Ricart-Agrawala Algorithm

- Optimization of Lamport's algorithm:

Lamport's Algorithm

Requesting the CS

- P_i sends message **REQUEST(t_i, i)** + enqueues the request in *request_queue*$_i$
- when P_i receives a request from P_j, it enqueues it and returns a **REPLY** to P_i

P_i executes the CS only when:

- has received a msg with timestamp > t_i from everybody
- its own request is the first in the *request_queue*$_i$

Releasing the CS:

- when done, a process remove its request from the queue + sends a timestamped **RELEASE** msg. to everybody else
- upon receiving a RELEASE message from P_i, a process removes P_i's request from its request queue

Ricart-Agarwala Algorithm

Requesting the CS

- P_i sends message **REQUEST(t_i, i)**
- when P_i receives a request from P_j, it returns a **REPLY** to P_i if it is not requesting or executing the CS, or if it made a request but with a larger timestamp. Otherwise, the request is **deferred.**

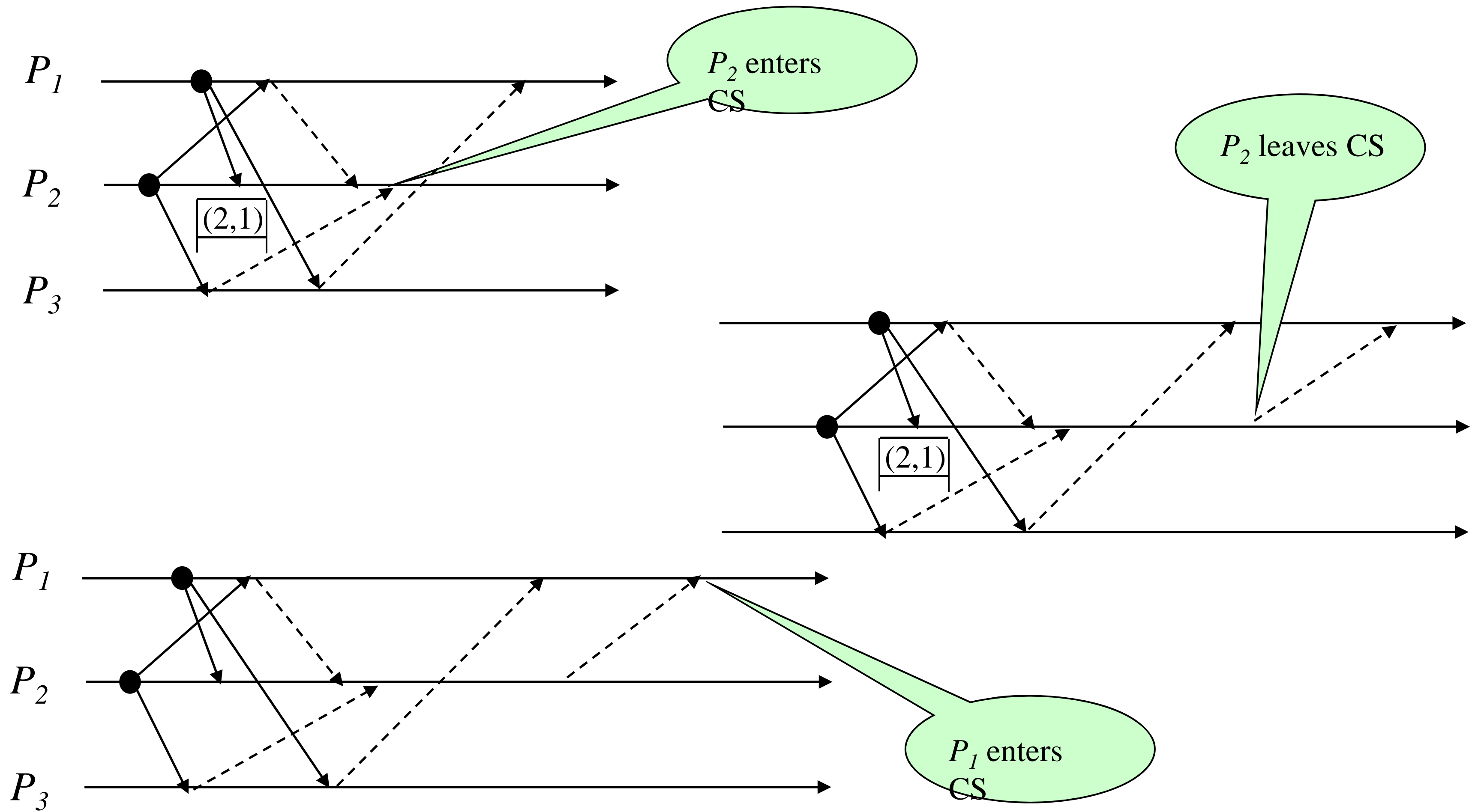
P_i executes the CS only when:

- has received a **REPLY** from everybody

Releasing the CS:

- when done, a process sends a **REPLY** to all deferred requests

Ricart-Agrawala Algorithm Example



Ricart-Agrawala: proof of correctness

- Assumption: Lamport's clock is used
- Proof by contradiction:
  - assume P_i and P_j are executing the CS at the same time
  - assume request timestamp of is P_i smaller than that of P_j
  - this means P_i issued its own request first and then received P_j's request, otherwise P_j request timestamp would be smaller
  - for P_i and P_j to execute the CS concurrently means P_i sent a REPLY to P_j before exiting the CS
  - Contradiction: a process is not allowed to send a REPLY if the timestamp of its request is smaller than the incoming one
- Therefore it cannot be that P_i and P_j are executing the CS at the same time!

Maekawa's Algorithm

- Difference with respect to previous algorithms:
  - a site does not request permission from every other site but only from a subset - called *request set*
- The requests sets of any two sites have at least one site in common:

$$\forall i\ \forall j : 1 \leq i,j \leq N :: R_i \cap R_j \neq \emptyset$$

- The basic idea is that each pair of sites is going to have a third site mediating conflicts between the pair

Maekawa's algorithm steps

- Requesting the CS
  - S_i sends a message **REQUEST(*i*)** to all the sites in R_i
  - when S_j receives a request from S_i, it returns a **REPLY** to S_i if it has not sent a **REPLY** since receiving the latest **RELEASE** message. Otherwise the request is enqueued.
- Executing the CS
  - A site S_i executes the CS only after receiving **REPLY** messages from all the sites in R_i
- Releasing the CS
  - When done, a site S_i sends a **RELEASE** message to all the sites in R_i
  - When a site receives a RELEASE message, it sends a REPLY message to the next site waiting in the queue and removes it

Construction of the request set

- The requests sets are constructed to satisfy the following conditions:
  - $\forall i \ \forall j : 1 \leq i,j \leq N :: R_i \cap R_j \neq \emptyset$
    - necessary for correctness
  - $\forall i \ : 1 \leq i \leq N :: S_i \in R_i$
    - necessary for correctness (note: this condition, like the need for FIFO comm., is really needed only in the extended version of the algorithm)
  - $\forall i \ : 1 \leq i \leq N :: / R_i / = K$
    - all R_i have equal size, so all sites do equal work to access the CS
  - Any site S_i is contained in K of the R_is
    - the same number of site is requesting permission from each site

More on the request set

- All the previous conditions are satisfied if N can be expressed as:

$$N = K(K - 1) + 1$$

  (examples: N = 3 and K = 2, N = 7 and K = 3, etc.)

  – Note that, for large N, K ≈ $\sqrt{N}$

- Otherwise one of the last two conditions must be relaxed

  – for example, $|R_i| = K$ not longer true for all i

Notes on Maekawa's algorithm

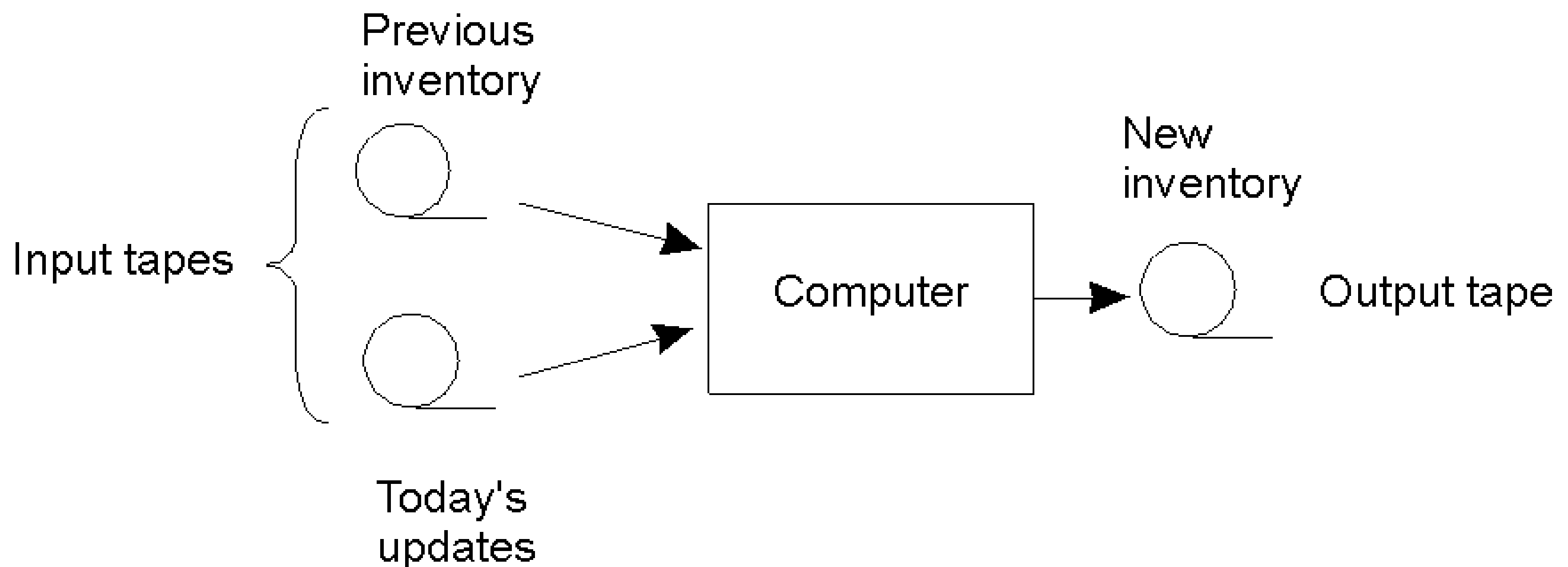
- Performance:
  - $3\sqrt{N}$ messages are needed for execution of the CS
  - synchronization delay is $2T$
- Problem: the algorithm is deadlock prone!
  - there is a variant of the algorithm that can prevent the deadlock by using a priority-based preempting scheme
  - this variant requires additional messages (up to $5\sqrt{N}$)

Database systems

- Database: a collection of shared data objects (d1, d2, … dn) that can be accessed by users
  - every database has some correctness constraints defined on it (called *consistency assertions* or *integrity constraint*)
  - a database is said to be *consistent* if the values of its data satisfy these constraints
- A user interacts with a database through complex operations called *transactions*
  - a transaction consists of a sequence of read, write, compute statements that refers to data objects in the database
  - examples: on-line booking, bank teller operations, ...

The Transaction Model (1)

- Old method of updating a master tape is fault tolerant.
  - Contrast with modern online database that is updated in place



The Transaction Model (2)

- The atomicity of transactions can be re-created with special primitives.

Primitive	**Description**
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

The Transaction Model (3)

- Example
  - a) Transaction to reserve three flights commits
  - b) Transaction aborts when third flight is unavailable

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

(b)