Distributed Systems

CS425/ECE428

02/19/2020

Today's agenda

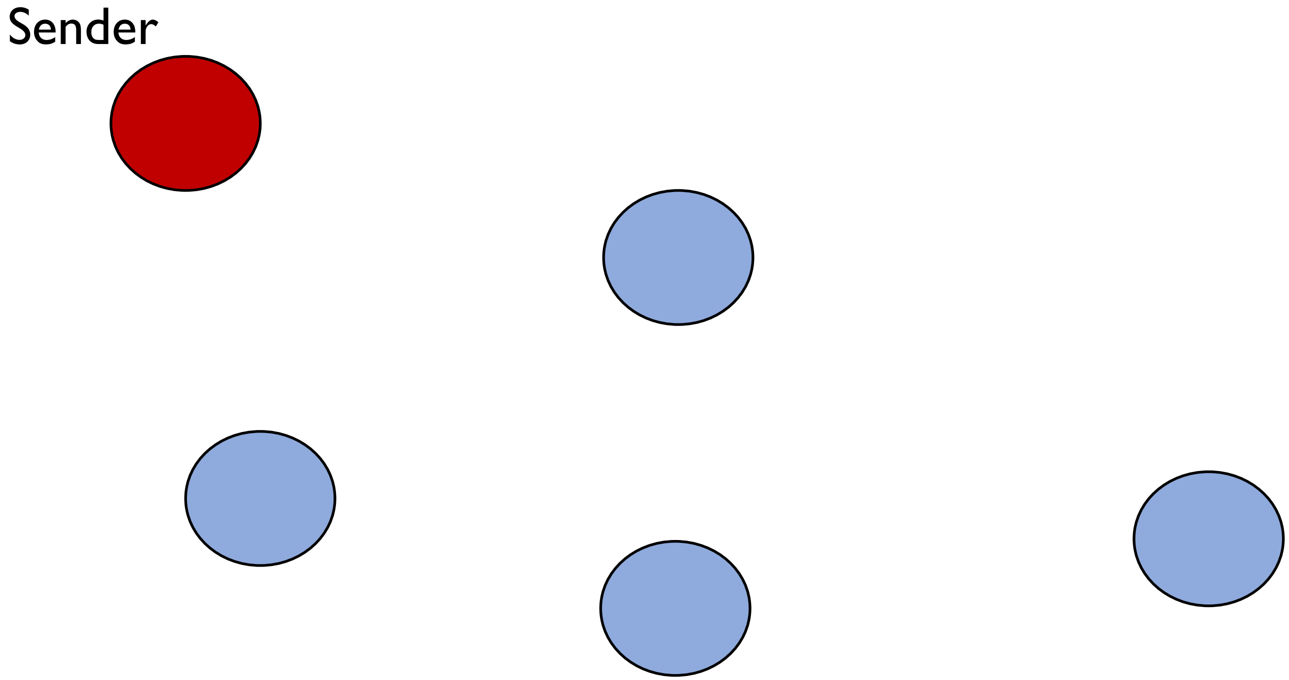
- **Wrap-up Multicast**
  - Tree-based multicast and gossip
- **Mutual Exclusion**
  - Chapter 15.2
- **Acknowledgement:**
  - Materials largely derived from Prof. Indy Gupta.

Recap: Multicast

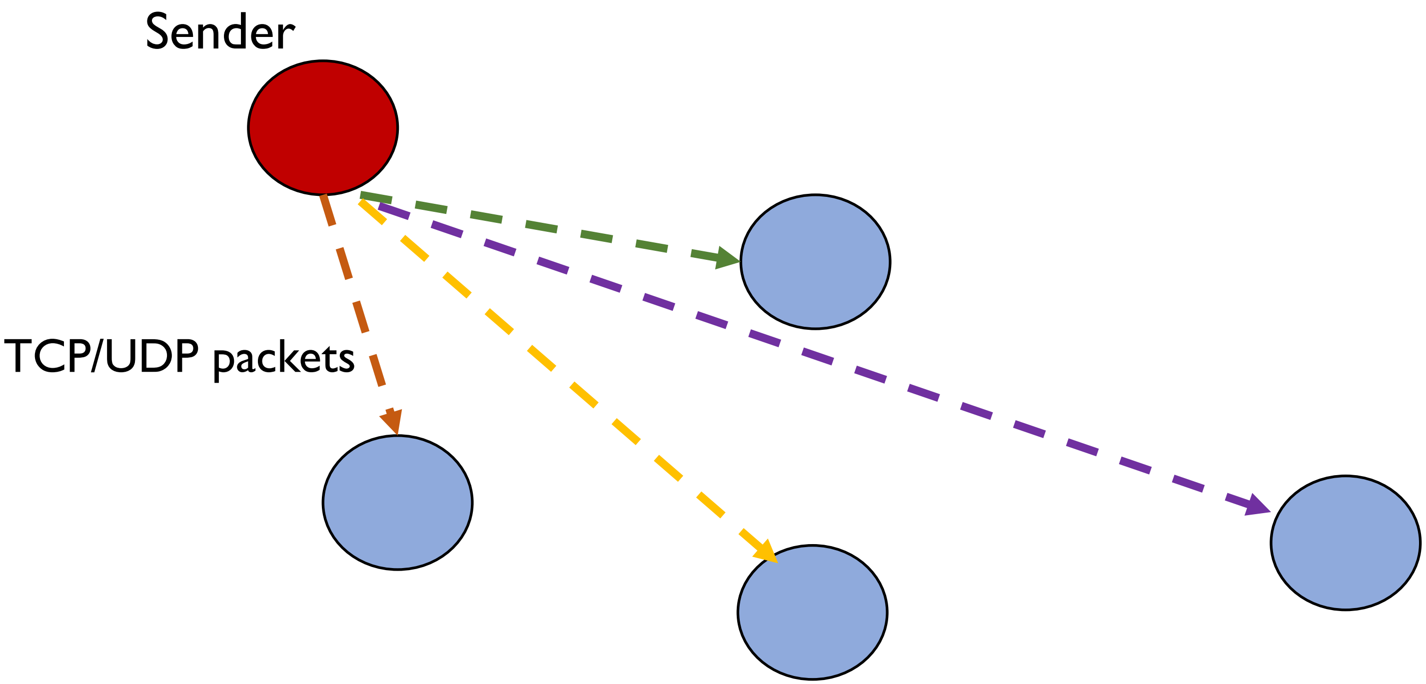
- Multicast is an important communication mode in distributed systems.

- Applications may have different requirements:
  - Basic
  - Reliable
  - Ordered: FIFO, Causal, Total
  - Combinations of the above.

B-Multicast

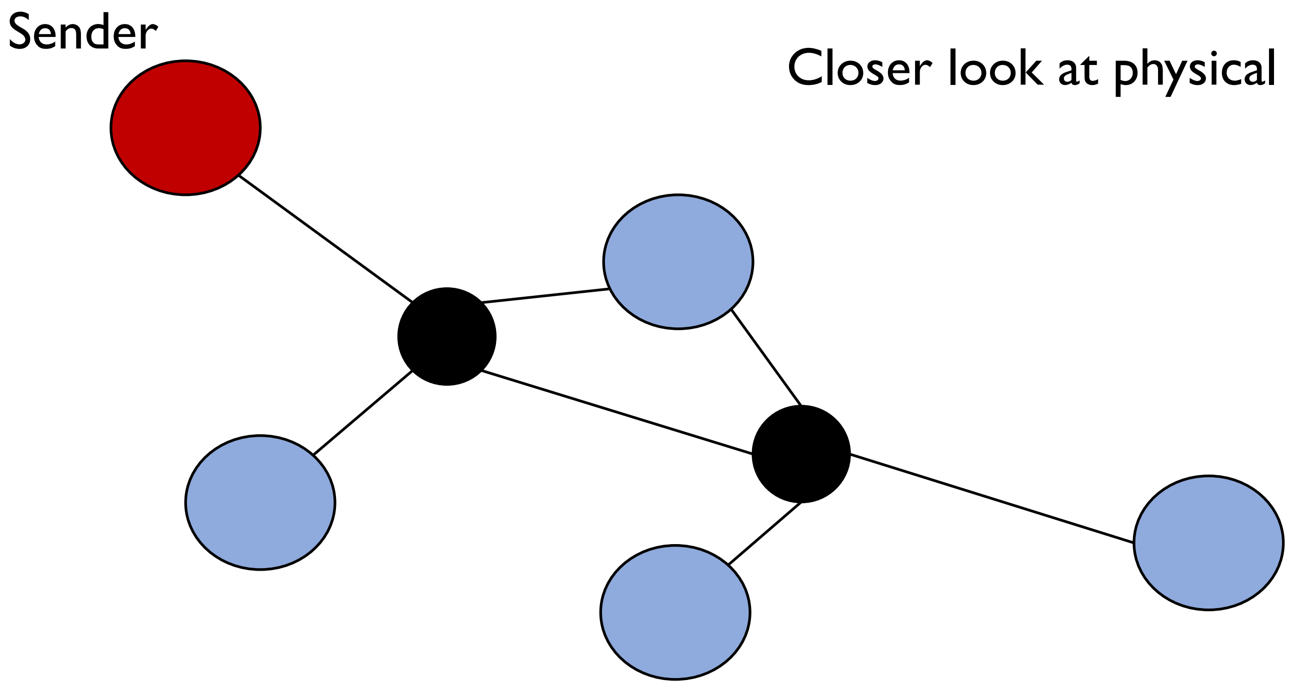


B-Multicast using unicast sends

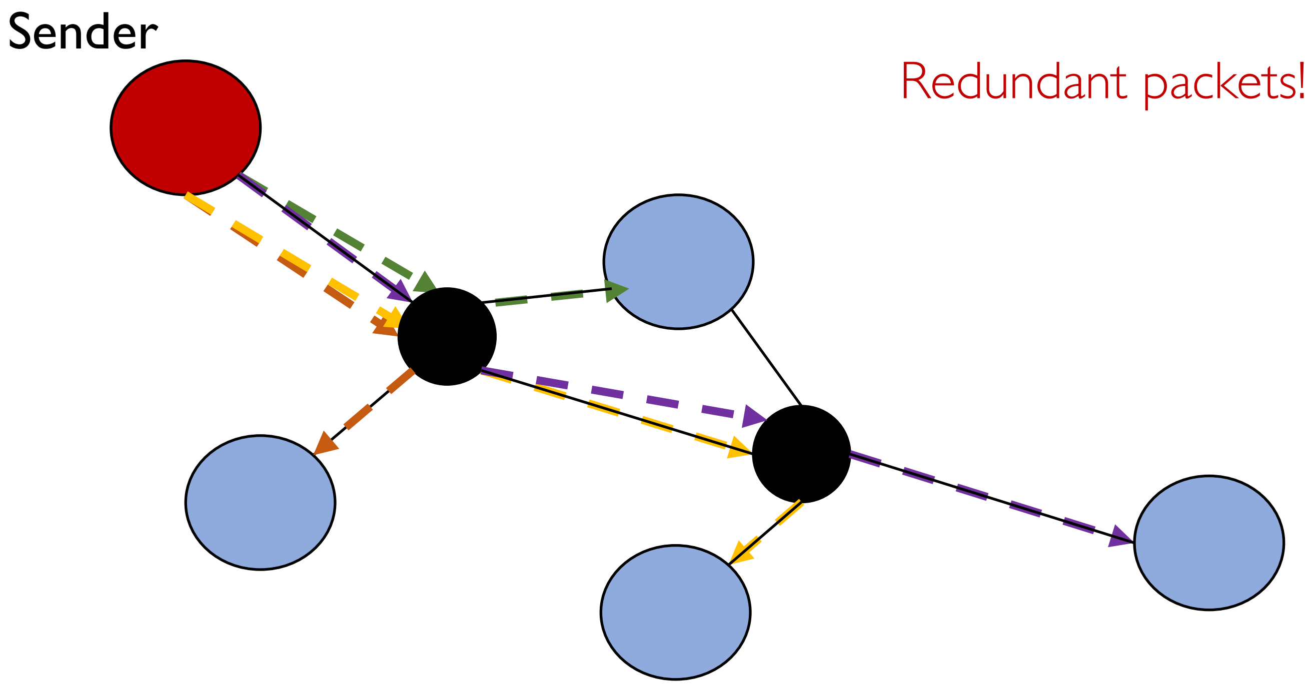


B-Multicast using unicast sends

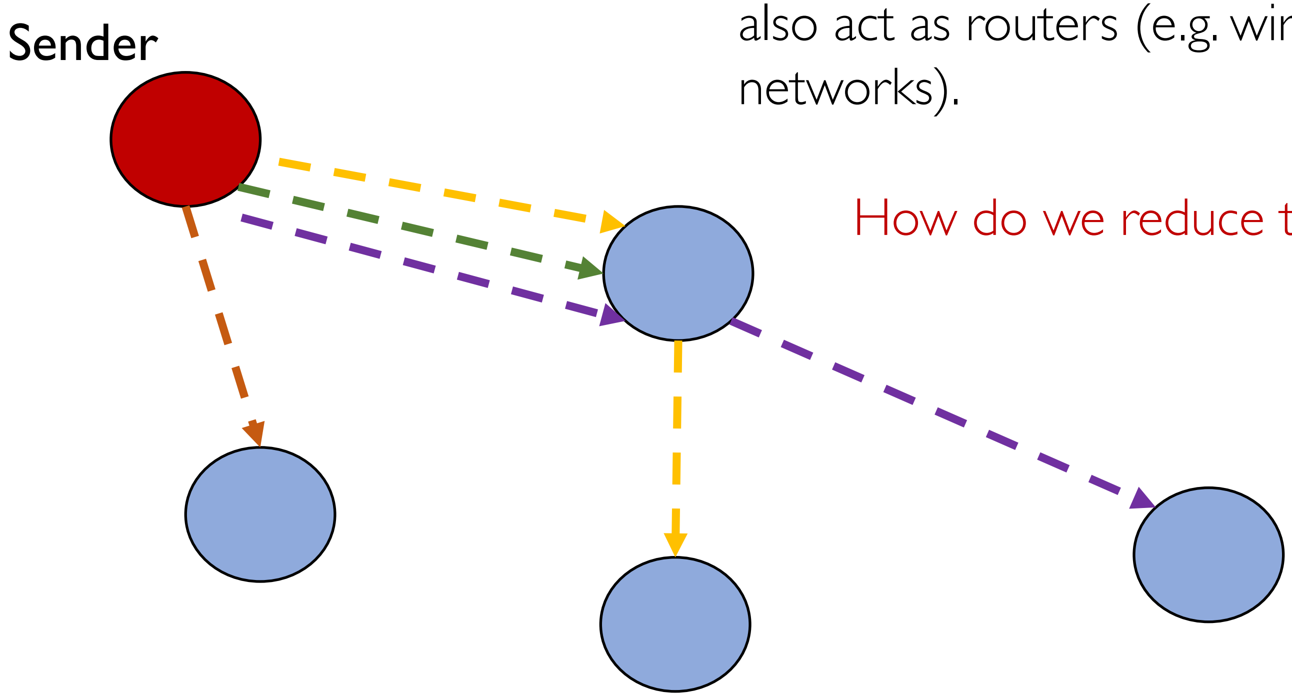
Closer look at physical network paths.



B-Multicast using unicast sends



B-Multicast using unicast sends

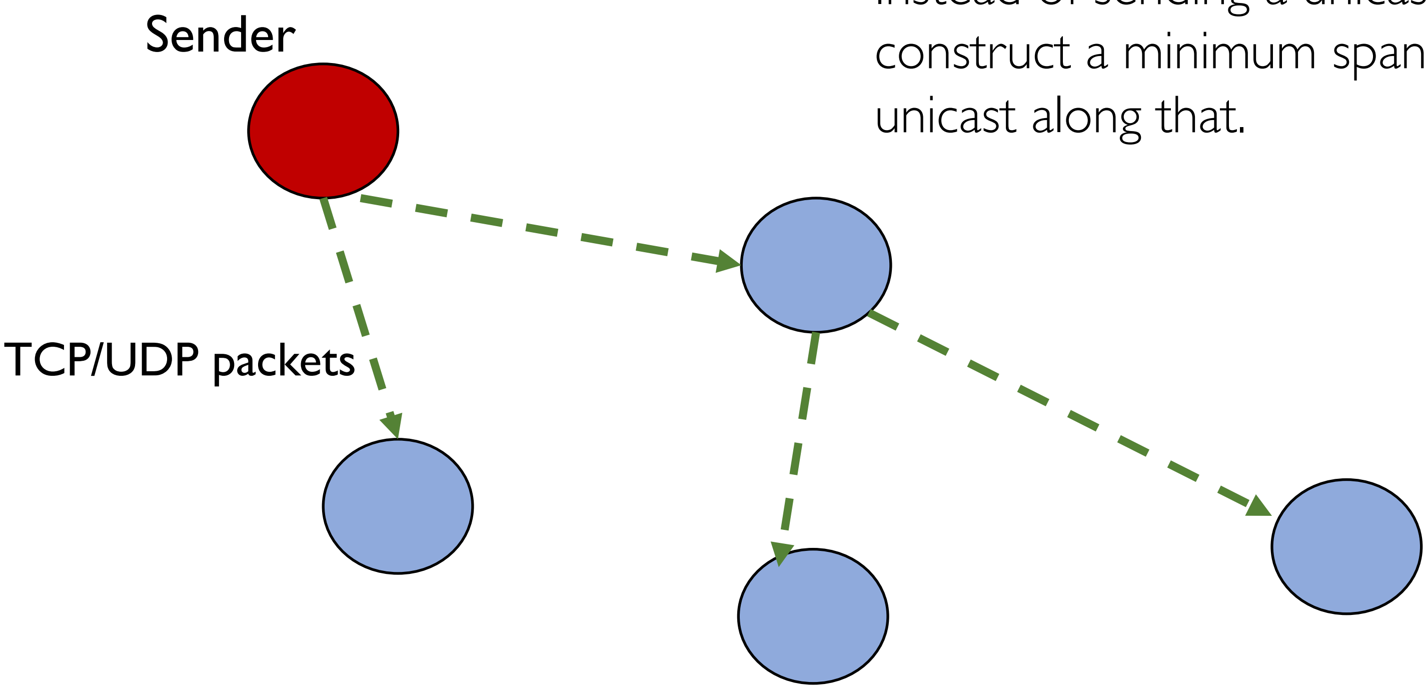


Similar redundancy when individual nodes also act as routers (e.g. wireless sensor networks).

How do we reduce the overhead?

Tree-based multicast

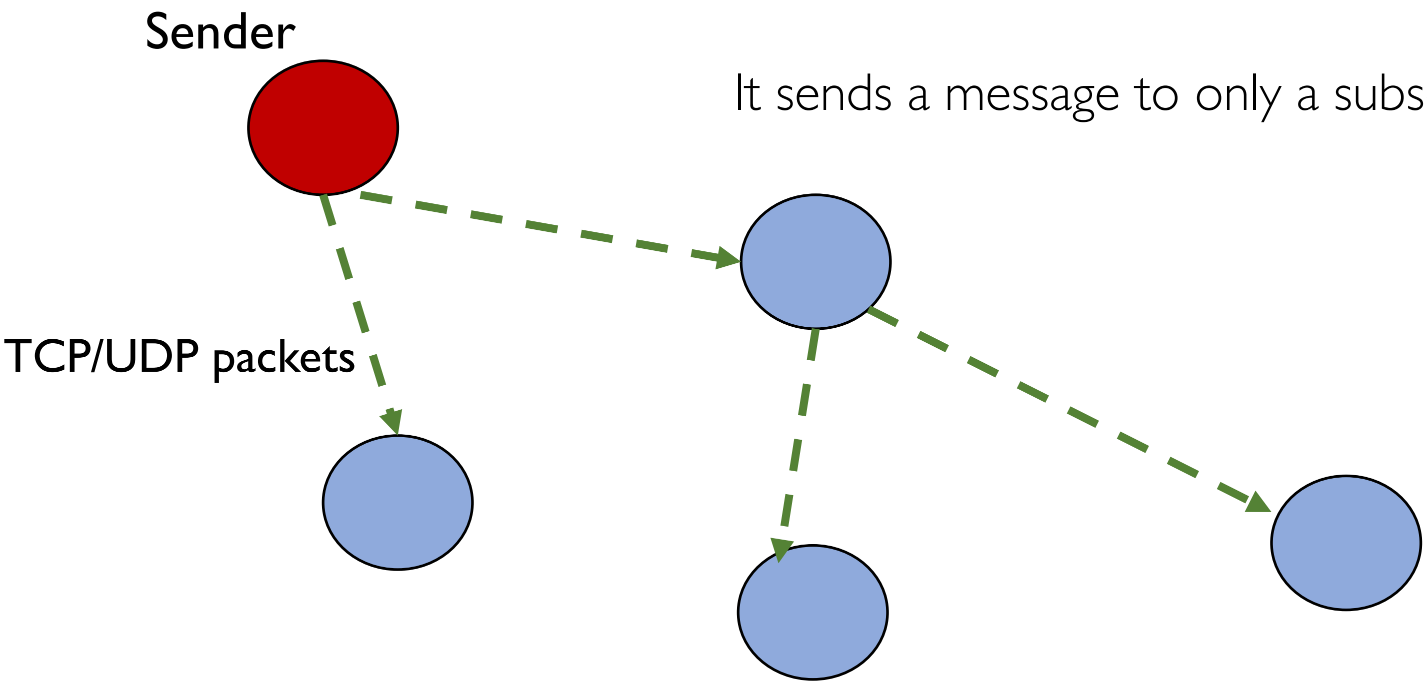
Instead of sending a unicast to all nodes, construct a minimum spanning tree and unicast along that.



Tree-based multicast

A process does not directly send messages to *all* other processes in the group.

It sends a message to only a subset of processes.

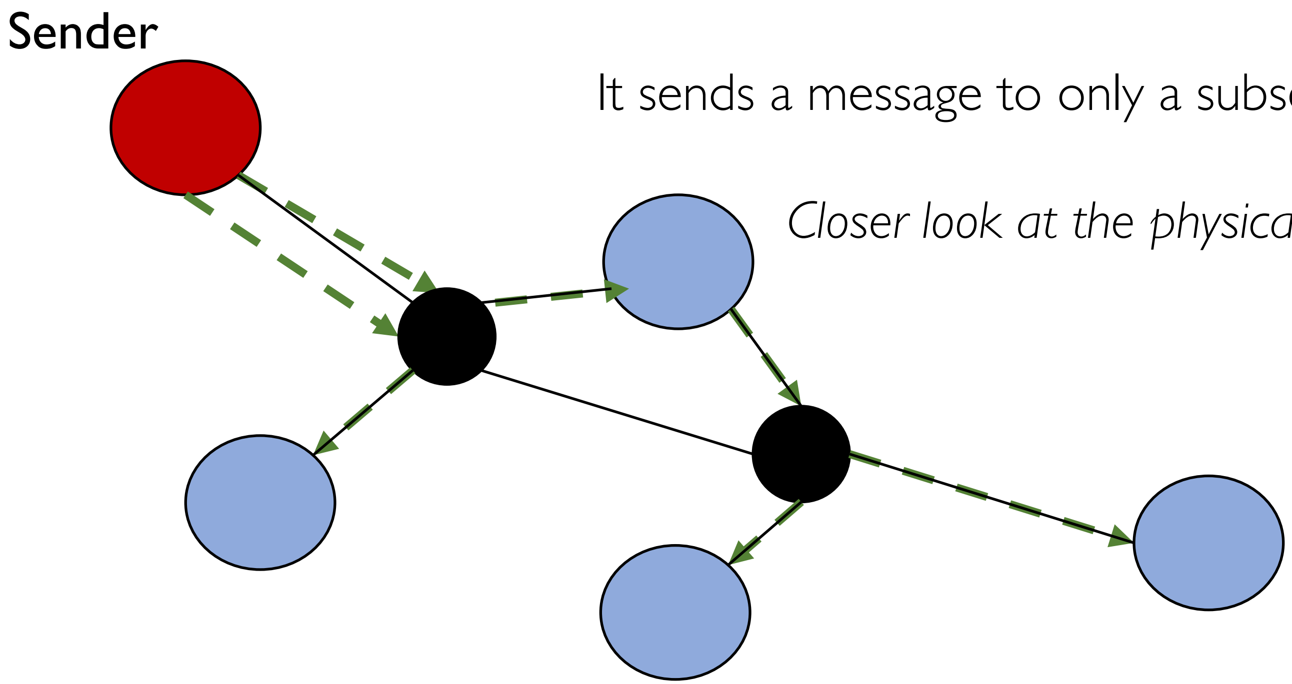


Tree-based multicast

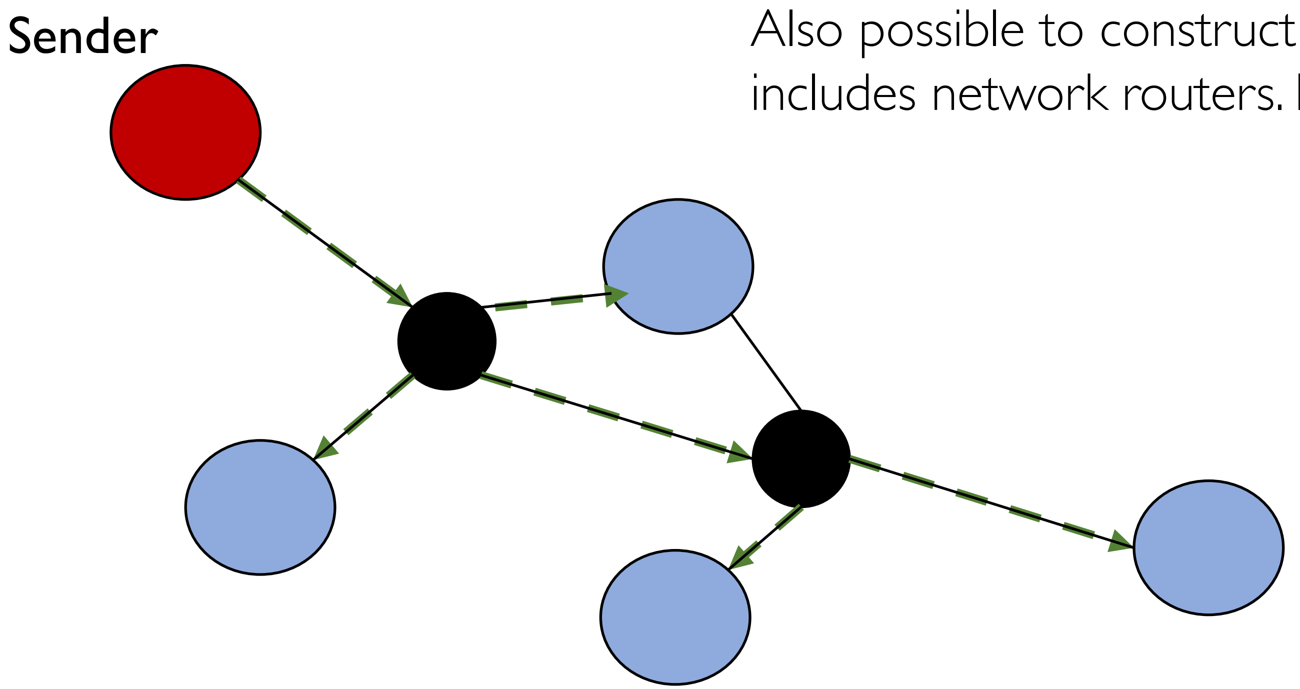
A process does not directly send messages to *all* other processes in the group.

It sends a message to only a subset of processes.

Closer look at the physical network.

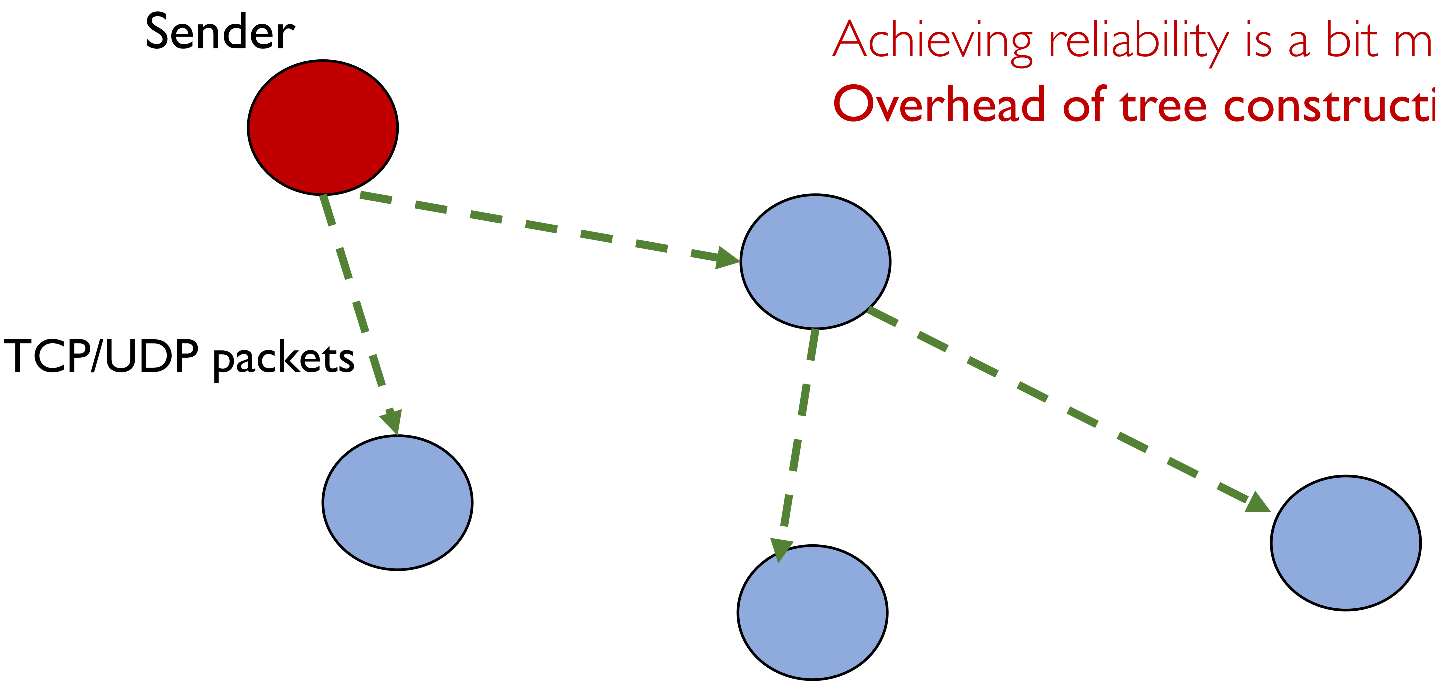


Tree-based multicast



Also possible to construct a tree that includes network routers. **IP multicast!**

Tree-based multicast

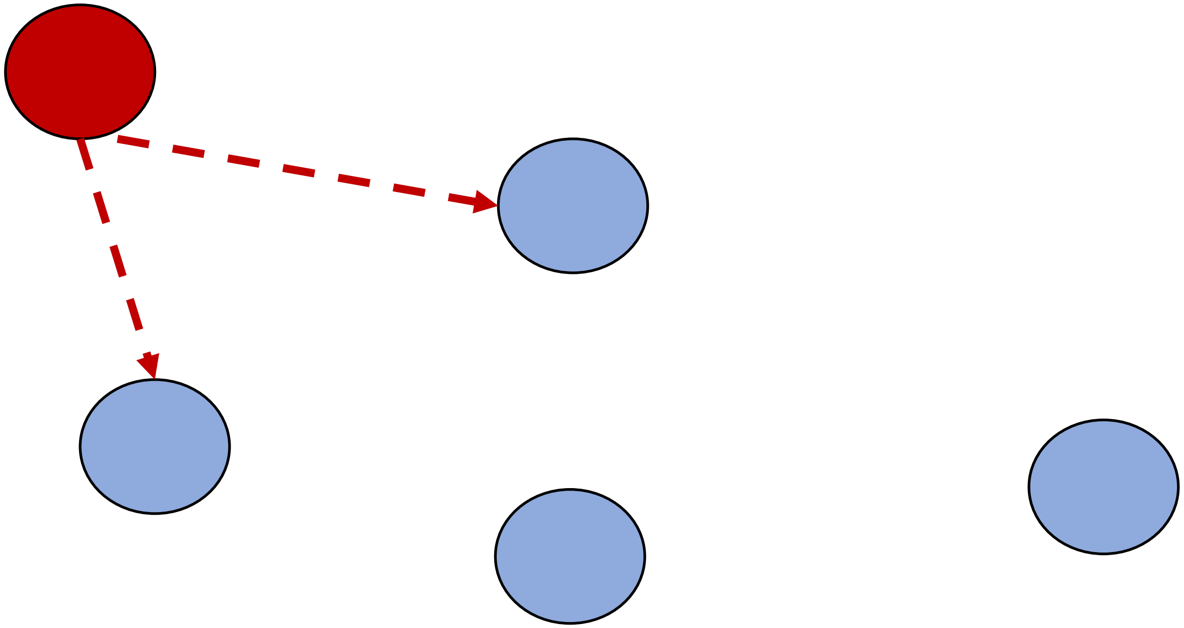


Achieving reliability is a bit more tricky.

Overhead of tree construction and repair.

Third approach: Gossip

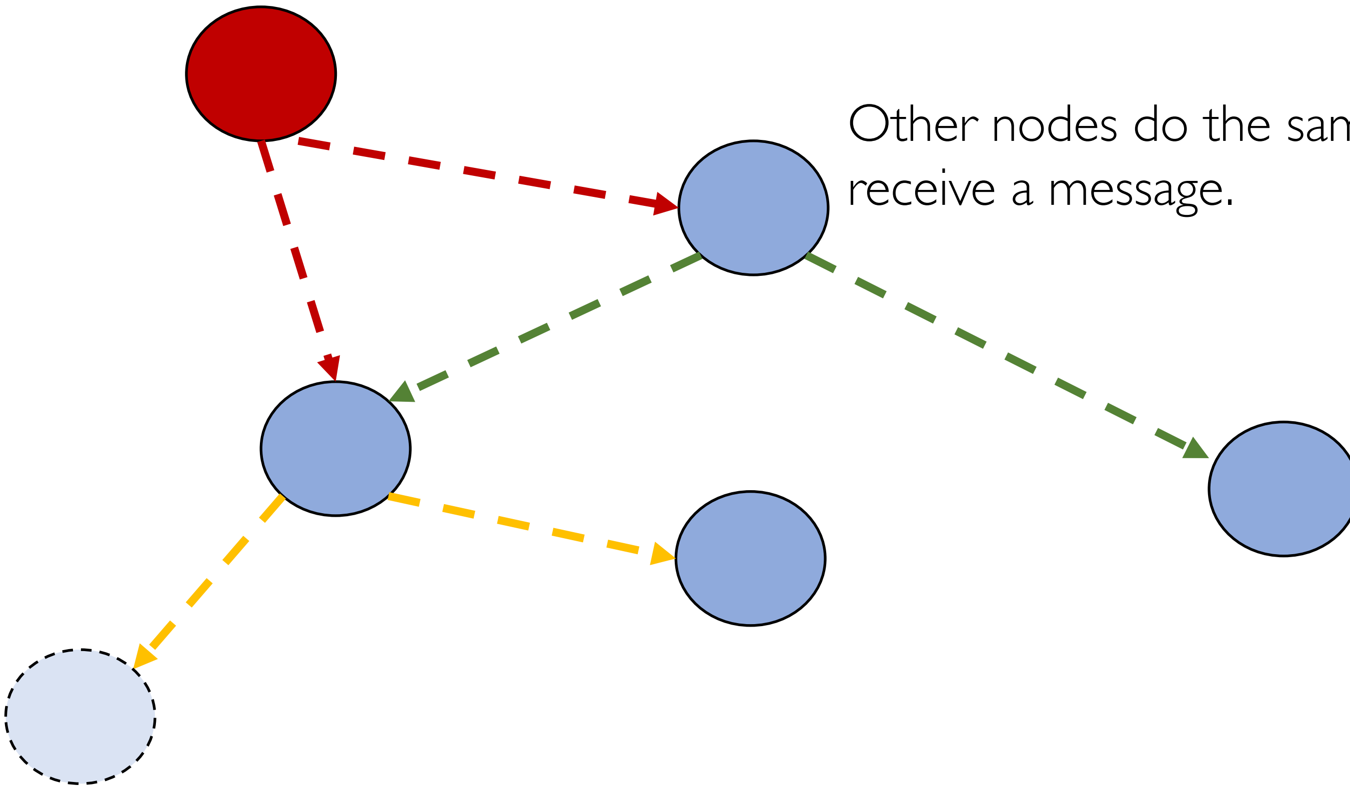
Transmit to b random targets.



Third approach: Gossip

Transmit to b random targets.

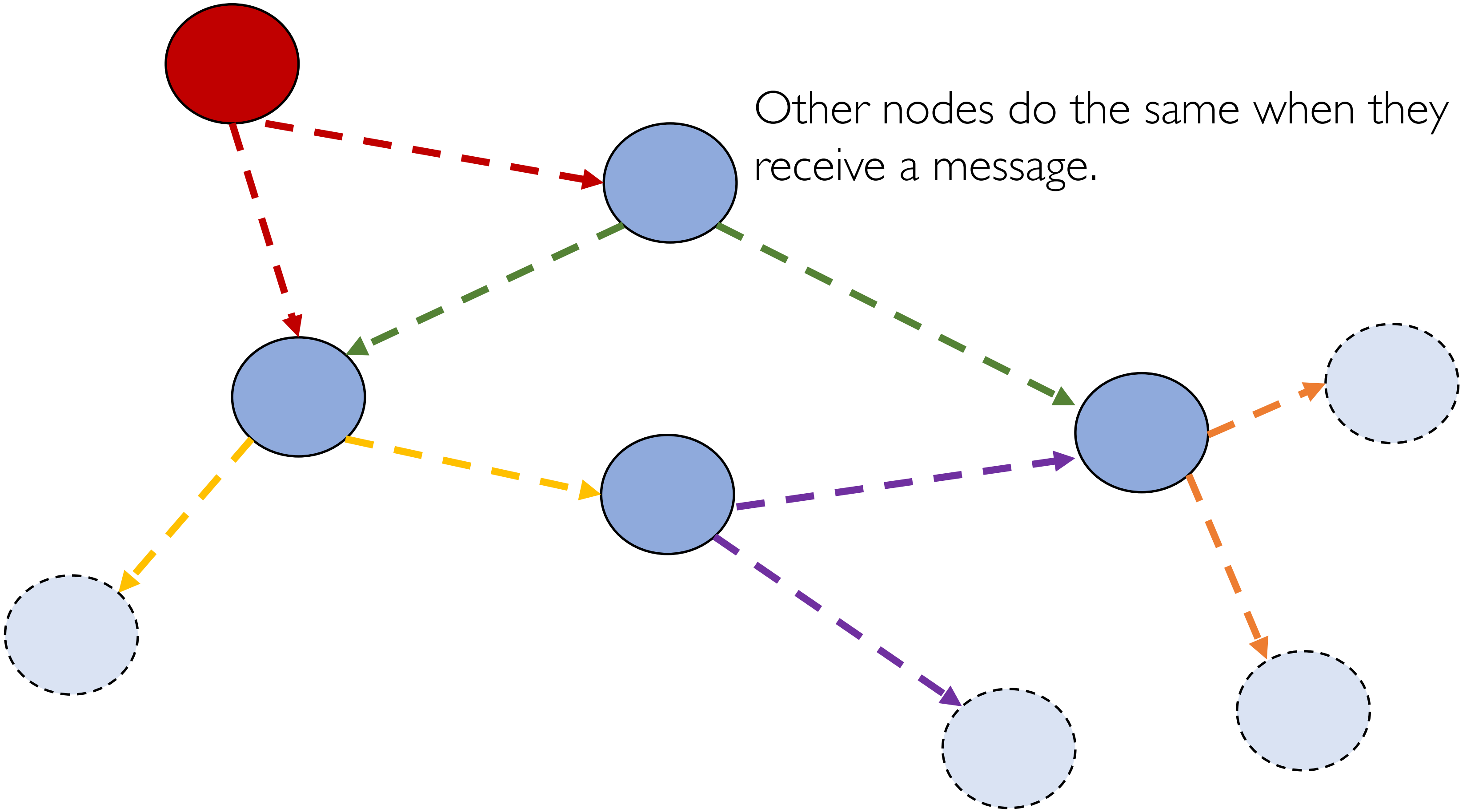
Other nodes do the same when they receive a message.



Third approach: Gossip

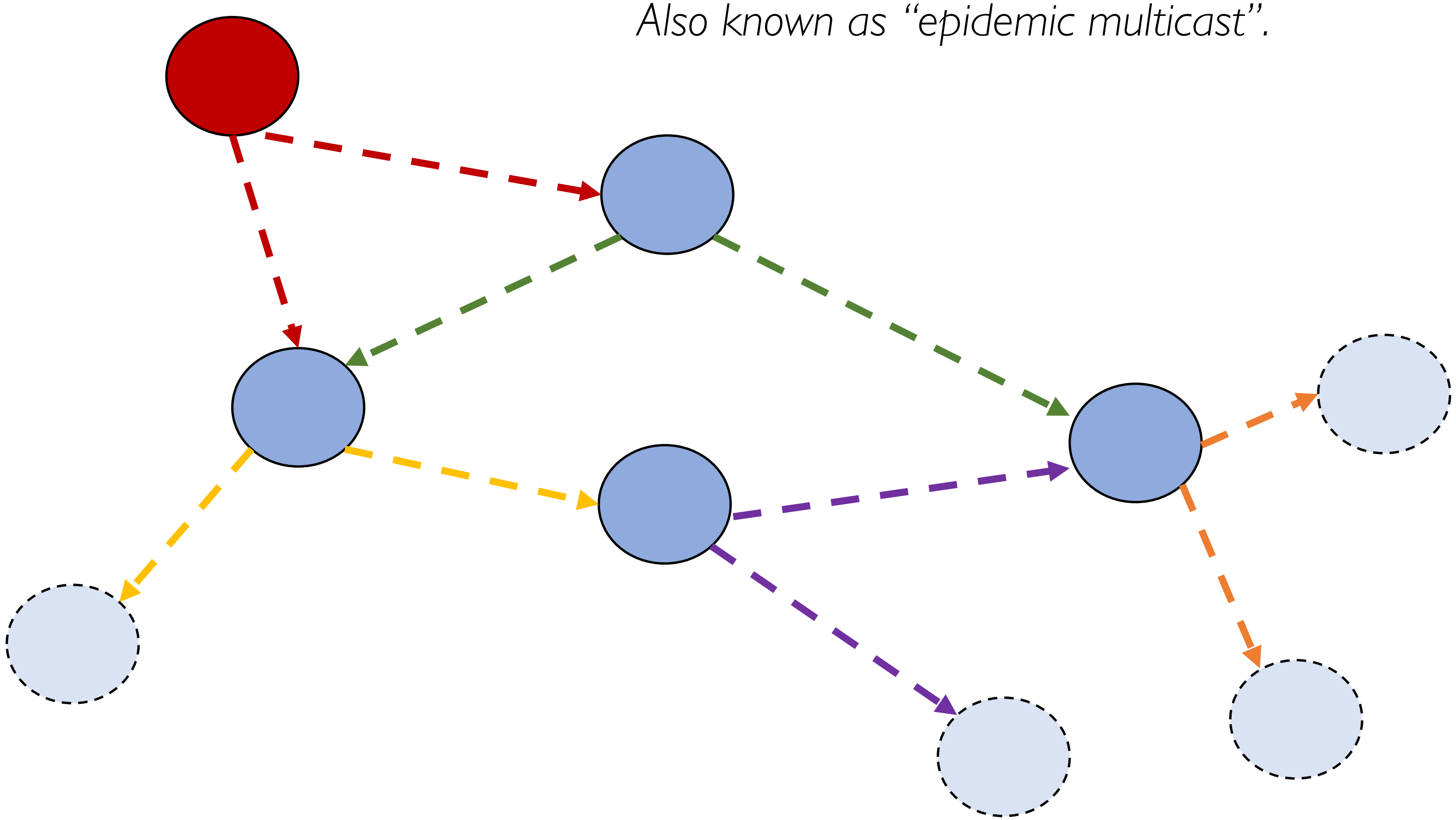
Transmit to b random targets.

Other nodes do the same when they receive a message.



Third approach: Gossip

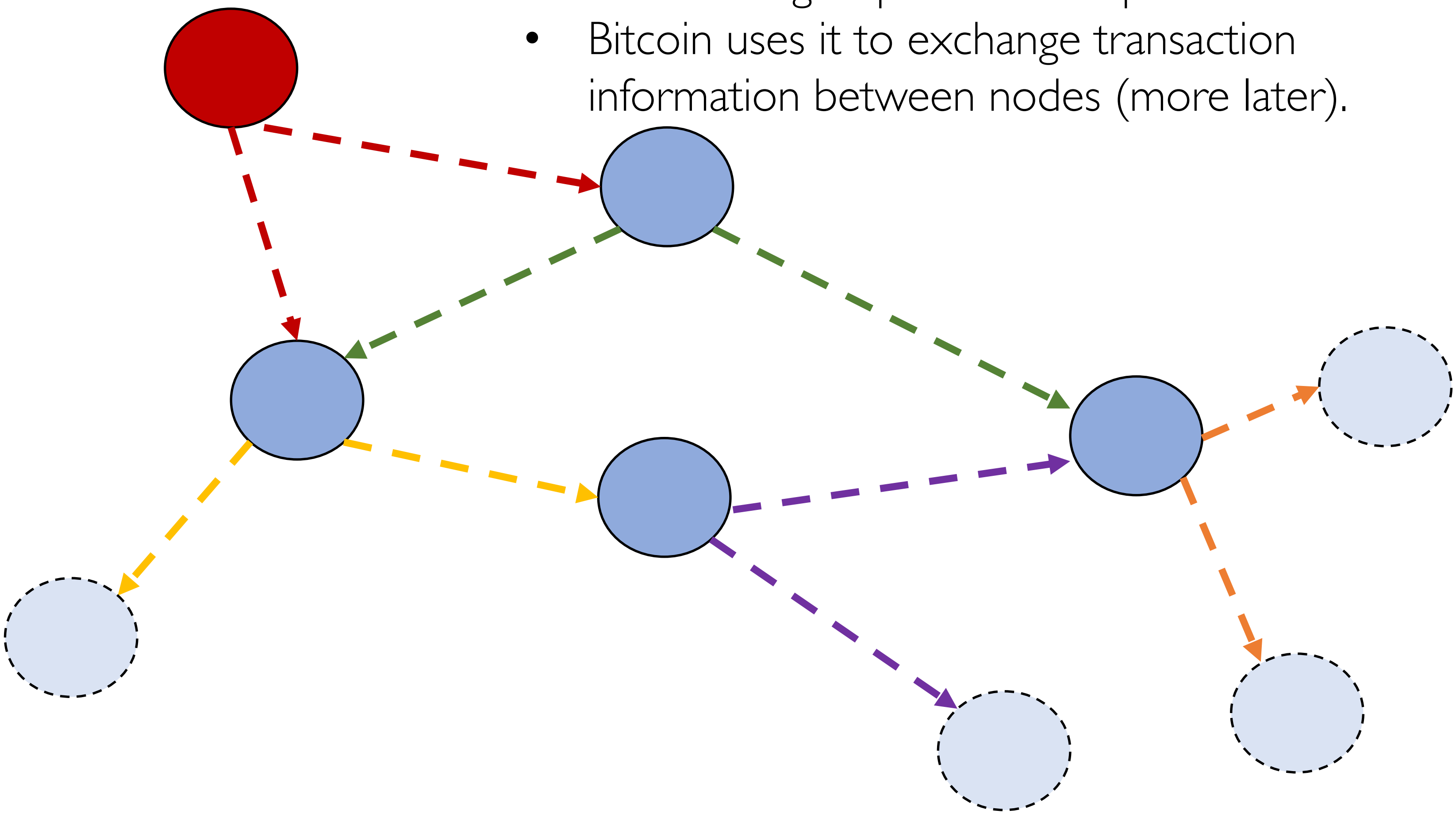
No "tree-construction" overhead.
More efficient than unicasting to all receivers.
Also known as "epidemic multicast".



Third approach: Gossip

Used in many real-world systems:

- Facebook's distributed datastore uses it to determine group membership and failures.
- Bitcoin uses it to exchange transaction information between nodes (more later).



Multicast Summary

- Multicast is an important communication mode in distributed systems.
- Applications may have different requirements:
  - Basic
  - Reliable
  - Ordered: FIFO, Causal, Total
  - Combinations of the above.
- Underlying mechanisms to spread the information:
  - Unicast to all receivers.
  - Tree-based multicast, and gossip: sender unicasts messages to only a subset of other processes, and they spread the message further.
  - Gossip is more scalable and more robust to process failures.

Today's agenda

- Wrap-up Multicast
  - Tree-based multicast and gossip
- Mutual Exclusion
  - Chapter 15.2
- Acknowledgement:
  - Materials largely derived from Prof. Indy Gupta.

Why Mutual Exclusion?

- Bank's Servers in the Cloud: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of $1000 concurrently from the bank's cloud server
  - Both ATMs add $10,000 to this amount (locally at the ATM)
  - Both write the final amount to the server
  - What's wrong?

Why mutual exclusion?

- Bank's Servers in the Cloud: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of $1000 concurrently from the bank's cloud server
  - Both ATMs add $10,000 to this amount (locally at the ATM)
  - Both write the final amount to the server
  - You lost $10,000!
- The ATMs need *mutually exclusive* access to your account entry at the server
  - or, mutually exclusive access to executing the code that modifies the account entry.

More uses of mutual exclusion

- Distributed file systems
    - Locking of files and directories
- Accessing objects in a safe and consistent way
    - Ensure at most one server has access to object at any point of time
- In industry
    - Chubby is Google's locking service

Problem Statement for mutual exclusion

- ***Critical Section* Problem:**
  - Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.

- Each process can call three functions
  - enter() to enter the critical section (CS)
  - AccessResource() to run the critical section code
  - exit() to exit the critical section

Our bank example

ATM1:

```
enter();
  // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
  // AccessResource() end
exit(); // exit
```

ATM2:

```
enter();
  // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
  // AccessResource() end
exit(); // exit
```

Mutual exclusion for a single OS

- If all processes are running in one OS on a machine (or VM):
  - Semaphores
  - Mutexes
  - Condition variables
  - Monitors
  - …

Processes Sharing an OS: Semaphores

- Semaphore == an integer that can only be accessed via two special functions
- Semaphore S=1; // Max number of allowed accessors.

wait(S) (or P(S) or down(S)):

while(1) { // each execution of the while loop is atomic
if (S > 0) {
S--;
break;
}
}

enter()

signal(S) (or V(S) or up(s)):

S++; // atomic

exit()

> *Atomic* operations are supported via hardware instructions such as compare-and-swap, test-and-set, etc.

Our bank example

ATM1:

```
enter();
  // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
 // AccessResource() end
exit(); // exit
```

ATM2:

```
enter();
  // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
 // AccessResource() end
exit(); // exit
```

Our bank example

Semaphore S=1; // shared

ATM1:

```
wait(S);
  // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
 // AccessResource() end
signal(S); // exit
```

ATM2:

```
wait(S);
  // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
 // AccessResource() end
signal(S); // exit
```

Mutual exclusion in distributed systems

- Processes communicating by passing messages.
- Cannot share variables like semaphores!
- *How do we support mutual exclusion in a distributed system?*

Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
    - Central server algorithm
    - Ring-based algorithm
    - Ricart-Agrawala Algorithm
    - Maekawa Algorithm

Mutual Exclusion Requirements

- Need to guarantee 3 properties:
  - Safety (essential):
    - At most one process executes in CS (Critical Section) at any time.
  - Liveness (essential):
    - Every request for a CS is granted eventually.
  - Ordering (desirable):
    - Requests are granted in the order they were made.

System Model

- Each pair of processes is connected by reliable channels (such as TCP).
- Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
- Processes do not fail.
  - Fault-tolerant variants exist in literature.

Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm

Central Server Algorithm

- Elect a central master (or leader)
- Master keeps
  - A **queue** of waiting requests from processes who wish to access the CS
  - A special **token** which allows its holder to access CS
- Actions of any process in group:
  - enter()
    - Send a request to master
    - Wait for token from master
  - exit()
    - Send back token to master

Central Server Algorithm

- Master Actions:
  - On receiving a request from process P*i*

    if (master has token)

    Send token to P*i*

    else

    Add P*i* to queue
  - On receiving a token from process P*i*

    if (queue is not empty)

    Dequeue head of queue (say P*j*), send that process the token

    else

    Retain token

Analysis of Central Algorithm

- Safety – at most one process in CS
  - Exactly one token
- Liveness – every request for CS granted eventually
  - With *N* processes in system, queue has at most *N* processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- Ordering:
  - FIFO ordering guaranteed in order of requests received at master
  - Not in the order in which requests were sent or the order in which processes enter CS!

Analysis of Central Algorithm

- Safety – at most one process in CS
  - Exactly one token
- Liveness – every request for CS granted eventually
  - With *N* processes in system, queue has at most *N* processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- Ordering:
  - FIFO ordering guaranteed in order of requests received at master
  - Not in the order in which requests were sent or the order in which processes enter CS!

Analyzing Performance

Three metrics:

- **Bandwidth**: the total number of messages sent in each *enter* and *exit* operation.
- **Client delay**: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
  - *We will focus on the client delay for the enter operation.*
- **Synchronization delay**: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting). Measure of the *throughput* of the system.

Analysis of Central Algorithm

- **Bandwidth**: the total number of messages sent in each *enter* and *exit* operation.
    - 2 messages for enter
    - 1 message for exit
- **Client delay**: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
    - 2 message latencies or 1 round-trip (request + grant) on enter.
- **Synchronization delay**: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)
    - 2 message latencies (release + grant)

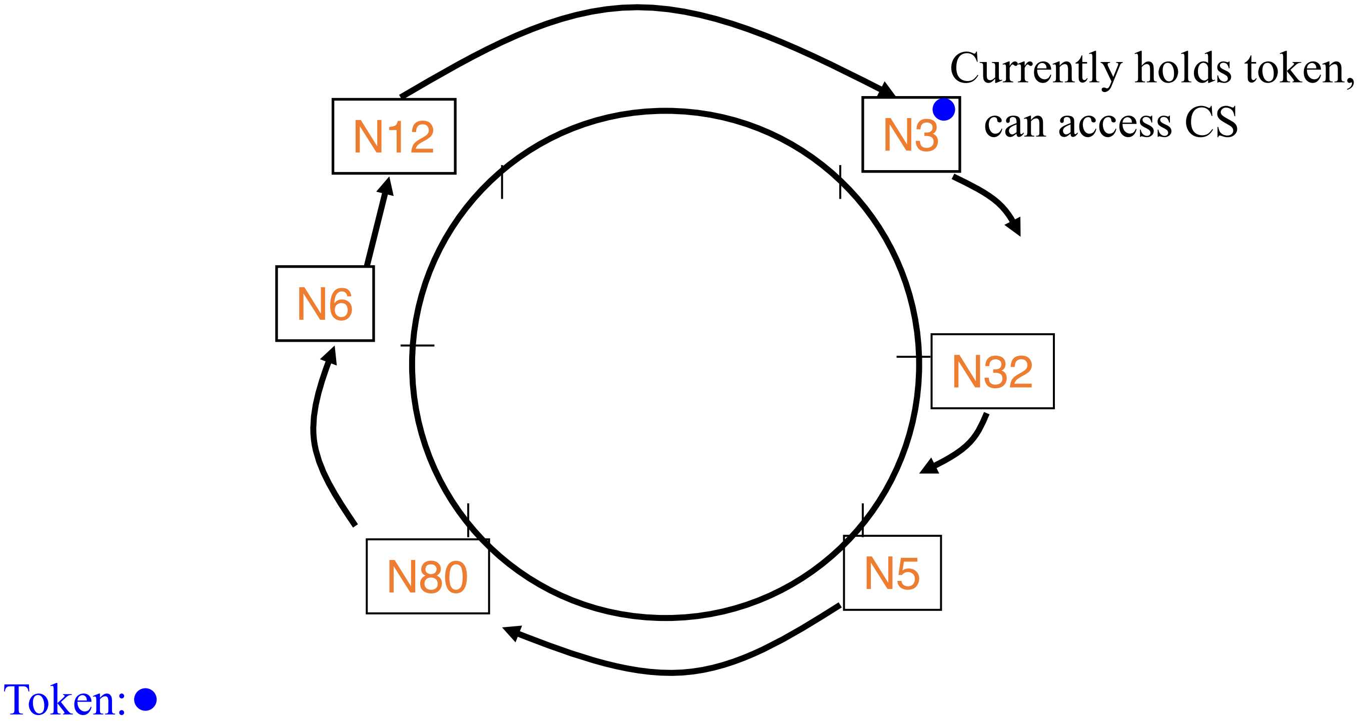
Limitations of Central Algorithm

- The master is the performance bottleneck and single point of failure.

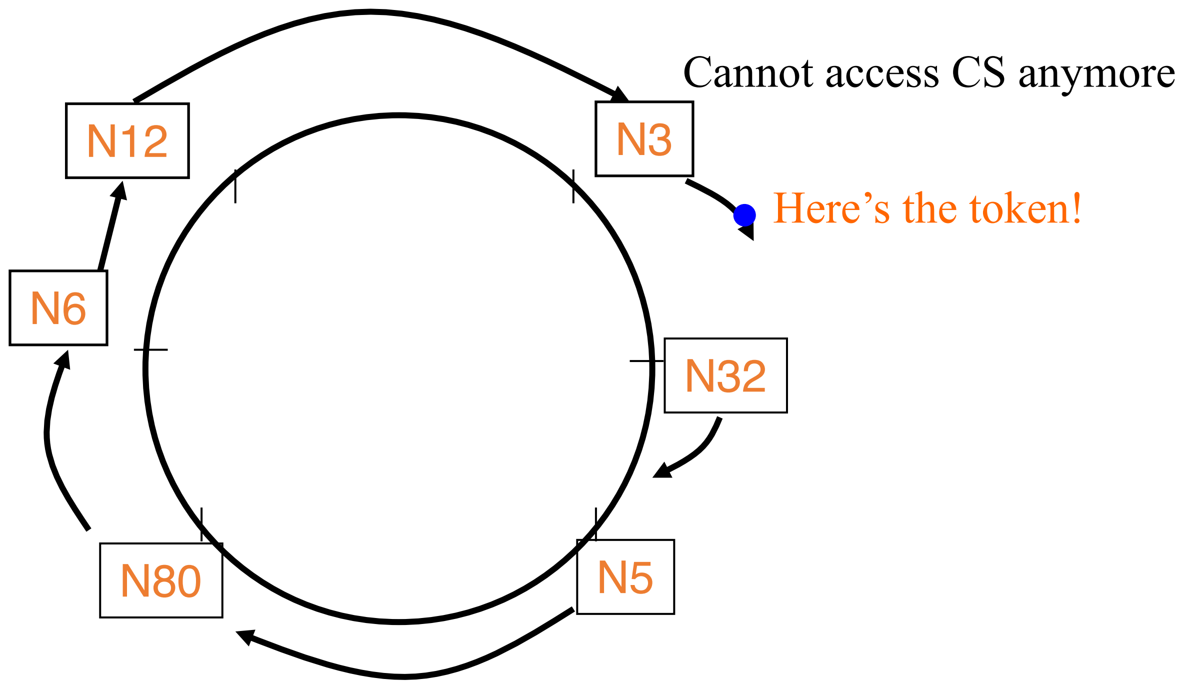
Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm

Ring-based Mutual Exclusion

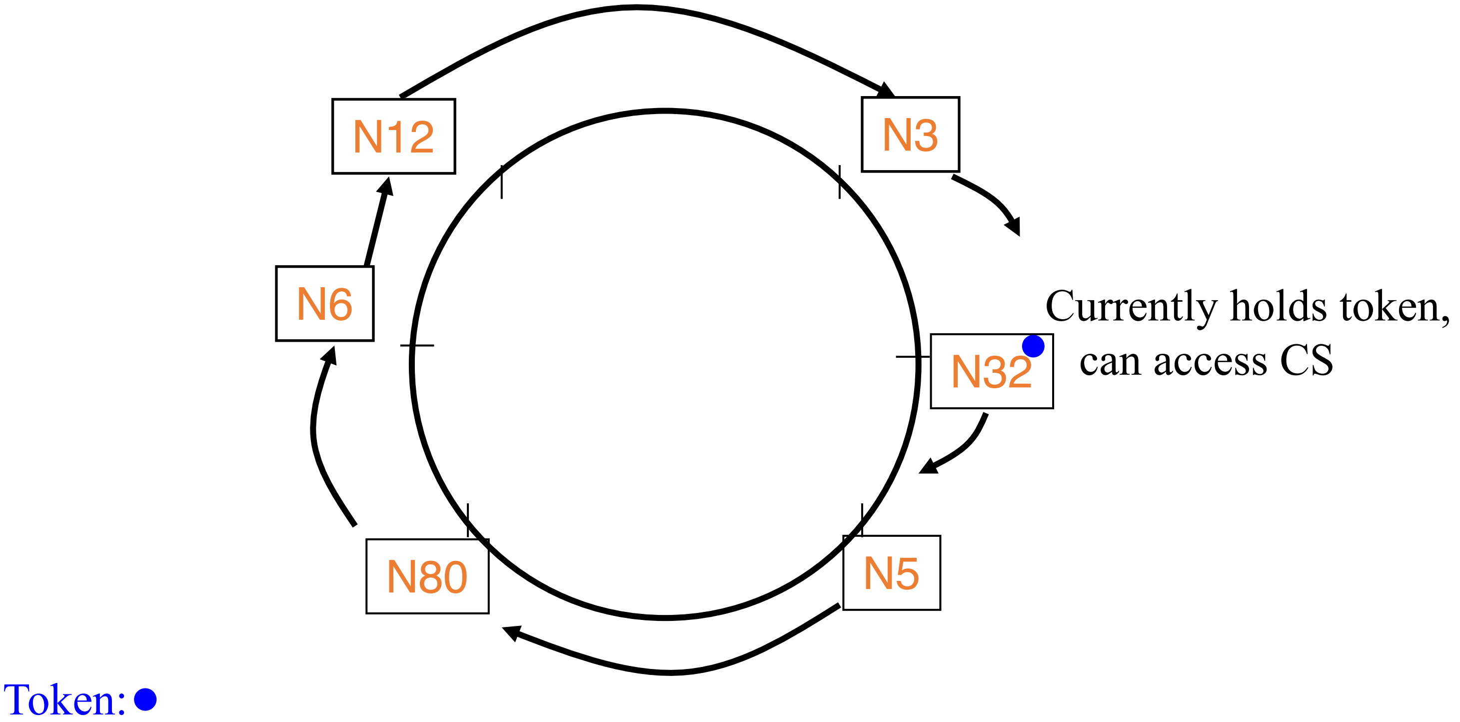


Ring-based Mutual Exclusion



Token: ●

Ring-based Mutual Exclusion



Ring-based Mutual Exclusion

- *N* Processes organized in a virtual ring
- Each process can send message to its successor in ring
- Exactly 1 token
- enter()
  - Wait until you get token
- exit() // already have token
  - Pass on token to ring successor
- If receive token, and not currently in enter(), just pass on token to ring successor

Analysis of Ring-based algorithm

- Safety
  - Exactly one token
- Liveness
  - Token eventually loops around ring and reaches requesting process (no failures)
- Ordering
  - Token not always obtained in order of enter events.

Analysis of Ring-based algorithm

- Safety
  - Exactly one token
- Liveness
  - Token eventually loops around ring and reaches requesting process (no failures)
- Ordering
  - Token not always obtained in order of enter events.

Analysis of Ring-based algorithm

- Bandwidth
  - Per enter, 1 message at requesting process but up to *N* messages throughout system.
  - 1 message sent per exit.
  - *Constantly consumes bandwidth even when no process requires entry to the critical section (except when a process is executing critical section).*

Analysis of Ring-based algorithm

- Client delay:
  - Best case: just received token
  - Worst case: just sent token to neighbor
  - 0 to *N* message transmissions after entering enter()
- Synchronization delay between one process' exit() from the CS and the next process' enter():
  - Best case: process in enter() is successor of process in exit()
  - Worst case: process in enter() is predecessor of process in exit()
  - Between 1 and (*N-1*) message transmissions.
- *Can we improve upon this O(n) client and synchronization delays?*

Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm

Ricart-Agrawala's Algorithm

- Classical algorithm from 1981
- Invented by Glenn Ricart (NIH) and Ashok Agrawala (U. Maryland)

- No token
- Uses the notion of causality and multicast.
- Has lower waiting time to enter CS than Ring-Based approach.

Key Idea: Ricart-Agrawala Algorithm

- enter() at process P*i*
    - multicast a request to all processes
        - Request: <T, P*i*>, where T = current Lamport timestamp at P*i*
    - Wait until *all* other processes have responded positively to request
- Requests are granted in order of causality.
- <T, P*i*> is used lexicographically: P*i* in request <T, P*i*> is used to break ties (since Lamport timestamps are not unique for concurrent events).

Messages in RA Algorithm

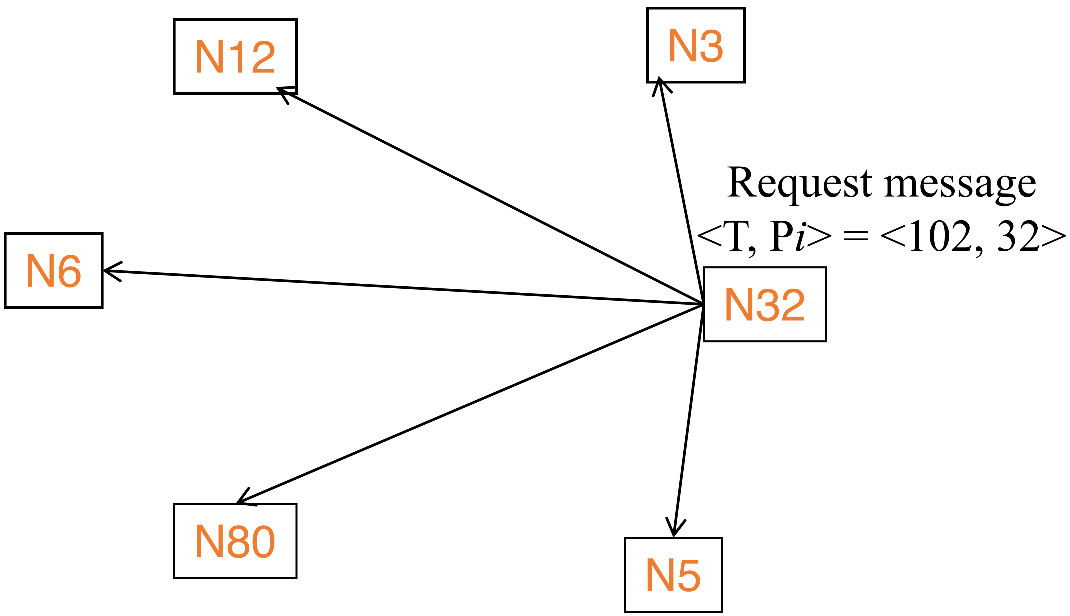
- enter() at process Pi
  - set state to <u>Wanted</u>
  - multicast "Request" <Ti, Pi> to all processes, where Ti = current Lamport timestamp at Pi
  - wait until <u>all</u> processes send back "Reply"
  - change state to <u>Held</u> and enter the CS
- On receipt of a Request <Tj, j> at Pi (i ≠ j):
  - if (state = <u>Held</u>) or (state = <u>Wanted</u> & (Ti, i) < (Tj, j))

    // lexicographic ordering in (Tj, j), Ti is Lamport timestamp of Pi's request

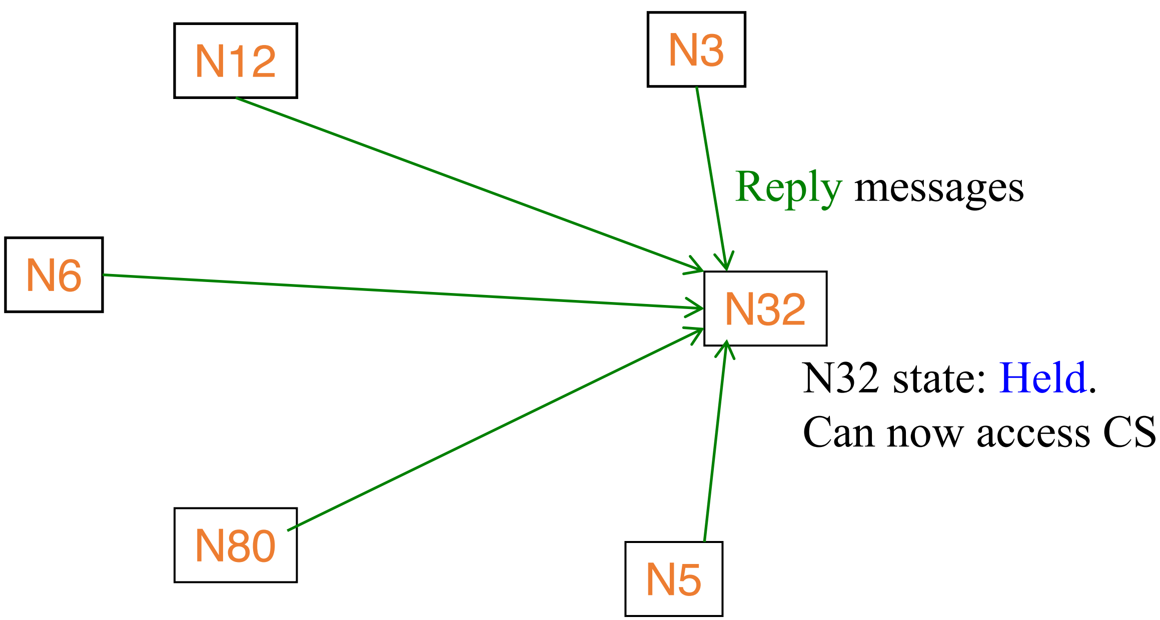
    add request to local queue (of waiting requests)

    else send "Reply" to Pj
- exit() at process Pi
  - change state to <u>Released</u> and "Reply" to <u>all</u> queued requests.

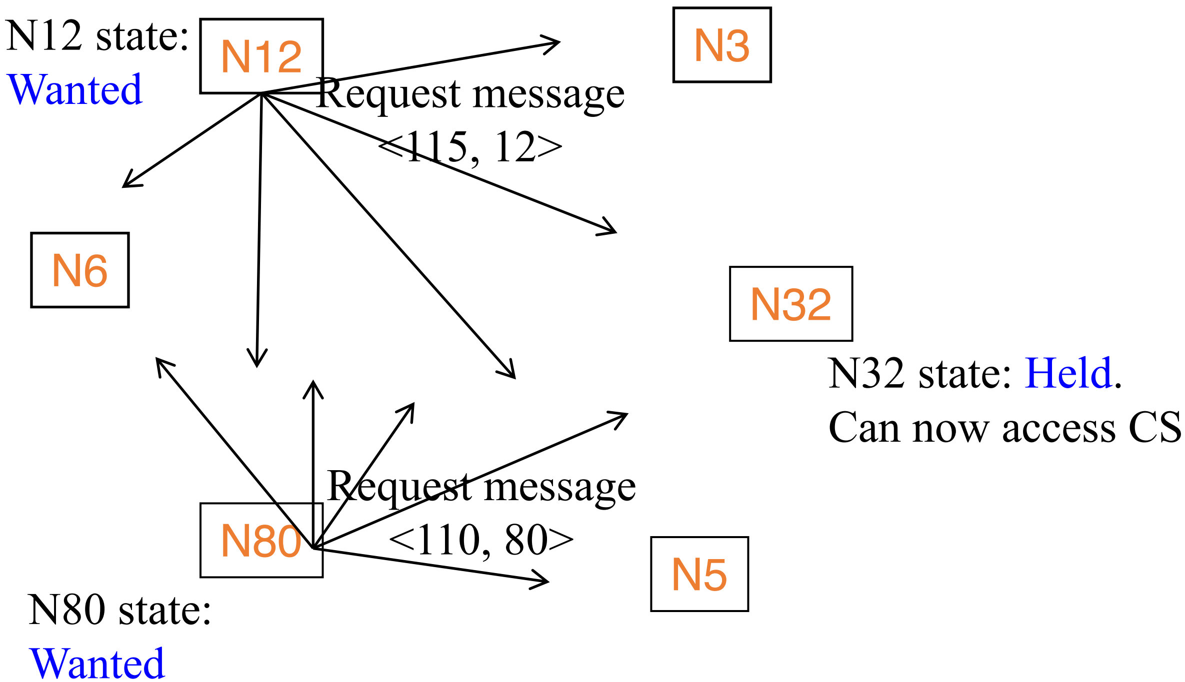
Example: Ricart-Agrawala Algorithm



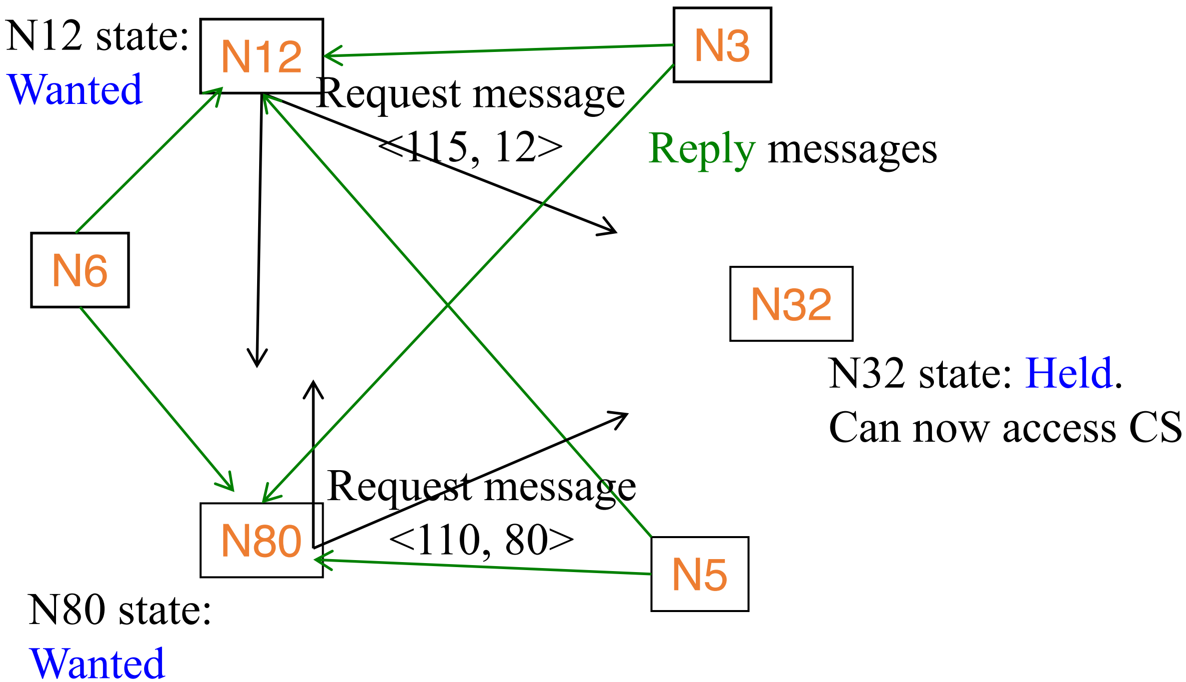
Example: Ricart-Agrawala Algorithm



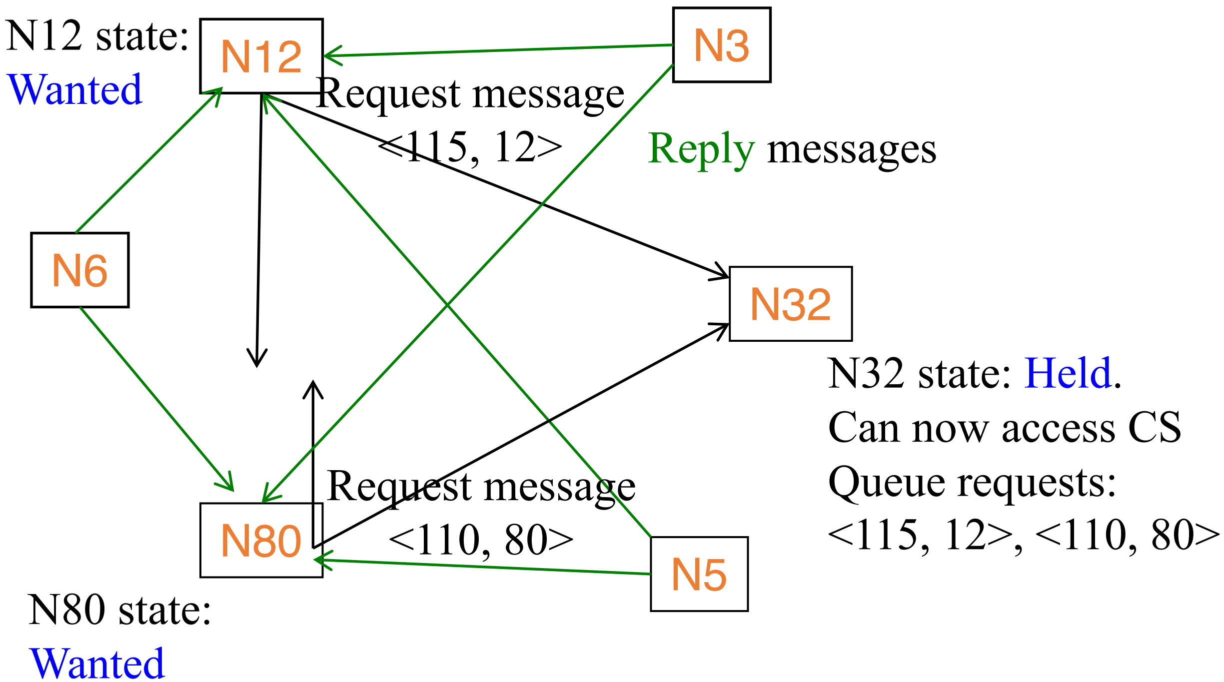
Example: Ricart-Agrawala Algorithm



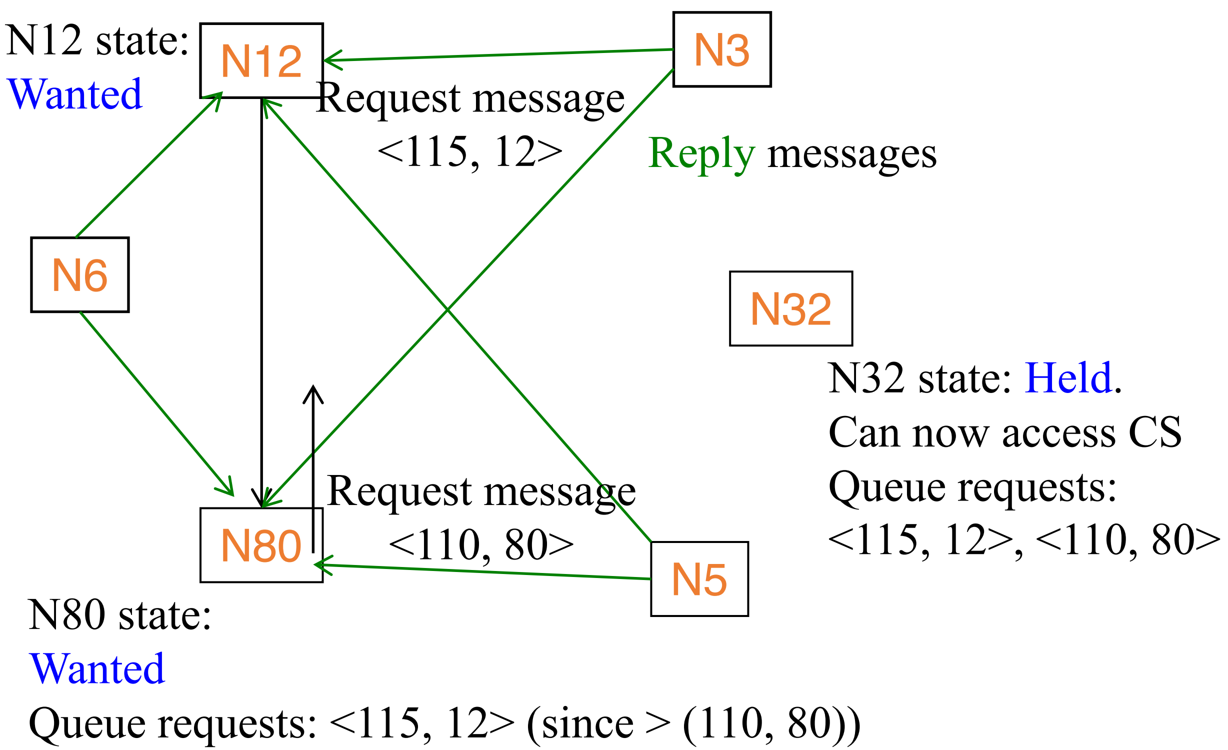
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm

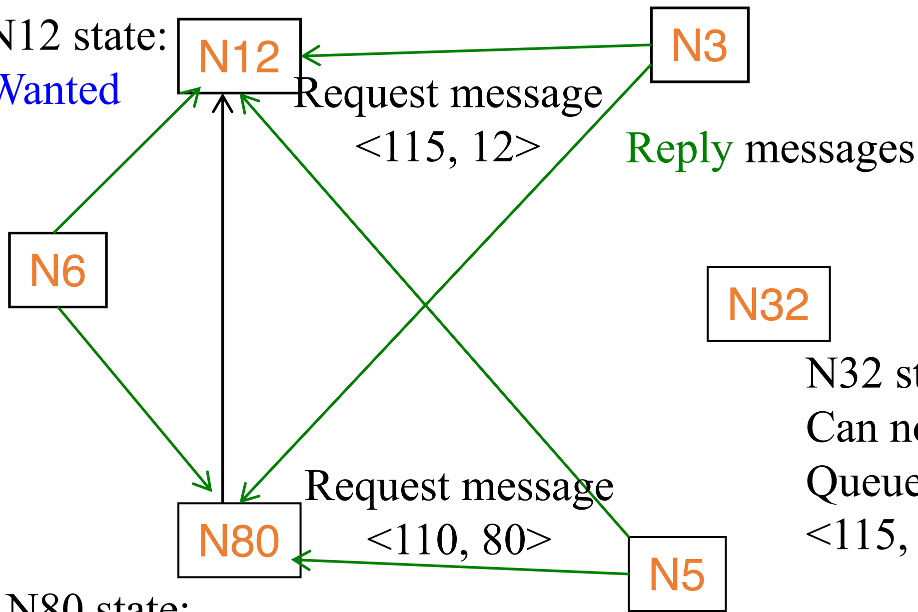


Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm

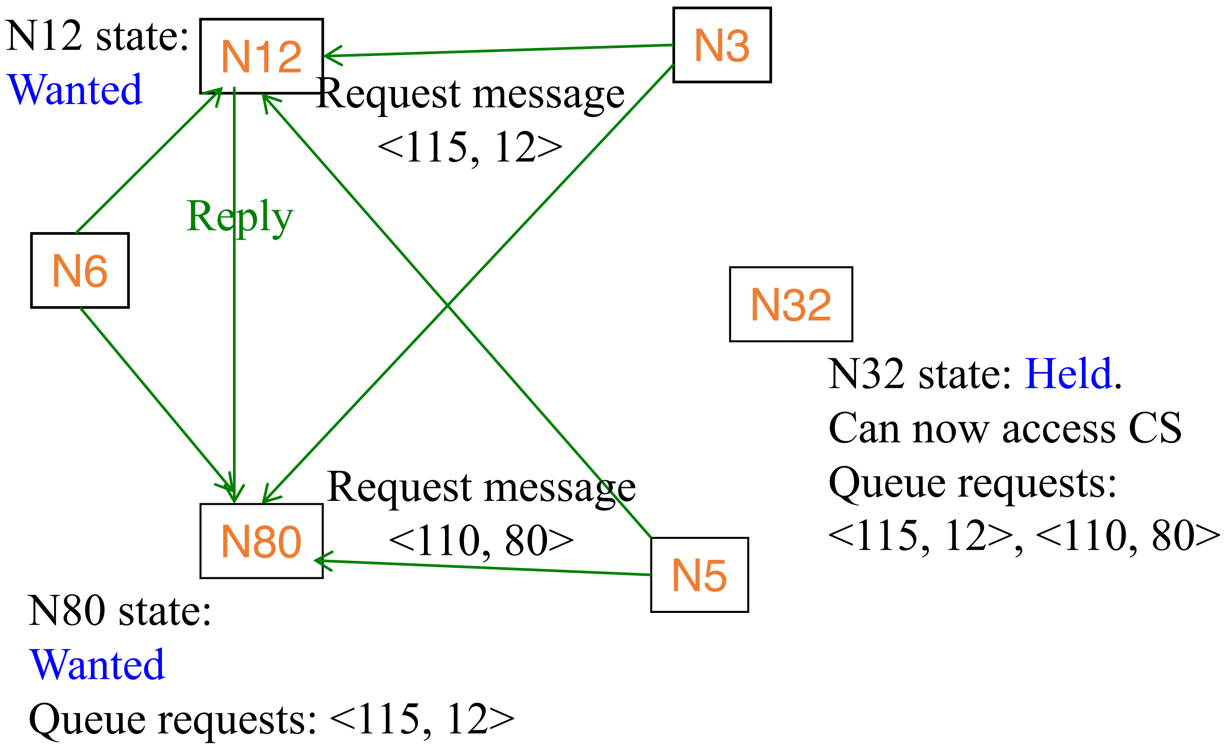
N12 state:
Wanted



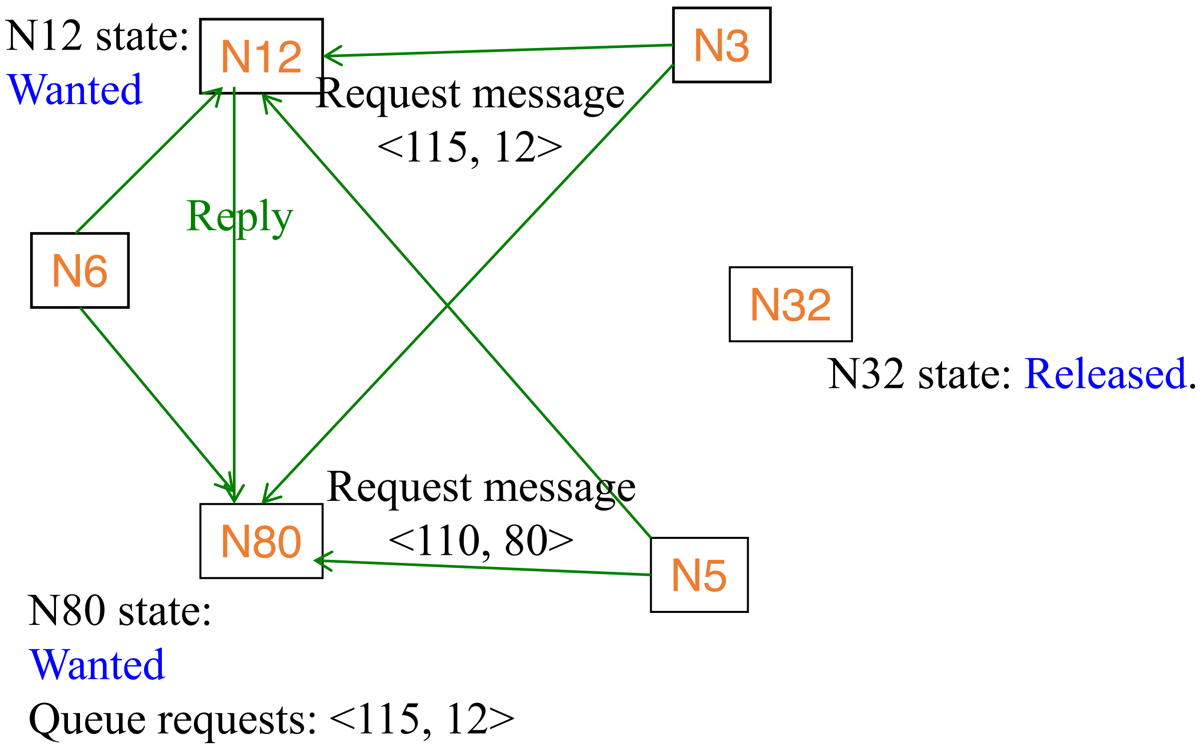
N32 state: Held.
Can now access CS
Queue requests:
<115, 12>, <110, 80>

N80 state:
Wanted
Queue requests: <115, 12> (since > (110, 80))

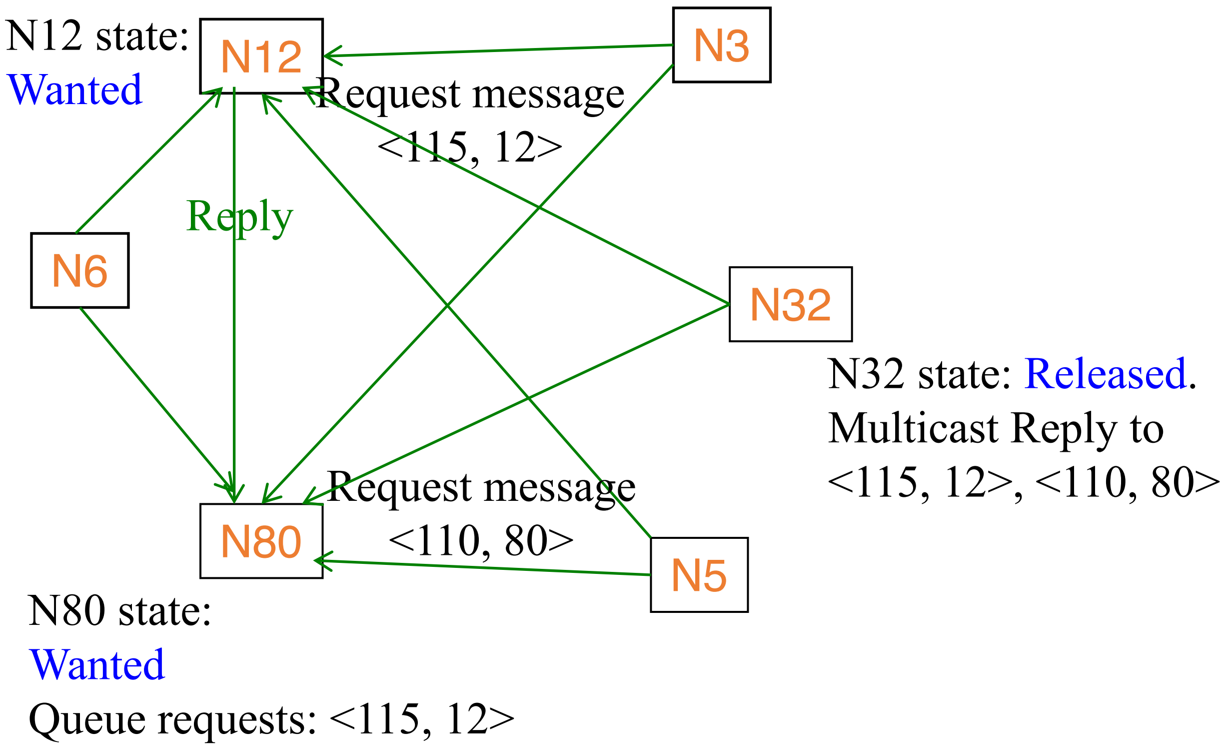
Example: Ricart-Agrawala Algorithm



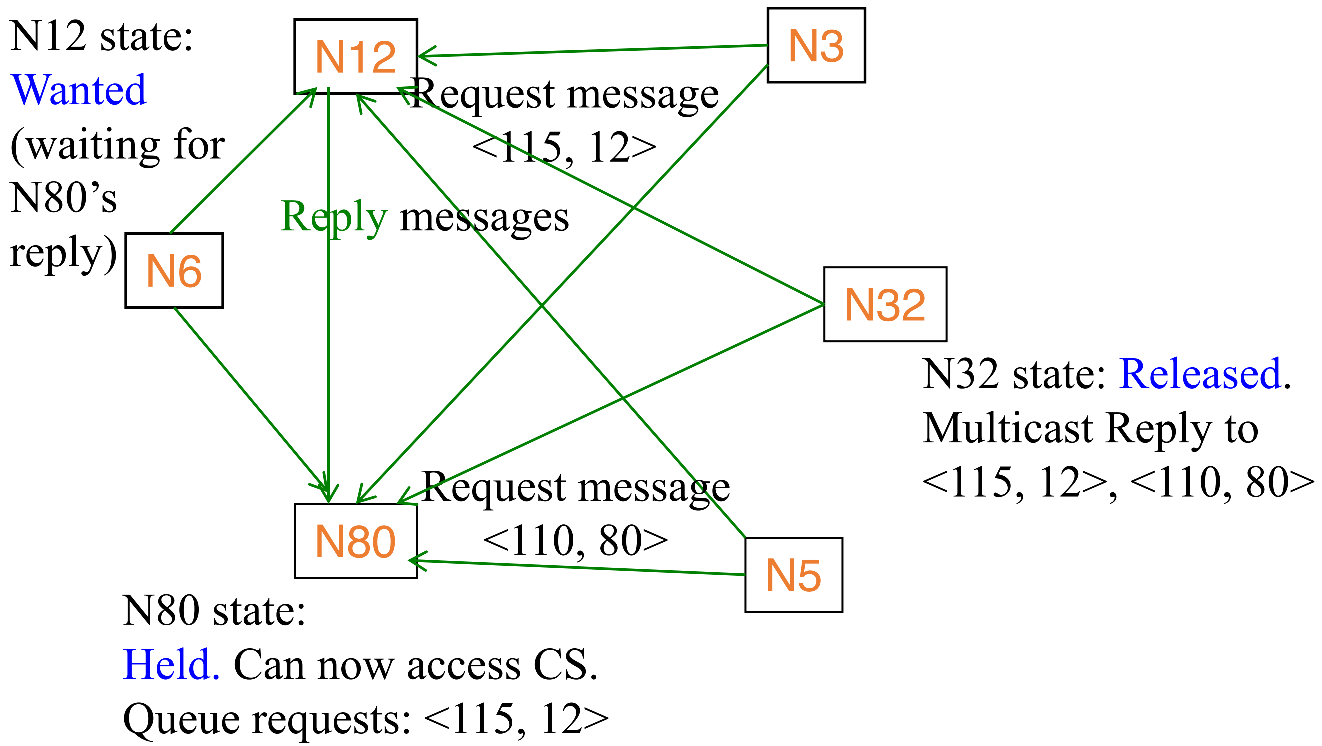
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



Next Class

- Analysis of Ricart-Agrawala algorithm.
- Maekawa algorithm for mutual exclusion.