Distributed Systems: Coordination & Agreement

Agenda

- Synchronous vs. asynchronous systems
- Distributed mutual exclusion
  - Mutual exclusion with shared variables (recap)
  - Algorithms for distributed mutual exclusion
- Leader election
  - Problem definition
  - Algorithms

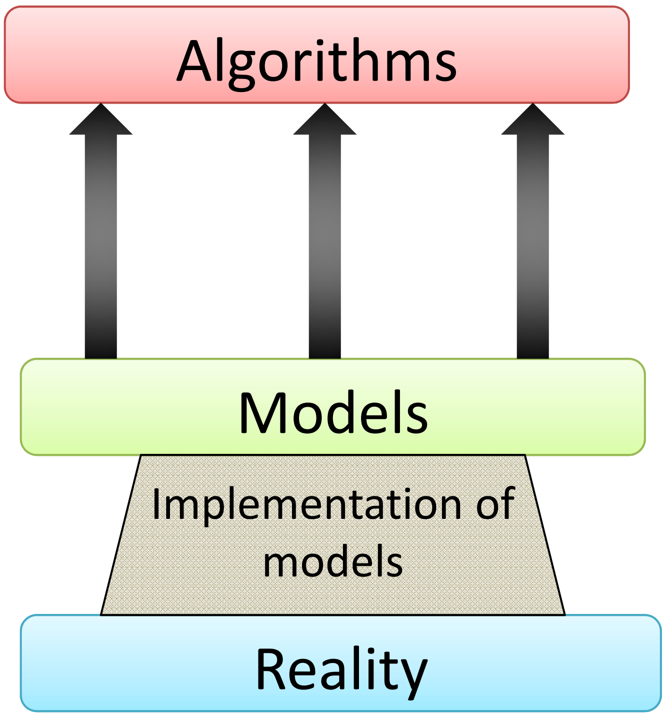
DISTRIBUTED SYSTEM MODELS

Distributed system model

Of theoretical relevance for designing algorithms

- Model captures all the **assumptions** made about the system
- Including network, clocks, processor, etc.
- Algorithms always **assume** a certain model
- Some algorithms are only possible in **stronger models** (more restrictions)
- Model is **theoretical**: whether its assumptions hold in practice is a different question

"All models are wrong, but some are useful"



Synchronous vs. asynchronous model

Property	Synchronous system model	Asynchronous system model
Clocks	Bound on drift	No bound on drift
Processes	Bound on execution time	No bound on execution time
Network	Bound on latency	No bound on latency

- There are other models.

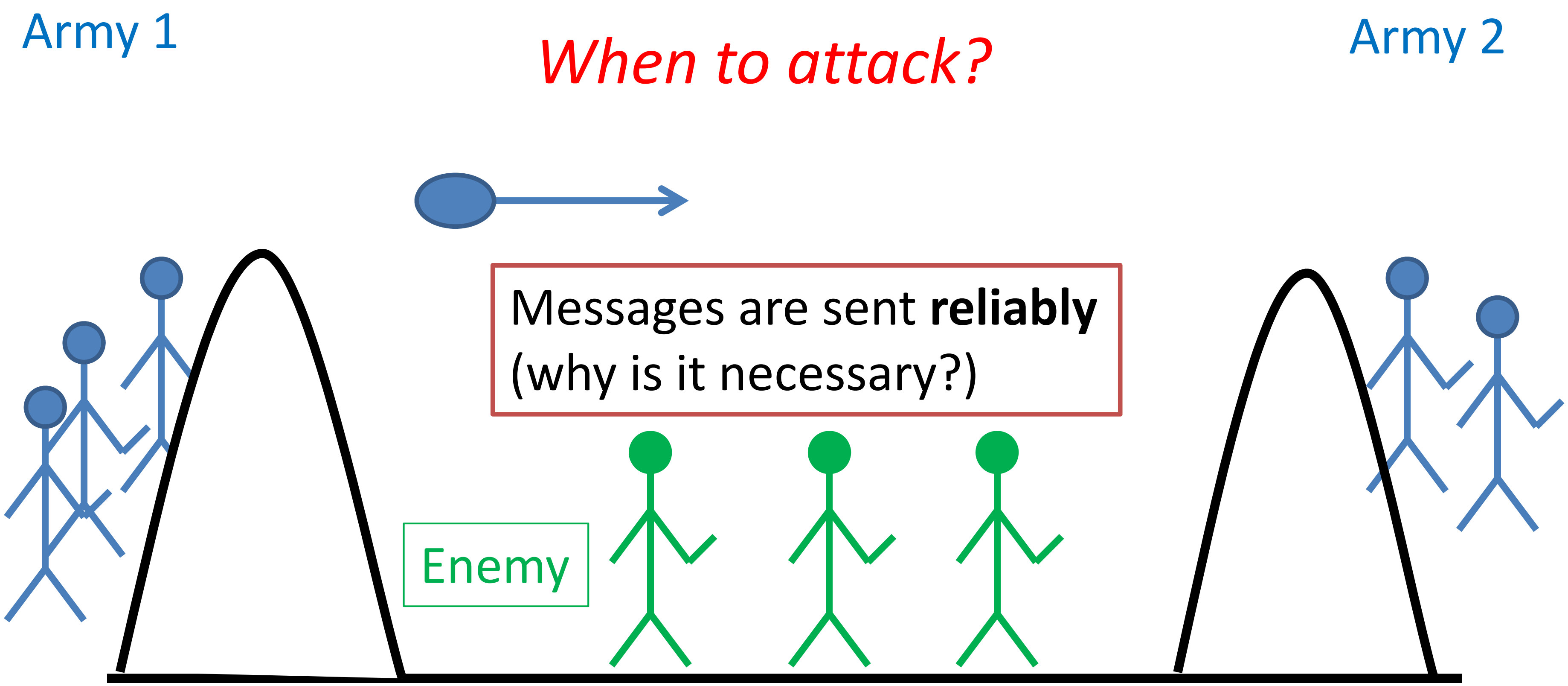
Two General's Problem (Agreement)

Armies need to agree on:

A thought experiment

Who leads the attack?

When to attack?



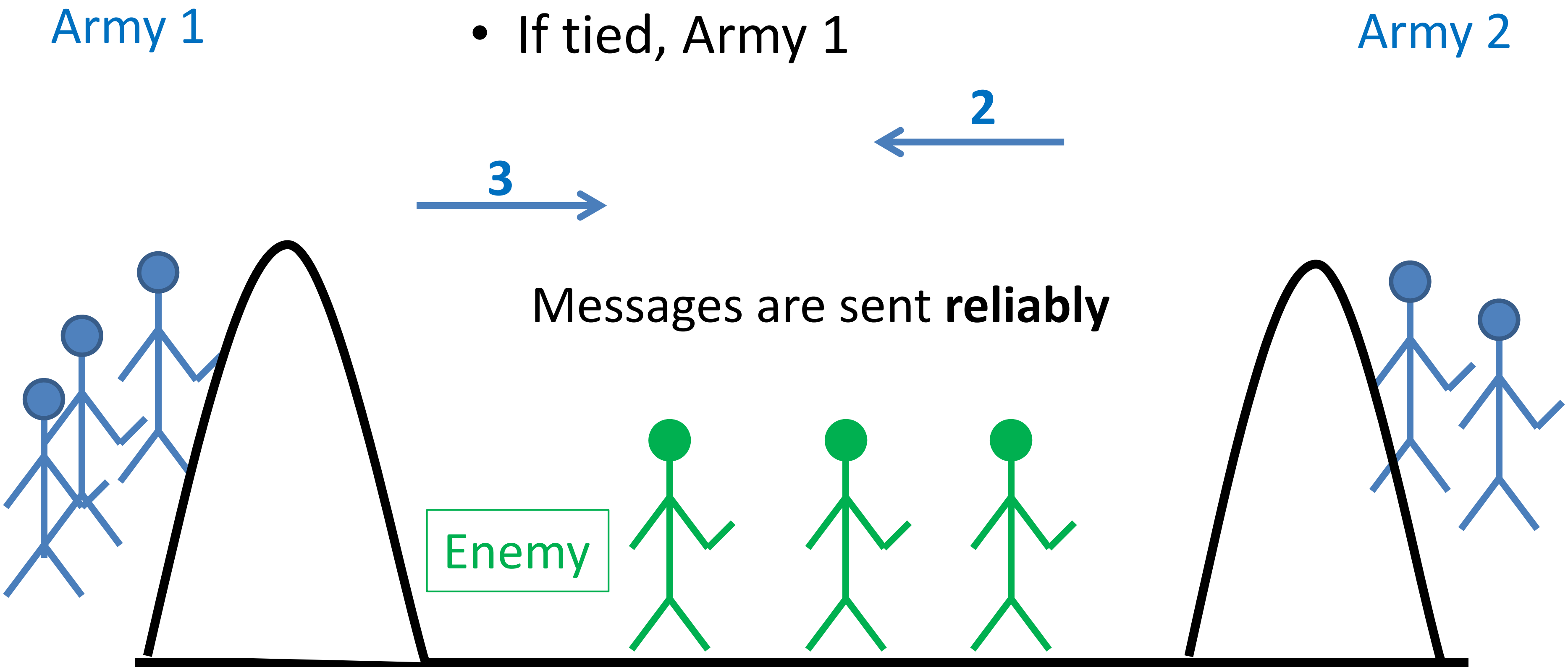
Armies are safe if they don't attack (or win) (Safety)

Armies need to coordinate attack to win. (Liveness)

Synchronous vs. asynchronous agreement

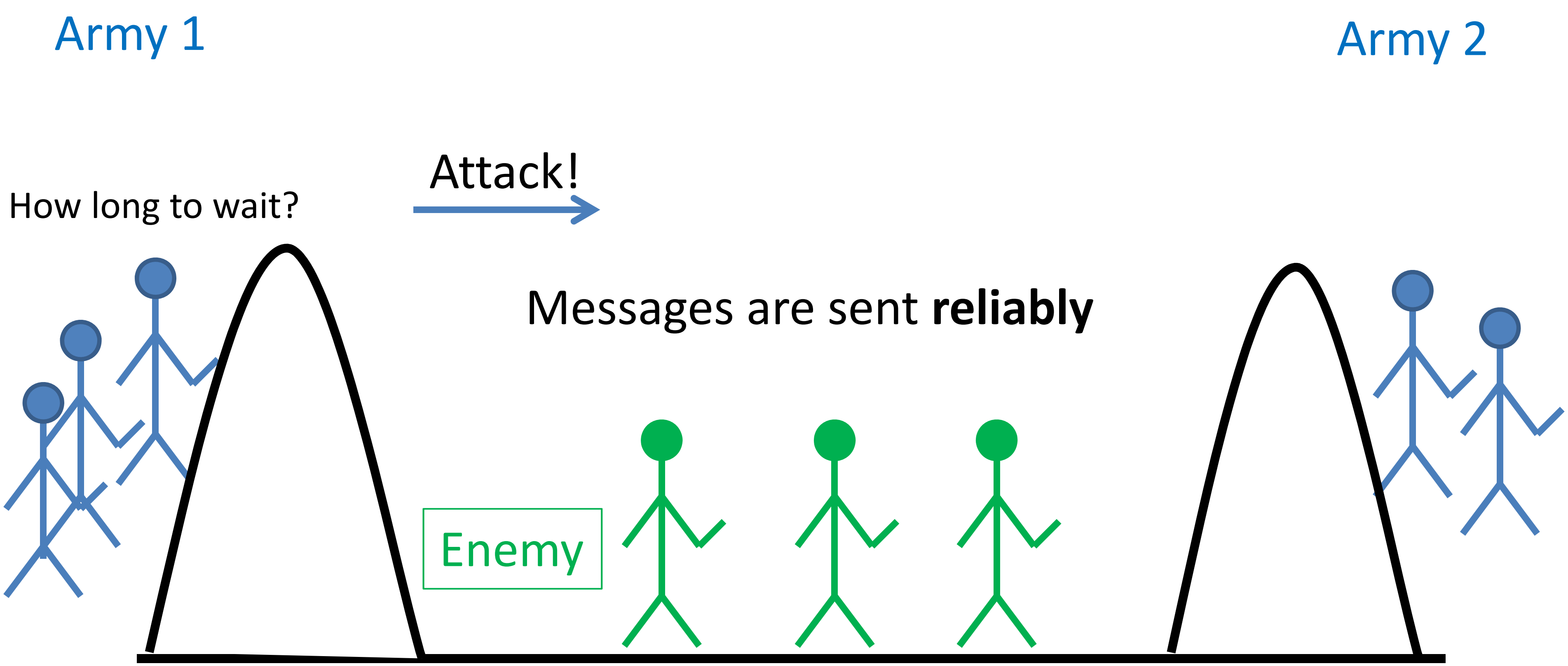
Who leads the attack?

- Largest army
- If tied, Army 1



Asynchronous agreement

When to attack? No bound on delivery!



Synchronous agreement

When to attack?

Message takes at least ***min*** time and at most ***max*** time to arrive

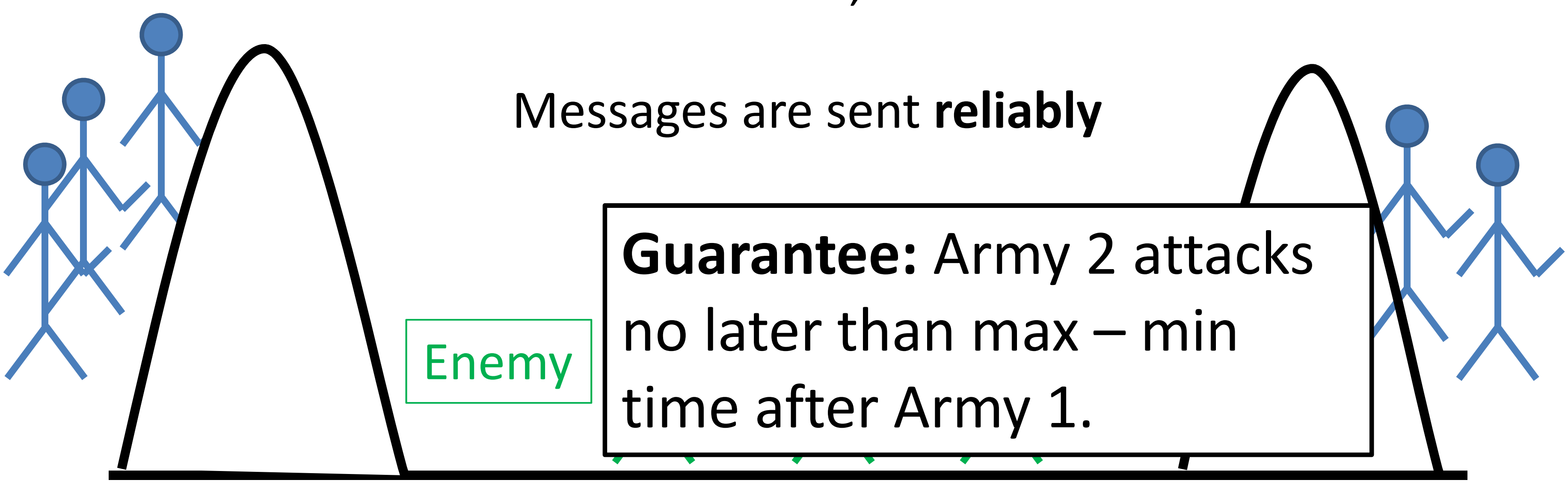
Army 1

Army 2

Attack!

Waits for min time, then attacks

Messages are sent **reliably**



Guarantee: Army 2 attacks no later than max – min time after Army 1.

Some takeaways

- Internet and many practical distributed applications are closer to asynchronous than synchronous model
- A solution valid for asynchronous distributed systems is also valid for synchronous ones (synchronous model is a stronger model)
- Many design problems cannot be solved in an asynchronous world (e.g., *when* vs. who leads attack)
- Apply **timeouts** and **timing assumptions** to reduce uncertainty and to bring elements of the synchronous model into the picture

You should have come across this before. E.g., ***Operating Systems*** *or* ***Computer Organiztion***

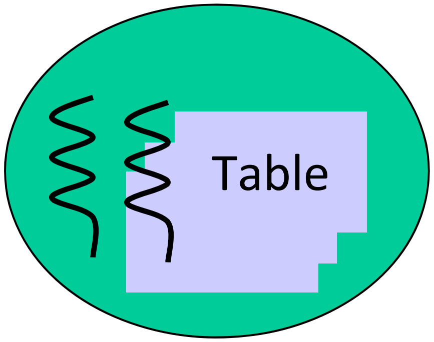
RECAP: MUTUAL EXCLUSION

Problem: Access to shared variables

- Imaging a **globally shared variable** `counter` in a process accessible to multiple threads
  - For example, the **key-value records** managed by a storage server (or more complex data structure)

Let's dissect the issue in detail

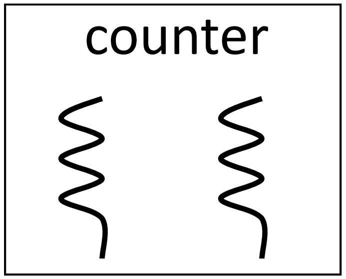
Shared data & synchronization



What may happen if multiple threads concurrently access shared process state (e.g., variables in memory)?

Concurrently manipulating shared data

- Two threads execute concurrently as part of the same process
- Shared variable (e.g., global variable)
  - `counter = 5`
- Thread 1 executes
  - `counter++`
- Thread 2 executes
  - `counter--`
- *What are **all the possible values** of* `counter` *after Thread 1 and Thread 2 finish executing?*



Machine-level implementation

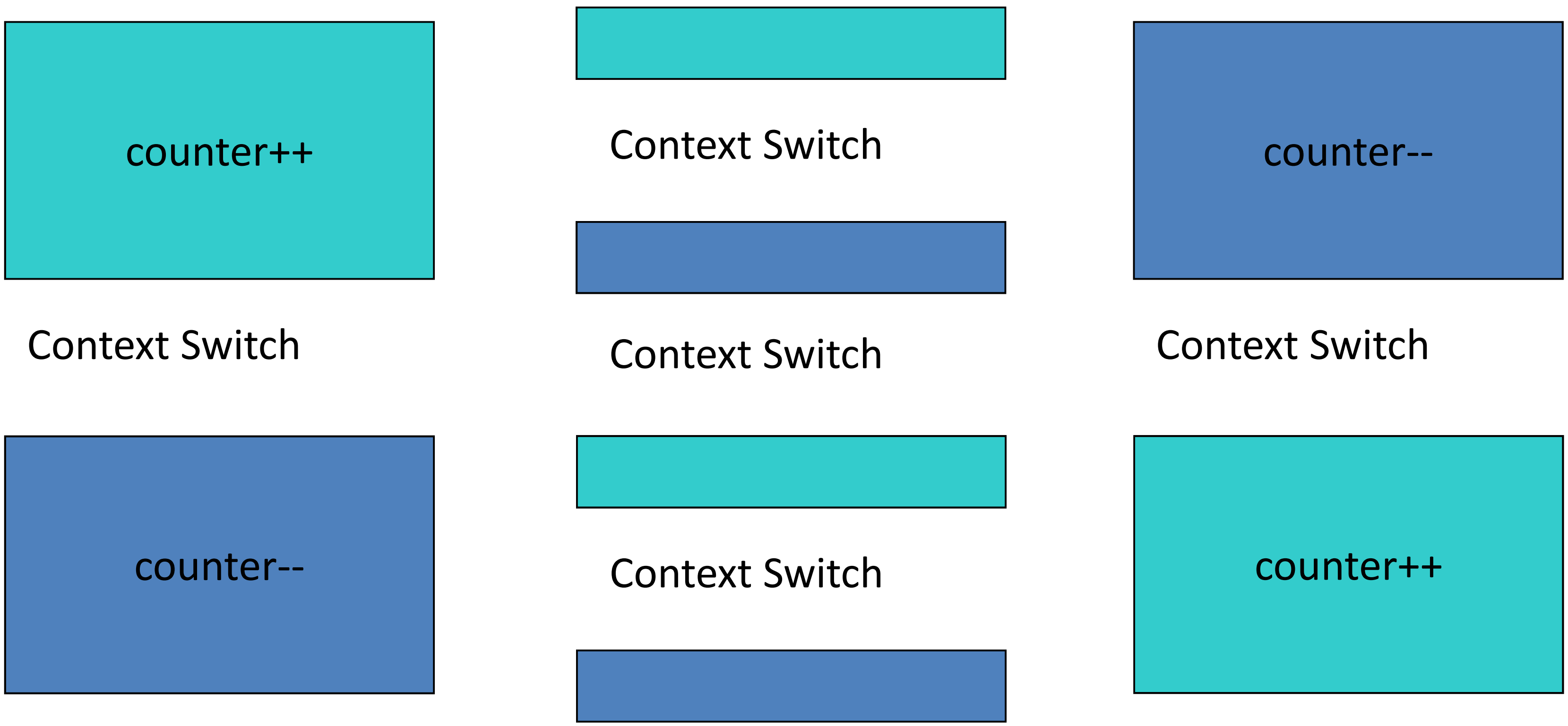
- Implementation of "counter++"

$$
\begin{aligned}
\textbf{register}_1 &= \textbf{counter} \\
\textbf{register}_1 &= \textbf{register}_1 + 1 \\
\textbf{counter} &= \textbf{register}_1
\end{aligned}
$$

- Implementation of "counter--"

$$
\begin{aligned}
\textbf{register}_2 &= \textbf{counter} \\
\textbf{register}_2 &= \textbf{register}_2 - 1 \\
\textbf{counter} &= \textbf{register}_2
\end{aligned}
$$

Possible execution sequences



Interleaved execution

- Assume **counter** is 5 and interleaved execution of counter++ (in T_1) and counter-- (in T_2)

T_1: r_1	**=**	**counter**	(*register$_1$ = 5*)
T_1: r_1	**=**	**$r_1 + 1$**	(***register$_1$ = 6***)
T_2 : r_2	**=**	**counter**	(*register$_2$ = 5*)
T_2 : r_2	**=**	**$r_2 - 1$**	(***register$_2$ = 4***)
T_1 : **counter**	**=**	**r_1**	(***counter = 6***)
T_2 : counter	**=**	**r_2**	(***counter = 4***)

Context switch

- The value of **counter** may be 4 or 6, whereas **the correct result should be 5**!

Race condition

- **Race condition**
  - **Several** threads **manipulate shared data concurrently**. The **final value** of the data **depends** upon **which thread finishes last**.
- In our example (interleaved execution) of counter++ with counter--
- To prevent race conditions, concurrent processes must be **synchronized**!

The moral of this story

- The statements
  counter++;
  counter--;
  must each be executed ***atomically***.
- Atomic operation means an operation that **completes in its entirety without interruption**.
- This is achieved through **synchronization primitives**
- Shared variable accessed in **critical section,** protected by synchronization primitives
- Known as the **critical section problem** or as **mutual exclusion**

"*Don't worry*", in distributed systems (the ones we look at), there is no shared memory – but, …

DISTRIBUTED MUTUAL EXCLUSION

Distributed mutual exclusion

- In distributed systems, mutual exclusion is more complex due to lack of:
  - Shared memory
  - Timing issues
  - Lack of a global clock
  - Event ordering
- Applications
  - Accessing a shared resource in distributed systems
  - One active Bigtable master to coordinate

Critical section (CS) problem: No shared memory

- System with *n* processes
- Processes access shared resources in CS
- **Coordinate access to CS via message passing**
- Application-level protocol for accessing CS
  - Enter_CS() – enter CS, block if necessary
  - ResourceAccess() – access shared resource in CS
  - Exit_CS() – leave CS

Current assumptions

Clearly, not practical, more of theoretical nature

- System is asynchronous
  - No bound on delays, no bound on clock drift, etc.
- Processes do not fail
- Message delivery is reliable
  - Any message sent, is eventually delivered intact and exactly once
- Client processes are well-behaved and spent finite time accessing resources in CS

Mutual exclusion requirements

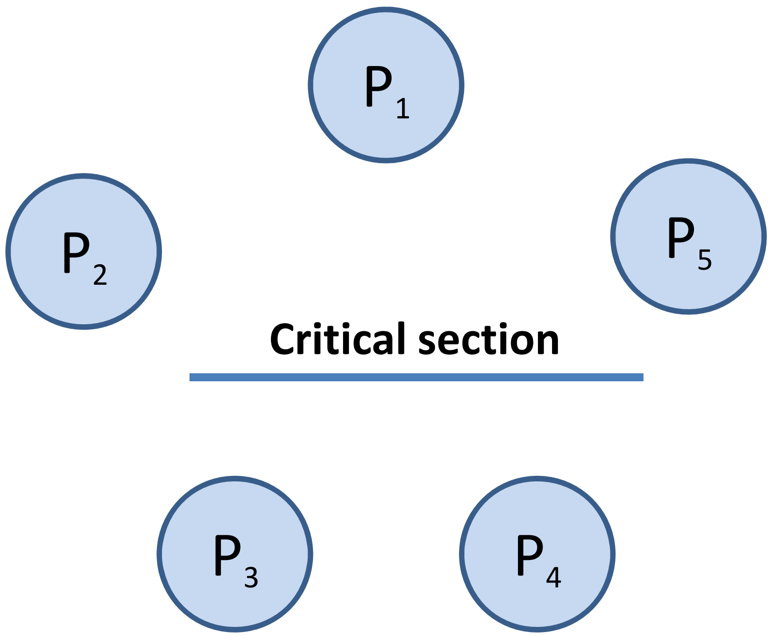
- **Safety** – correctness
  - At most one process in the critical section at a time
- **Liveness –** progress (something good happens)
  - Requests to enter/exit CS eventually succeed
  - No deadlock
- **Fairness** (order & starvation)
  - If one request to enter CS **happened-before** another one, then entry to CS is granted in that **order**
  - Requests are ordered such that no process enters the critical section twice while another waits to enter (i.e., **no starvation**)

Possible performance metrics

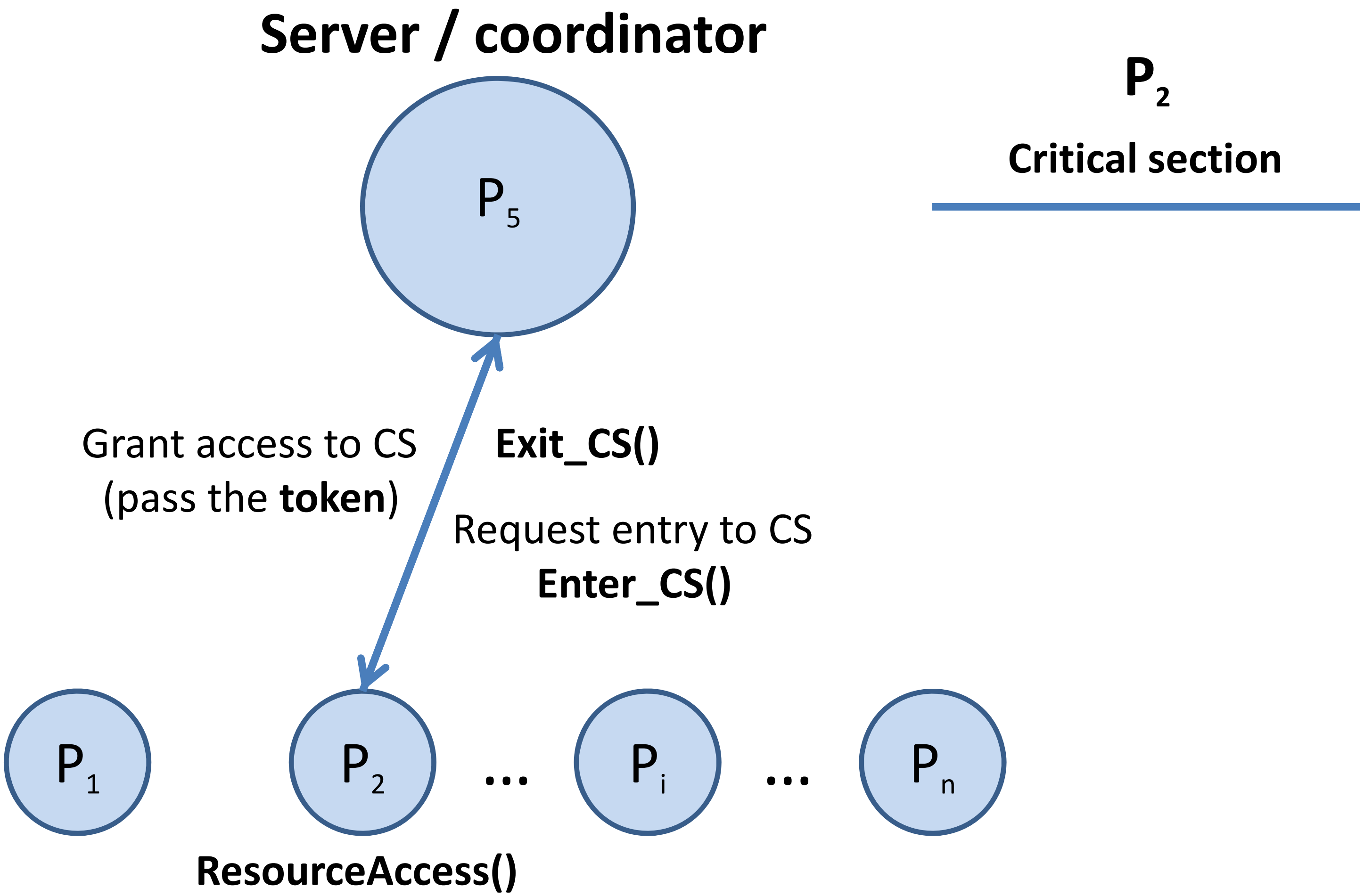
- **Bandwidth:** Number of messages sent, received or both
- **Synchronization delay**: Time between one process **exiting** critical section and next one **entering**
- **Client delay**: Delay **at entry** and **exit** (response time)
- We do not measure client access to resources protected by the critical section (**assume finite**)

Centralized strategy

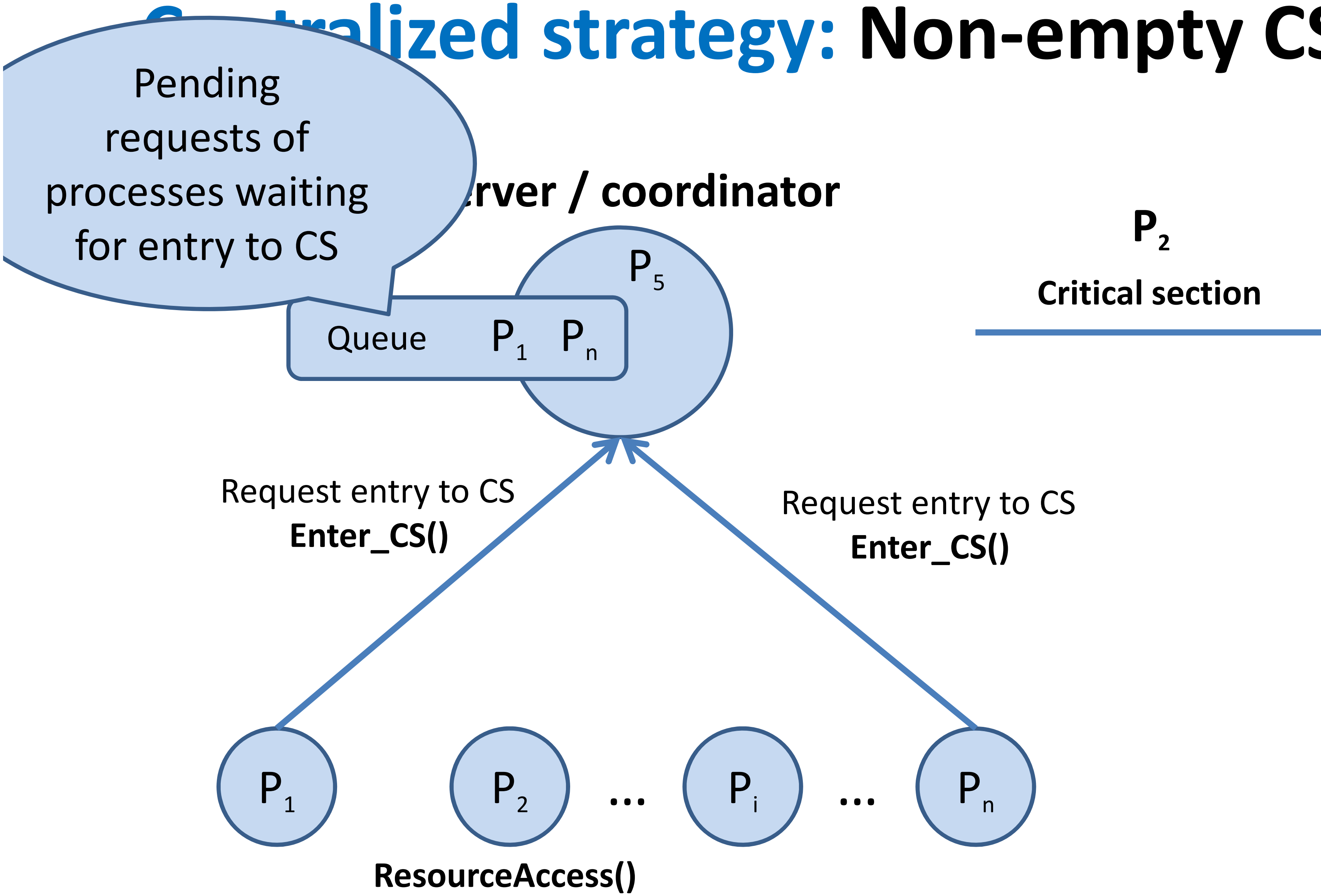
1. Elect a leader (details, cf. soon)



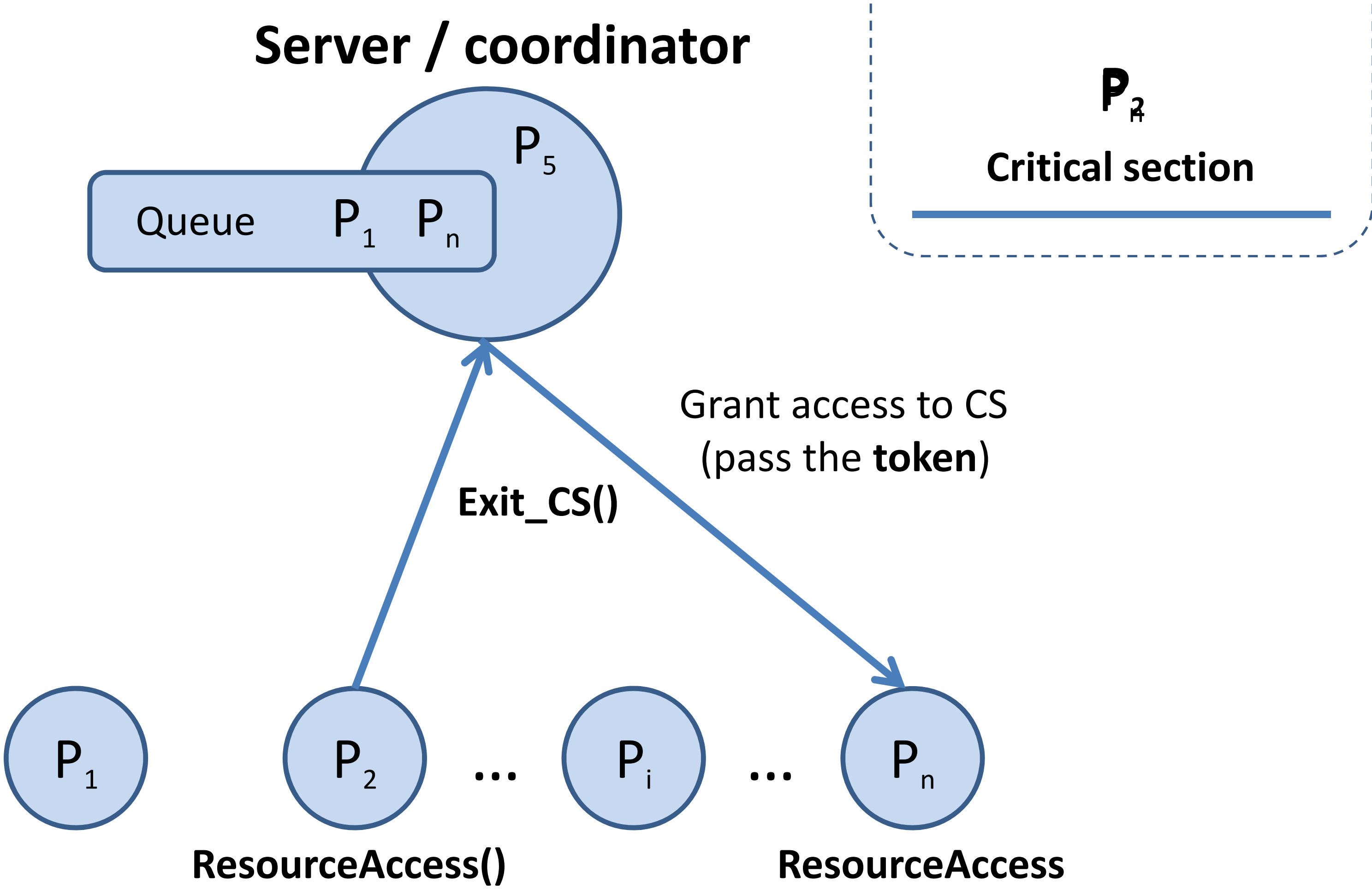
Centralized strategy: Empty CS



Centralized strategy: Non-empty CS



Centralized strategy: Exit CS



Centralized strategy analysis I

- Meets requirements: Safety, liveness, no starvation
- ***Does solution meet the ordering requirement?***
- Advantages
  - **Simple to implement**
- Disadvantages
  - **Single point of failure**
  - Bottleneck, network congestion, timeout
- Deadlock potential for multiple resources with separate servers

Centralized strategy analysis II

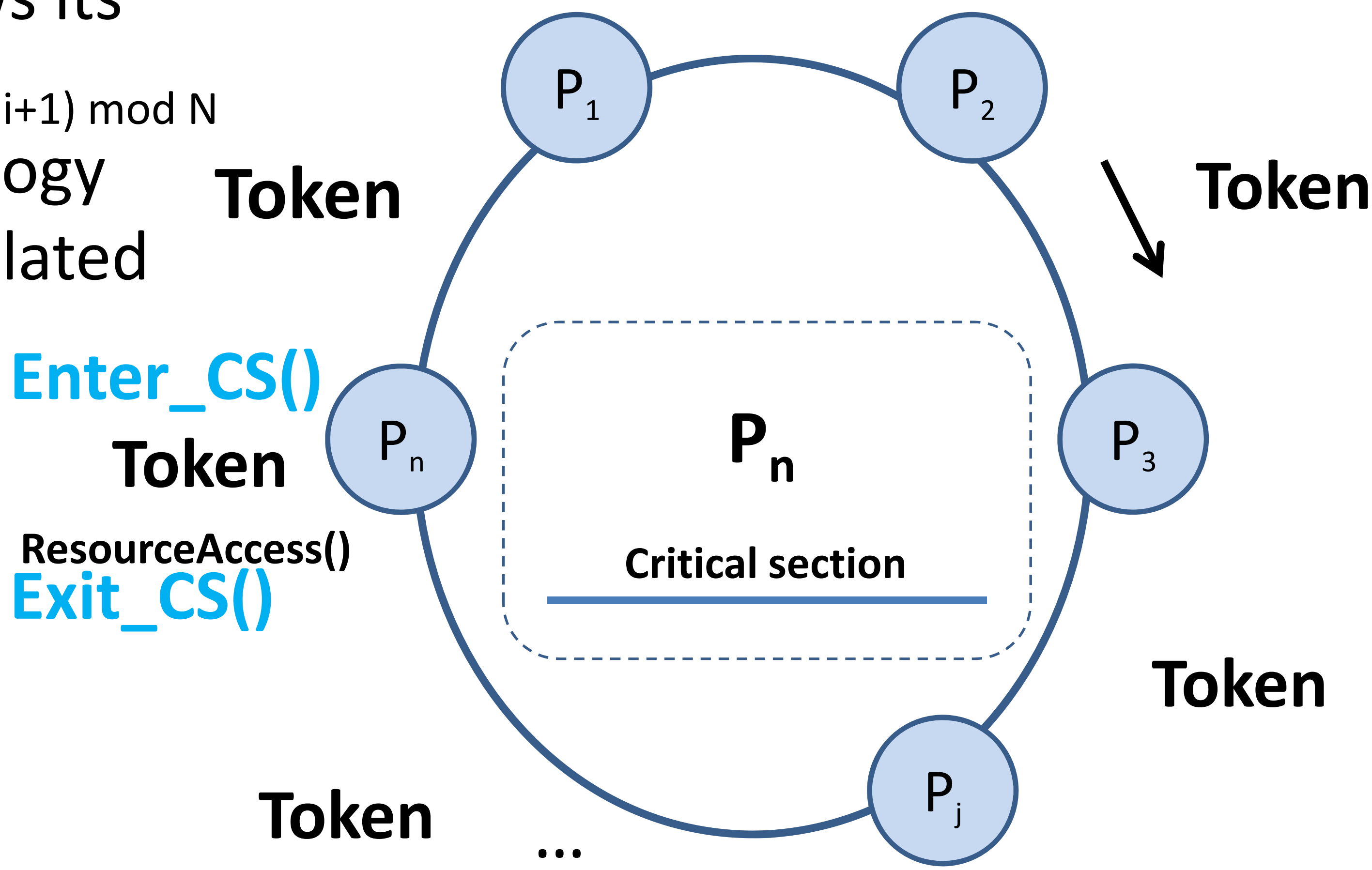
- Enter_CS()
  - **Two messages**: Request & Grant
  - One round of communication (RTT delay)
- Exit _CS()
  - **One message**: Release message
  - No delay for the process in CS

Distributed strategy

- In our distributed strategies, **the same decision made on each node**, independent of the other nodes in the system.
- Selected algorithms
  - Ring-based algorithm
  - **Logical clocks-based algorithm (Lamport, 1976)**
  - Ricart & Agrawala, 1981
  - Maekawa, 1985
  - Many more

Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



Ring-based algorithm analysis

- **Safe**: Node enters CS only if it holds token
- **Live**: Since finite work is done by each node (can't re-enter), token eventually gets to each node
- **Fair**: Ordering is based on ring topology, no starvation (pass token between accesses)
- **Performance**
  - **Constantly consumes network bandwidth**, even when no processes seek entry, except when inside CS
  - **Synchronization delay**: Between 1 and *N* messages
  - **Client delay**: 0 to *N* messages for entry; 0 for exit

Potential problems with ring-based algorithm
(Due to our assumption, do not apply here.)

- Node crash
- Lost token
- Duplicate token
- Timeouts on token passing

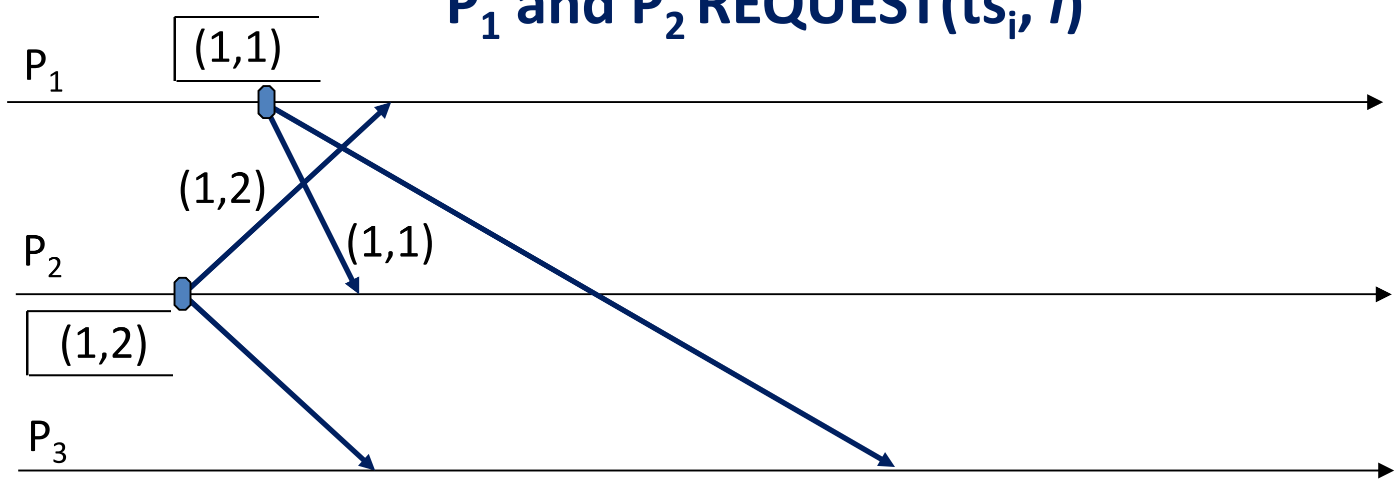
Lamport's algorithm

- System of *n* processes
- (ts_i, i) – logical clock timestamp of process $\mathbf{P_i}$
- Non-token based approach uses logical timestamps to order requests for CS
- Smaller timestamps have priority over larger ones
- Request queue maintained at each process ordered by timestamp (**a priority queue**!)
- Assume message delivered in FIFO order

Lamport's algorithm

- **P_i requesting** CS
  - Broadcast **REQUEST(ts_i, *i*)** message to all processes
    - Place request in ***request_queue$_i$***
    - (ts_i, *i*) denotes timestamp of request

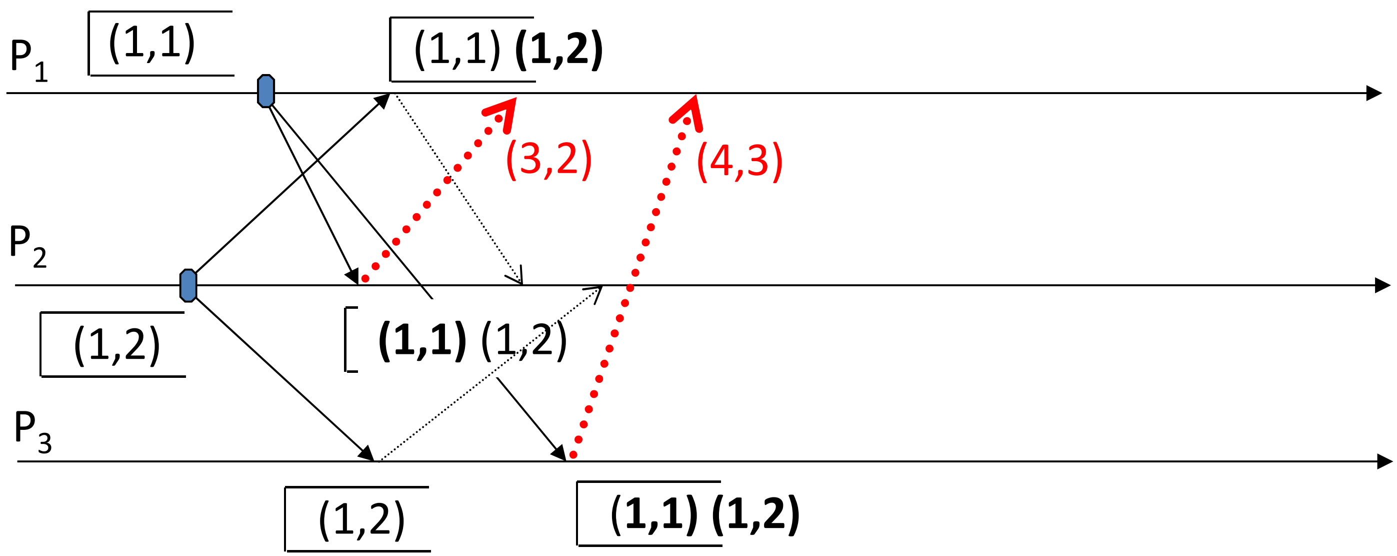
P_1 and P_2 REQUEST(ts_i, *i*)



Lamport's algorithm

- **P_k** receiving a request to enter CS
  - When **P_k** receives a REQUEST(ts_i, *i*) message from **P_i**,
    - It places **P_i**'s request into ***$request\_queue_k$***
    - Sends a timestamped REPLY message to **P_i**

REPLY(ts_k, *k*) – only replies to P_i's request are shown

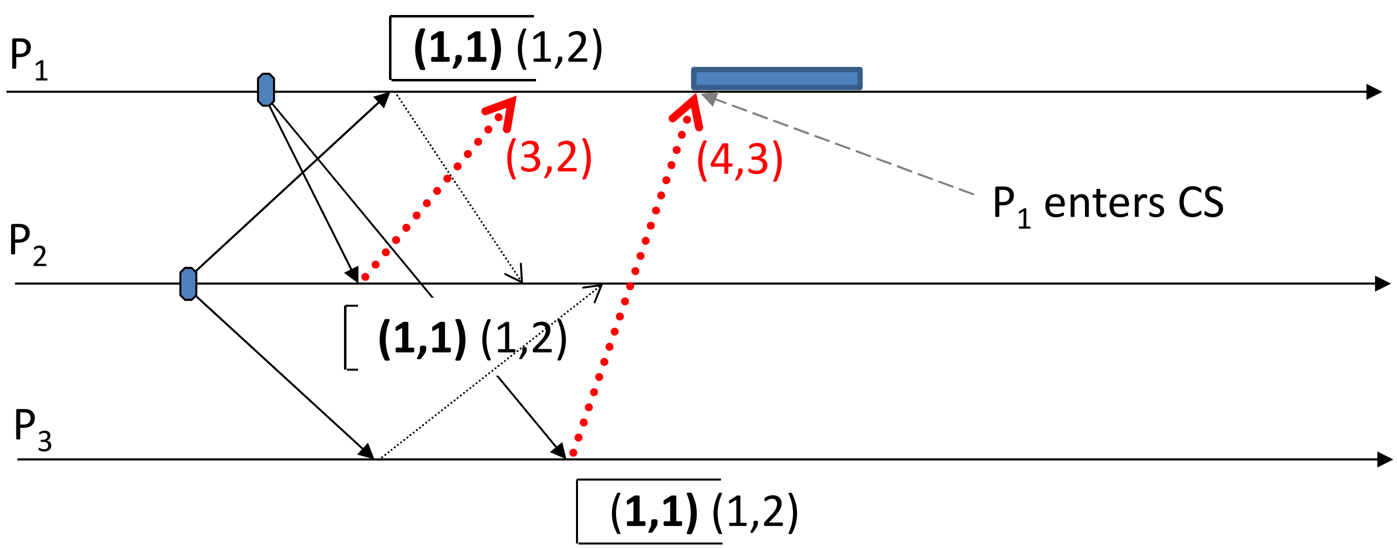


Lamport's algorithm

P_i requesting CS – P_i enters when following conditions hold

1. P_i has received a message **with timestamp larger than (ts_i, *i*) from every other processes**
2. P_i's request is at top of ***request_queue$_i$***

P_1 sent REQ(1, 1) and received REPLY(3,2) and REPLY(4,3)

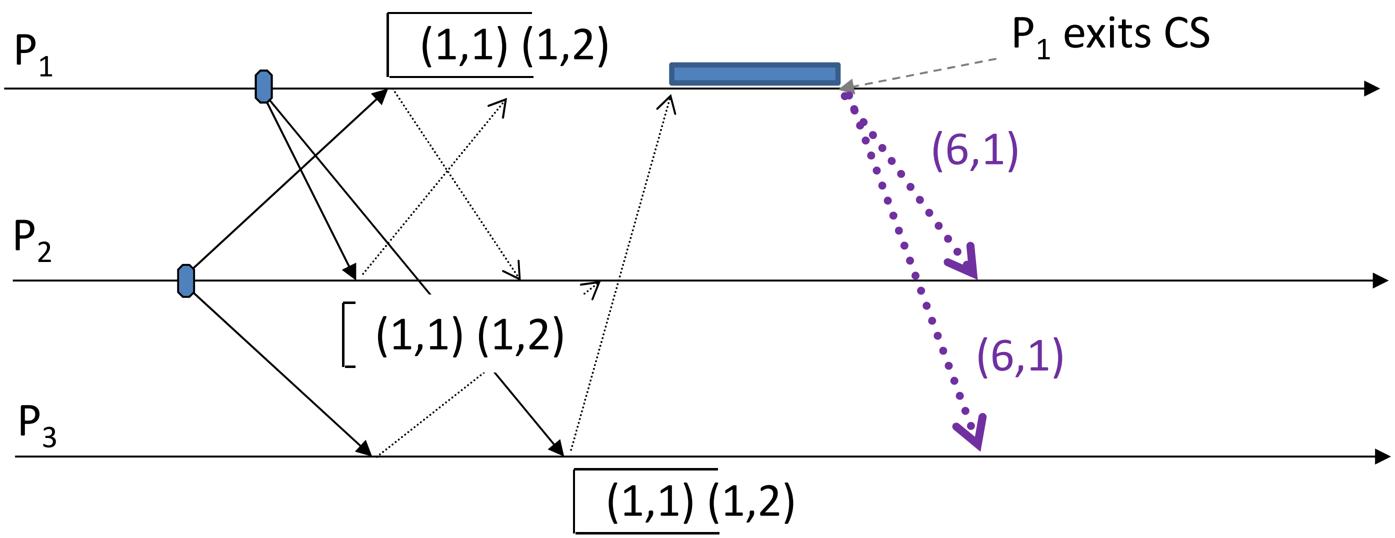


Lamport's algorithm

P_i releasing CS

- Removes request from top of the request queue
- Broadcasts a timestamped **RELEASE** message to all processes

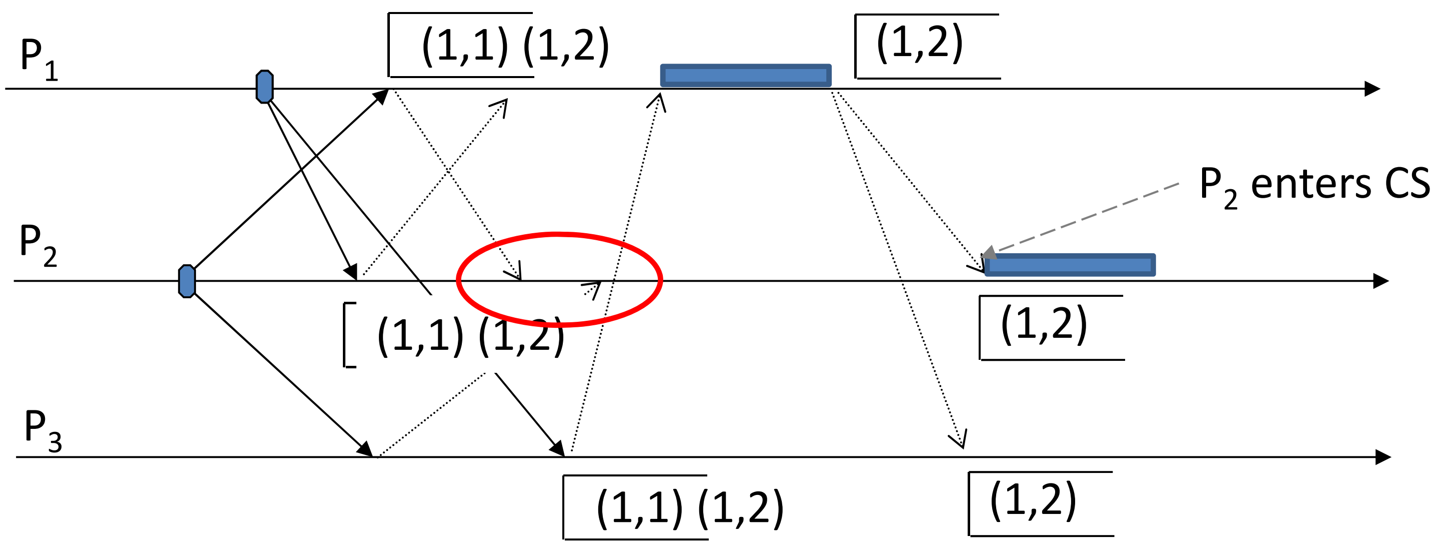
P_1 - RELEASE(ts_i, *i*)



Lamport's algorithm

P_k receiving a release message

- When **P_k** receives a **RELEASE** message from **P_i**, **P_k** removes **P_i**'s request from its request queue



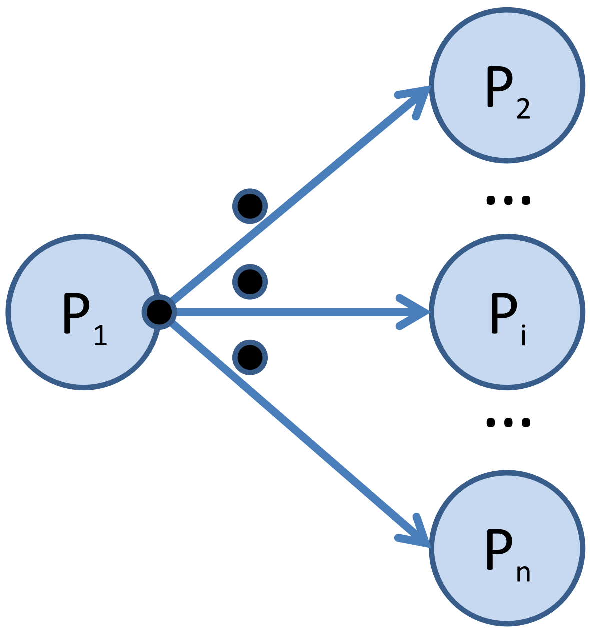
Performance

- 3(N-1) messages per CS invocation
  - (N - 1) REQUEST
  - (N - 1) REPLY
  - (N - 1) RELEASE messages

Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among *n* processes)

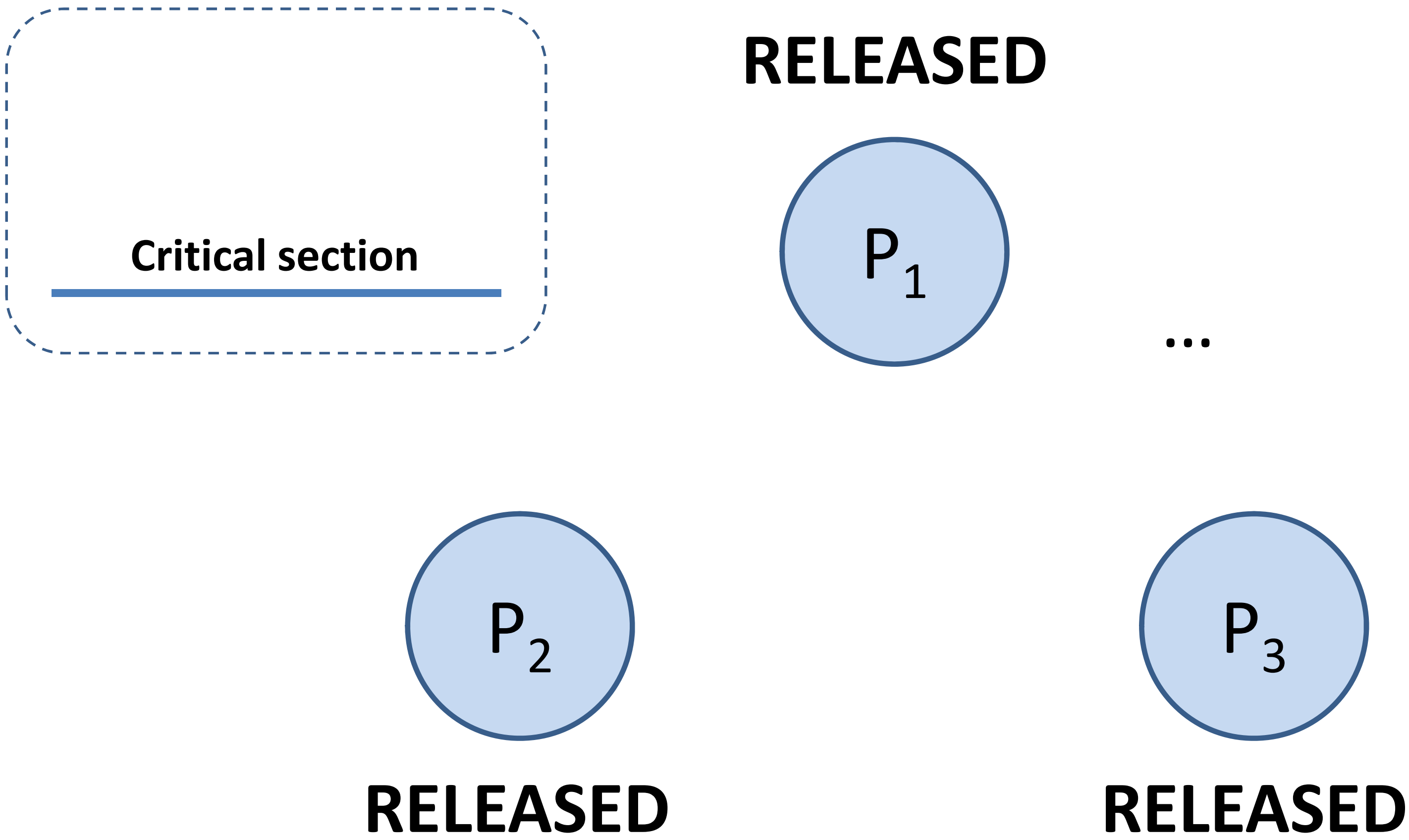
- **Basic idea**
  - Processes wanting to enter CS, **broadcast a request to all processes**
  - Enter CS, once **all** have **granted request**
- Use **Lamport timestamps** to order requests: <T, P_i>, T the timestamp, P_i the process identifier



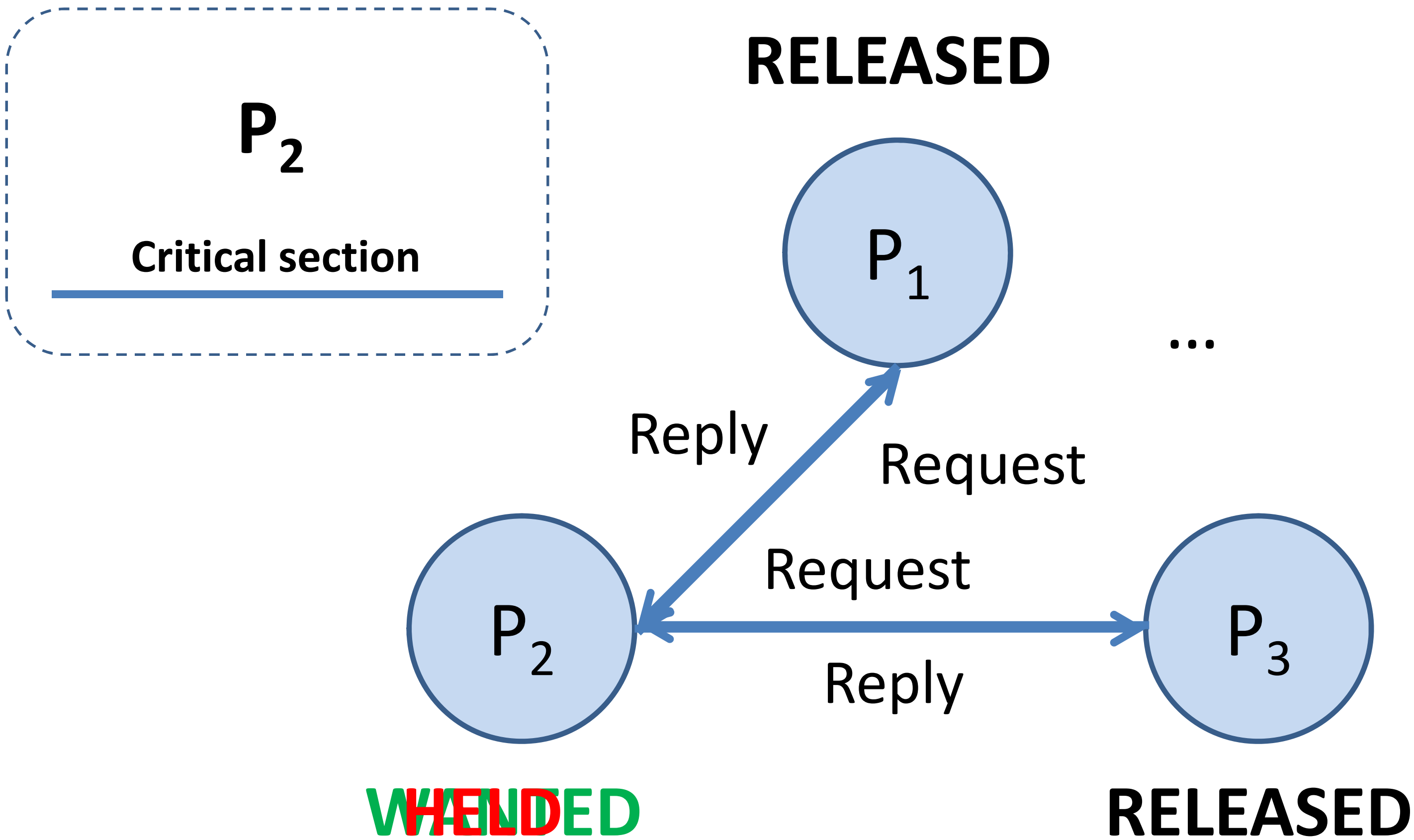
Ricart & Agrawala: Distributed strategy

- Each process is in one of three states
  - **Released** - Outside the CS, Exit_CS()
  - **Wanted** - Wanting to enter CS, Enter_CS()
  - **Held** - Inside CS, RessourceAccess()
- If a process requests to enter CS and **all** other **processes** are **in the *Released state***, **entry** is **granted** by each process
- If a process, $P_{i,}$ requests to enter CS and another process, $P_{k,}$ is inside the CS (***Held state***), then P_k will not reply, until it is finished with the CS

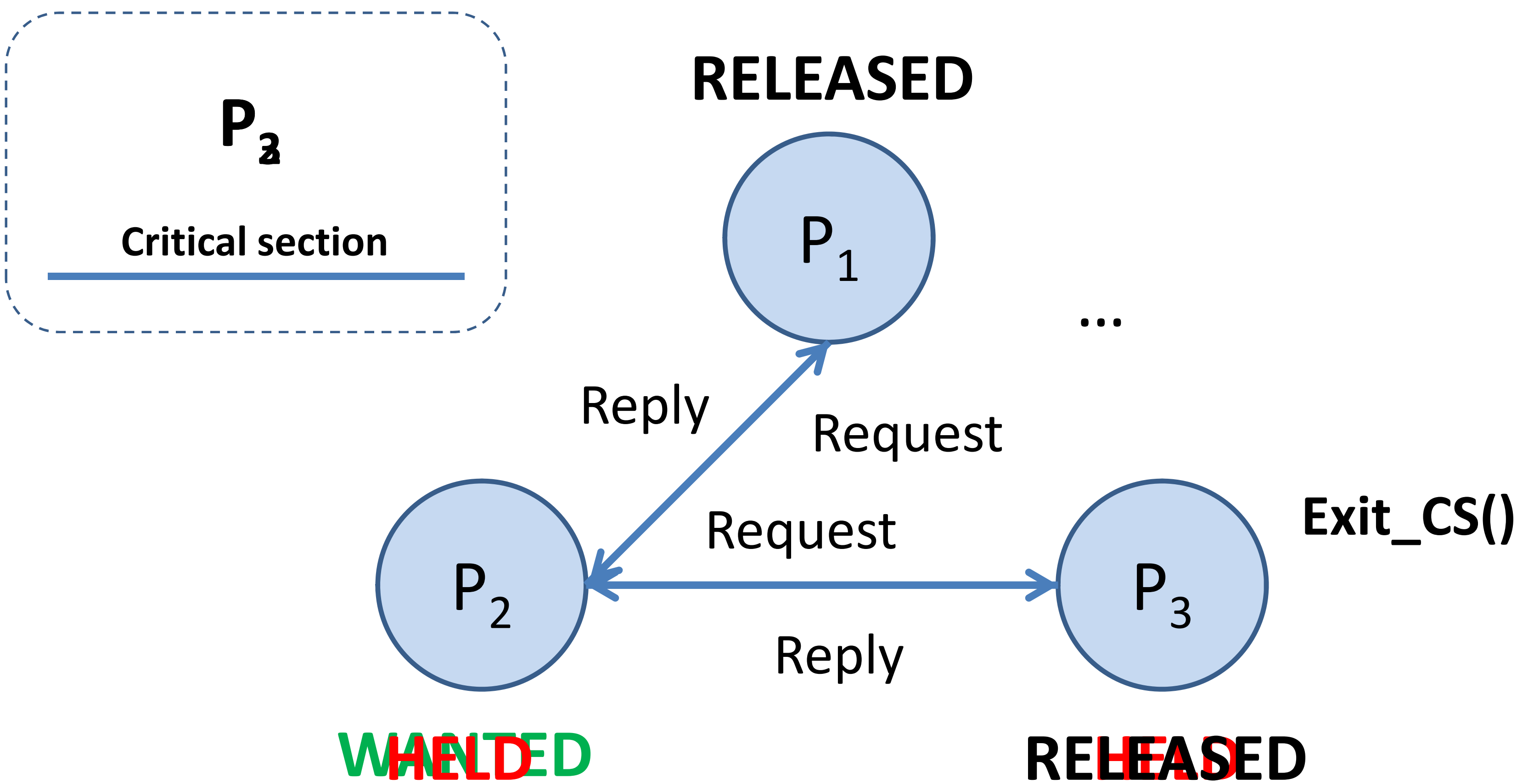
Initialization



Requesting entry to CS: *All Released*

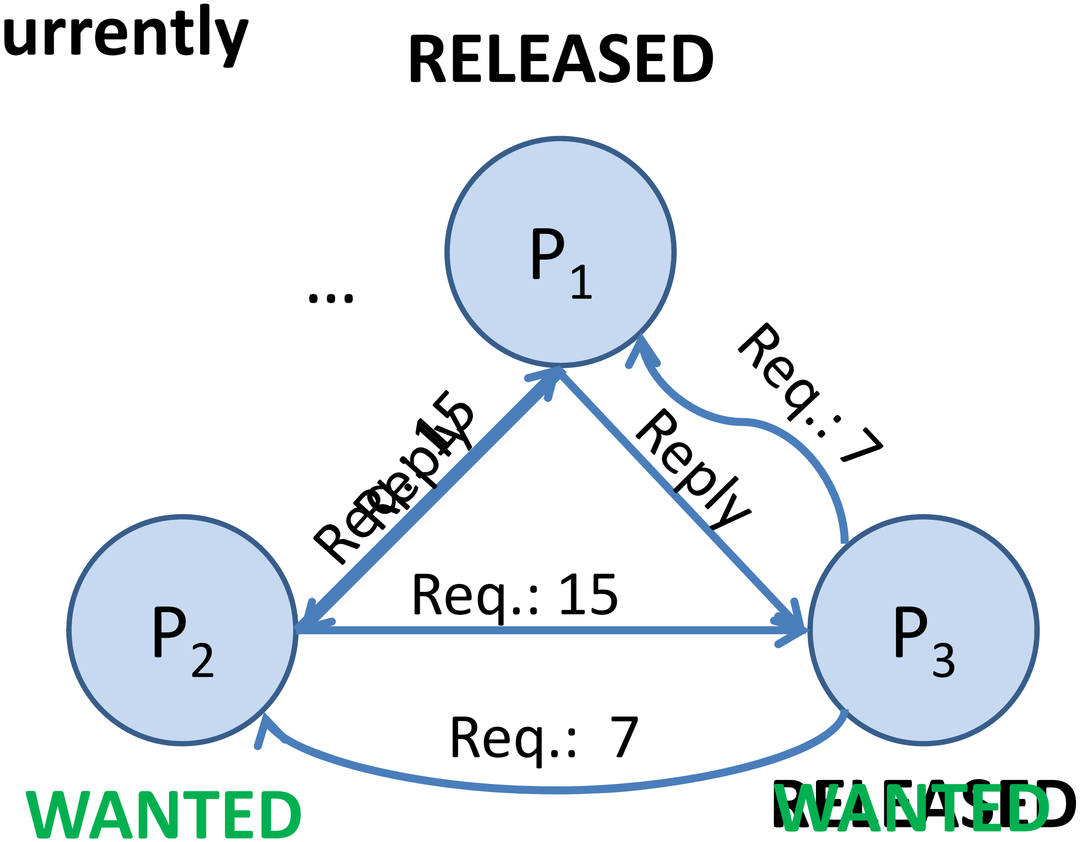


Requesting entry to CS:
Request while Held



Concurrent entry requests to CS

P_2 and P_3 request entry to CS concurrently



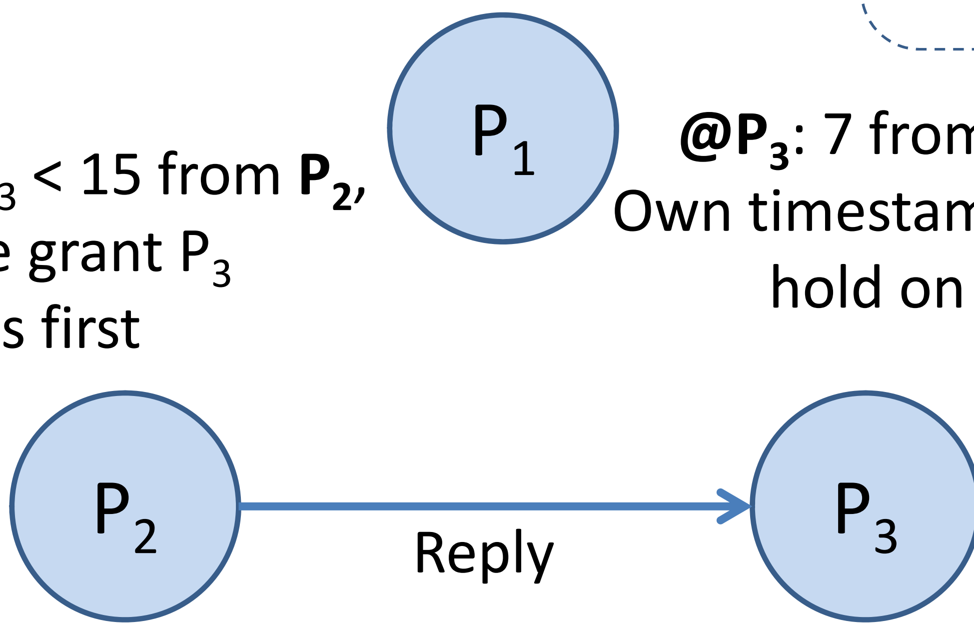
Concurrent entry requests to CS

P_2 and P_3 request entry to CS concurrently

P_3

Critical section

RELEASED



@P_2: 7 from P_3 < 15 from **P_2**,
therefore grant P_3
access first

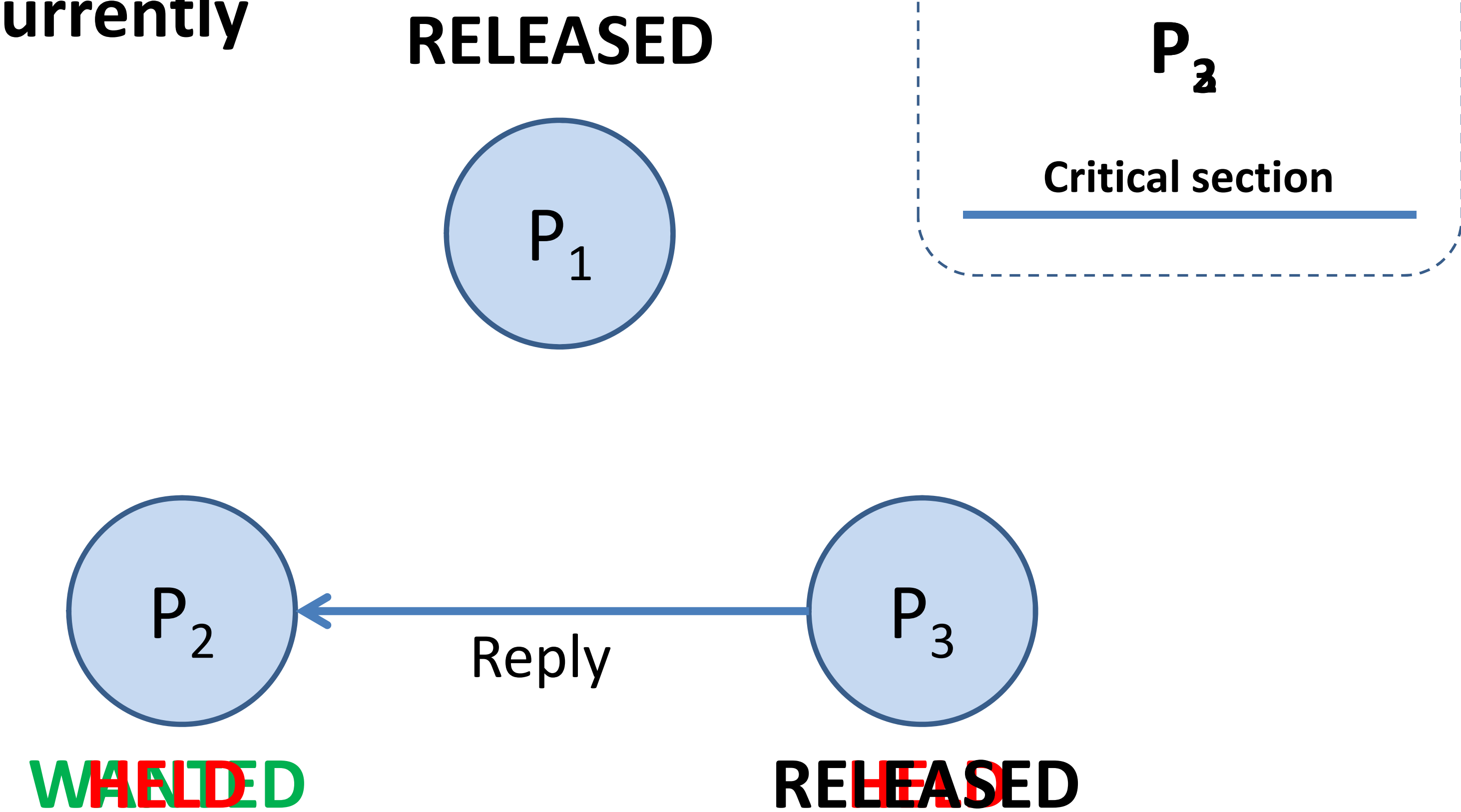
@P_3: 7 from **P_3** < 15 from P_2,
Own timestamp lower, therefore,
hold on to reply to P_2

WANTED

WANTED **HELD**

Concurrent entry requests to CS

P_2 and P_3 request entry to CS concurrently



Reminder: Subtlety about timestamps

- Use **Lamport timestamps** to order requests: <T, P_i>, T the timestamp, P_i the process identifier
- If for two timestamps
  - <T, P> and <S, Q>, T=S, then break ties by looking at process identifiers
  - Gives rise to a total order over timestamps

Ricart & Agrawala algorithm

Enter_CS()

state = WANTED

Multicast **request** to all processes

T = request`s timestamp

wait until ((n-1) acks received)

state = HELD

On initialization

state = RELEASED

Exit_CS()

state = RELEASED

Reply to all queued requests

j's timestamp (its own)

On receiving a request with < T_i, P_i> at P_j(i ≠ j)

if (state==HELD or (state==WANTED and (T, P_j) < (T_i, P_i)))

queue request from P_i without replying

else

send a reply to P_i

j

Potential fault-tolerance issues
(Our assumptions bracketed them out.)

- None of the algorithms tolerate message loss
- Ring-based algorithm cannot tolerate crash of single process
- Centralized algorithm can tolerate crash of processes that are neither in the CS, nor have requested entry
- Lamport, R&A can not tolerate faults

LEADER ELECTION

Leader election

- **Problem**: A group of processes, P_1, …, P_n, **must agree** on some **unique P_k** to be the "**leader**"
- Often, the leader then **coordinates another activity**
- Election runs when leader (a.k.a., coordinator) failed
- Any process who hasn't heard from the leader in some predefined time interval **may call for an election**
- **False alarm** is a possibility (new elections initiated, while current leader still alive)
- **Several processes** may initiate **elections concurrently**
- Algorithm should allow for process crash during election

Applications of leader election

- Berkeley clock synchronization algorithm
- Centralized mutual exclusion algorithm
- Leader election for choosing the master in Hbase / Bigtable (using ZooKeeper/Chubby)
- Choosing the master among the 5 servers of Chubby or ZooKeeper cell
- *Primary-backup replication algorithms*
- *Two-phase commit protocol*

As compared to mutual exclusion

- Losers return to what they were doing ...

  ... **instead of waiting**
- **Fast election** is important ...

  ... not starvation avoidance
- **All processes** must know result ...

  ... not just the winner

ME can be reduced to LE!
(e.g., Hbase wants LE,
ZooKeeper provides ME)

Process identifier

- Elected **leader** must be **unique**
- Active process with **largest identifier** wins
- Identifier could be any “useful value”
  - I.e., **unique** & **totally ordered value**
- E.g., based on OS process identifiers, IP adr., port
- E.g., based on least computational load
  - <1/load, i>, load > 0, i is process ID to break ties
- Each process, $\mathbf{P_i}$, has a variable $\mathbf{elected_i}$ that holds the value of the leader or is “$\perp$” (undefined)

Election algorithm requirement

- **Safety**
  - A participating process, $\mathbf{P_i}$, has variable $\mathbf{elected_i}$ **=** **"$\perp$"** or $\mathbf{elected_i = P}$**,** where P is chosen as the non-crashed process at the end of the election run with the **largest identifier.**
  - **Only one leader at a time!**
- **Liveness**
  - **All processes participate** in the election and **eventually** either set $\mathbf{elected_i \neq}$ **"$\perp$"** or **crash**.

Chang & Roberts Ring-based algorithm, 1978

Essentially three phases

1. **Initialization**
2. **Election phase** (concurrent calls for election)
   - Determine election winner (voting phase)
   - Reach point of message originator
3. Announce leader phase (**victory announcement phase**)

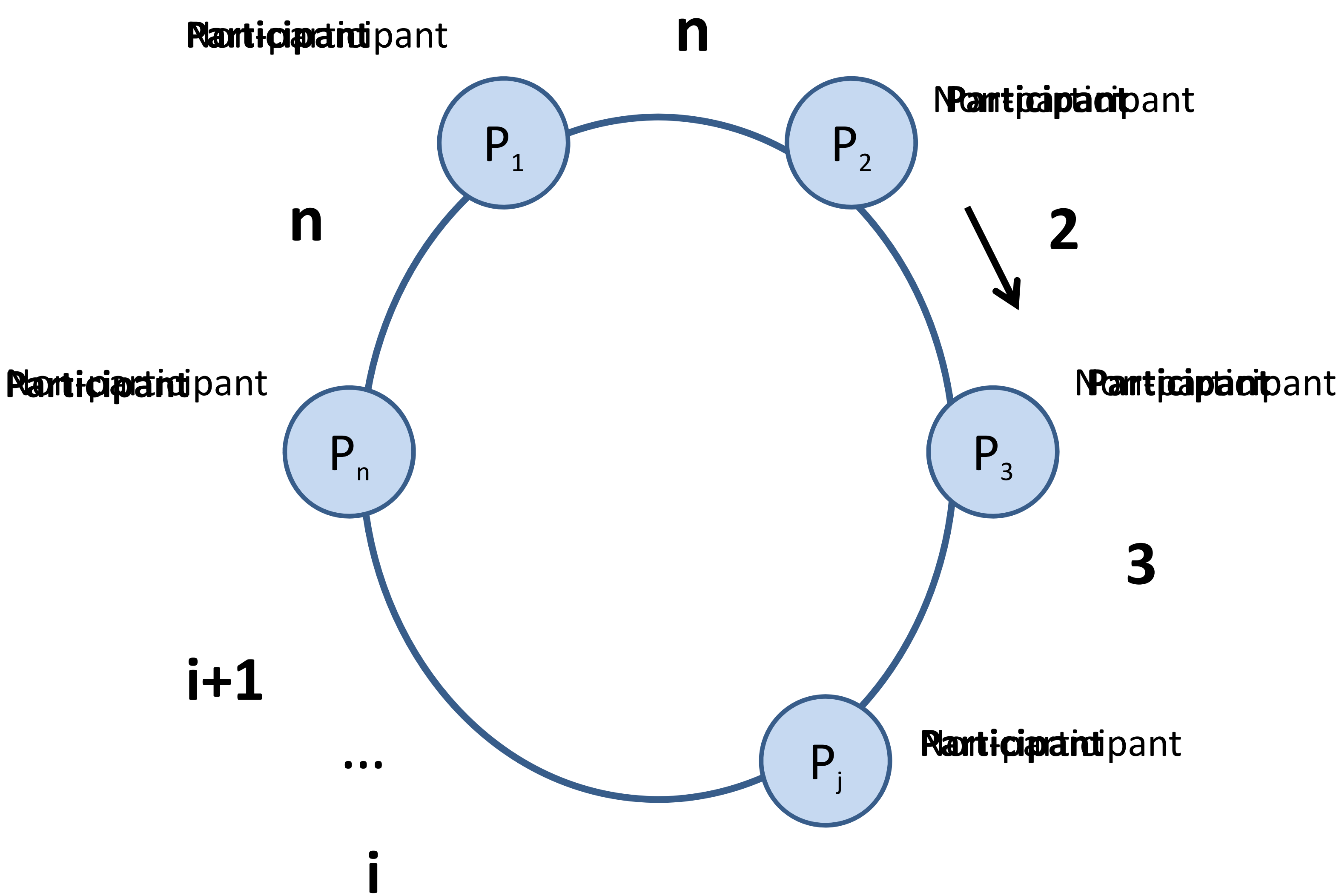
Ring-based election algorithm I

- Construct a ring (cf. ring-based mutual exclusion)
- Assume, each P has **unique ID (e.g., unique integer)**
- Assume, no failures and asynchronous system, **but failures can happen before the election!**
- Any P_i may begin an election by sending an **election message** to its successor (i.e., after detecting leader failure)
- Election message holds P_i's ID

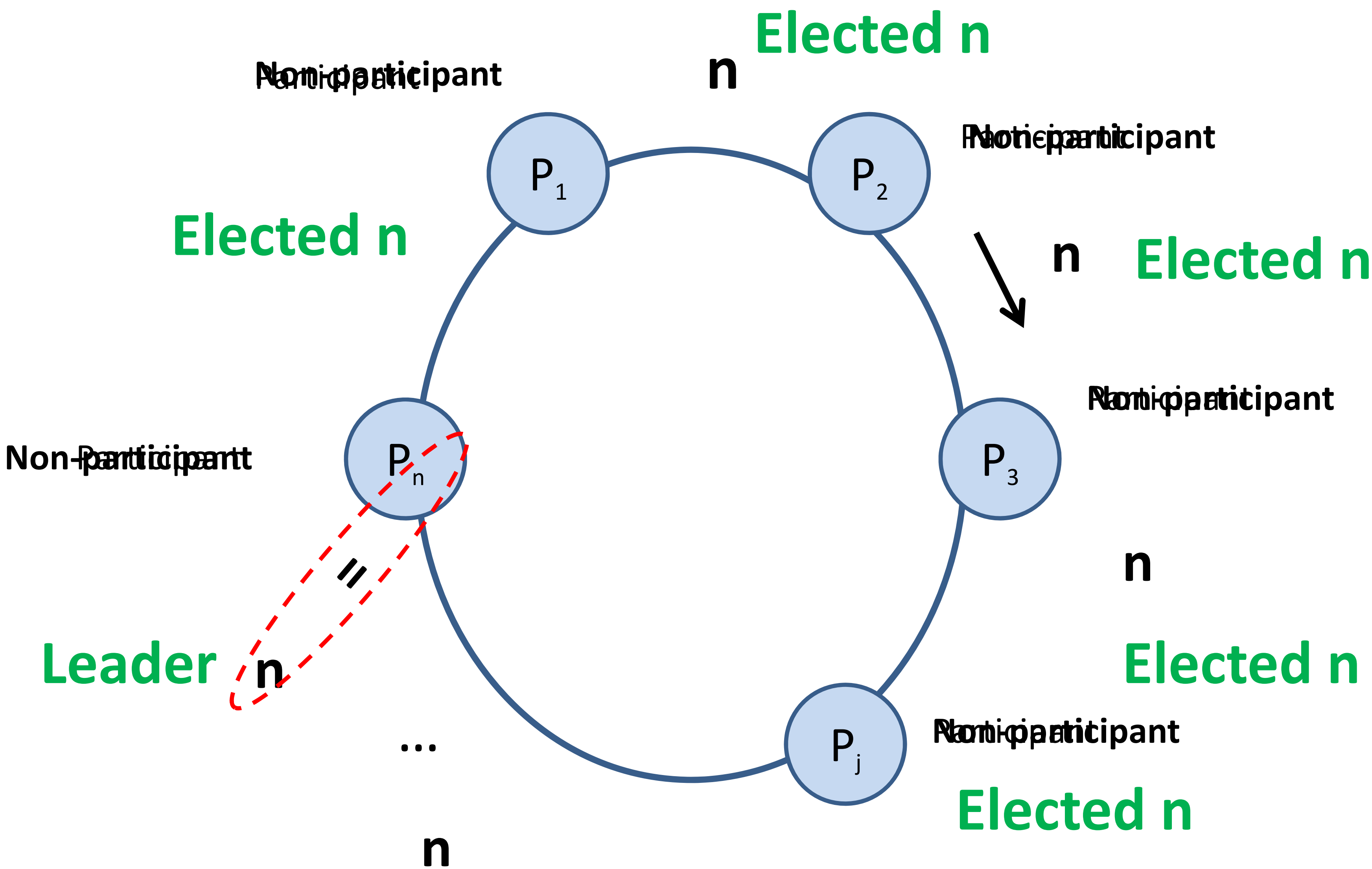
Ring-based election algorithm II

- Election message holds P_i's ID
- Upon receiving an election message, P_k **compares its own ID to ID it received in election message**
  - If received message ID is **greater**: **Forward** election message
  - If … **smaller**: **Forward** election message **with P_k's ID,** unless P_k has already sent a message, i.e., has participated in current election run
  - If … **equal**: **P_k is now leader**. Forward victory message to notify all other processes

Ring-based algorithm:
Calling an Election (determine winner)



Ring-based algorithm:
Calling an Election (origin & victory)



Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$ → **Participant**
Forward S_{ID}

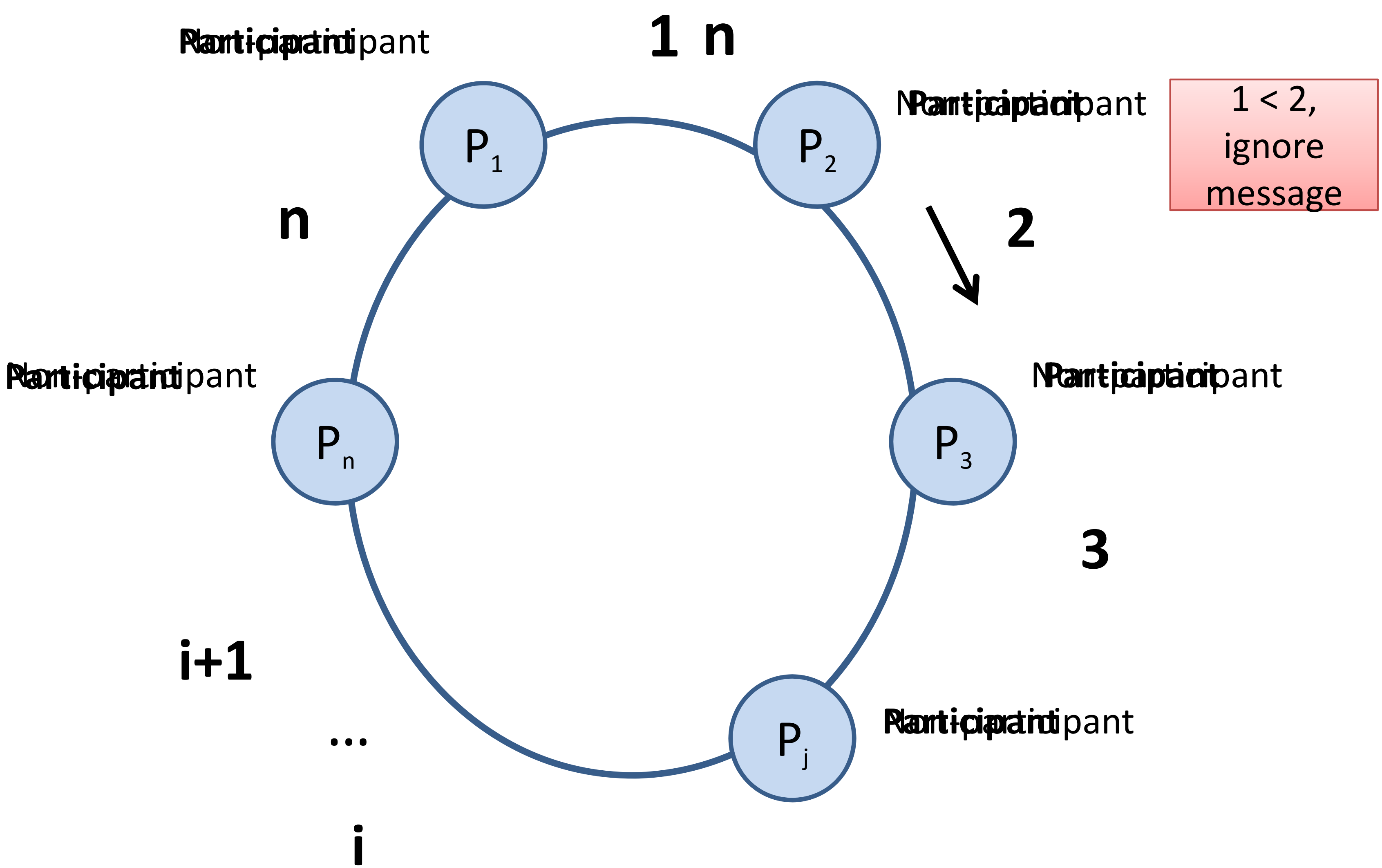
If $\mathbf{S_{ID} = R_{ID}}$, it follows **R** elected as leader

$S_{ID} > R_{ID}$ → **Non-participant**
Forward S_{ID}

$S_{ID} < R_{ID}$ → **Participant**
No forwarding (own ID already sent)

$S_{ID} < R_{ID}$ → **Non-participant**
Forward R_{ID}

Ring-based algorithm:
Concurrent election start



Ring-based election algorithm analysis

- Worst case: 3N -1 messages
  - *N-1* messages to reach highest ID from lowest ID
  - *N* messages to reach point of origin
  - Leader announcement takes another *N* messages
- Safety: even with multiple processes starting election
- Liveness: guaranteed progress *if no failures during the election occur*

Bully algorithm, 1982

- Assumes each process has a unique ID, reliable message delivery, and synchronous system
- Assumes processes know each others' IDs and can communicate with one another
  – Higher IDs have priority
  – Can "bully" lower numbered processes
- Initiated by any process that **suspects leader failure**
- Tolerates processes crashing during elections

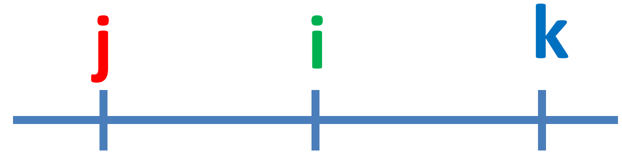
Bully algorithm messages

- Operates with three types of **messages**
  - ***Election*** announces an election
  - ***Answer*** responds to an election message
  - ***Coordination*** announces identity of leader

- Algorithm is triggered by any process that detects leader to have crashed (synchronous system)

@ P_i

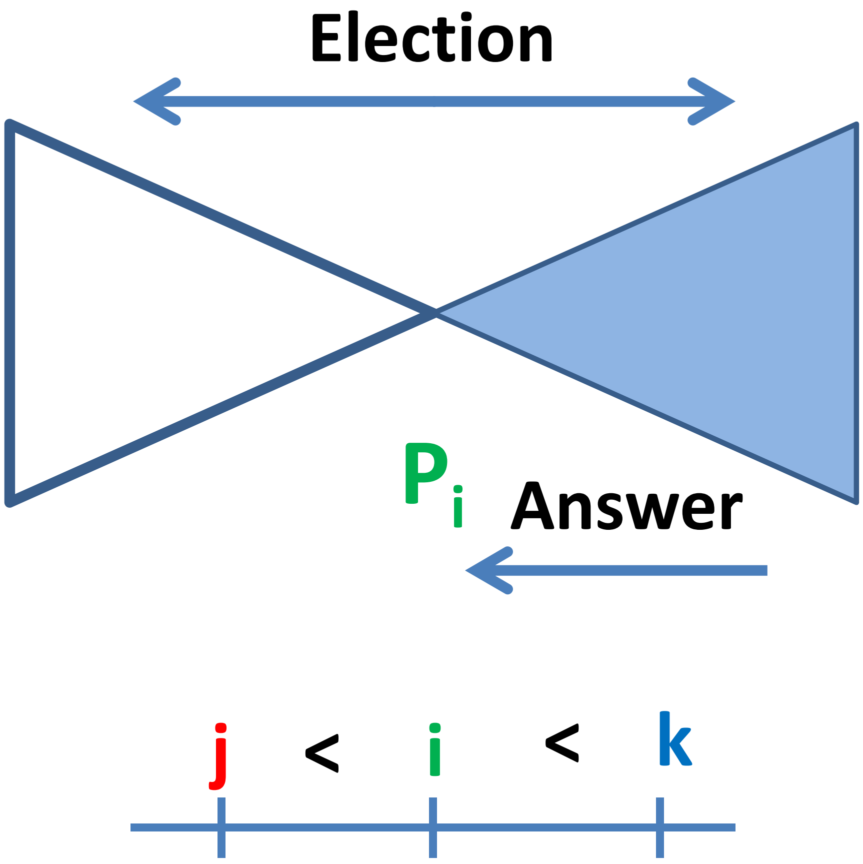
P_i detects failure of leader



For any $j < i$ and any $i < k$

1. Broadcasts **election message**
2. Any P_k receiving election message **responds with answer message** and starts another election
3. Any P_j receiving election message **does not respond**
4. If P_i **does not receive any answer message (timeout)** then it **broadcasts victory** via **coordination message**
5. If P_i **does receive answer message(s)** then waits to receive **coordination message**
6. Restarts election, if no coordination message received (failure happened)

Election & answer



Upon crash of a process

- Suppose process eventually recovers (*no problem if it stays down, why?*)
- Process may determine that it has the highest identifier, thus, pronounce itself as leader
  - Even though system may have an existing leader (elected during crash)
- New process “**bullies**” current leader