Distributed Systems

global state – observation

Johan Montelius

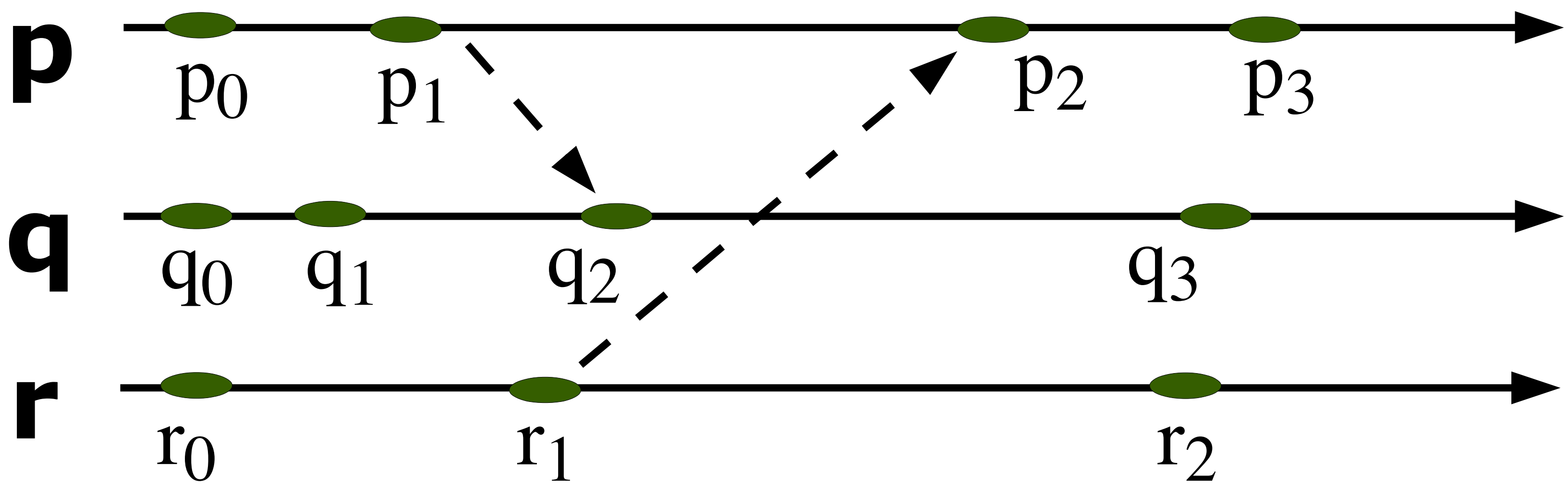
Leandro Navarro

Global state

- Time is very much related to the notion of global state.
- If we cannot agree on a time how should we agree on a global state.
- Global state is important:
  - garbage collection
  - dead-lock detection
  - termination
  - debugging

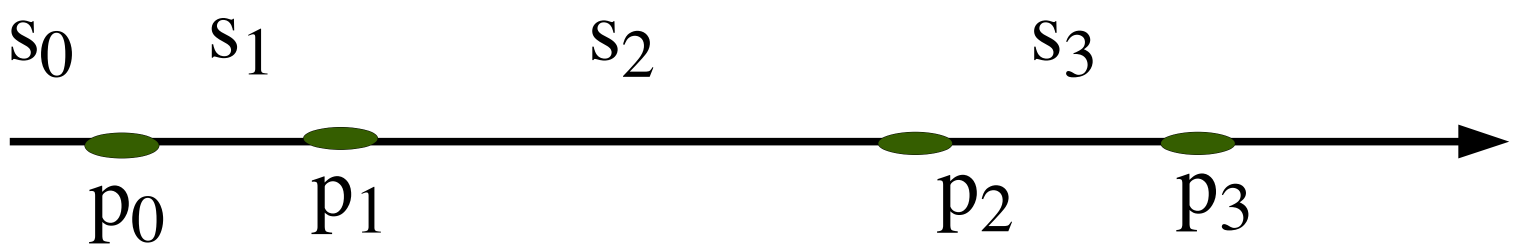
Global state

- Given a partial order of events, can we say anything about the state of the system?



History and state

- The history of a process is a sequence of events: <p_0, p_1, ..>
- The state of a process, s_i, is the state before the i'th event.

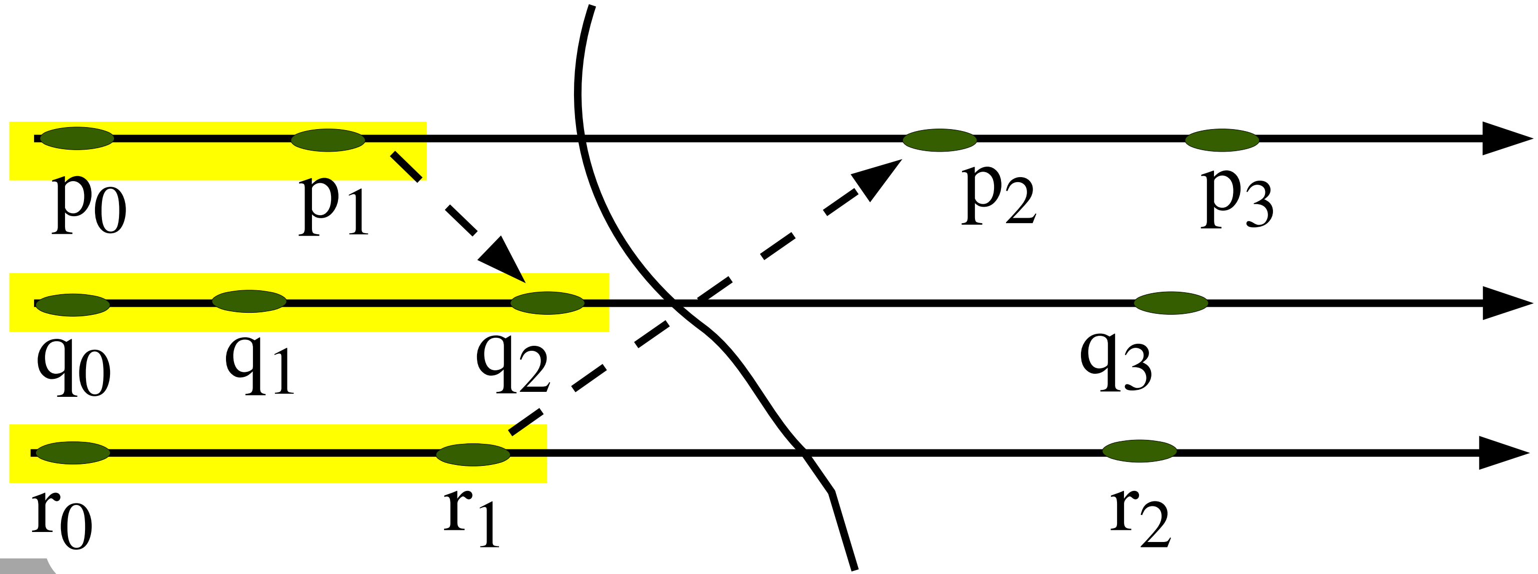


History and state

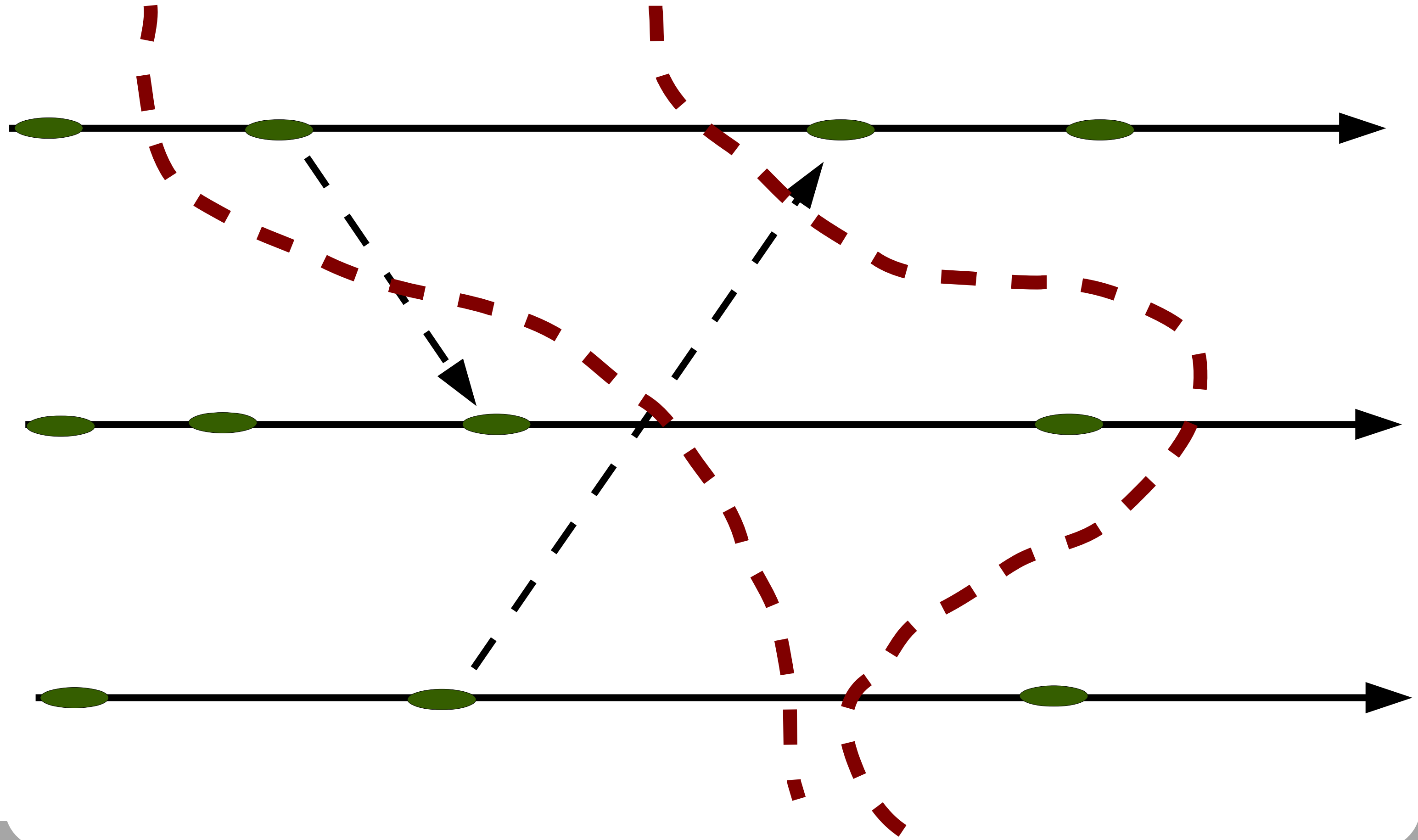
- Is the state of a process the history of events?
- What is the global state of a distributed system?
  - the union of histories of all processes
- Do all unions make sense?

Global history and cuts

- The global history is the union of all histories.
- A *cut* is the global history up to a specific event in each history.

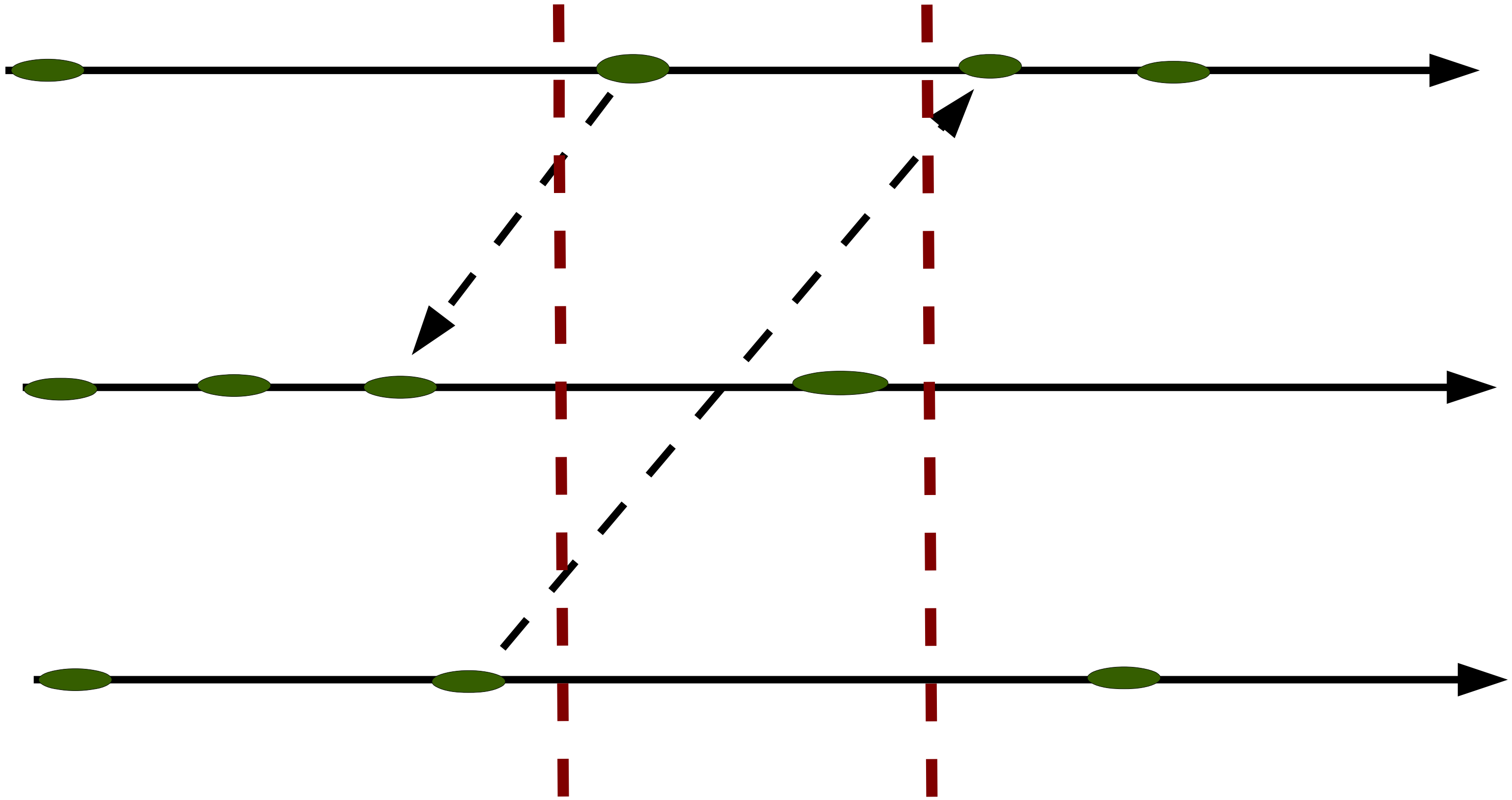


different cuts



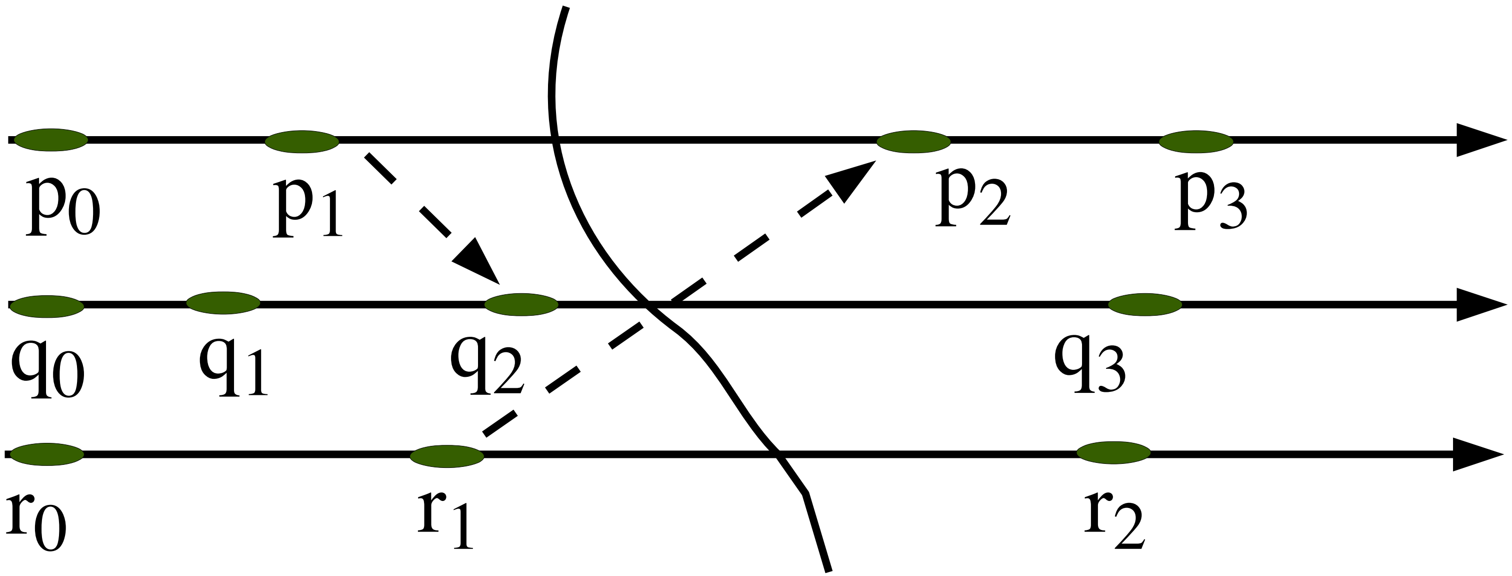
different cuts – rubber band transformations

- Abstract from real time but: preserve causal relations, otherwise concurrent



Consistent cuts

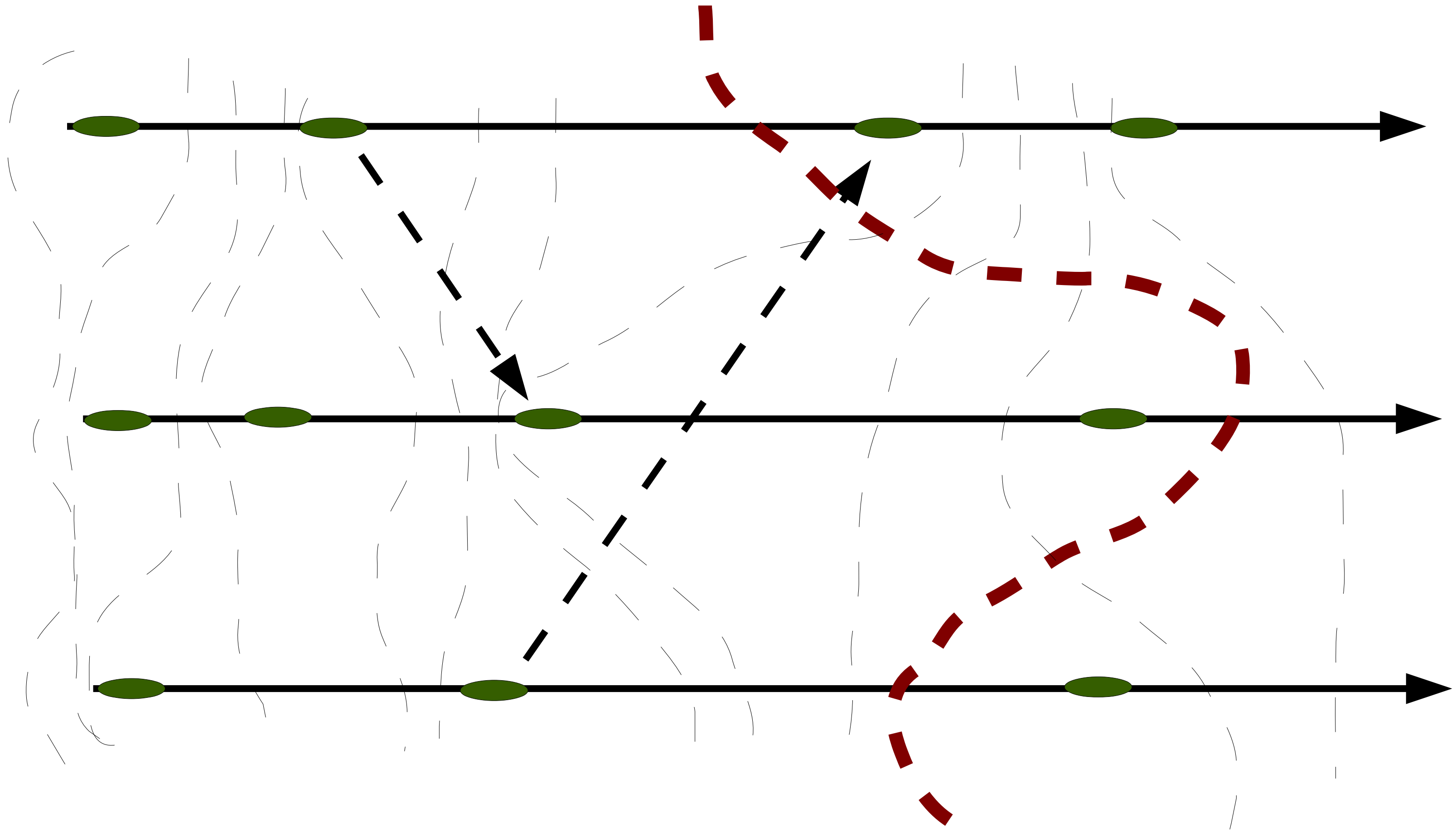
- How can we tell if a cut is *consistent?*
  - *For each event e in the cut:*
    - *if* e *happened before* e' *then*
    - e *is also in the cut.*



Consistent global state

- A consistent cut corresponds to a consistent global state.
  - it is a possible state without contradictions
  - an actual execution might not have passed through the state

consistent cut

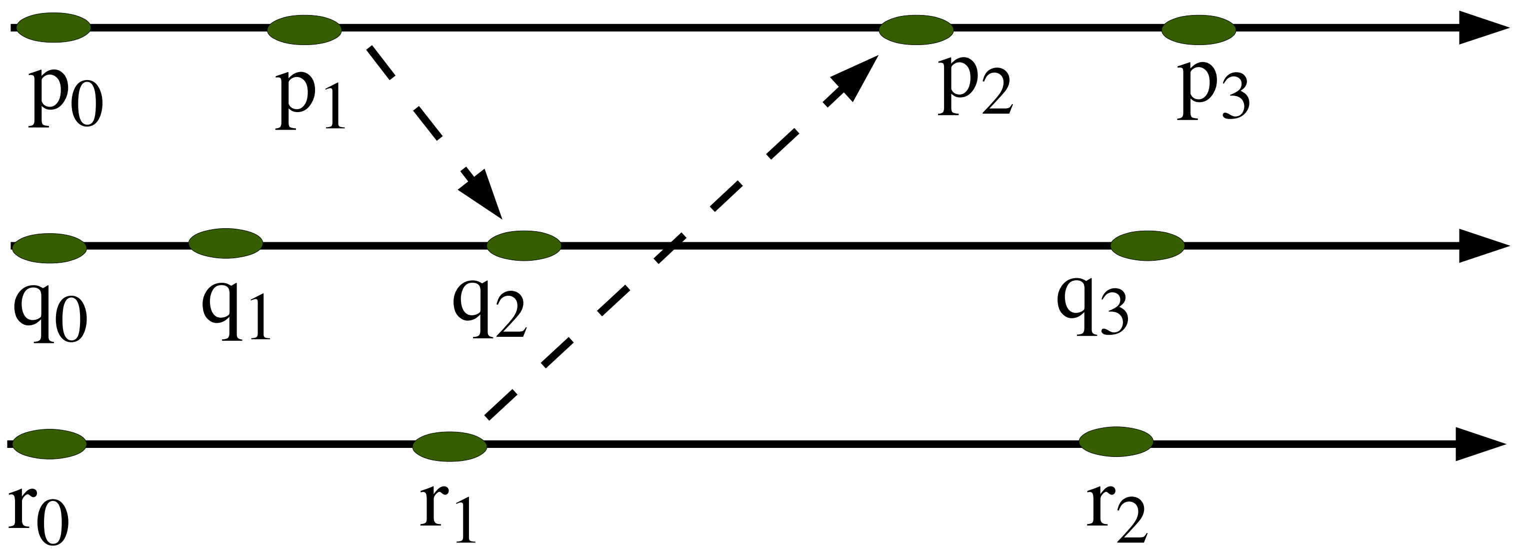


Linearization

- A *run* is a total ordering of all events in a global history that is consistent with each local history.
- A *linearization* or *consistent run* is a run that describes transitions between consistent global states.
- A state S' is reachable from state S if there is a linearization from S to S'.

Linearization

$$\{p_0, p_1, q_0, r_0, q_1, r_1, p_2, p_3, q_2, r_2, q_3\}$$



Why is this important?

- If we can collect all events and know the *happened before order,* then we can construct <u>all possible linearizations</u>.
- We know that the actual execution took one of these paths.
- Can we say something about the execution even though we do not know which path was taken?

Global predicates

- A *global state predicate* is a property that is true or false for a global state.
  - stable: if a predicate becomes true it remains true for all reachable states.

    e.g. deadlock, termination, object is garbage, ...
  - non-stable: if a predicate can become true and then later become false

    e.g. $x=y$, …

Properties of executions

- Safety
  - a predicate is never true in any state reachable from S_0.
- Liveness
  - a predicate that eventually evaluates to true for any linearization from S_0.

Predicates

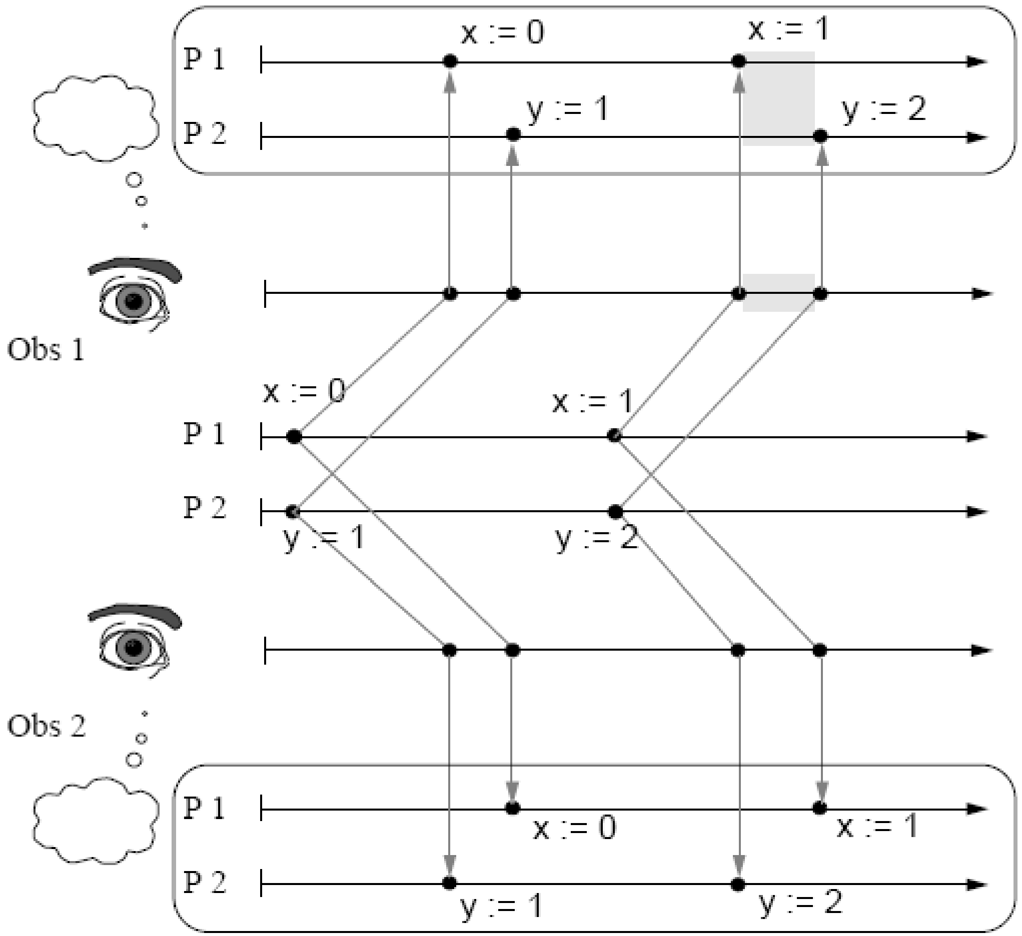
- If we can show that a *stable predicate* is true given a snapshot then it is also true in the execution.
  - garbage collection
  - dead-lock detection
- A *non-stable* predicate might be true in the snapshot but not necessarily during the actual execution.

Possibly and definitely

- We want to know if a non-stable predicate *possibly* occurred or *definitely* occurred.
- We have to look at all possible execution states.
  - How do we reconstruct all possible execution states?
  - Is the simple snapshot enough?

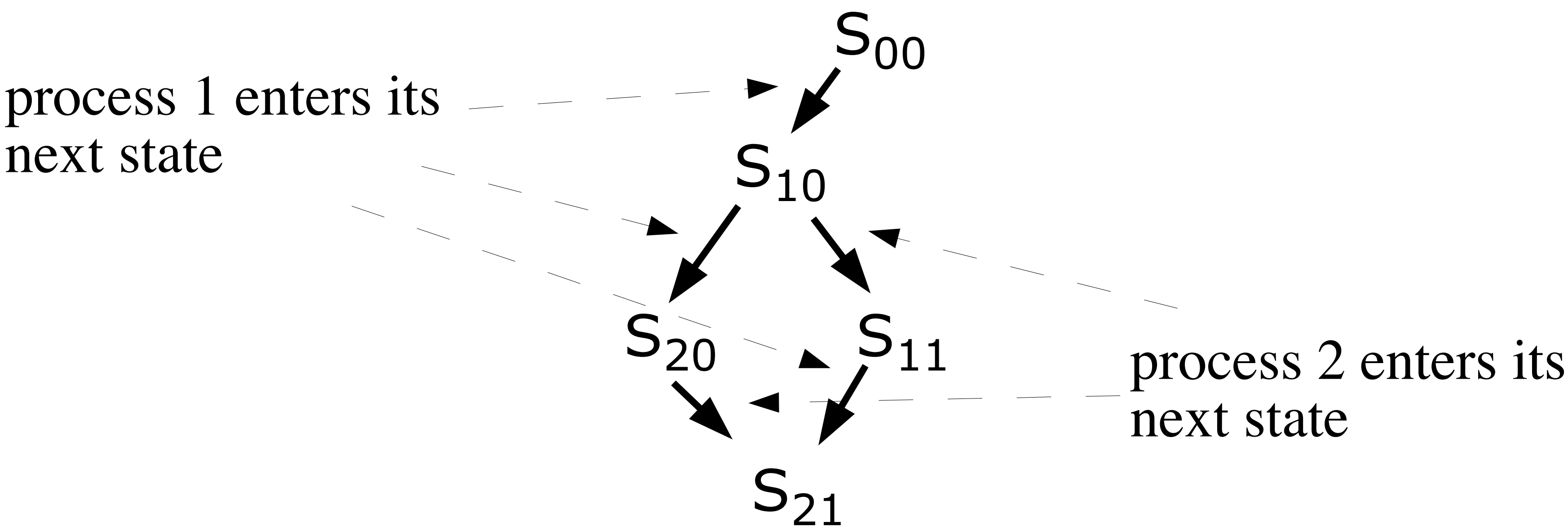
Global predicate (ex)

- Does (x=y) hold?

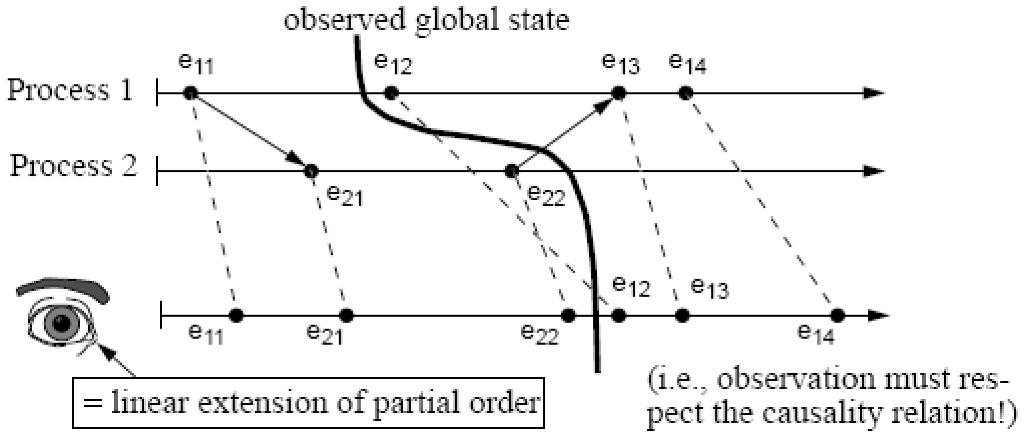


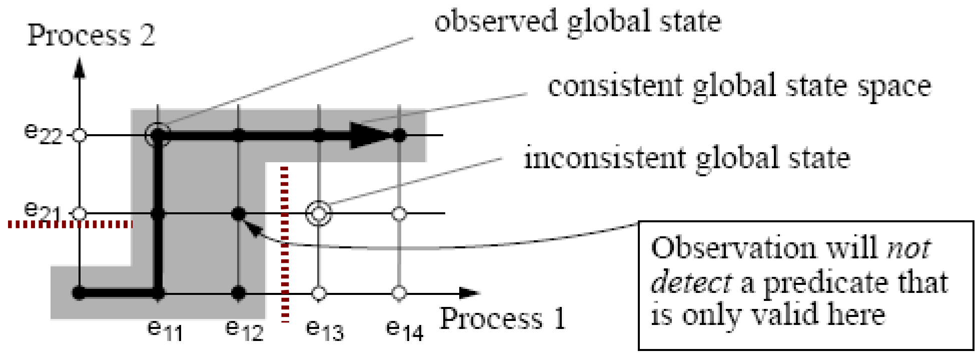
Global state lattice

- Consistent global states form a lattice where each edge is a *possible transition* of a process.



The lattice of consistent states

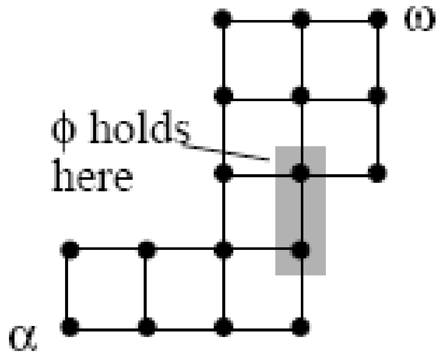




Possibly true

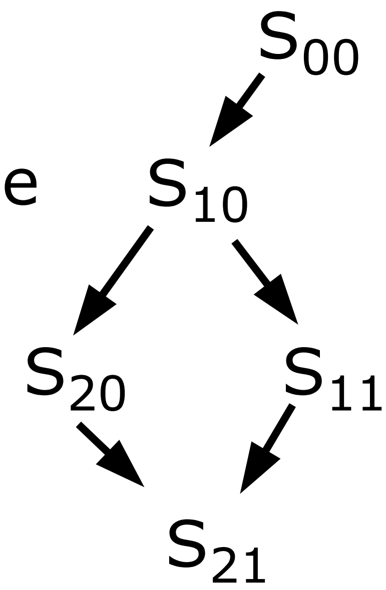
- At least one observer sees Φ

  No more than one traffic light green →
  Possibly(Φ)=false



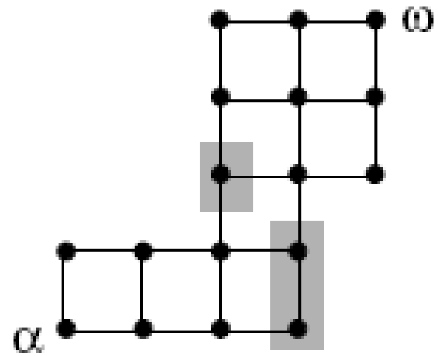
possibly ϕ holds

$p(S_{20})$ = true

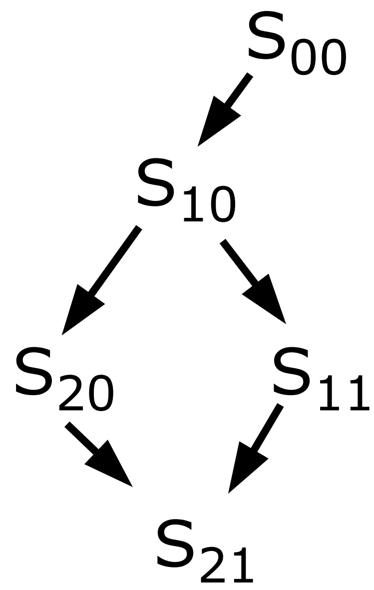


Definitely true

- All observers see Φ



definitely ϕ holds



$$p(S_{20}) = \text{true}$$
$$p(S_{11}) = \text{true}$$

Possibly and definitely

- If a predicate is *true in any* consistent global state of the lattice then it is *possibly true* in the execution.
- If we *cannot find a path* from the initial state to the final state without reaching a state for which a predicate is true then the predicate is *definitely true* during the execution.

Election

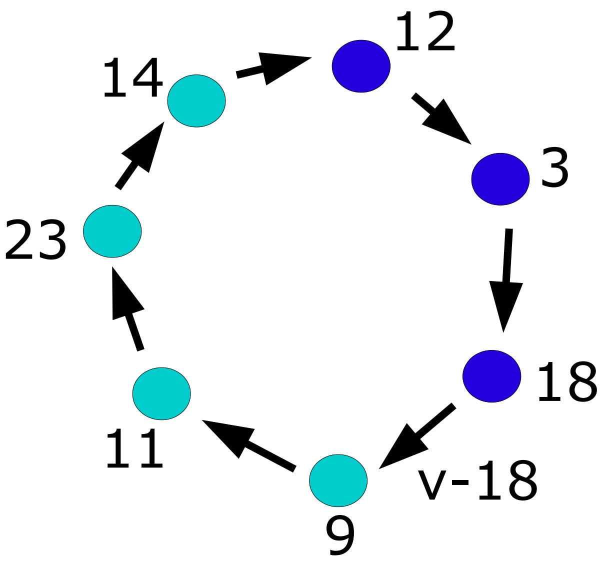
- **Algorithms to choose a unique process to play a particular role**
- Many algorithms require a server but if no node is assigned to be the server or if the server crashes we need a new server.
- Assumptions:
  - any node can *call an election* but it can only call one at a time
  - a node is either *participant* or *non-participant*
  - nodes have identifiers that are ordered

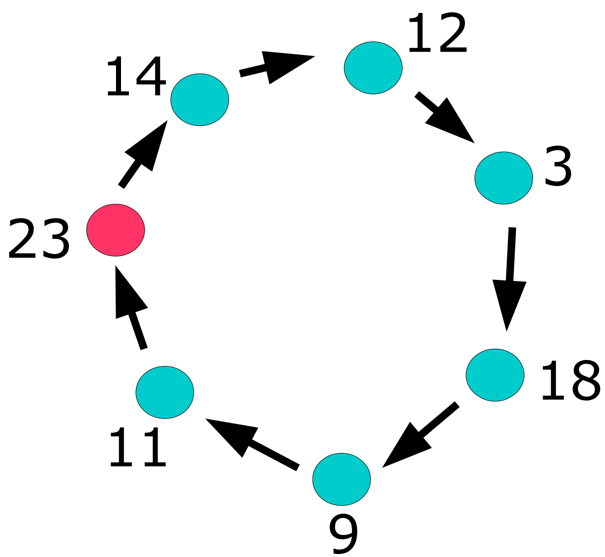
Election

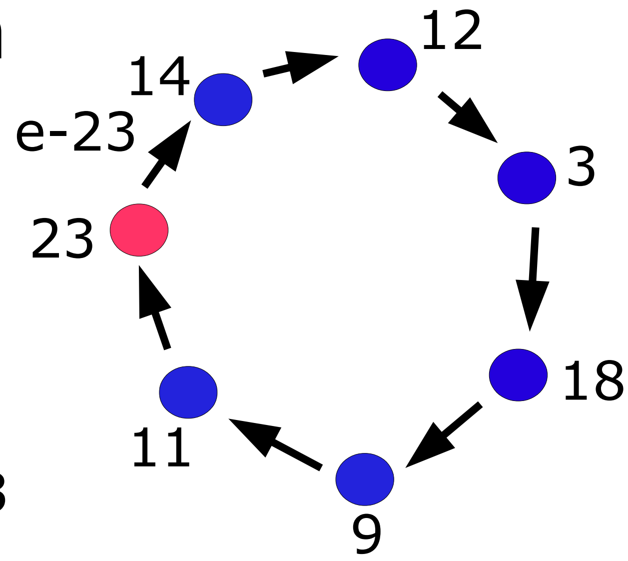
- Requirements
  - Safety: a *participant* is either non-decided or decided with P, a unique non crashed node with largest ID
  - Liveness: all nodes eventually *participate* and decide on a elected node, or crash
- Efficiency
  - number of messages
  - turnaround time: delay/#messages from *call* to *close*

Ring-based election

- Elect "coordinator" node (largest Id)
- Any process can begin election







Ring-based election

- Every process non-participant, any can start
- Election, node i: status(p_i) ← participant, send {election, i}
- P_j on {election, i}: status(pi) ← participant

  if(i>j) forward {election, I}

  if (j>i)

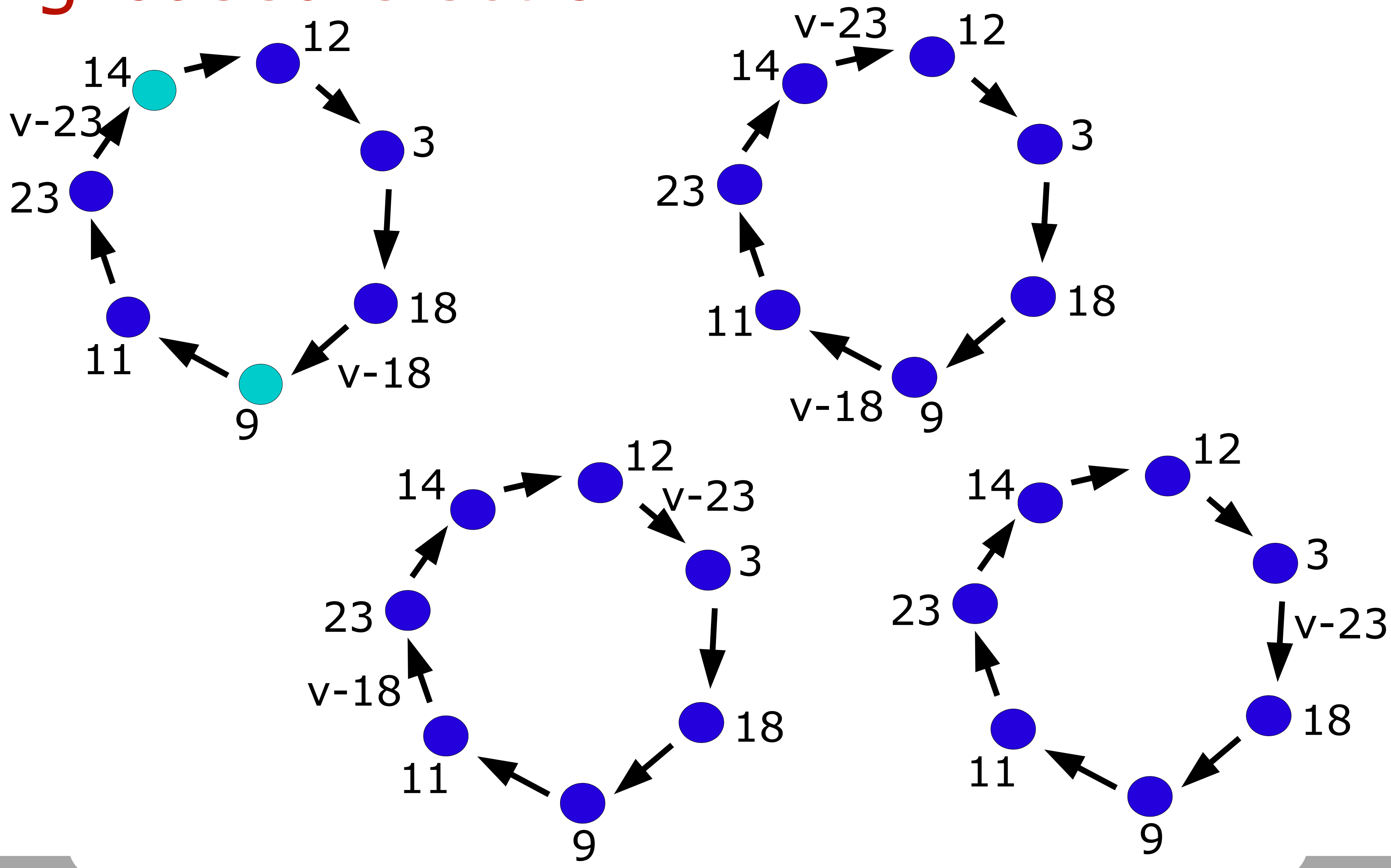
  if ! participant send {election, j}

  else discard // participant

  if (i==j) P_j ← coordinator; status ← elected;

  send {elected, j}

Ring-based election



Ring-based election

- Requirements
  - safety
  - liveness
- Efficiency
  - messages: best case, worst case?
  - turnaround:
- Failure
  - hmm, ...

+recent

Consensus – Paxos

Johan Montelius

Leandro Navarro

Consensus - Paxos

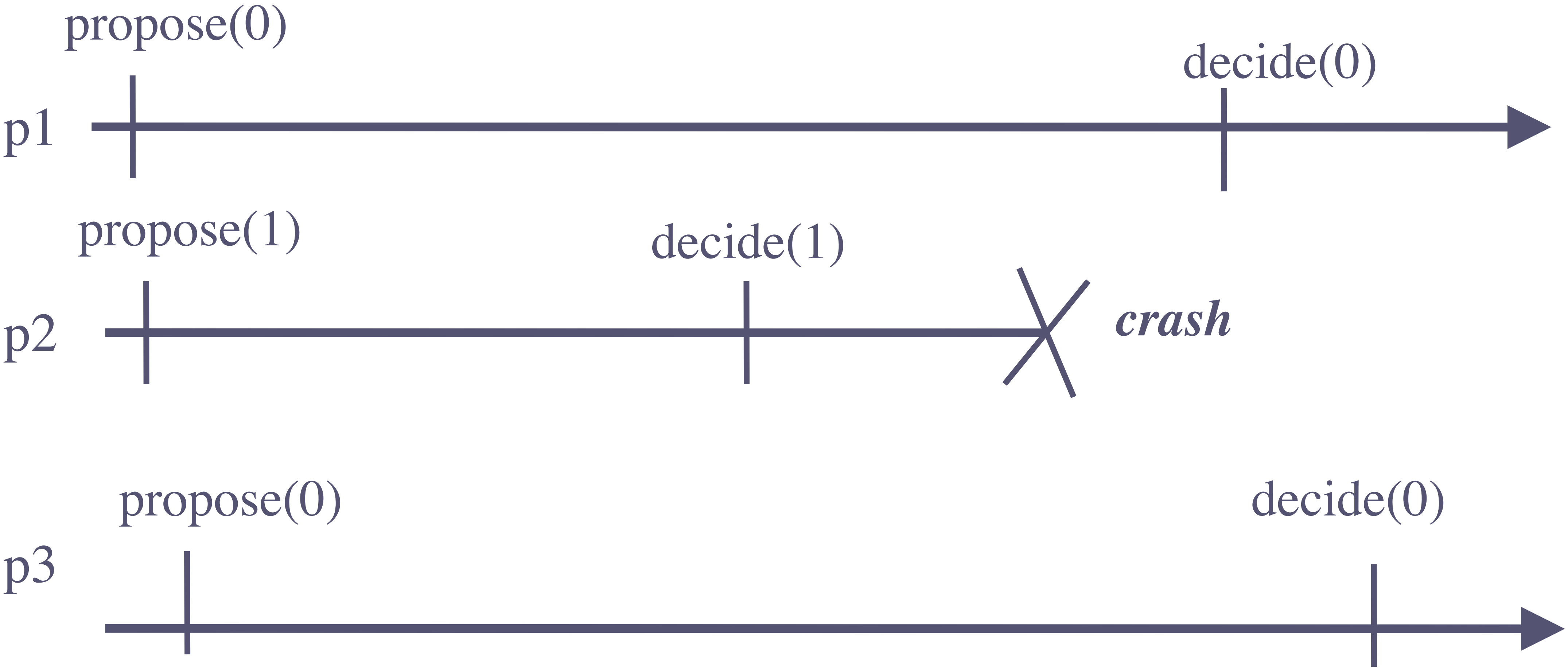
- In the consensus problem, the processes propose values and have to agree on one among these values

- Solving consensus is key to solving many problems in distributed computing (e.g., total order broadcast, atomic commit, terminating reliable broadcast)
- … in the presence of faults !

Consensus

- **Validity**: Any value decided is a value proposed
- **Agreement**: No two correct processes decide differently
- **Termination**: Every correct process eventually decides
- **Integrity**: No process decides twice

Consensus


propose(0)
decide(0)
p1
propose(1)
decide(1)
crash
p2
propose(0)
decide(0)
p3

Paxos

- **Paxos is a consensus protocol based on gaining votes from a quorum**
- Nodes can crash but are restarted and will remember where in the protocol they were
- Messages can take arbitrary long time to be delivered, get lost or duplicated but not corrupted
  - Messages signed, if corruption is an issue

Outline

- Proposers:
  - send request with sequence number (**n**)
- Acceptors:
  - promise not to accept a proposal with lower sequence number
- Proposer:
  - collect promises and cast a ballot (value **v**)
- Acceptors
  - accept if they have not promised else

Proposer

- Operates in rounds, each round using a higher unique sequence number
- In a round
  - Send prepare messages: sequence number (**n**)
  - Collect all replies

    If majority for n:
  - Choose proposal (value):
    keep the proposal with highest sequence number, or choose one if not reported
  - Request votes for the proposal:
    send accept messages (n,v)
  - If quorum vote for, we have reached consensus
  - If not back-off and do another round

Acceptor

- Keeps track of:
  - a sequence number below which it will never accept a proposal
  - the accepted value with the highest sequence number that it has voted for
- If requested to prepare
  - Ok if not promised else
  - return accepted value and sequence number
- If requested to vote
  - Ok, if not promised otherwise

Failures

- Acceptors need never reply on anything
  - the protocol will never end in more than one value being selected by a quorum
- A proposer can abort and restart anytime
  - must select unique sequence numbers
- Progress is not guaranteed
  - two proposers can fight forever over a quorum
- Strategy
  - elect one distinguished proposer

Summary

- Paxos is a quorum-based consensus protocol
- Will agree on one unique value even if:
  - nodes fail and restart
  - messages are lost
- Acceptors must remember
  - made promises
  - accepted value of highest sequence

Replication

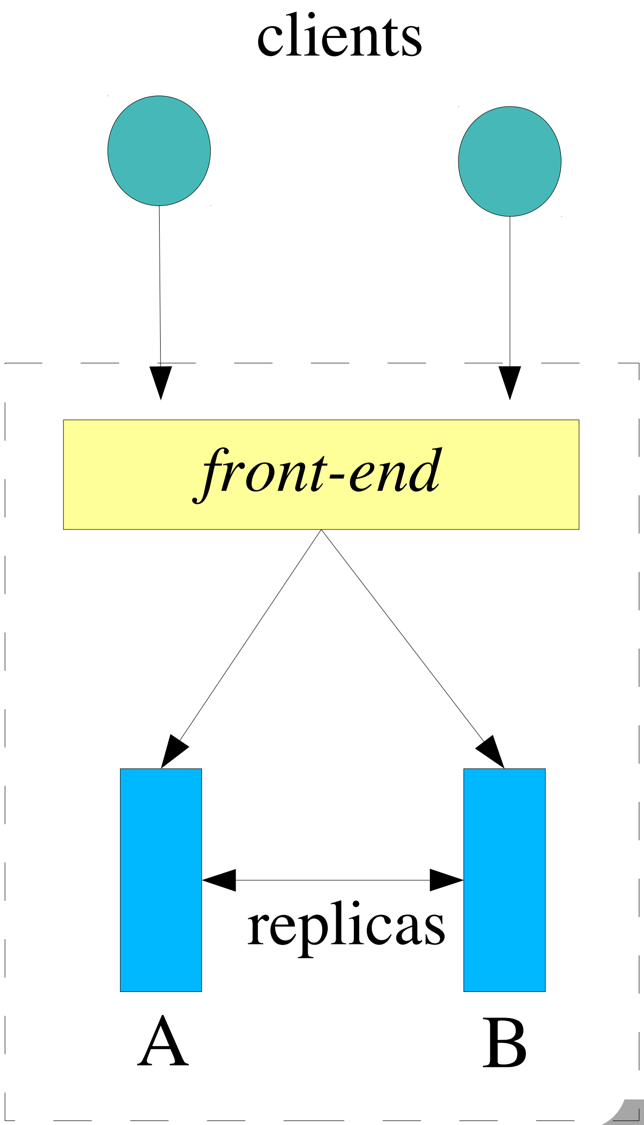
- The problem we have:
  - servers might be unavailable
  - low performance
- The solution:
  - keep duplicates at different servers
- What we could get:
  - Better performance (for more resources)
  - Increased availability (despite failures, weak?)
  - Fault tolerance (correct behaviour despite ...)

Building a fault-tolerant service

- When building a fault-tolerant service with replicated data and functionality the service:
  - should produce the same results as a non-replicated service
  - should respond despite node crashes
  - ... if the cost is not too high

A first try...

- two replicas
- replica acknowledge operation then copy to peer
- front-end uses one replica and switch if replica fails



Let's see...

front-end directs requests to replica B

Client 1

```
setBalanceB(x,10);
  - B fails -
setBalanceA(y,20);
```

Client 2

```
getBalanceA(y);-> 20
getBalanceA(x);-> 0
```

A fault-tolerant service - not

- This does not give us a correct service, but why?
- What are the requirements of a correct service?

Client 1	**Client 2**
`setBalance`$_{\mathbf{B}}$`(x,10);`	
`setBalance`$_{\mathbf{A}}$`(y,20);`	
	`getBalance`$_{\mathbf{A}}$`(y);-> 20`
	`getBalance`$_{\mathbf{A}}$`(x);-> 0`

Correct behavior

- When talking about correct behavior we look at the sequence of operations and their returned values.

`setBalance`$_{\mathbf{B}}$`(x,10);` `setBalance`$_{\mathbf{A}}$`(y,20);`	`getBalance`$_{\mathbf{A}}$`(y);-> 20` `getBalance`$_{\mathbf{A}}$`(x);-> 0`

What is a correct behavior

- A replicated service is said to be *linearizable* if for *any execution* there is *some interleaving* that ...
  - meets the specification of a non-replicated service
  - matches the *real time* order of operations in the real execution

A less restricted

- A replicated service is said to be *sequentially consistent* if for any *execution* there is *some interleaving* that ...
  - meets the specification of a non-replicated service
  - matches the *program order* of operations in the real execution at each client

No “real time”
No total order on all ops
Ony program order at each client

Sequential consistency

- Can we find an interleaving with a correct behavior.

Client 1

```
setBalanceB(x,10);
setBalanceA(y,20);
```

Client 2

```
getBalanceA(y);-> 20
getBalanceA(x);-> 0
```

$getBalance_A(x);\text{-> } 0,\ setBalance_B(x,10),\ setBalance_A(y,20),\ getBalance_A(y);\text{-> } 20$

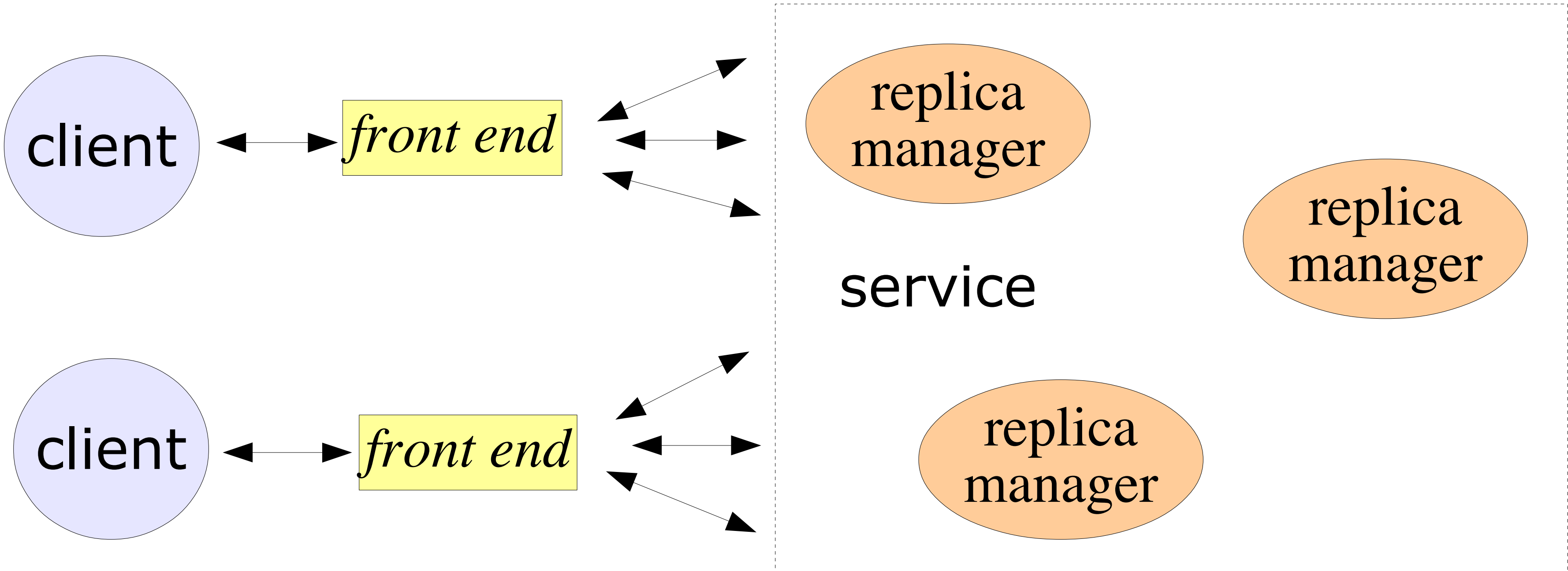
Sequentially consistent

- Is this behavior sequentially consistent?
- linearizable?

Client 1	**Client 2**
`setBalance`$_{\mathbf{B}}$`(x,10);`	
	`getBalance`$_{\mathbf{A}}$`(y);-> 0`
	`getBalance`$_{\mathbf{A}}$`(x);-> 0`
`setBalance`$_{\mathbf{A}}$`(y,20);`	

System model

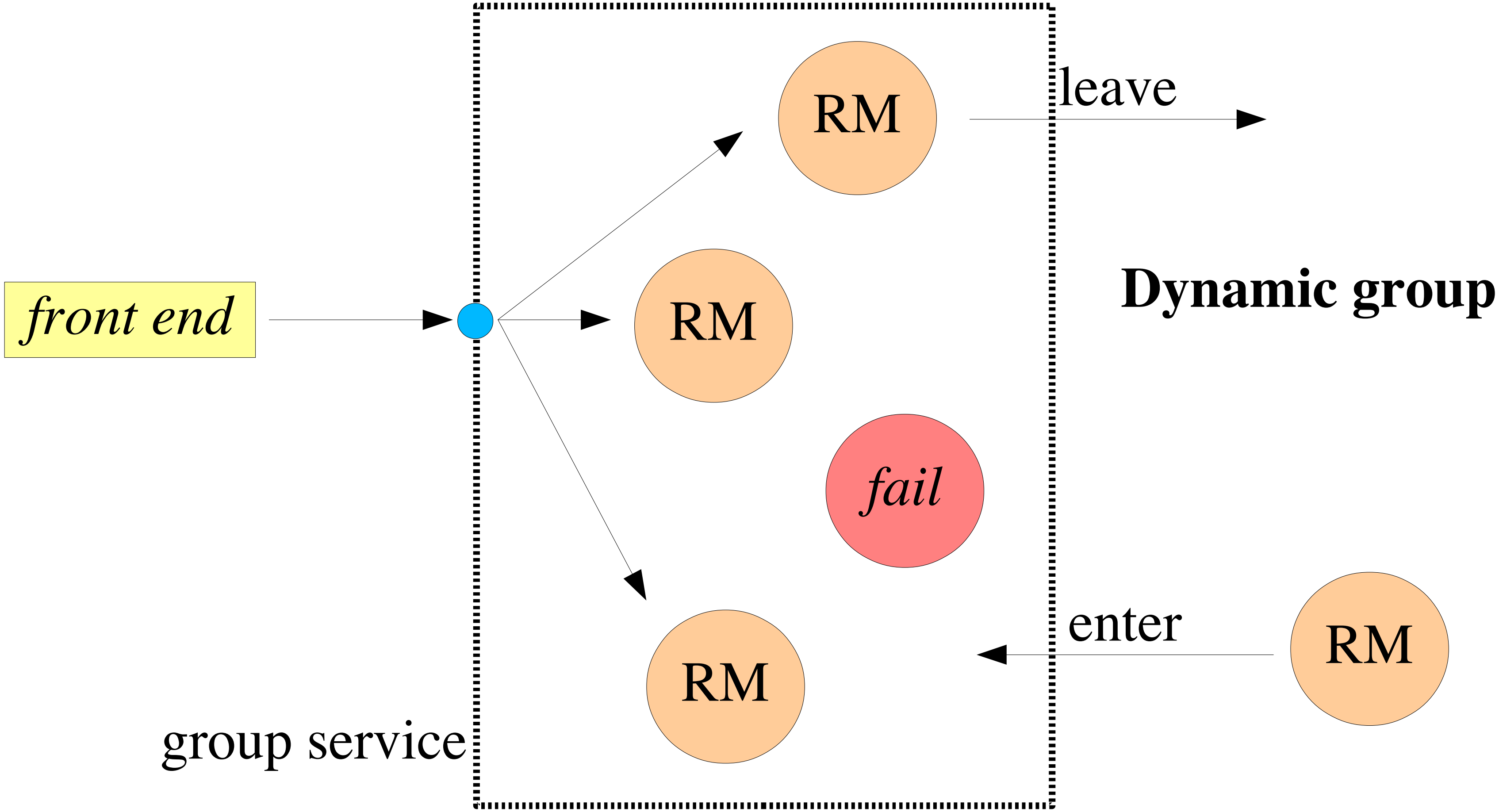
- Asynchronous system, nodes fail only by crashing, no network partitions.



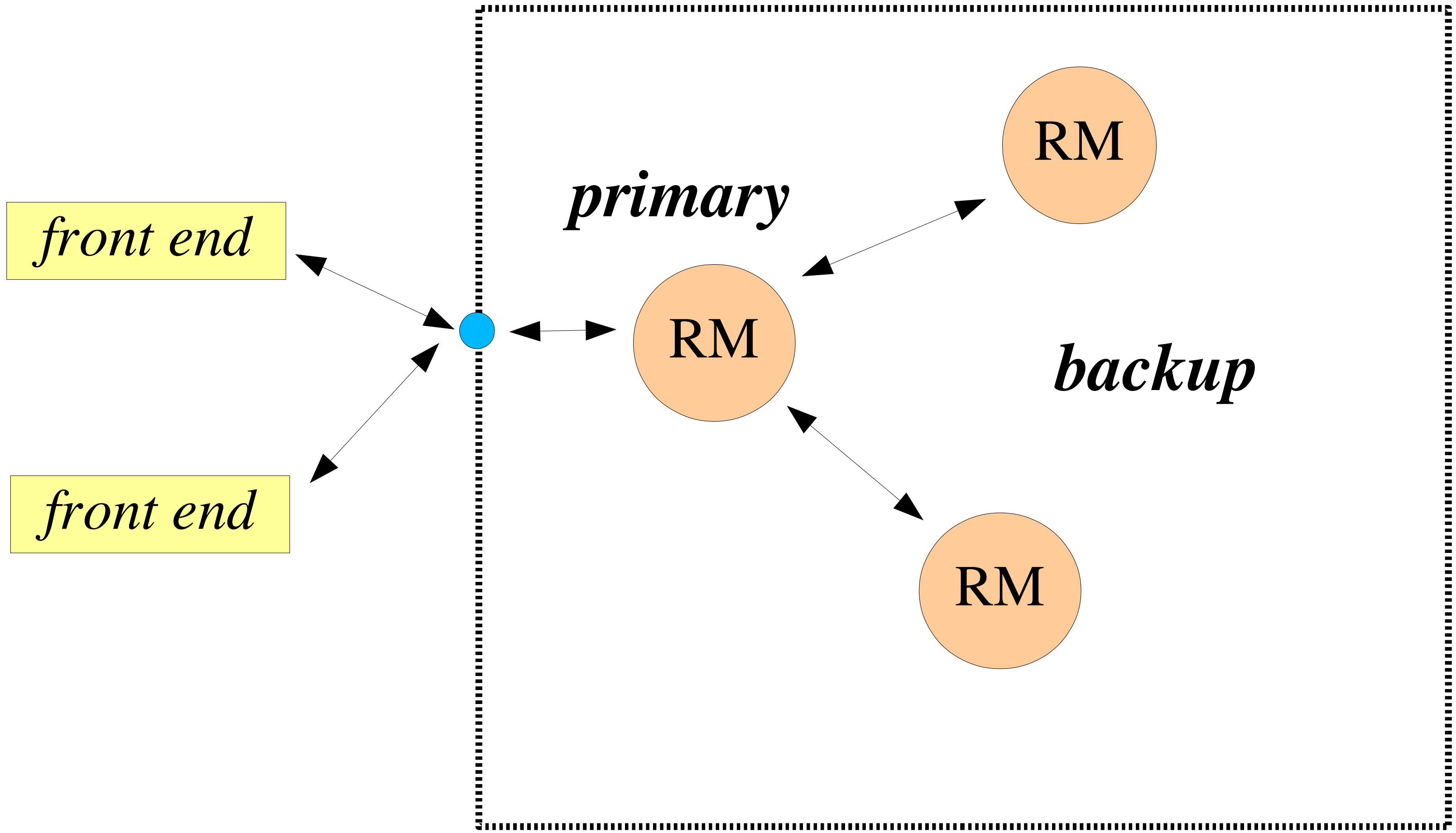
Life of a request

- Request
  - front-end delivers a request
- Coordination
  - replica managers decide on order
- Execution
  - tentative execution that can be aborted
- Agreement
  - reaching a consensus
- Response
  - front end collect one or more responses

Group membership service



passive replication



Passive replication

- Request
  - Front end: request with unique identifier → primary
- Coordination
  - Primary: takes each request atomically
    checks if new request, or resend response
- Execution
  - Primary: execute and store response
- Agreement
  - Primary: send (updated state, response, id) to
    backup nodes and backups reply ACK to primary
- Respond
  - Primary: response → front end → client

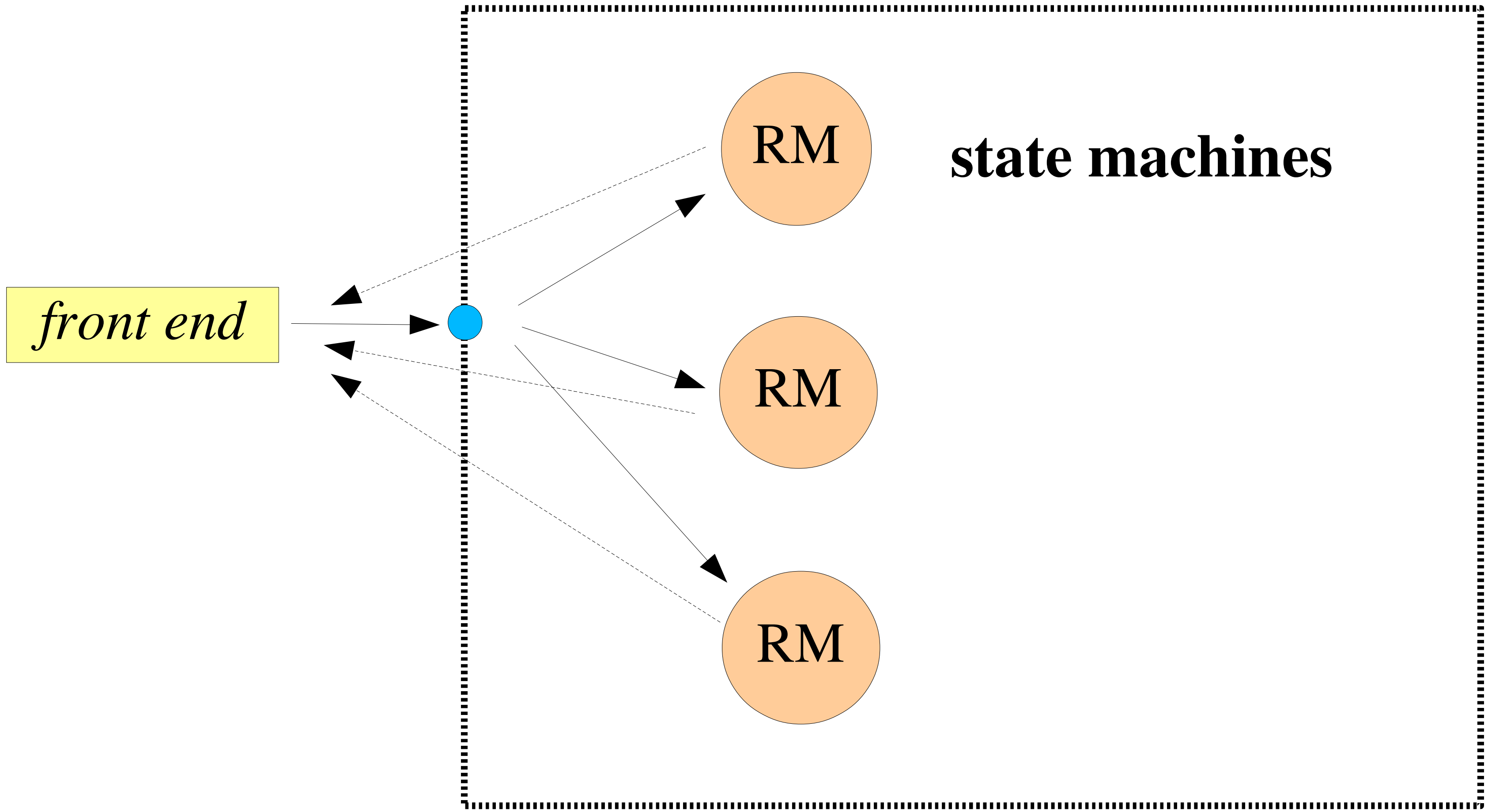
Is it linearizable?

- The primary replica manager will serialize all operations.
- If the primary fails, it retains linearizability if backup takes over where the primary left off.

primary crash

- Assume the primary crash:
  - Backups will receive new *view* with primary missing, new primary is selected.
- Request is resent
  - If agreement was reached last time the reply is stored, if not the execution is re-done.

active replication



Active replication

- Request
  - Front end: multicasted to group, unique identifier
    one request at a time
- Coordination
  - deliver request in *total order*
- Execution
  - all replicas are identical
- Agreement
  - not needed
- Response
  - sent all to front end, (e.g.) first reply to client

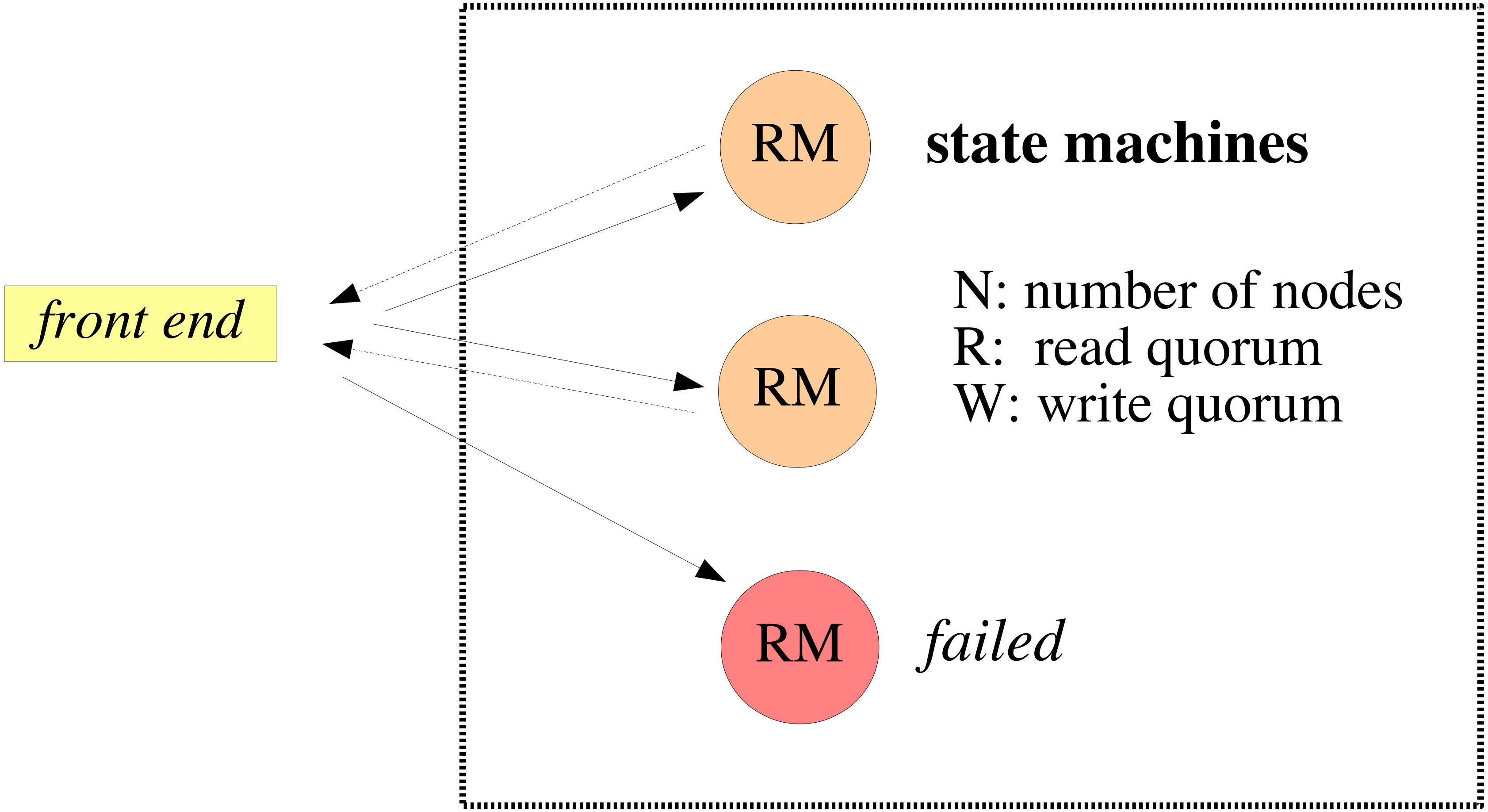
Active replication

- Sequential consistency:
  - All replicas execute the same sequence of operations.
  - All replicas produce the same answer.
- Linearizability:
  - Total order multicast does not preserve real-time order.

Quorum based

- Can we have a static group (nodes might fail but they will be restarted) and solve consistency using a quorum.
- Why would we like to do this?

Quorum based



What would happen if...

- $W < N$, $R = N$
  - can we handle this?
- $R < N$, $W = N$
  - special case: $R = 1$, $W = N$
- $W > N/2$, $R + W > N$
  - if W nodes fail?

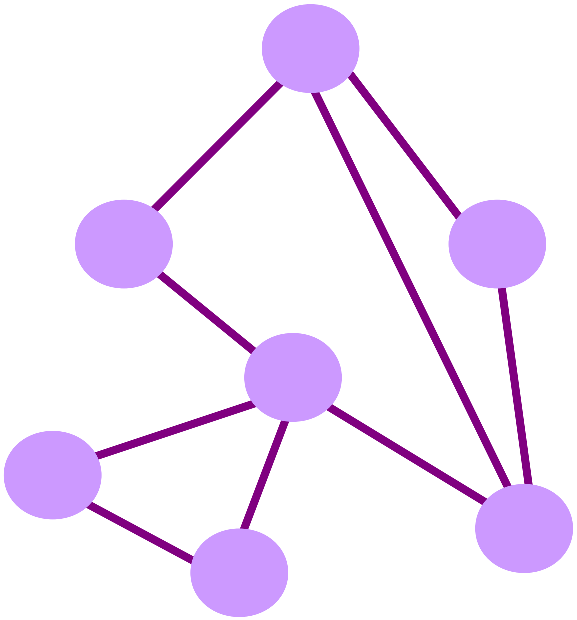
Summary

- Replicating objects used to achieve fault tolerant services.
- Services should (?) provide single image view as defined by sequential consistency.
- Passive replication
- Active replication
- Quorum based replication

How to Compute Paths?

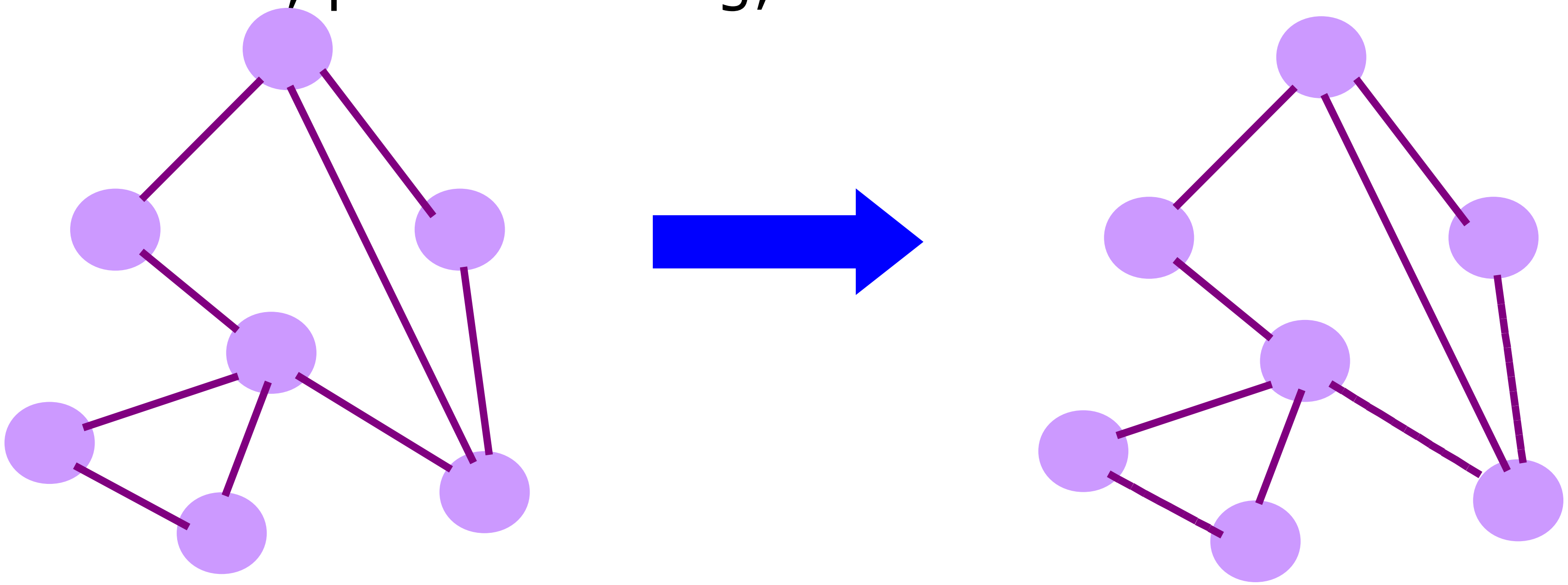
Different Ways to Represent Paths

- Static model
  - *What* is computed, i.e., what is the outcome
  - Not *how* the computation is performed
- Trade-offs
  - State to represent the paths
  - Efficiency of the paths
  - Ability to support multiple paths
  - Complexity of path computation



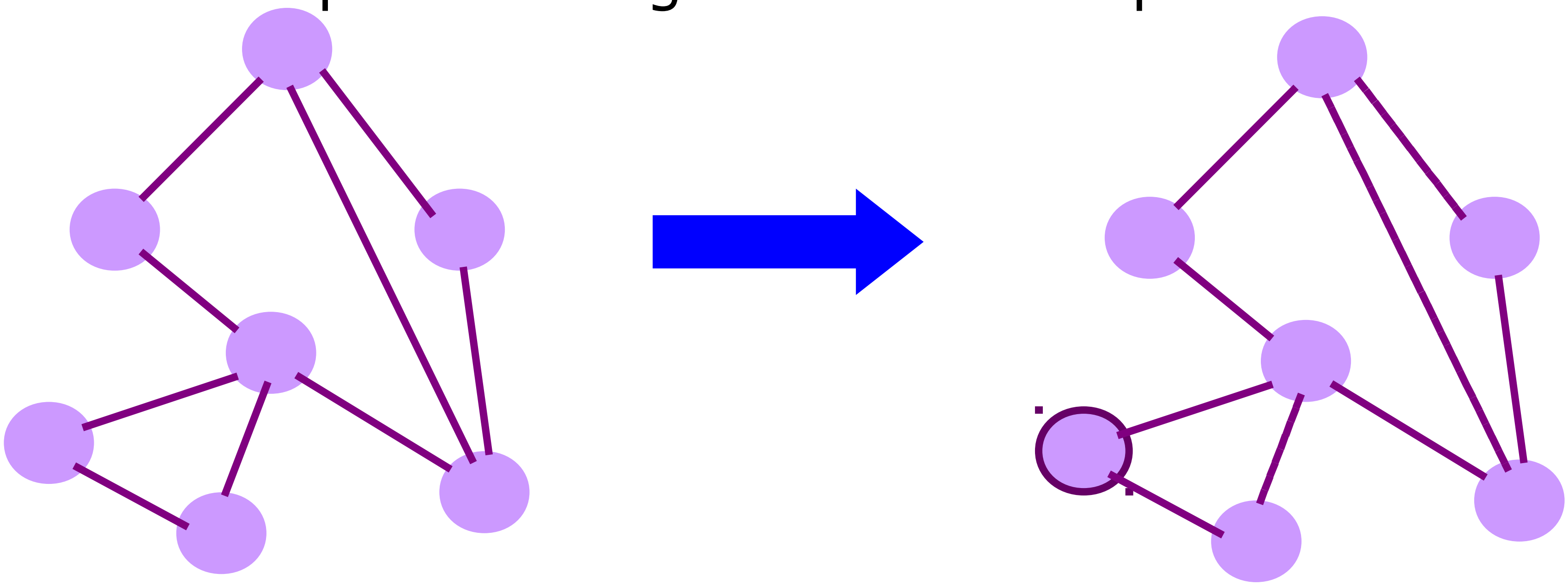
Spanning Tree

- One tree that reaches every node
  - Single path between each pair of nodes
  - No loops, so can support broadcast easily
  - But, paths are long, and some links not used



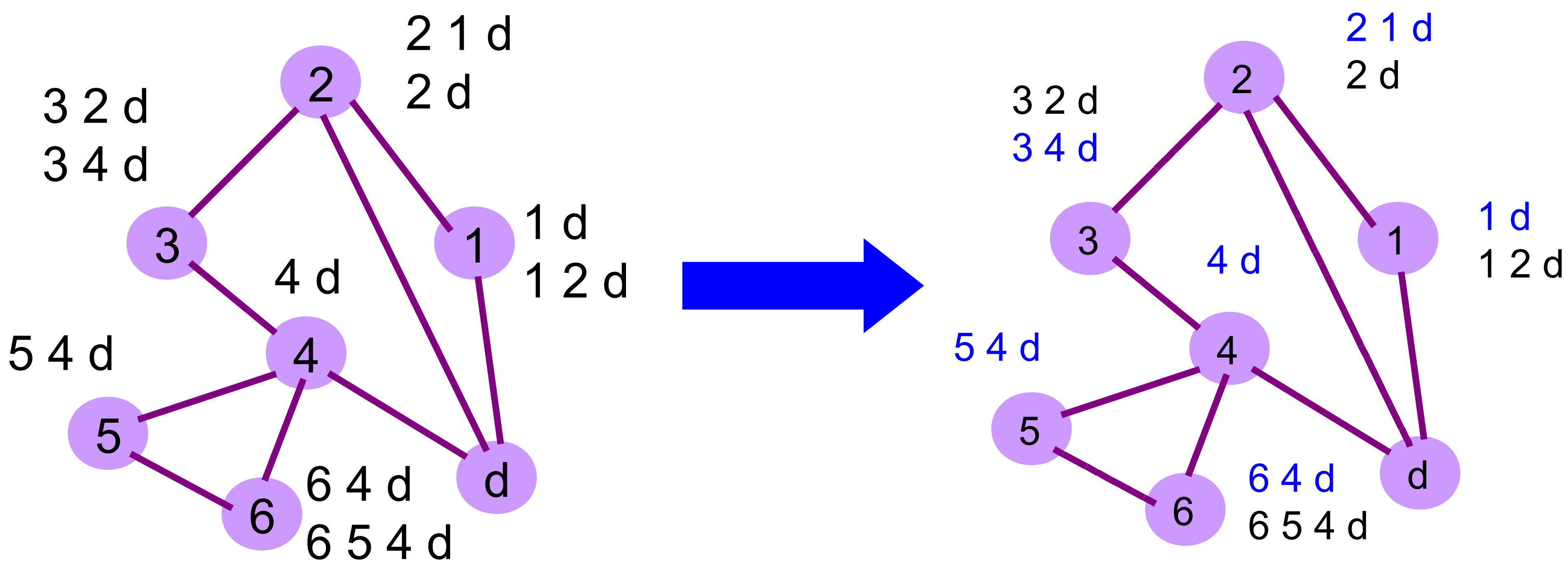
Shortest Paths

- Shortest path(s) between pairs of nodes
  - A shortest-path tree rooted at each node
  - Min hop count or min sum of edge weights
  - Multipath routing is limited to Equal Cost MultiPath



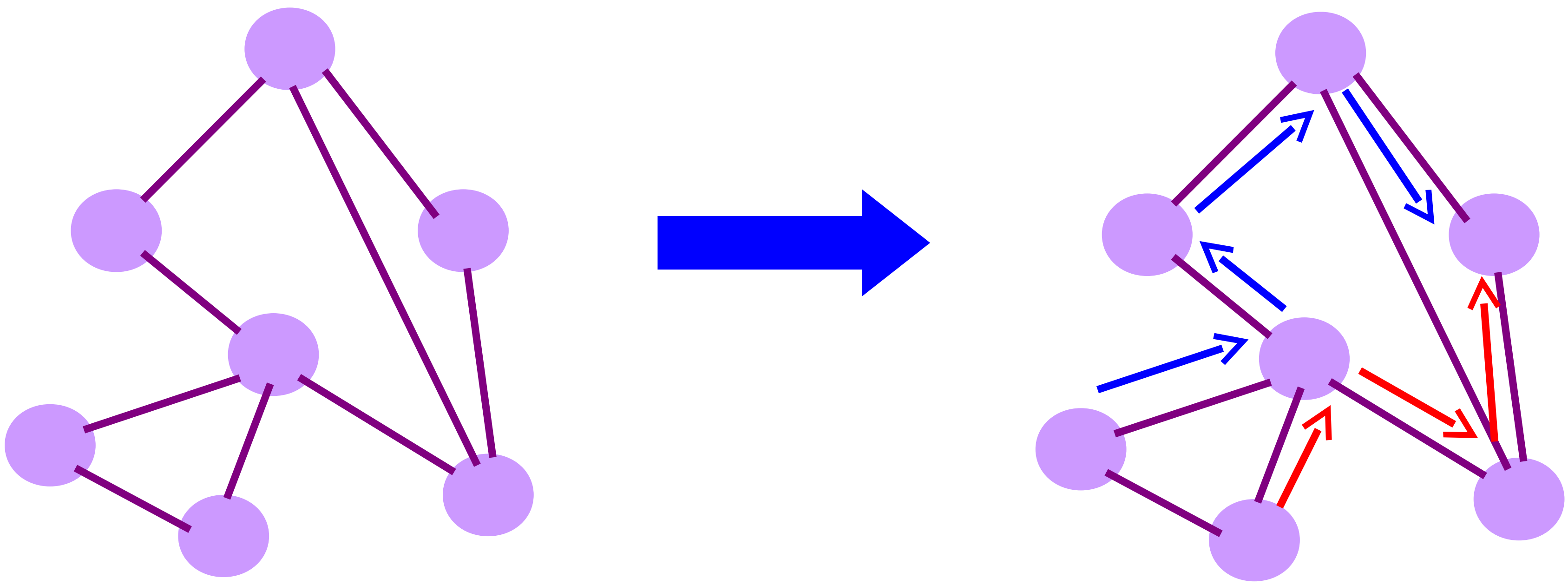
Locally Policy at Each Hop

- Locally best path
  - Local policy: each node picks the path it likes best
  - … among the paths chosen by its neighbors



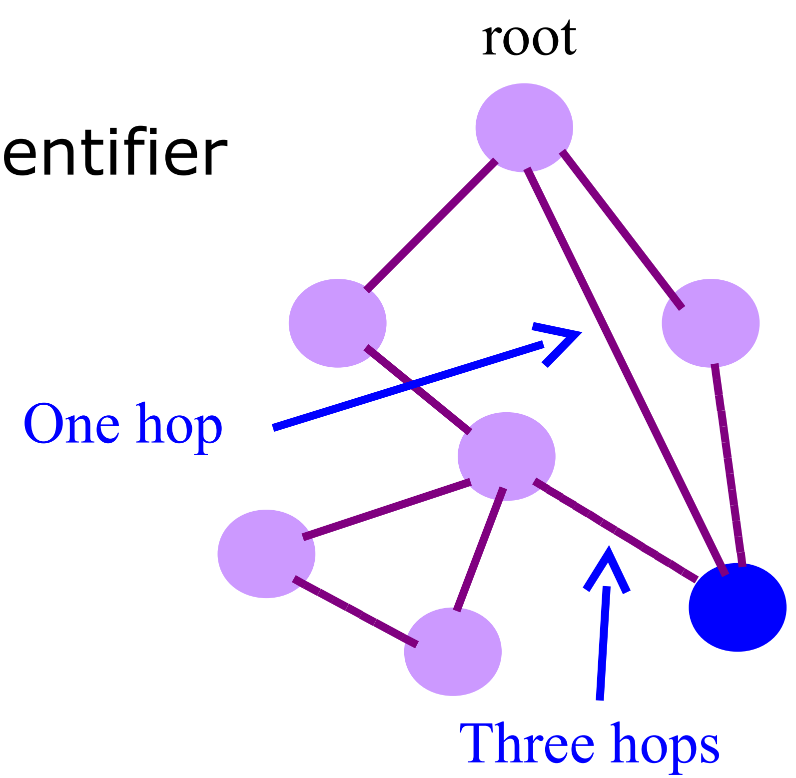
End-to-End Path Selection

- End-to-end path selection
  - Each node picks its own end to end paths
  - … independent of what other paths other nodes use
  - More state and complexity in the nodes



Spanning Tree Algorithm

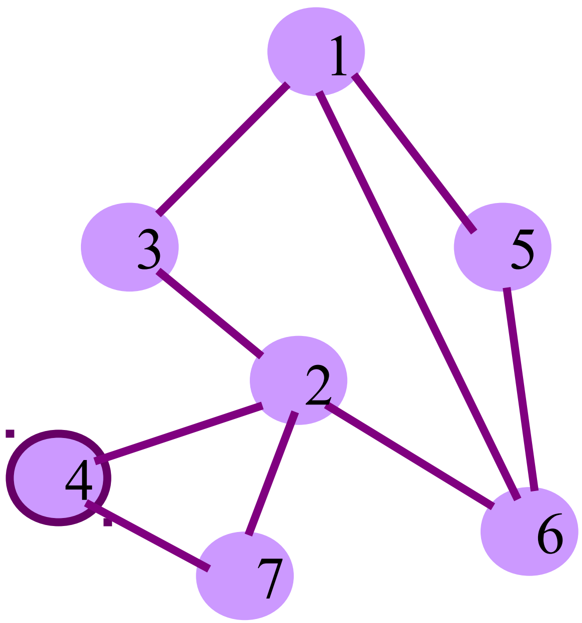
- Elect a root
  - The switch with the smallest identifier
  - And form a tree from there
- Algorithm
  - Repeatedly talk to neighbors
    - "I think node Y is the root"
    - "My distance from Y is d"
  - Update based on neighbors
    - Smaller id as the root
    - Smaller distance d+1



Used in Ethernet LANs

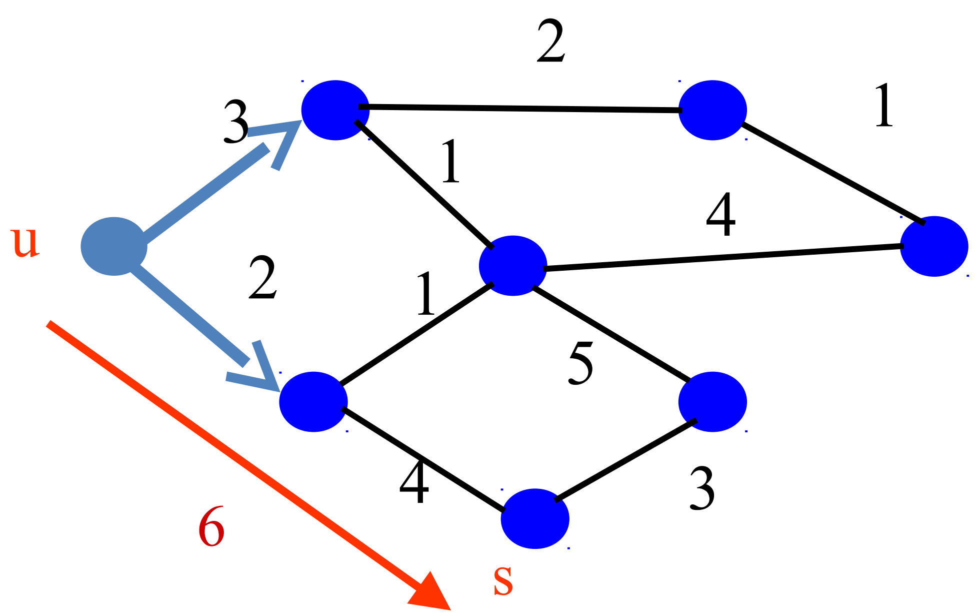
Spanning Tree Example: Switch #4

- Switch #4 thinks it is the root
  - Sends (4, 0, 4) message to 2 and 7
- Switch #4 hears from #2
  - Receives (2, 0, 2) message from 2
  - … and thinks that #2 is the root
  - And realizes it is just one hop away
- Switch #4 hears from #7
  - Receives (2, 1, 7) from 7
  - But, this is a longer path, so 4 prefers 4-2 over 4-7-2
  - And removes 4-7 link from the tree



Shortest-Path Problem

- Compute: *path costs* to all nodes
  - From a given source u to all other nodes
  - Cost of the path through each outgoing link
  - Next hop along the least-cost path to s



Link State: Dijkstra's Algorithm

- Flood the topology information to all nodes
- Each node computes shortest paths to other nodes

<u>Initialization</u>

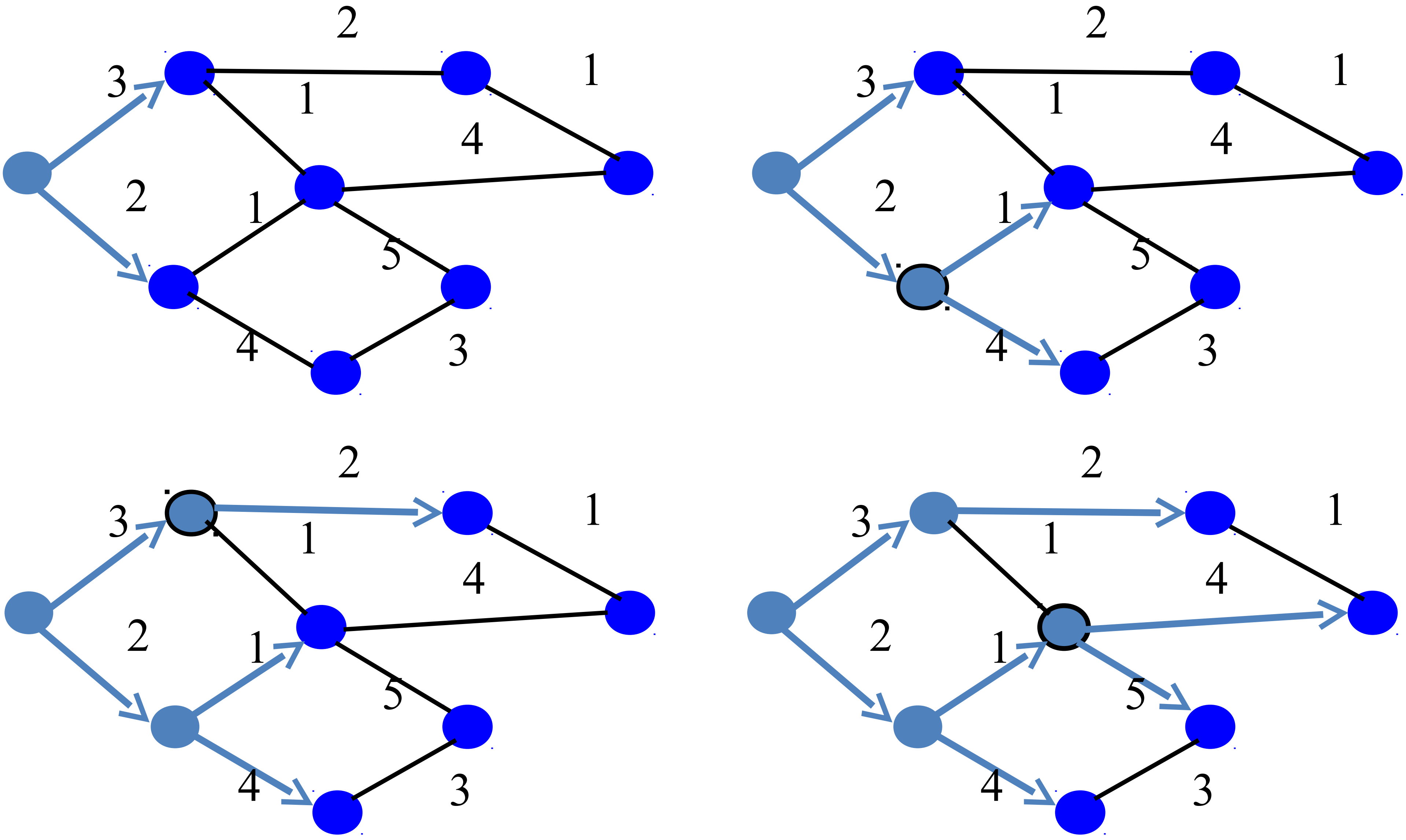
```
S = {u}
for all nodes v
  if (v is adjacent to u)
    D(v) = c(u,v)
  else D(v) = ∞
```

<u>Loop</u>

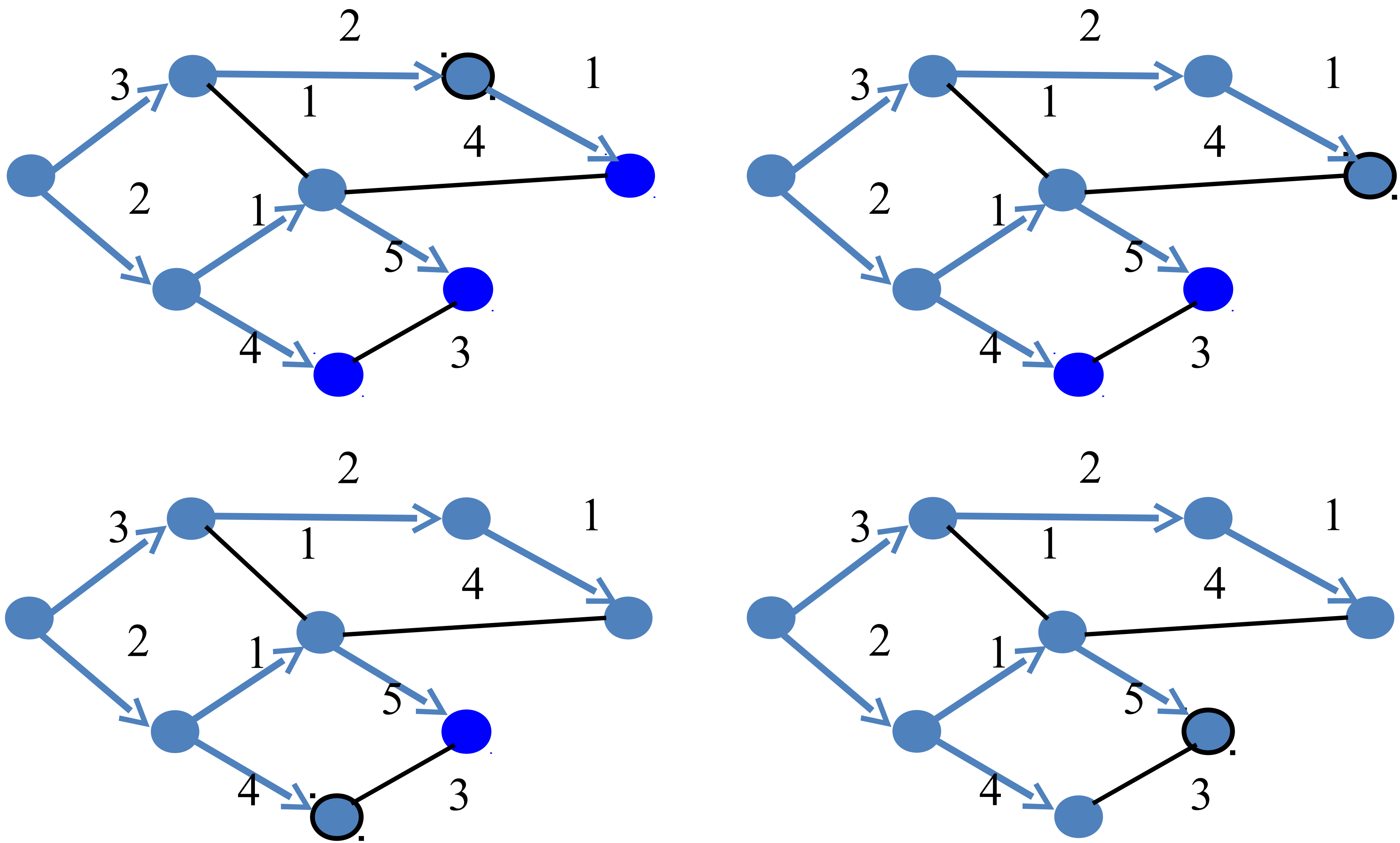
```
add w with smallest D(w) to S
update D(v) for all adjacent v:
  D(v) = min{D(v), D(w) + c(w,v)}
until all nodes are in S
```

Used in OSPF and IS-IS

Link-State Routing Example

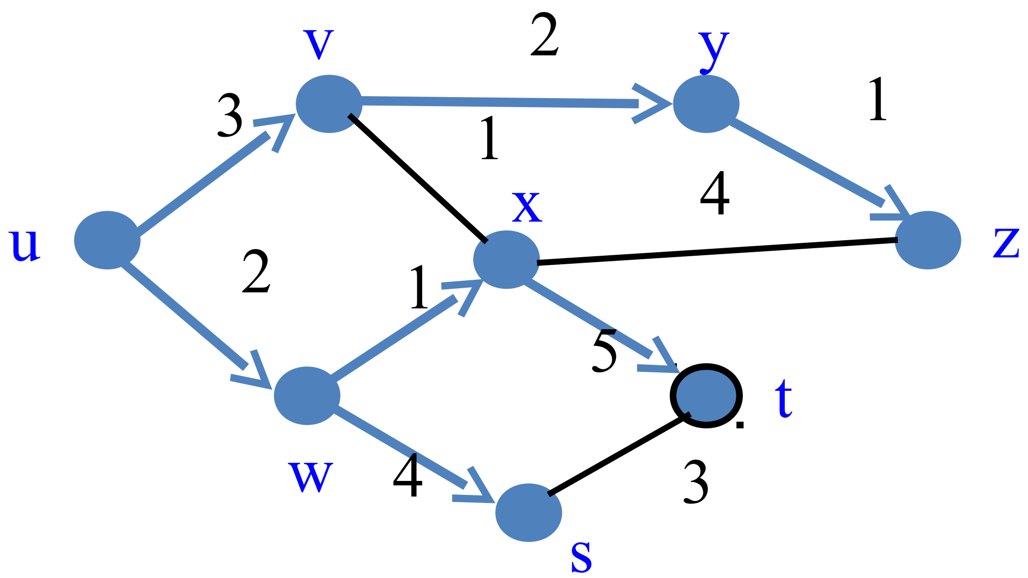


Link-State Routing Example (cont.)



Link State: Shortest-Path Tree

Shortest-path tree from u

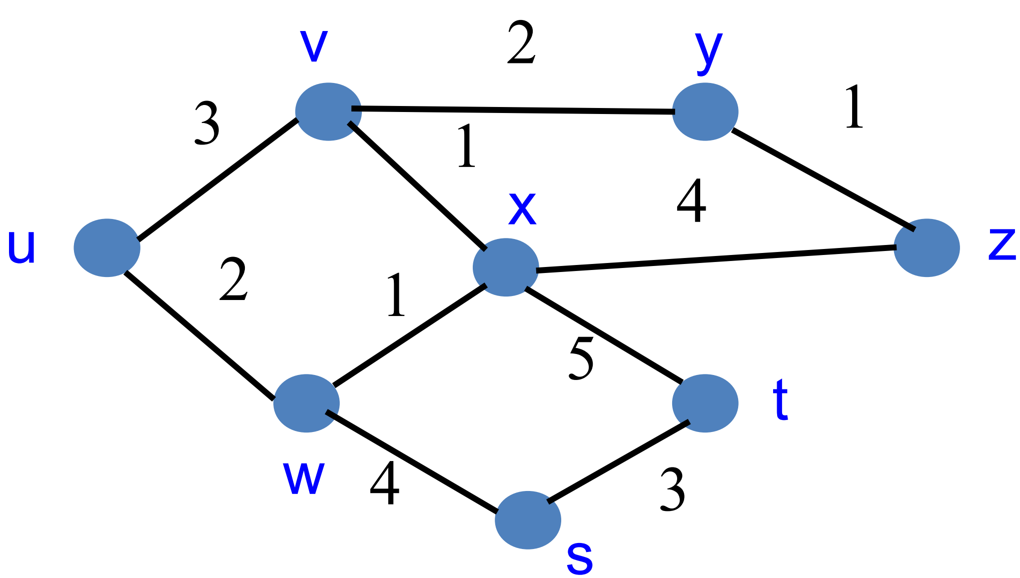


Forwarding table at u

	link
v	(u,v)
w	(u,w)
x	(u,w)
y	(u,v)
z	(u,v)
s	(u,w)
t	(u,w)

Distance Vector: Bellman-Ford Alg

- Define distances at each node x
  - $d_x(y)$ = cost of least-cost path from x to y
- Update distances based on neighbors
  - $d_x(y) = \min \{c(x,v) + d_v(y)\}$ over all neighbors v

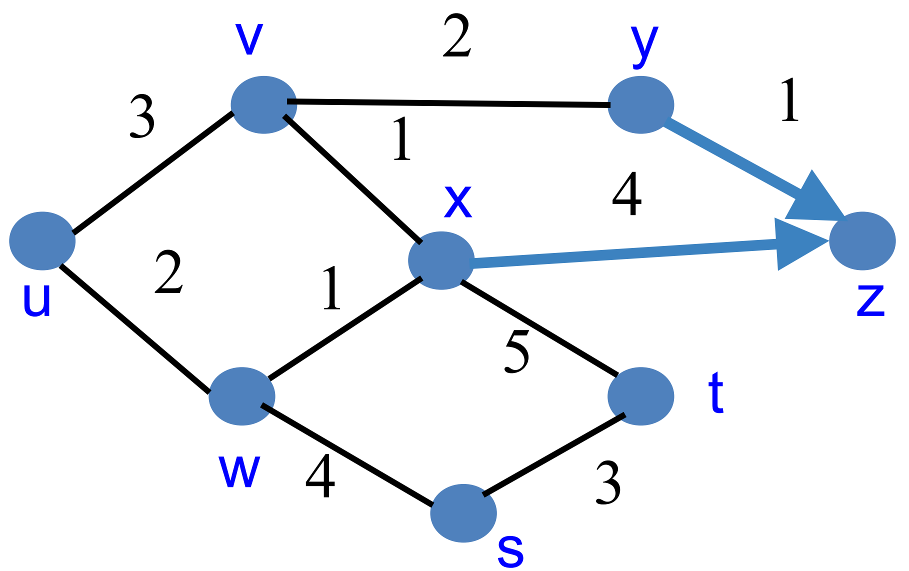


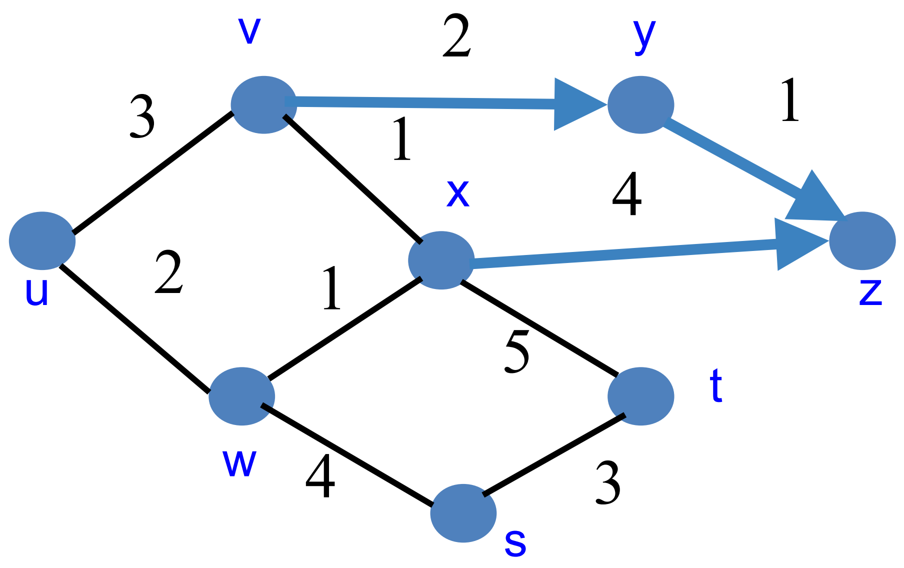
$$d_u(z) = \min\{c(u,v) + d_v(z),\ c(u,w) + d_w(z)\}$$

Used in RIP and EIGRP

Distance Vector Example

To z:



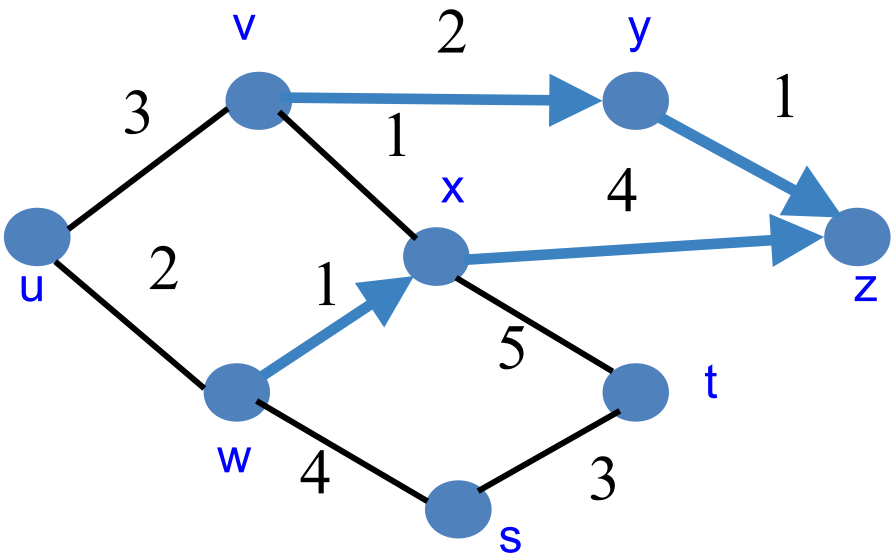


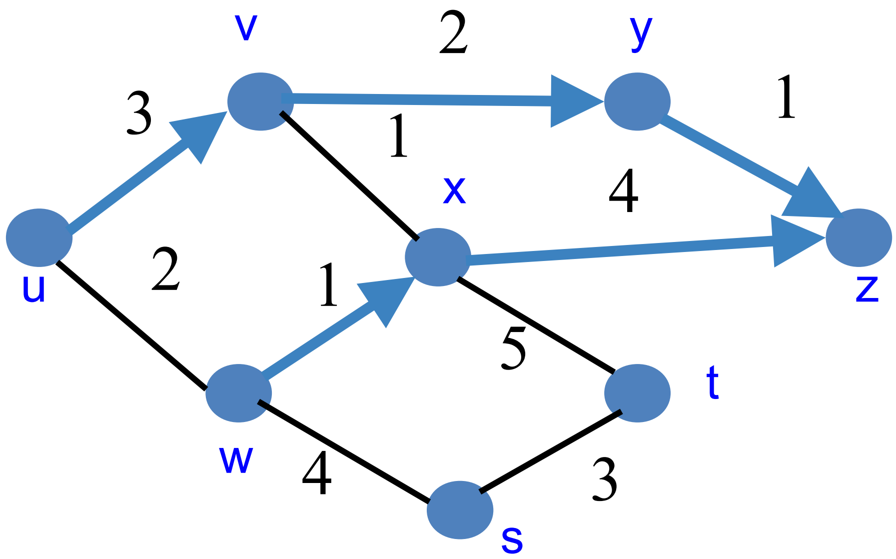
$d_y(z)=1$

$d_x(z)=4$

$d_v(z)= \min\{2+d_y(z),$
$1+d_x(z)\}$
$= 3$

Distance Vector Example (Cont.)





$d_w(z) = \min\{1 + d_x(z),$
$4 + d_s(z),$
$2 + d_u(z)\}$
$= 5$

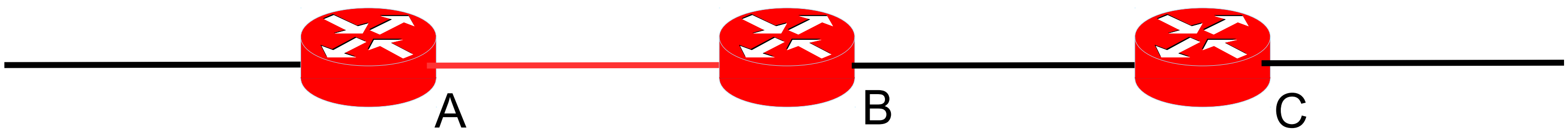
$d_u(z) = \min\{3 + d_v(z),$
$2 + d_w(z)\}$
$= 6$

Comparison

- Link state
  - All routers know (all paths, all routers) of the net
  - Link state info flooded → all routers have a consistent copy of the link-state database
  - Each router constructs its own relative shortest-path tree, itself as root, for all routes
- Distance vector
  - Involves: distance (metric) to destination + vector (direction) to get there
  - Routing info only exchanged among direct neighb
  - Short-sighted: ignores where neighb learned route

Routing Information Protocol (RIP)

- Distance Vector protocol
  - Number of hops, $\infty = 16$
  - UDP, port 520
  - Routers send RIP updates every 30 secs
  - Neighbor router is down: 180 secs
- RIP versions:
  - RIPv1 (1988) no subnet info, no auth, bcast table
  - RIPv2 (1993-1998), mcast table to 224.0.0.9
  - RIPng (1997), IPv6



Convergence: Count to infinity

- A-B fails, B knows, C still says A is 2 hops away from C (through B), B believes B-C-?-A
- Split horizon:
  - prohibiting a router from advertising a route back onto the interface from which it was learned
  - Poisoned reverse: adding entries with ∞ cost
  - Send updates when change (before 30 sec timer)
  - Hold down timer (CISCO), 280 secs w/o updates

Open Shortest Path First (OSPF)

- Link State protocol

  Routers monitor neighbour routers and nets

  When any change, Link State Advertisement sent to all routers (IP, mcast 224.0.0.5)

  Routers maintain LS database with LSAs
- Metric can include link bitrate, delay, etc
- No convergence (count to infinity) problems
- Interior Gateway Protocol