iteration
  - apply triangle inequality for each destination independently
  - Triangle inequality: $(LENGTH[k] > (LENGTH_j[k] + weight_{j,i})$
  - Node i estimates $weight_{ij}$ using RTT or queuing delay to neighbor j

Single Source Shortest Path: Async Bellman-Ford

- Weighted graph, no cycles with negative weight
- No node has global view; only local topology
- exponential $\Omega(c^n)$ number of messages and exponential $\Omega(c^n \cdot d)$ time complexity in the worst case, where c is some constant
- If all links have equal weight, the algorithm computes the minimum-hop path; the minimum-hop routing tables to all destinations are computed using $O(n^2 \cdot l)$ messages

Async Distributed Bellman-Ford: Code

(local variables)
int $length \longleftarrow \infty$
set of int $Neighbors \longleftarrow$ set of neighbors
set of int $\{weight_{i,j}, weight_{j,i} \mid j \in Neighbors\} \longleftarrow$ the known values of the weights of incident links

(message types)
UPDATE

(1) **if** $i = i_0$ **then**
(1a) $length \longleftarrow 0$;
(1b) **send** UPDATE$(i_0, 0)$ to all neighbours; **terminate**.

(2) When UPDATE$(i_0, length_j)$ arrives from j:
(2a) **if** $(length > (length_j + weight_{j,i}))$ **then**
(2b) $length \longleftarrow length_j + weight_{j,i}$; $parent \longleftarrow j$;
(2c) **send** UPDATE$(i_0, length)$ to all neighbors;

All-All Shortest Paths: Floyd-Warshall

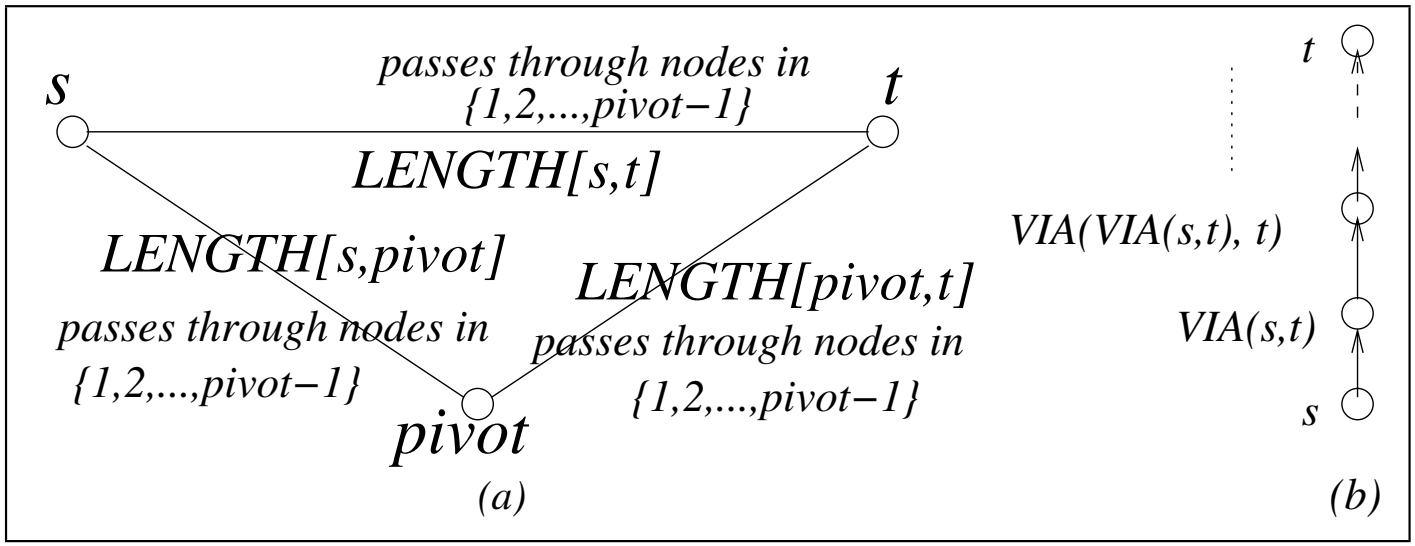

Figure 5.6: (a) Triangle inequality for Floyd-Warshall algorithm. (b) *VIA* relationships along a branch of the sink tree for a given (s, t) pair

All-All Shortest Paths: Floyd-Warshall

After *pivot* iterations of the outer loop,

Invariant

"$LENGTH[i, j]$ is the shortest path going through intermediate nodes from the set $\{i, \ldots, pivot\}$. $VIA[i, j]$ is the corresponding first hop."

```
(1) for pivot = 1 to n do
(2)   for s = 1 to n do
(3)       for t = 1 to n do
(4)           if LENGTH[s, pivot] + LENGTH[pivot, t] < LENGTH[s, t] then
(5)               LENGTH[s, t] ⟵ LENGTH[s, pivot] + LENGTH[pivot, t];
(6)               VIA[s, t] ⟵ VIA[s, pivot].
```

Complexity (centralized): $O(n^3)$

Distributed Floyd-Warshall (1)

- Row i of $LENGTH[1..n, 1..n]$, $VIA[1..n, 1..n]$ stored at i, which is responsible for updating the rows. (So, i acts as source.)
- Corresponding to centralized algorithm, line (4):
  - How does node i access remote datum $LENGTH[pivot, t]$ in each iteration $pivot$?
    - ⋆ *D*istributed (dynamic) sink tree: In any iteration $pivot$, all nodes $s \mid LENGTH[s, t] \neq \infty$ are on a *s*ink tree, with sink at t
  - How to synchronize execution of outer loop iteration at different nodes? (otherwise, algorithm goes wrong).
    - ⋆ Simulate "synchronizer": e.g., use *r*eceive to get data $LENGTH[pivot, *]$ from parent on sink tree

Distributed Floyd-Warshall: Data structures

(local variables)
array of int $LEN[1..n]$ // $LEN[j]$ is the length of the shortest known path from i to node j.
// $LEN[j] = weight_{ij}$ for neighbor j, 0 for $j = i$, ∞ otherwise
array of int $PARENT[1..n]$ // $PARENT[j]$ is the parent of node i (myself) on the sink tree rooted at j.
// $PARENT[j] = j$ for neighbor j, $\perp$ otherwise
set of int $Neighbours \longleftarrow$ set of neighbors
int $pivot, nbh \longleftarrow 0$

(message types)

IN_TREE($pivot$), NOT_IN_TREE($pivot$), PIV_LEN($pivot$, $PIVOT\_ROW[1..n]$)

// $PIVOT\_ROW[k]$ is $LEN[k]$ of node $pivot$, which is $LEN[pivot, k]$ in the central algorithm

// the PIV_LEN message is used to convey $PIVOT\_ROW$.

Distributed Floyd-Warshall: Code

```
(1) for pivot = 1 to n do
(2)   for each neighbour nbh ∈ Neighbours do
(3)       if PARENT[pivot] = nbh then
(4)           send IN_TREE(pivot) to nbh;
(5)       else send NOT_IN_TREE(pivot) to nbh;
(6)   await IN_TREE or NOT_IN_TREE message from each neighour;
(7)   if LEN[pivot] ≠ ∞ then
(8)       if pivot ≠ i then
(9)           receive PIV_LEN(pivot, PIVOT_ROW[1..n]) from PARENT[pivot];
(10)      for each neighbour nbh ∈ Neighbours do
(11)          if IN_TREE message was received from nbh then
(12)              if pivot = i then
(13)                  send PIV_LEN(pivot, LEN[1..n]) to nbh;
(14)              else send PIV_LEN(pivot, PIVOT_ROW[1..n]) to nbh;
(15)      for t = 1 to n do
(16)          if LEN[pivot] + PIVOT_ROW[t] < LEN[t] then
(17)              LEN[t] ⟵ LEN[pivot] + PIVOT_ROW[t];
(18)              PARENT[t] ⟵ PARENT[pivot].
```

Distributed Floyd-Warshall: Dynamic Sink Tree

Rename $LENGTH[i, j]$, $VIA[i, j]$ as $LEN[j]$, $PARENT[j]$ in distributed algorithm
$\Longrightarrow$ $LENGTH[i, pivot]$ is $LEN[pivot]$
At any node i, in iteration $pivot$:

- **iff** $LEN[pivot] \neq \infty$ at node i, then $pivot$ distributes $LEN[*]$ to all nodes (including i) in sink tree of $pivot$
- Parent-child edges in sink tree need to be IDed. How?
  1. A node sends IN_TREE to $PARENT[pivot]$; NOT_IN_TREE to other neighbors
  2. Receive IN_TREE from $k \Longrightarrow k$ is a child in sink tree of $pivot$
- Await IN_TREE or NOT_IN_TREE from each neighbor.
  This send-receive is synchronization!
- $pivot$ broadcasts $LEN[*]$ down its sink tree.
  This send-receive is synchronization!
- Now, all nodes execute triangle inequality in pseudo lock-step

Time Complexity: $O(n^2)$ execution/node, + time for n broadcasts
Message complexity: n iterations;

- 2 IN_TREE or NOT_IN_TREE msgs of size $O(1)$ per edge: $O(l)$ msgs
- $\leq n - 1$ PIV_LEN msgs of size $O(n)$: $O(n)$ msgs

Total $O(n(l + n))$ messages; Total $O(nl + n^3)$ message space

Distributed Floyd-Warshall: Sink Tree

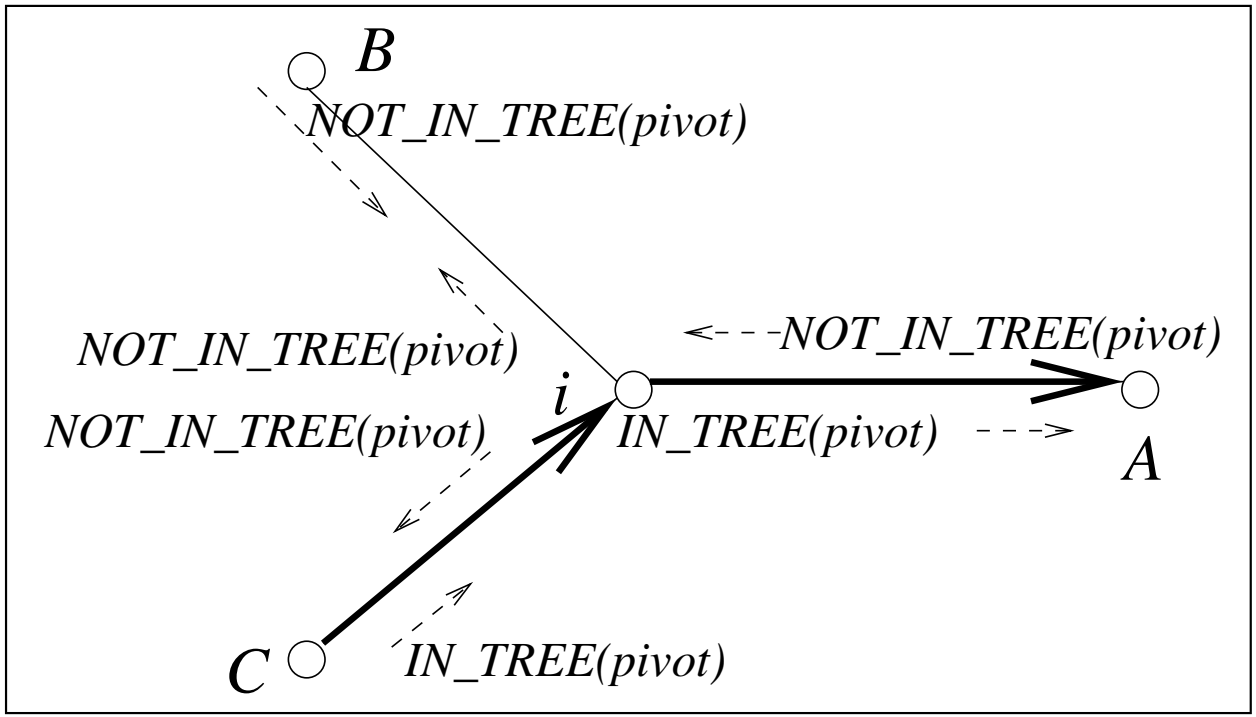

Figure 5.7: Identifying parent-child nodes in sink tree

Constrained Flooding (no ST)

- FIFO channels; duplicates depected using seq. nos.
- Asynchronous flooding:
  - used by Link State Routing in IPv4
  - Complexity: $2l$ messages worst case; Time: d sequential hops
- Synchronous flooding (to learn one datum from each processor):
  - $STATEVEC[k]$ is estimate of k's datum
  - Message complexity: $2ld$ messages, each of size n
  - Time complexity: d rounds

Async Constrained Flooding (no ST)

(local variables)
array of int $SEQNO[1..n] \longleftarrow \overline{0}$
set of int *Neighbors* $\longleftarrow$ set of neighbors
(message types)
UPDATE

(1) To send a message M:
(1a) **if** $i = root$ **then**
(1b) $\quad SEQNO[i] \longleftarrow SEQNO[i] + 1$;
(1c) $\quad$ **send** UPDATE$(M, i, SEQNO[i])$ to each $j \in Neighbors$.

(2) When UPDATE$(M, j, seqno_j)$ arrives from k:
(2a) **if** $SEQNO[j] < seqno_j$ **then**
(2b) $\quad$ Process the message M;
(2c) $\quad SEQNO[j] \longleftarrow seqno_j$;
(2d) $\quad$ **send** UPDATE$(M, j, seqno_j)$ to $Neighbors/\{k\}$
(2e) **else** discard the message.

Sync Constrained Flooding (no ST)

Algorithm learns all nodes identifiers

(local variables)
array of int $STATEVEC[1..n] \longleftarrow \overline{0}$
set of int $Neighbors \longleftarrow$ set of neighbors
(message types)
UPDATE

(1) $STATEVEC[i] \longleftarrow$ local value;
(2) **for** $round = 1$ **to** diameter d **do**
(3) **send** UPDATE($STATEVEC[1..n]$) to each $j \in Neighbors$;
(4) **for** $count = 1$ **to** $|Neighbors|$ **do**
(5) **await** UPDATE($SV[1..n]$) from some $j \in Neighbors$;
(6) $STATEVEC[1..n] \longleftarrow max(STATEVEC[1..n], SV[1..n])$.

Minimum Spanning Tree (MST): Overview

Assume undirected weighted graph. If weights are not unique, assume some tie-breaker such as nodeIDs are used to impose a total order on edge weights.

- Review defns: forest, spanning forest, spanning tree, MST
- Kruskal's MST:
  - Assume forest of graph components
  - maintain sorted list of edges
  - In each of $n-1$ iterations, identify minimum weight edge that connects two different components
  - Include the edge in MST
  - $O(l \ log \ l)$
- Prim's MST:
  - Begin with a single node component
  - In each of $n-1$ iterations, select the minimum weight edge incident on the component. Component expands using this selected edge.
  - $O(n^2)$ (or $O(n \ log \ n)$ using Fibonacci heaps in dense graphs)

GHS Synchronous MST Algorithm: Overview

Gallagher-Humblet-Spira distributed MST uses Kruskal's strategy. Begin with forest of graph components.

- MWOE (minimum weight outgoing edge): "outgoing" is logical, i.e., indicates direction of expansion of component
- Spanning trees of connected components combine with the MWOEs to still retain the spanning tree property in combined component
- Concurrently combine MWOEs:
  - after k iterations, $\leq \frac{n}{2^k}$ components $\Longrightarrow$ at most $log\ n$ iterations
- Each component has a *leader* node in an iteration
- Each iteration within a component has 5 steps, triggered by leader
  - broadcast-convergecast phase: leader identifies MWOE
  - broadcast phase: (potential) leader for next iteration identified
  - broadcast phase: among merging components, 1 leader is selected; it identifies itself to all in the new component

Minimum Weight Outgoing Edge: Example

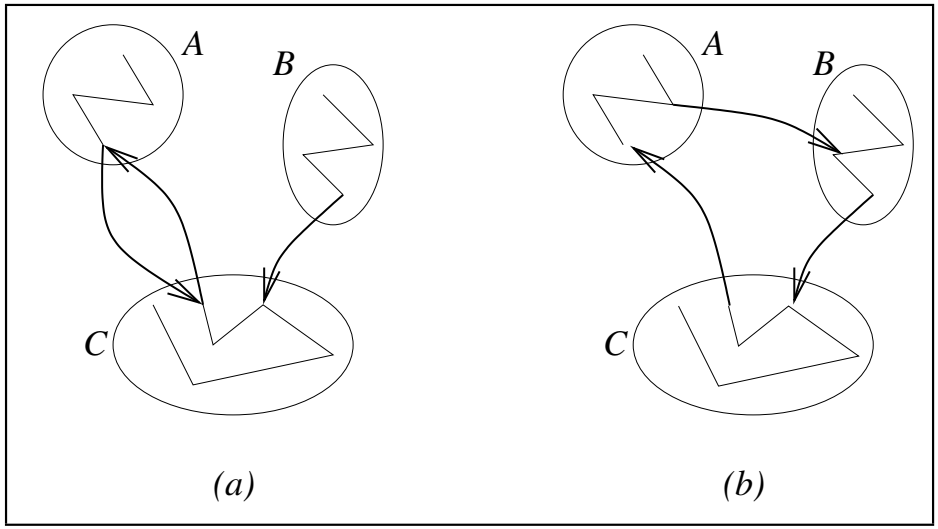

Figure 5.8: Merging of MWOE components. (a) Cycle $len = 2$ possible. (b) Cycle $len > 2$ not possible.

Observation 5.1

For any spanning forest $\{(N_i, L_i) \mid i = 1 \ldots k\}$ of graph G, consider any component (N_j, L_j). Denote by λ_j, the edge having the smallest weight among those that are incident on only one node in N_j. Then an MST for G that includes all the edges in each L_i in the spanning forest, must also include edge λ_i.

MST Example

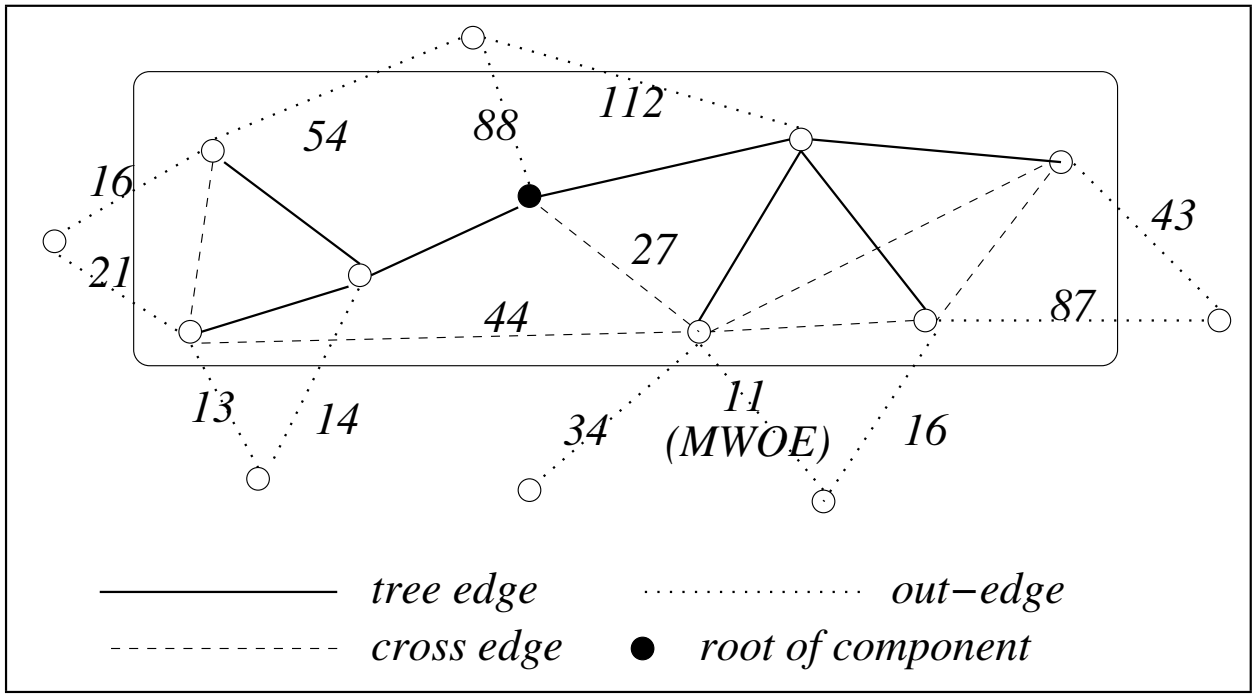

Figure 5.9: Phases within an iteration in a component.
(a) Root broadcasts SEARCH_MWOE; (b) Convergecast REPLY_MWOE occurs. (c) Root broadcasts ADD_MWOE; (d) If the MWOE is also chosen as the MWOE by the component at the other end of the MWOE, the incident process with the higher ID is the leader for the next iteration; and broadcasts NEW_LEADER.

Sync GHS: Message Types

(message types:)

SEARCH_MWOE(*leader*) // broadcast by current leader on tree edges

EXAMINE(*leader*) // sent on non-tree edges after receiving SEARCH_MWOE

REPLY_MWOES(*local_ID*, *remote_ID*) // details of potential MWOEs are convergecast to leader

ADD_MWOE(*local_ID*, *remote_ID*) // sent by leader to add MWOE and identify new leader

NEW_LEADER(*leader*) // broadcast by new leader after merging components

Sync GHS: Code

leader = *i*;
for *round* = 1 **to** *log*(*n*) **do** // each merger in each iteration involves at least two components

1. **if** *leader* = *i* **then**
   broadcast SEARCH_MWOE(*leader*) along marked edges of tree.
2. On receiving a SEARCH_MWOE(*leader*) message that was broadcast on marked edges:
   1. Each process *i* (including *leader*) sends an EXAMINE message along unmarked (i.e., non-tree) edges to determine if the other end of the edge is in the same component (i.e., whether its leader is the same).
   2. From among all incident edges at *i*, for which the other end belongs to a different component, process *i* picks its incident MWOE(localID,remoteID).
3. The leaf nodes in the MST within the component initiate the **convergecast** using REPLY_MWOEs, informing their parent of their MWOE(localID,remoteID). All the nodes participate in this convergecast.
4. **if** *leader* = *i* **then**
   await convergecast replies along marked edges.
   Select the minimum MWOE(localID,remoteID) from all the replies.
   broadcast ADD_MWOE(localID,remoteID) along marked edges of tree.
   // To ask process *localID* to mark the (*localID*, *remoteID*) edge,
   // i.e., include it in MST of component.
5. **if** an MWOE edge gets marked by both the components on which it is incident **then**
   1. Define *new_leader* as the process with the larger ID on which that MWOE is incident (i.e., process whose ID is *max*(*localID*, *remoteID*)).
   2. *new_leader* identifies itself as the leader for the next round.
   3. *new_leader* **broadcasts** NEW_LEADER in the newly formed component along the marked edges announcing itself as the leader for the next round.

GHS: Complexity

- $log\ n$ rounds (synchronous)
- Time complexity: $O(n\ log\ n)$
- Message complexity:
  - In each iteration, $O(n)$ msgs along tree edges (steps 1,3,4,5)
  - In each iteration, l EXAMINE msgs to determine MWOEs

  Hence, $O((n + l) \cdot log\ n)$ messages
- Correctness requires synchronous operation
  - In step (2), EXAMINE used to determine if unmarked neighbor belongs to same component. If nodes of an unmarked edge are in different levels, problem!
  - Consider EXAMINE sent on edge (j, k), belonging to same component. But k may not have learnt it belongs to new component and new leader ID; and replies +ve
  - Can lead to cycles.

MST (asynchronous)

- Synchronous GHS *simulated* using extra msgs/steps.
  - New leader does BC/CC on marked edges of new component.
    - ⋆ In Step (2), recipient of EXAMINE can delay response if in old round
    - ⋆ $n \cdot log\ n$ extra messages overall
  - On involvement in a new round, inform each neighbor
    - ⋆ Send EXAMINE when all nbhs along unmarked edges in same round
    - ⋆ $l \cdot log\ n$ extra messages overall
- Engineer!! asynchronous GHS:
  - msg $O(n\ log\ n + l)$ time: $O(n \log\ n\,(l + d))$
  - Challenges
    - ⋆ determine levels of adjacent nodes
    - ⋆ repeated combining with singleton components $\Longrightarrow$ $log\ n$ becomes n
    - ⋆ If components at different levels, coordinate search for MWOEs, merging

Synchronizers

Definition

Class of transformation algorithms that allow a synchronous program (designed for a synchronous system) to run on asynchronous systems.

- Assumption: failure-free system
- Designing tailor-made async algo from scratch may be more efficient than using synchronizer

Process safety

Process i is safe in round r if all messages sent by i have been received.

Implementation key: signal to each process when it is safe to go to next round, i.e., when all msgs to be received have arrived

Synchronizers: Notation

$$M_a = M_s + (M_{init} + rounds \cdot M_{round}) \quad (1)$$

$$T_a = T_s + T_{init} + rounds \cdot T_{round} \quad (2)$$

- M_s: # messages in the synchronous algorithm.
- *rounds*: # rounds in the synchronous algorithm.
- T_s: time for the synchronous algorithm.
  Assuming one unit (message hop) per round, this equals *rounds*.
- M_{round}: # messages needed to simulate a round,
- T_{round}: # sequential message hops to simulate a round.
- M_{init}, T_{init}: # messages, # sequential message hops to initialize async system.

Synchronizers: Complexity

	Simple synchronizer	α synchronizer	β synchronizer	γ synchronizer
M_{init}	0	0	$O(n \cdot log(n) + \|L\|)$	$O(kn^2)$
T_{init}	d	0	$O(n)$	$n \cdot log(n)/log(k)$
M_{round}	$2\|L\|$	$O(\|L\|)$	$O(n)$	$O(L_c)$ $(\leq O(kn))$
T_{round}	1	$O(1)$	$O(n)$	$O(h_c)$ $(\leq O(log(n)/log(k)))$

The message and time complexities for synchronizers.
h_c is the greatest height of a tree among all the clusters.
L_c is the number of tree edges and designated edges in the clustering scheme for the γ synchronizer.

d is the graph diameter.

Simple Synchronizer

- A process sends each neighbor 1 message/round
  Combine messages or send dummy message
- On receiving a msg from each neighbor, go to next round.
- Neighbors P_i, P_j may be only one round apart
- P_i in $round_i$ can receive msg from only $round_i$ or $round_i + 1$ of neighbor.
- Initialization:
  - Any process may start round x.
  - In d time units, all processes would be in round x.
  - $T_{init} = d, M_{init} = 0$.
- Complexity: $M_{round} = 2|L|, T_{round} = 1$.

α Synchronizer

- P_i in round r moves to $r+1$ if all neighbors are *safe* for round r.
- When neighbor P_j receives ack for each message it sent, it informs P_i (and its other neighbors) that it is safe.

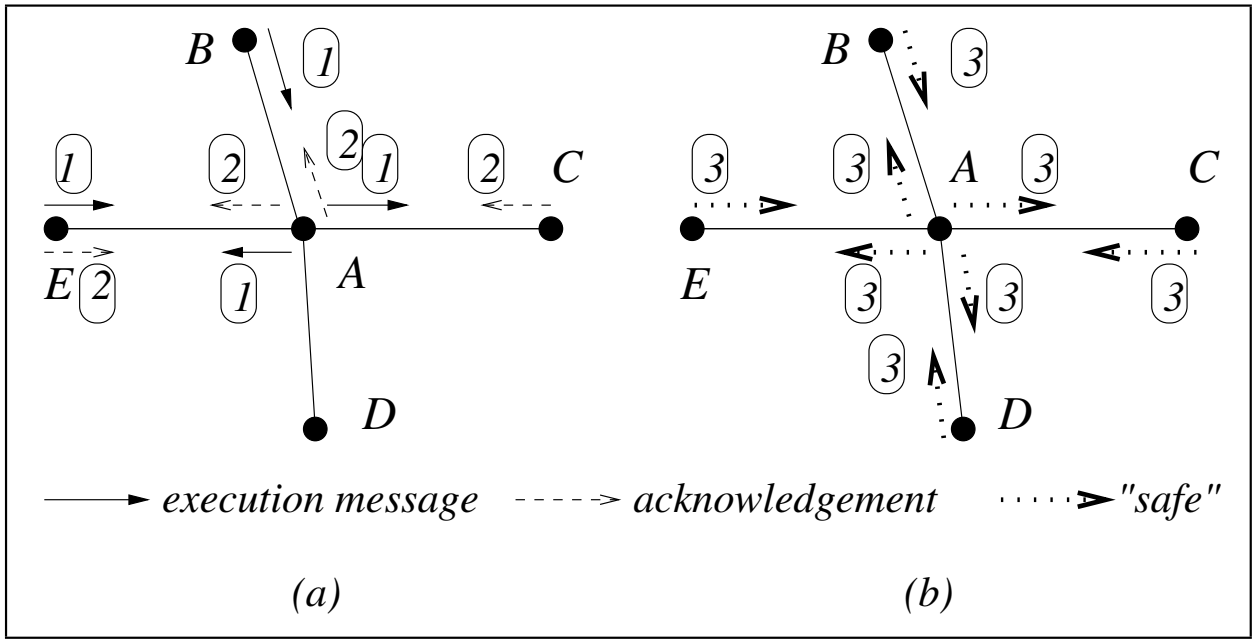

Figure 5.10: Example. (a) Execution msgs (1) and acks (2). (b) "I am safe" msgs (3).

α Synchronizer: Complexity

- Complexity:
  - l' msgs $\Rightarrow$ l' acks; transport layer acks $\Rightarrow$free!
  - $2|L|$ messages/round to inform neighbors of safety.
    $M_{round} = O(|L|).T_{round} = O(1)$.
- Initialization: None. Any process may spontaneously wake up.

β Synchronizer

Initialization: rooted spanning tree, $O(n \log n + |L|)$ messages, $O(n)$ time.
Operation:

- Safe nodes initiate convergecast (CvgC)
- intermediate nodes propagate CvgC when their subtree is safe.
- When root becomes safe and receives CvgC from all children, initiates tree broadcast to inform all to move to next round.

Complexity: l' acks for free, due to transport layer.

- $M_{round} = 2(n-1)$
- $T_{round} = 2 \log n$ average; $2n$ worst case

γ Synchronizer: Clusters

- Set of clusters; each cluster has a spanning tree
- Intra-cluster: β synchronizer over tree edges
- Inter-cluster: α synchronizer over *designated* inter-cluster edges. (For 2 neighboring clusters, 1 inter-cluster edge is *designated*.)

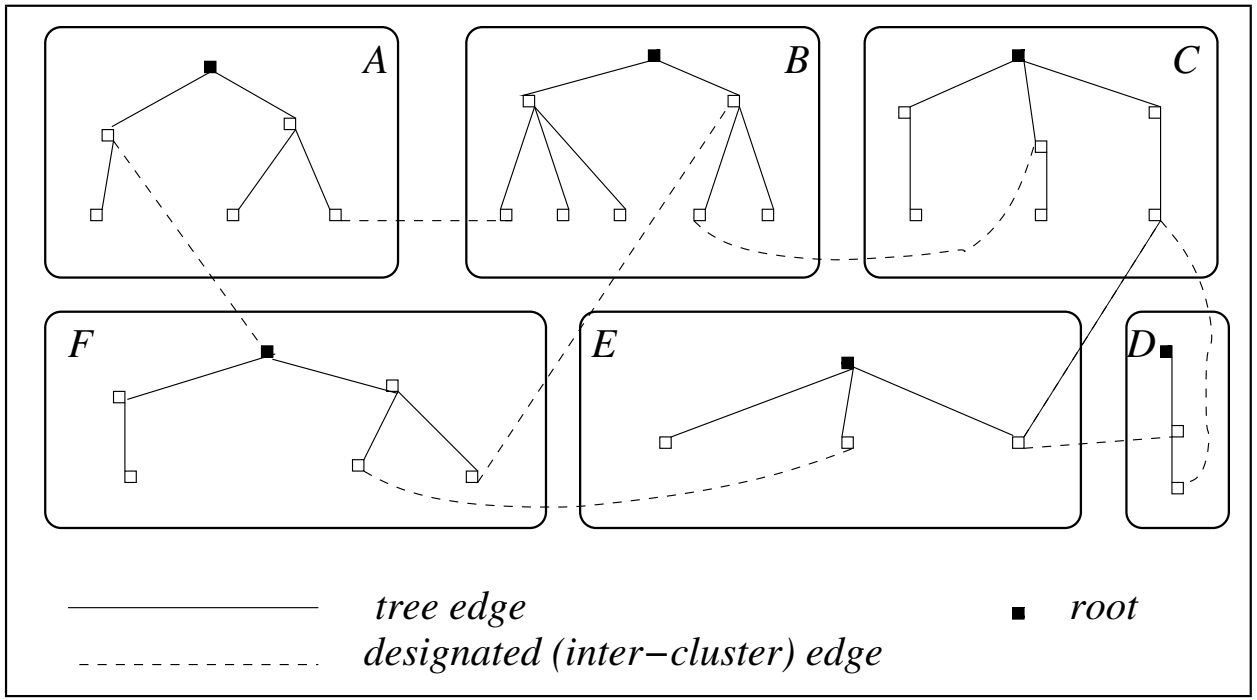

Figure 5.11: Cluster organization. Only tree edges and inter-cluster designated edges are shown.

γ Synchronizer: Operation and Complexity

- Within cluster, β synchronizer executed
- Once cluster is stabilized, α synchronizer over inter-cluster edges
- To convey stabilization of inter-cluster α synchronizer, within a cluster, CvgC and BC phases over tree
- This CvgC initiated by leaf nodes once neighboring clusters are stabilized.
- $M_{round} = O(L_c)$, $T_{round} = O(h_c)$.

γ Synchronizer: Code

(message types)
Subtree_safe // β synchronizer phase's convergecast within cluster
This_cluster_safe // β synchronizer phase's broadcast within cluster
My_cluster_safe // embedded inter-cluster α synchronizer's messages across cluster boundaries
Neighboring_cluster_safe // Convergecast following inter-cluster α synchronizer phase
Next_round // Broadcast following inter-cluster α synchronizer phase

for each *round* **do**

1. **(β synchronizer phase:)** This phase aims to detect when all the nodes within a cluster are safe, and inform all the nodes in that cluster.
    1. Using the spanning tree, leaves initiate the **convergecast** of the '*Subtree_safe*' message towards the root of the cluster.
    2. After the convergecast completes, the root initiates a **broadcast** of '*This_cluster_safe*' on the spanning tree within the cluster.
    3. **(Embedded α synchronizer:)**
        1. During this broadcast in the tree, as the nodes get engaged, the nodes also send '*My_cluster_safe*' messages on any incident *designated* inter-cluster edges.
        2. Each node also awaits '*My_cluster_safe*' messages along any such incident *designated* edges.
2. **(Convergecast and broadcast phase:)** This phase aims to detect when all neighboring clusters are safe, and to inform every node within this cluster.
    1. **(Convergecast:)**
        1. After the broadcast of the earlier phase (1.2) completes, the leaves initiate a convergecast using '*Neighboring_cluster_safe*' messages once they receive any expected '*My_cluster_safe*' messages (step (1.3)) on all the *designated* incident edges.
        2. An intermediate node propagates the convergecast once it receives the '*Neighboring_cluster_safe*' message from all its children, and also any expected '*My_cluster_safe*' message (as per step (1.3)) along *designated* edges incident on it.
    2. **(Broadcast:)** Once the convergecast completes at the root of the cluster, a '*Next_round*' message is broadcast in the cluster's tree to inform all the tree nodes to move to the next round.

Maximal Independent Set: Definition

- For a graph (N, L), an *independent set* of nodes N', where $N' \subset N$, is such that for each i and j in N', $(i, j) \notin L$.
- An independent set N' is a *maximal independent set* if no strict superset of N' is an independent set.
- A graph may have multiple MIS; perhaps of varying sizes.
  The largest sized independent set is the *maximum independent set*.
- Application: wireless broadcast - allocation of frequency bands (mutex)
- NP-complete

Luby's Randomized Algorithm, Async System

Iteratively:

- Nodes pick random nos, exchange with nbhs
- Lowest number in neighborhood wins (selected in MIS)
- If neighbor is selected, I am eliminated ($\Rightarrow$ safety)
- Only neighbors of selected nodes are eliminated ($\Rightarrow$ correctness)

Complexity:

- In each iteration, ≥ 1 selected, ≥ 1 eliminated $\Rightarrow \leq n/2$ iterations.
- Expected # iterations $O(log, n)$ due to randomized nature.

Luby's Maximal Independent Set: Code

```
(variables)
set of integer Neighbours                                   // set of neighbours
real random_i                                               // random number from a sufficiently large range
boolean selected_i                                          // becomes true when P_i is included in the MIS
boolean eliminated_i                                        // becomes true when P_i is eliminated from the candidate set
(message types)
RANDOM(real random)                                         // a random number is sent
SELECTED(integer pid, boolean indicator)                    // whether sender was selected in MIS
ELIMINATED(integer pid, boolean indicator)                  // whether sender was removed from candidates


(1a) repeat
(1b) if Neighbours = ∅ then
(1c)       selected_i ←— true; exit();
(1d) random_i ←— a random number;
(1e) send RANDOM(random_i) to each neighbour;
(1f) await RANDOM(random_j) from each neighbour j ∈ Neighbours;
(1g) if random_i < random_j (∀j ∈ Neighbours) then
(1h)       send SELECTED(i, true) to each j ∈ Neighbours;
(1i)       selected_i ←— true; exit();  // in MIS
(1j) else
(1k)       send SELECTED(i, false) to each j ∈ Neighbours;
(1l)       await SELECTED(j, ⋆) from each j ∈ Neighbours;
(1m)       if SELECTED(j, true) arrived from some j ∈ Neighbours then
(1n)            for each j ∈ Neighbours from which SELECTED(⋆, false) arrived do
(1o)                 send SELECTED(i, true) to j;
(1p)            eliminated_i ←— true; exit();  // not in MIS
(1q)       else
(1r)            send ELIMINATED(i, false) to each j ∈ Neighbours;
(1s)            await ELIMINATED(j, ⋆) from each j ∈ Neighbours;
(1t)            for all j ∈ Neighbours do
(1u)                 if ELIMINATED(j, true) arrived then
(1v)                      Neighbours ←— Neighbours \ {j};
(1w) forever.
```

Maximal Independent Set: Example

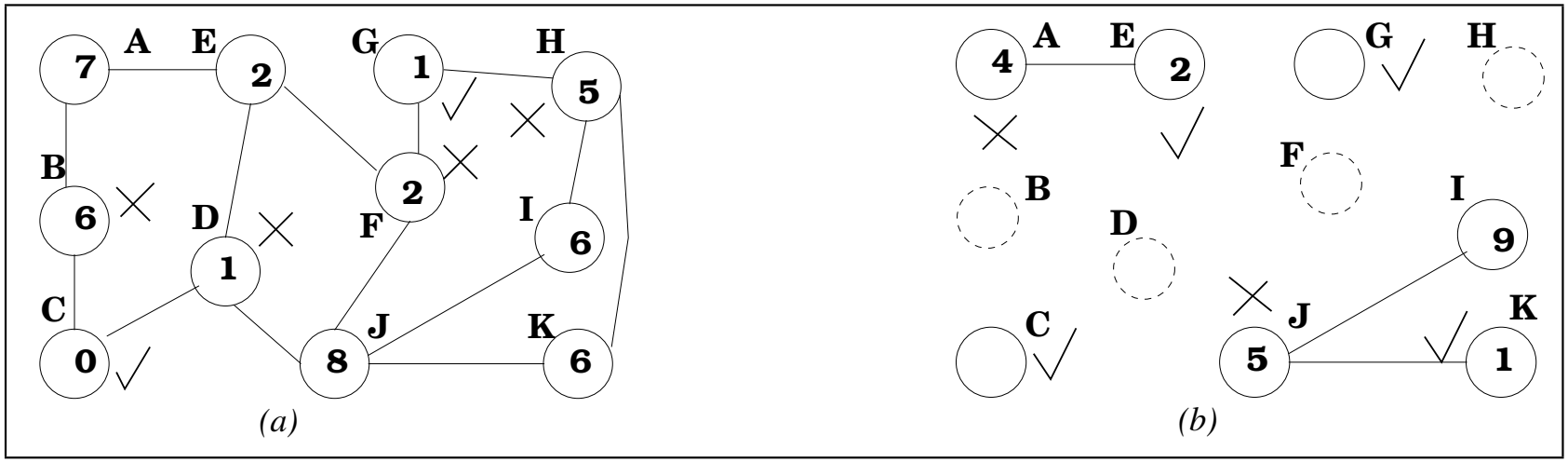

Figure 5.12: (a) Winners and losers in round 1. (b) Winners up to round 1, losers in round 2.

Third round: I is winner. MIS=$\{C, E, G, I, K\}$.

Note: $\{A, C, G, J\}$ is a smaller MIS.

Connected Dominating Set (CDS)

- A *dominating set* of graph (N, L) is a set $N' \subseteq N \mid$ each node in $N \setminus N'$ has an edge to some node in N'.
- A *connected dominating set* (CDS) of (N, L) is a dominating set N' such that the subgraph induced by the nodes in N' is connected.
- NP-Complete
  - Finding the minimum connected dominating set (MCDS)
  - Determining if there exists a dominating set of size $k < |N|$
- Poly-time heuristics: measure using approximation factor, stretch factor
  - Create ST; delete edges to leaves
  - Create MIS; add edges to create CDS
- Application: backbone for broadcasts

Compact Routing Tables (1)

Avoid tables of size n - large size, more processing time

- Hierarchical routing - hierarchical clustered network, e.g., IPv4
- Tree labeling schemes
  - Logical tree topology for routing
  - Node labels | dests reachable via link labeled by contiguous addresses $[x, y]$
  - Small tables but traffic imbalance

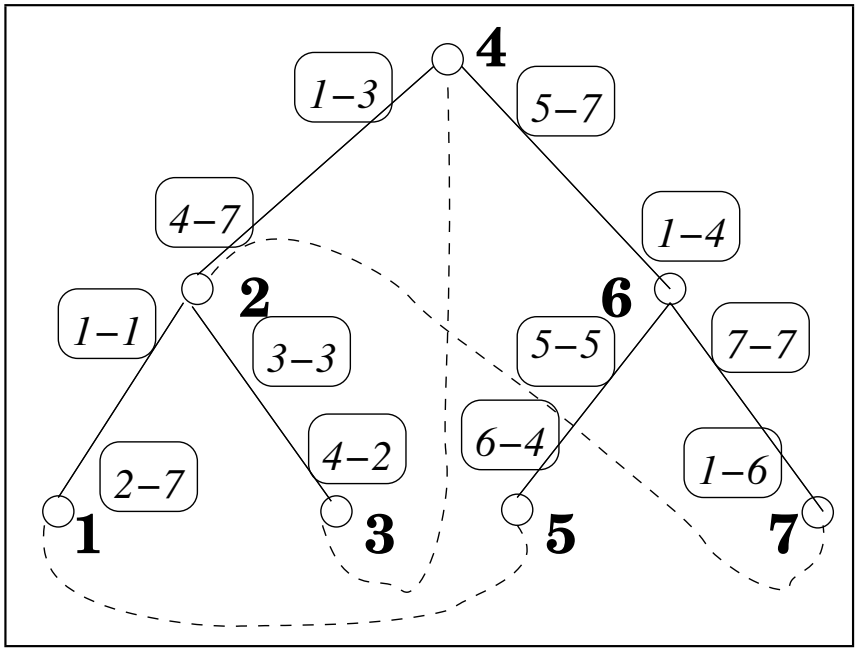

Figure 5.13: Tree label based routing tables. Tree edges labels in rectangles. Non-tree edges in dashed lines.

Compact Routing Tables (2)

- Interval routing:
  - Node labeling: $\mathcal{B}$ is a 1:1 mapping on N.
  - Edge labeling: $\mathcal{I}$ labels each edge in L by some subset of node labels $\mathcal{B}(N)$ | for any node x
    - ⋆ all destinations are covered ($\cup_{y \in Neighbours} \mathcal{I}(x, y) \cup \mathcal{B}(x) = N$) and
    - ⋆ there is no duplication of coverage ($\mathcal{I}(x, w) \cap \mathcal{I}(x, y) = \emptyset$ for $w, y \in Neighbours$).
  - For any s, t, there exists a path $\langle s = x_0, x_1 \ldots x_{k-1}, x_k = t \rangle$ where $\mathcal{B}(t) \in \mathcal{I}(x_{i-1}, x_i)$ for each $i \in [1, k]$.
  - Interval labeling possible for *every graph*!
  - No guarantee on path lengths; not robust to topology changes.
- Prefix routing: Node, channel labels from same domain, view as strings
  - To route: use channel whose label is longest prefix of dest.

Compact Routing Tables (3)

Stretch factor of a routing scheme r

$max_{i,j \in N} \{ \frac{distance_r(i,j)}{distance_{opt}(i,j)} \}$.

Designing compact routing schemes:

- rich in graph algorithmic problems
- Identify and prove bounds on efficiency of routes
- Different specialized topologies (e.g., grid, ring, tree) offer scope for easier results

Leader Election

- Defn: All processes agree on a common distinguished process (leader)
- Distributed algorithms not completely symmetrical; need a initiator, finisher process; e.g., MST for BC and CvgC to compute global function
- LeLang Chang Roberts (LCR) algorithm
  - Asynchronous unidirectional ring
  - All processes have unique IDs
  - Processes circulate their IDs; highest ID wins
  - Despite obvious optimizations, msg complexity $n \cdot (n-1)/2$; time complexity $O(n)$.
- Cannot exist deterministic leader election algorithm for anonymous rings
- Algorithms may be uniform

Leader Election - LCR algorithm: Code

(variables)
boolean *participate* ← *false* // becomes true when P_i is included in the MIS
(message types)
PROBE **integer** // contains a node identifier
SELECTED **integer** // announcing the result

(1) When a process wakes up to participate in leader election:
(1a) **send** PROBE(i) to right neighbor;
(1b) *participate* ⟵ *true*.

(2) When a PROBE(k) message arrives from the left neighbor P_j:
(2a) **if** *participate* = *false* **then** execute step (1) first.
(2b) **if** $i > k$ **then**
(2c) discard the probe;
(2d) **else if** $i < k$ **then**
(2e) **forward** PROBE(k) to right neighbor;
(2f) **else if** $i = k$ **then**
(2g) declare i is the leader;
(2h) circulate SELECTED(i) to right neighbor;

(3) When a SELECTED(x) message arrives from left neighbor:
(3a) **if** $x \neq i$ **then**
(3b) note x as the leader and forward message to right neighbor;
(3c) **else** do not forward the SELECTED message.

Leader Election: Hirschberg-Sinclair Algorithm

- Binary search in both directions on ring; token-based
- In each round k, each active process does:
  - Token circulated to 2^k nghbrs on both sides
  - P_i is a leader after round k iff i is the highest ID among 2^k nghbrs in both directions

    $\Rightarrow$ After round k, any pair of leaders are at least 2^k apart

    $\Rightarrow$ # leaders diminishes logarithmically as $n/2^k$
  - Only winner (leader) after a round proceeds to next round.
- In each round, max n msgs sent using supression as in LCR
- $log\ n$ rounds
- Message complexity: $O(n \cdot log\ n)$ (formulate exact expression)!
- Time complexity: $O(n)$.

Object Replication Problems

- Weighted graph (N, L), k users at $N_k \subseteq N$ nodes, r replicas of a object at $N_r \subseteq N$.
- What is the optimal placement of the replicas if $k > r$ and accesses are read-only?
  - Evaluate all choices for N_r to identify $\min(\sum_{i \in N_k, r_i \in N_r} dist_{i,r_i})$, where $dist_{i,r_i}$ is the cost from node i to r_i, the replica nearest to i.
- If Read accesses from each user in N_k have a certain frequency (or weight), the minimization function changes.
- Address BW of each edge.
- Assume user access is a Read with prob. x, and an Update with prob. $1 - x$. Update requires all replicas to be updated.
  - What is the optimal placement of the replicas if $k > r$?

Adaptive Data Replication: Problem Formulation

Network (V, E). Assume single replicated object.

- *Replication scheme*: subset R of V | each node in R has a replica.
- r_i, w_i: rates of reads and writes issued by i
- $c_r(i)$, $c_w(i)$: cost of a read and write issued by i.
- $\mathcal{R}$: set of all possible replication schemes.
- Goal: minimize cost of the replication scheme:

$$\min_{R \in \mathcal{R}} [\sum_{i \in V} r_i \cdot c_r(i) + \sum_{i \in V} w_i \cdot c_w(i)]$$

  Arbitrary graph: cost is NP-Complete
- Hence, assume tree overlay
- Assume one copy serializability, implemented by Read-One-Write-All (ROWA) policy.

Adaptive Data Replication over Tree Overlay

- All communication, set R on tree T overlay
- R: amoeba-like subgraph, moves to center-of-gravity of activity
  - Expands when Read cost is higher
  - Shrinks when Write cost is higher
  - Equilibrium-state R is optimal; converges in $d + 1$ steps once Read-Write pattern stabilizes
  - Dynamic activity: algorithm re-executed in epochs
- Read: From closest replica, along T. Use *parent* pointers.
- Write: To closest replica, along T. Then propagate in R.
  Use $R - neighbor$, set of neighbors in R.
- Implementation: (i) in R? (ii) $R - neighbor$, (iii) *parent*.

Adaptive Data Replication: Convergence (1)

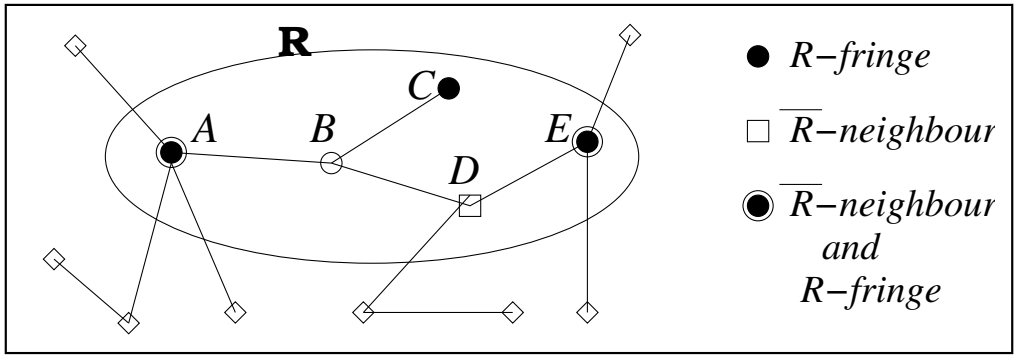

Figure 5.14: Nodes in ellipse belong to R.

- C is *R-fringe*
- A, E are *R-fringe* and $\overline{R}$-*neighbour*
- D is $\overline{R}$-*neighbour*

$\overline{R}$-neighbour: $i \in R$; and has at least one neighbour $j \notin R$.

R-fringe: $i \in R$; and has only one neighbour $j \in R$.
Thus, i is a leaf in the subgraph of T induced by R and j is parent of i.

singleton: $|R| = 1$ and $i \in R$.

Adaptive Data Replication: Tests

Tests at end of each epoch.

Expansion test: $\overline{R}$-*neighbour* node i includes neighbor j in R if $r > w$.

Contraction test: R-*fringe* node i excludes itself from R if $w > r$. Before exiting, seek permission from j to avoid $R = \emptyset$.

Switch test: Singleton node i transfers its replica to j if $r + w$ being forwarded by j is greater than $r + w$ that node i receives from all other nodes.

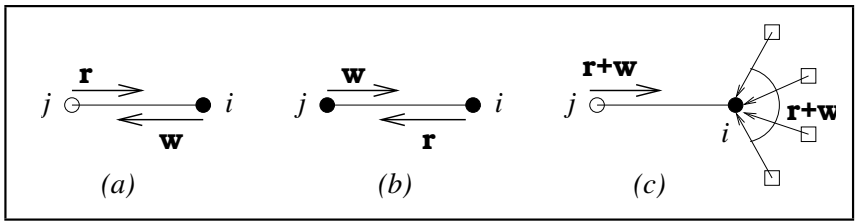

Figure 5.15: (a) Expansion test. (b) Contraction test. (c) Switch test.

$\overline{R}$-*neighbour* may also be R-*fringe* or *singleton*. In either case, the *expansion test* executed first; if it fails, *contraction test* or *switch test* is executed.

Adaptive Data Replication: Code (1)

(variables)
array of integer $Neighbours[1 \ldots b_i]$; // b_i neighbours in tree T topology
array of integer $Read\_Received[1 \ldots |b_i|]$; // jth element gives # reads from $Neighbours[j]$
array of integer $Write\_Received[1 \ldots |b_i|]$; // jth element gives # writes from $Neighbours[j]$
integer $write_i$, $read_i$; // # writes and # reads issued locally
boolean *success*;

(1) P_i determines which tests to execute at the end of each epoch:
(1a) **if** i is $\overline{R}$-*neighbour* and *R-fringe* **then**
(1b) **if** *expansion test* fails **then**
(1c) *reduction test*
(1d) **else if** i is $\overline{R}$-*neighbour* and *singleton* **then**
(1e) **if** *expansion test* fails **then**
(1f) *switch test*
(1g) **else if** i is $\overline{R}$-*neighbour* and not *R-fringe* and not *singleton* **then**
(1h) *expansion test*
(1i) **else if** i is $\overline{R} - neighbour$ and *R-fringe* **then**
(1j) *contraction test*.

(2) P_i executes *expansion test:*
(2a) **for** j **from** 1 **to** b_i **do**
(2b) **if** $Neighbours[j]$ not in R **then**
(2c) **if** $Read\_Received[j] > (write_i + \sum_{k=1 \ldots b_i, k \neq j} Write\_Received[k])$ **then**
(2d) send a copy of the object to $Neighbours[j]$; $success \longleftarrow 1$;
(2e) **return**(*success*).

Adaptive Data Replication: Code (2)

(variables)
array of integer $Neighbours[1 \ldots b_i]$; // b_i neighbours in tree T topology
array of integer $Read\_Received[1 \ldots |b_i|]$; // jth element gives # reads from $Neighbours[j]$
array of integer $Write\_Received[1 \ldots |b_i|]$; // jth element gives # writes from $Neighbours[j]$
integer $write_i$, $read_i$; // # writes and # reads issued locally
boolean *success*;

(3) P_i executes *contraction test:*
(3a) let $Neighbours[j]$ be the only neighbour in R;
(3b) **if** $Write\_Received[j] > (read_i + \sum_{k=1 \ldots b_i, k \neq j} Read\_Received[k])$ **then**
(3c) seek permission from $Neighbours[j]$ to exit from R;
(3d) **if** permission received **then**
(3e) *success* $\longleftarrow$ 1; inform all neighbours;
(3f) **return**(*success*).

(4) P_i executes *switch test:*
(4a) **for** j **from** 1 **to** b_i **do**
(4b) **if** $(Read\_Received[j] + Write\_Received[j]) >$
$[\sum_{k=1 \ldots b_i, k \neq j}(Read\_Received[k] + Write\_Received[k]) + read_i + write_i]$ **then**
(4c) transfer object copy to $Neighbours[j]$; *success* $\longleftarrow$ 1; inform all neighbours;
(4d) **return**(*success*).