

# **TDDD25**

## **Distributed Systems**

# **Replication**

**Christoph Kessler**

IDA  
Linköping University  
Sweden

2024

# Agenda

## REPLICATION

- 1. Motivation and Requirements**
- 2. Architectural Model**
- 3. Request Ordering**
- 4. Implementing Total and Causal Ordering**
- 5. Update Protocols**

# Motivation

**Replication** is the maintenance of on-line copies of data (files).

- Each copy is located on a separate **replica manager** (server).
- Each copy is called a **replica**.

## Benefits of replication:

- Increased **availability** and **fault tolerance**:
  - The system remains operational and available to the users despite failures.
  - Alternate copies of a replicated data can be used when a primary copy is unavailable.
- **Performance** enhancement:
  - Data shared between a large number of clients should not be held at a single server; such a single server becomes a bottleneck.
  - Data should be replicated on several servers, each one providing service to a group of users close to the server.
  - Thus, network traffic is also reduced.

# Main Requirements with Replication

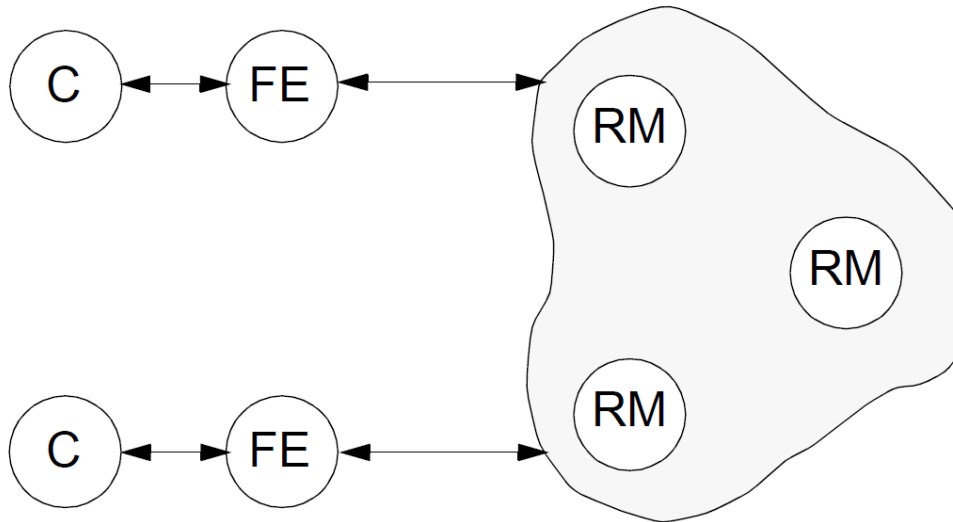
- **Replication transparency:**
  - The clients should not be aware that multiple physical copies of data exist.
- **Consistency:**
  - Consistency implies that any access from a client should be served with correct data (regardless of the replica manager it directly has access to)
  - What *correct* means, depends on the particular application:
    - ▶ In some situations, it is enough that all operations are *eventually* performed on all copies; it is acceptable that, at certain moments, different clients read *different* versions of the replicated data.
      - Question: *how* different?
    - ▶ Often, client access has to provide the **most recent** version of the data.

## Problems:

1. The **order** in which operations are performed on the different replicas.
2. Do we always need to **update** all replicas?  
If not, how can we guarantee that an access is always served with the latest version?
3. The effect of replication on **performance**:  
strong requirements on consistency can lead to significant overheads.

# Architectural Model

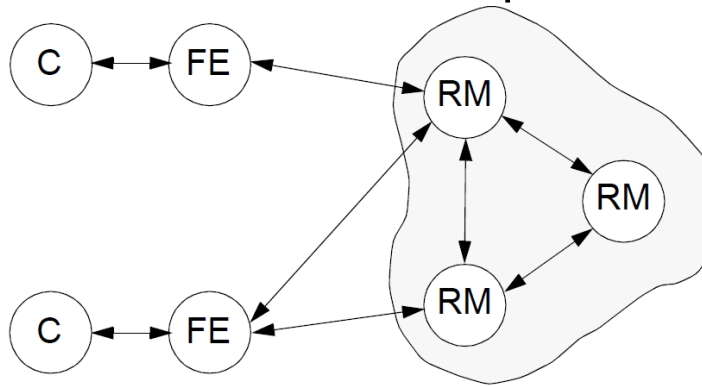
- **Client (C)**: makes a request (read or update)
- **Front-end (FE)**: proxy server, communicates with one or more replica managers (provides replication transparency)
- **Replica managers (RM)**: contain the replicas and perform operations on them



Different alternative models are possible, depending on the particular communication pattern between FEs and RMs, and between the different RMs.

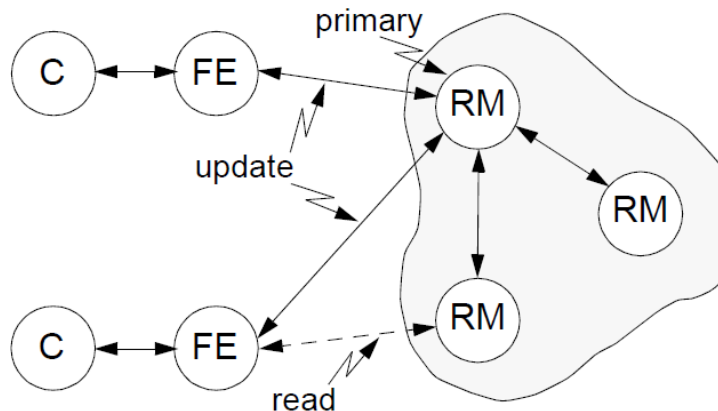
# Architectural Model

All RMs communicate with each other in order to agree on operations so that **coherence** is preserved between copies.



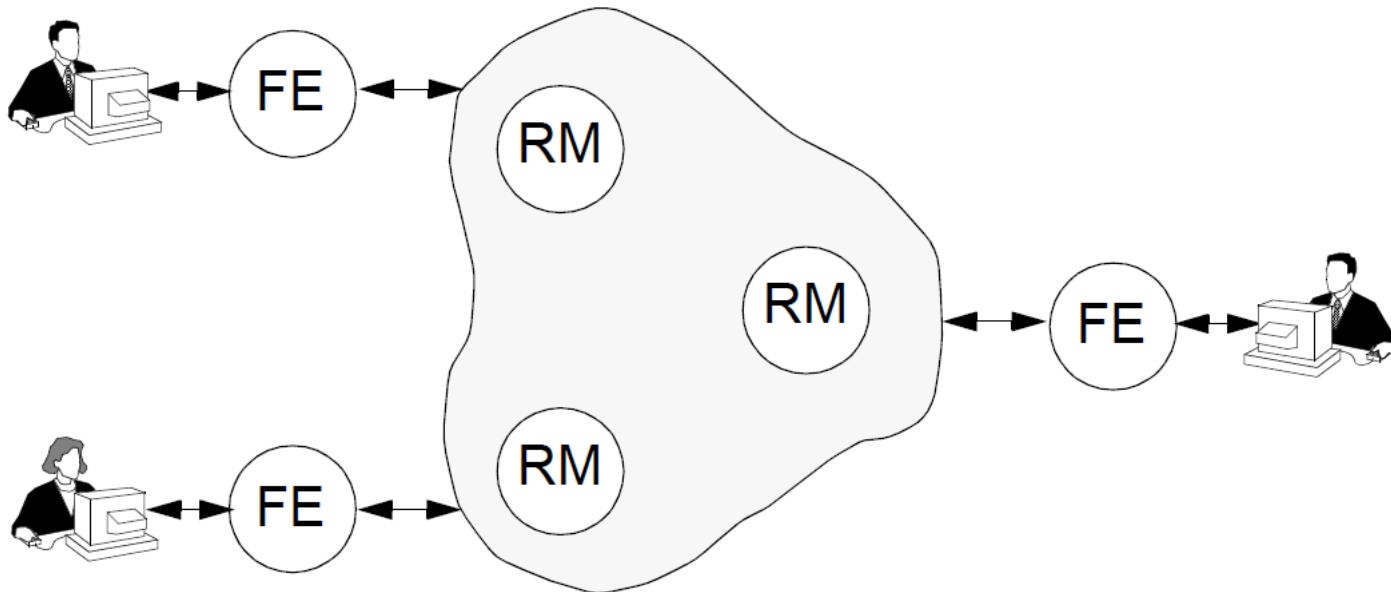
A "**primary**" RM coordinates the other RMs managing copies of the same data

- **Update requests** are directed from FEs to that primary RM, which propagates them to the other RMs.
- **Read requests** can be directed to any RM.



# Example: Bulletin Board System

- **Users** at different sites share a **bulletin board**.
- A **server** at each site hosts a **replica** of the board content.
- Each user can
  - **post** new items,
  - **select** a certain item to visualise, and
  - **respond** to a given message.



# Example: Bulletin Board System

- Items are displayed as available at a certain server, in the order in which they have been received.

Erik's view

Item	From	Subject
14	Johansson	weather
15	Ericsson	Java
16	Perkins	clocks
17	Johansson	Re:Java
18	Schmidt	Re: weather

Diana's view

Item	From	Subject
17	Perkins	clocks
18	Johansson	Re:Java
19	Pop	lab
20	Ericsson	Java
21	Schmidt	Re: weather
22	Larsson	bandy



# Request Ordering

Ordering of requests at the replica manager is sometimes essential in order to preserve consistency as required by the specific application.

- **Total ordering**

- If  $r_1$  and  $r_2$  are requests, then either  $r_1$  is processed before  $r_2$  at all replica managers or  $r_2$  is processed before  $r_1$  at all replica managers.

Erik's view

Item	From	Subject
17	Perkins	clocks
18	Ericsson	Java
19	Johansson	weather
20	Johansson	Re:Java
21	Schmidt	Re: weather
22	Larsson	bandy

Diana's view

Item	From	Subject
17	Perkins	clocks
18	Ericsson	Java
19	Johansson	weather
20	Johansson	Re:Java
21	Schmidt	Re: weather
22	Larsson	bandy

- In this case, users at different sites will see the items in identical order and can refer to them by their **number**.

# Request Ordering

## Causal ordering

- If two requests  $r_1$  and  $r_2$  are in a happened-before relation  $r_1 \rightarrow r_2$ , then  $r_1$  is processed before  $r_2$  at all replica managers.

Erik's view

Item	From	Subject
14	Johansson	weather
15	Ericsson	Java
16	Perkins	clocks
17	Johansson	Re:Java
18	Schmidt	Re: weather

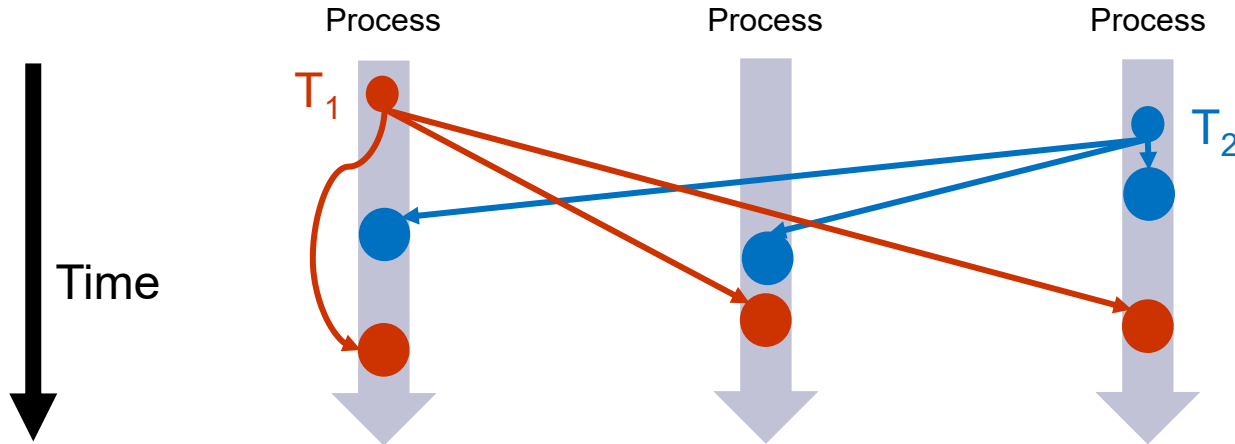
Diana's view

Item	From	Subject
17	Perkins	clocks
18	Larsson	bandy
19	Pop	lab
20	Ericsson	Java
21	Johansson	Re:Java

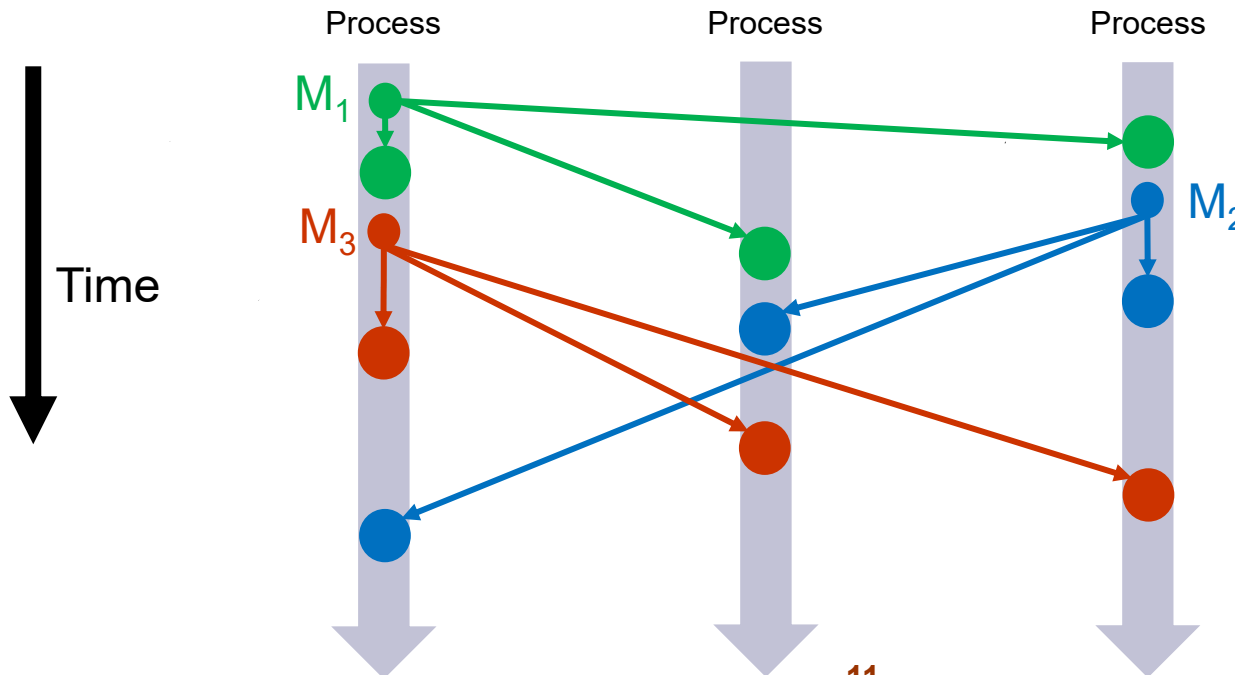
In this case, a user will never see an answer message before she has seen the initial message.

In general, total ordering does *not* necessarily imply causal ordering; it only means that all replica managers handle requests in the *same* (possibly, non-causal) order.

# Total vs. Causal Ordering of Multicast Messages / Requests



**Totally ordered**



**Causally ordered**  
for causally related  
messages  $M1 \rightarrow M2$ ,  
 $M1 \rightarrow M3$ ,  
but not totally ordered  
( $M2 \parallel M3$ )

# Implementing Total Ordering

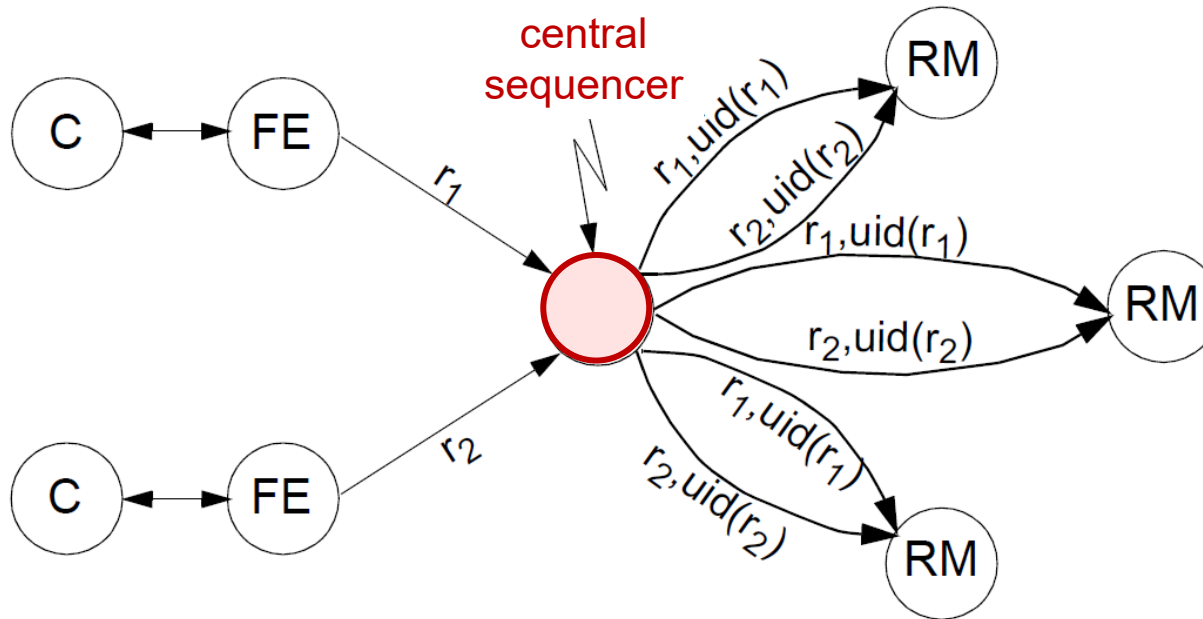
The basic idea:

- Assign **totally ordered identifiers**  $uid(r)$  to requests;
- Each replica manager makes the same ordering decision based on these identifiers.
- Notice: it is not sufficient the identifiers to be unique:
  - For a total ordering algorithm, it is needed that a site knows *when* to process a request  $r_1$  with unique identifier  $uid(r_1)$ , so that no other request  $r_2$  can arrive later so that  $uid(r_2) < uid(r_1)$ .

# Implementing Total Ordering

- Total ordering with **central sequencer**
- Total ordering based on **distributed agreement**

# Total Ordering with Central Sequencer



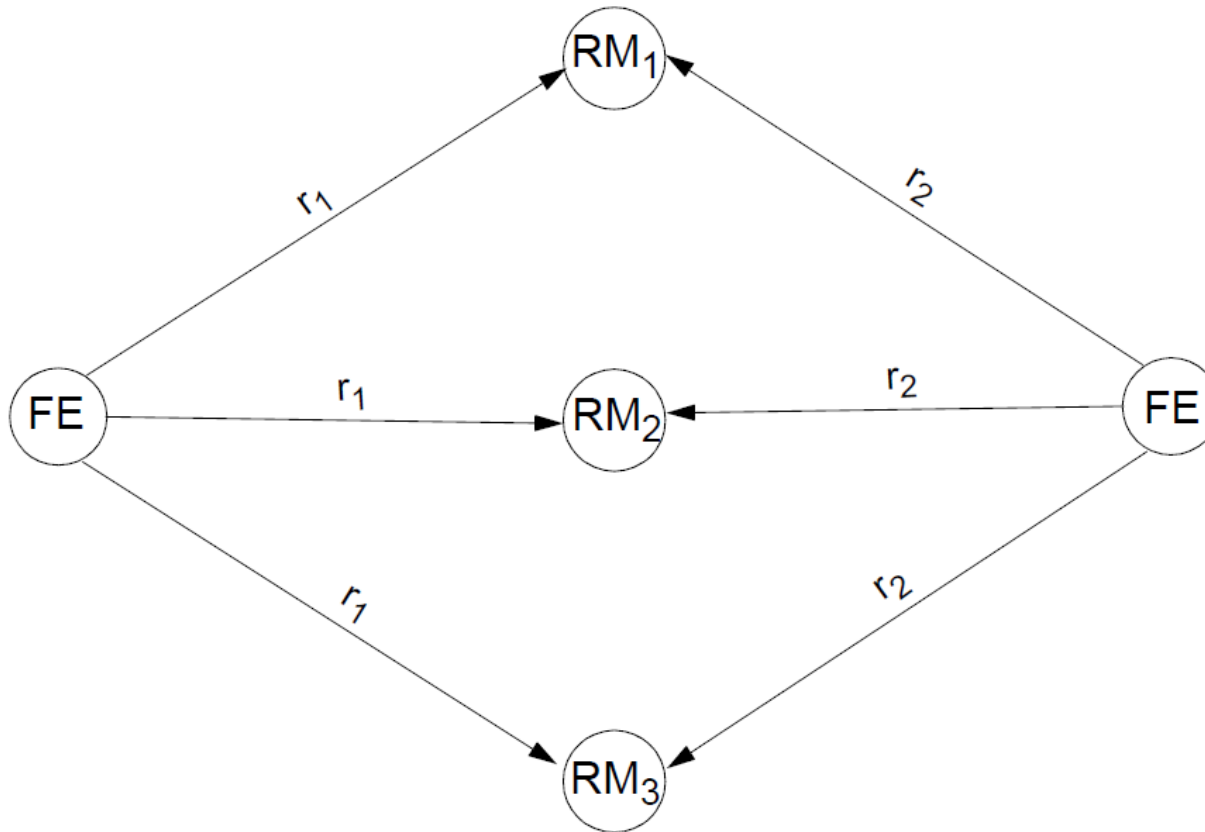
All requests are sent to the **sequencer**.

The sequencer assigns consecutive increasing identifiers to requests as it receives them, and forwards the requests with the corresponding identifier to the RMs.

- One of the RMs, appointed after election, can act as central sequencer.
- The sequencer becomes a performance bottleneck and a critical point of failure.

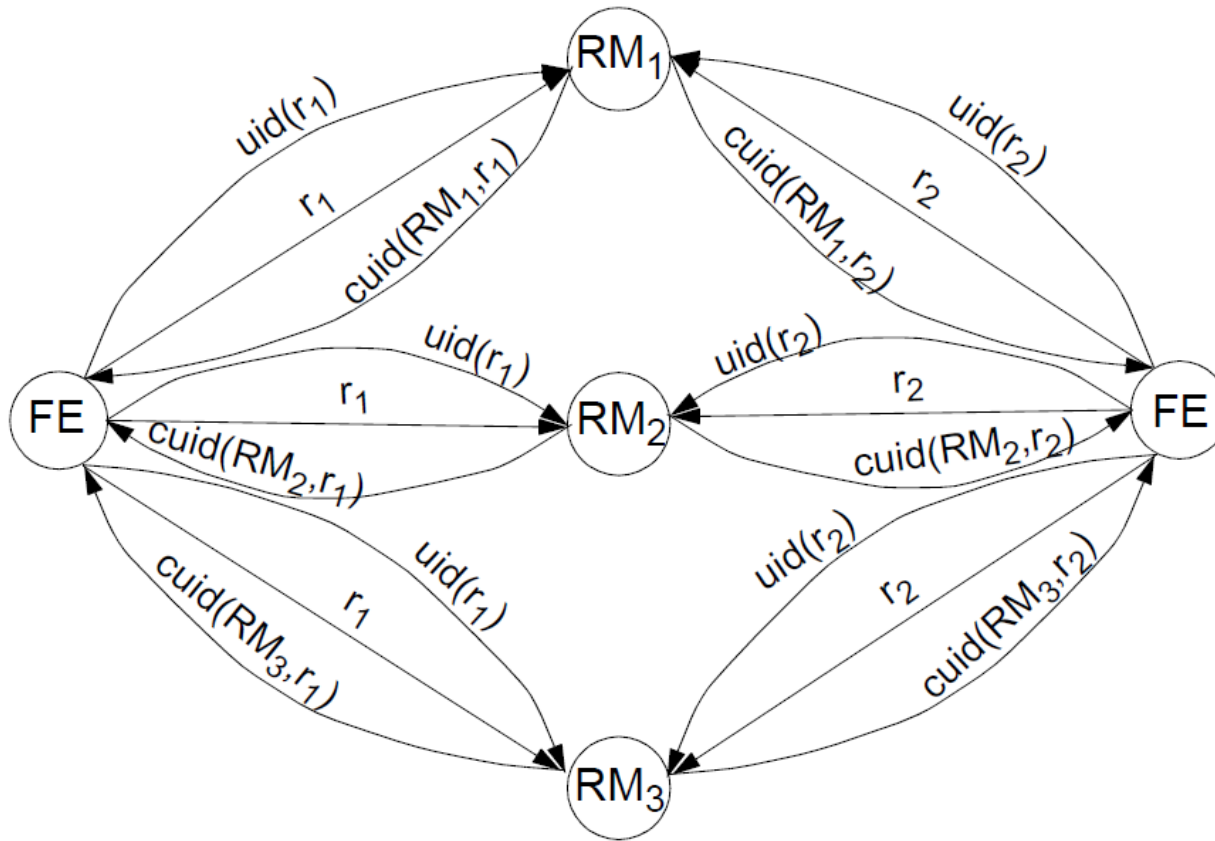
# Total Ordering Based on Distributed Agreement

- This method avoids the need for a centralized sequencer.



# Total Ordering Based on Distributed Agreement

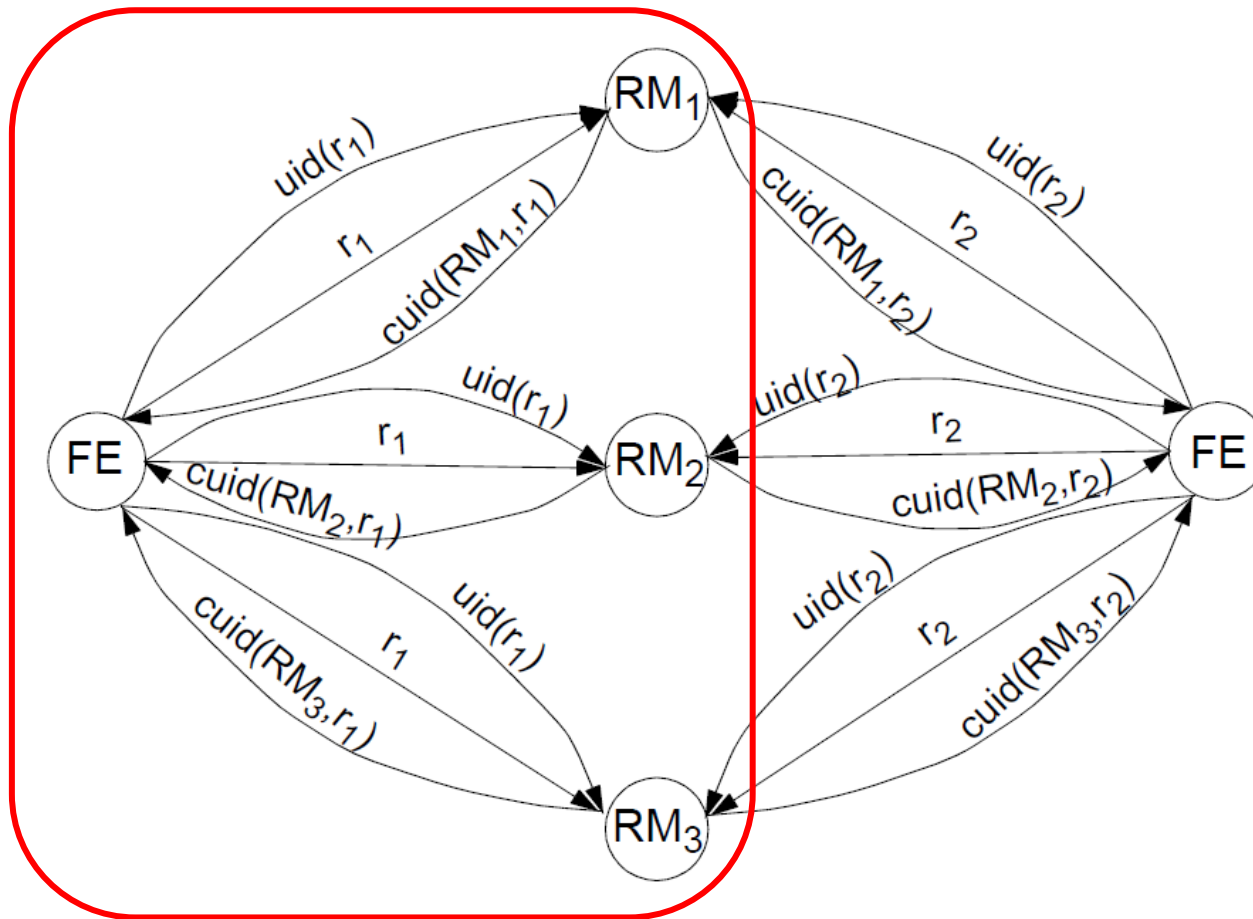
- This method avoids the need for a centralized sequencer.
- Identifiers are assigned to requests as result of distributed agreement



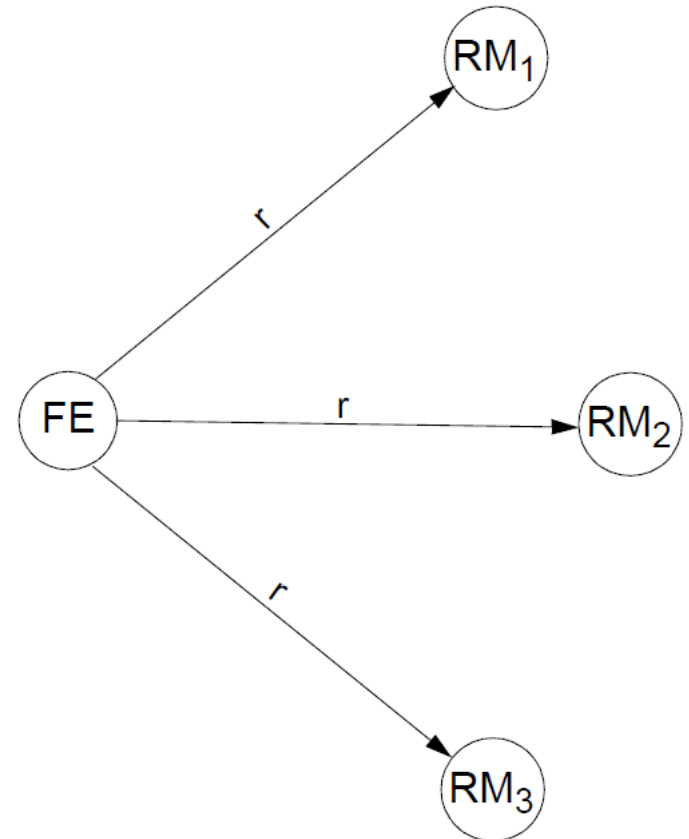


# Total Ordering Based on Distributed Agreement

- This method avoids the need for a centralized sequencer.
- Identifiers are assigned to requests as result of distributed agreement



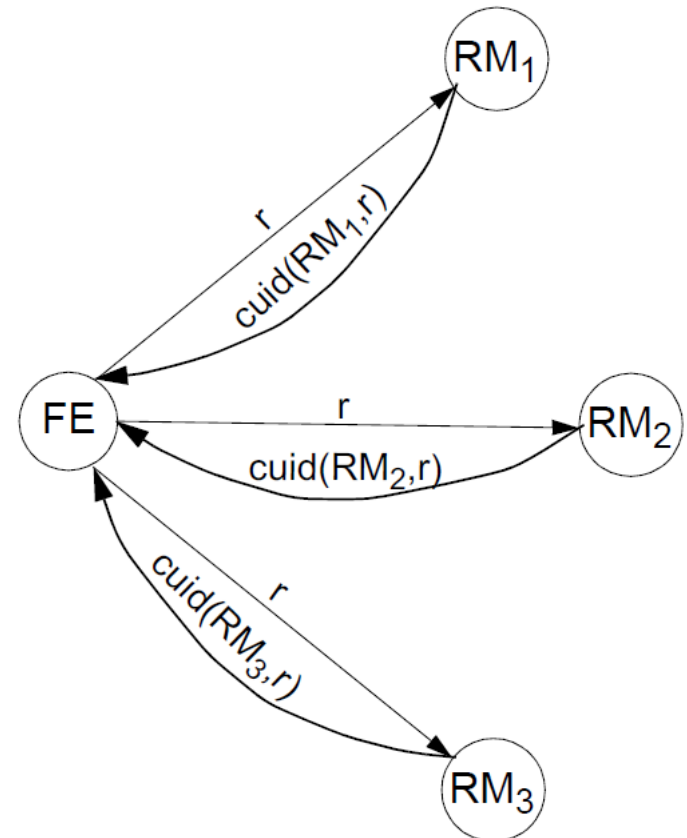
# Total Ordering Based on Distributed Agreement



# Total Ordering Based on Distributed Agreement

Unique identifiers are computed in two phases:

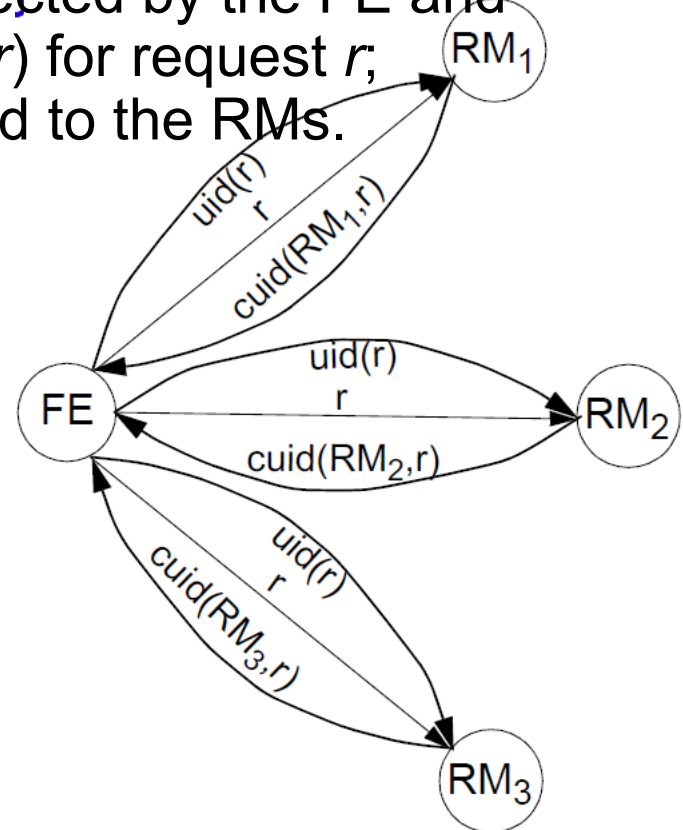
1. Each RM proposes a **candidate unique identifier**  $cuid(RM, r)$  for a request  $r$ ; the  $cuid$  is forwarded to the FE that issued the request.



# Total Ordering Based on Distributed Agreement

Unique identifiers are computed in two phases:

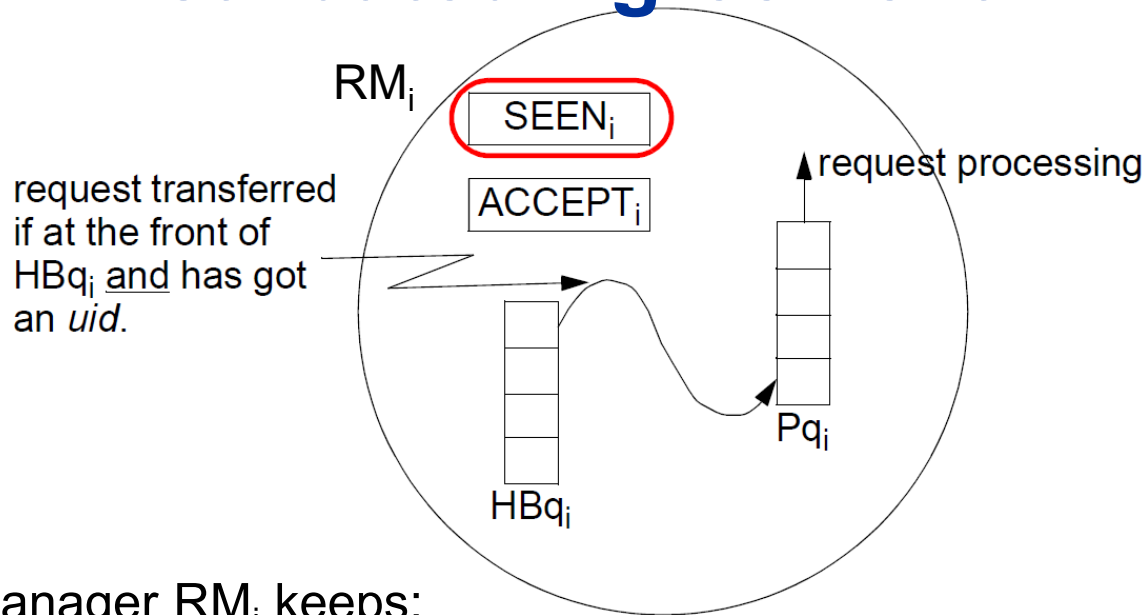
1. Each RM proposes a **candidate unique identifier**  $cuid(RM, r)$  for a request  $r$ ; the  $cuid$  is forwarded to the FE that issued the request.
2. One of the candidate identifiers is selected by the FE and it becomes the **unique identifier**  $uid(r)$  for request  $r$ ; the selected identifier is communicated to the RMs.



# Total Ordering Based on Distributed Agreement

- A replica manager  $RM_i$  has **seen** a request  $r$  once  $RM_i$  has received  $r$  and has proposed a  $cuid(RM_i, r)$  to be forwarded to the respective FE.
- A replica manager  $RM_i$  has **accepted** a request  $r$ , once  $RM_i$  knows the ultimate choice of  $uid(r)$  made for  $r$  by the respective FE.

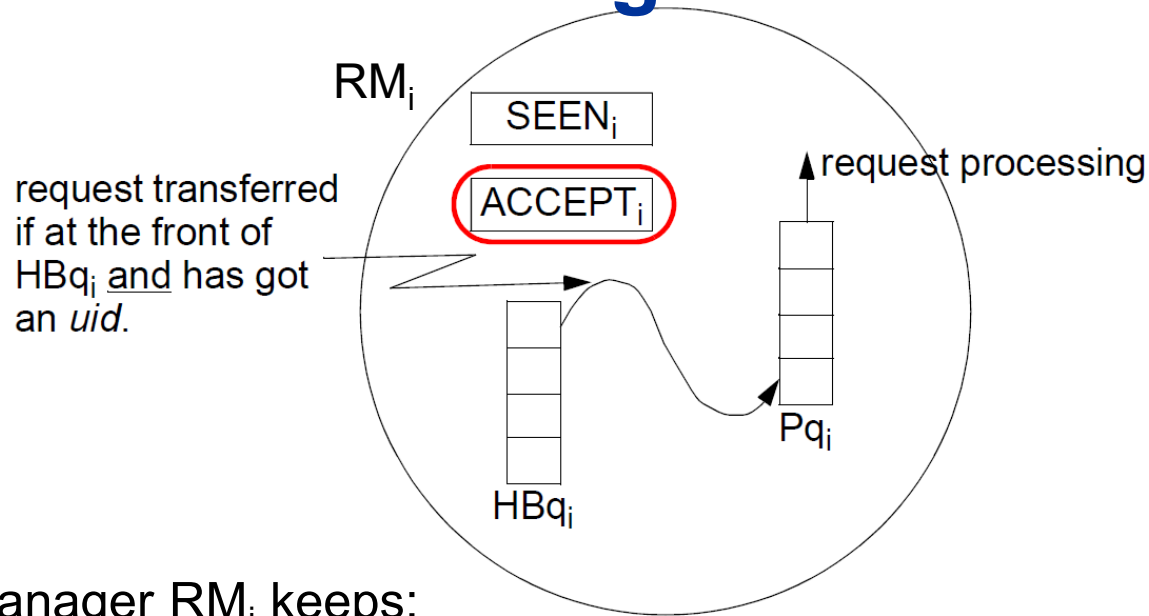
# Total Ordering Based on Distributed Agreement



Each replica manager  $RM_i$  keeps:

- $SEEN_i$ : the largest  $cuid(RM_i, r)$  assigned to any request  $r$  so far seen by  $RM_i$

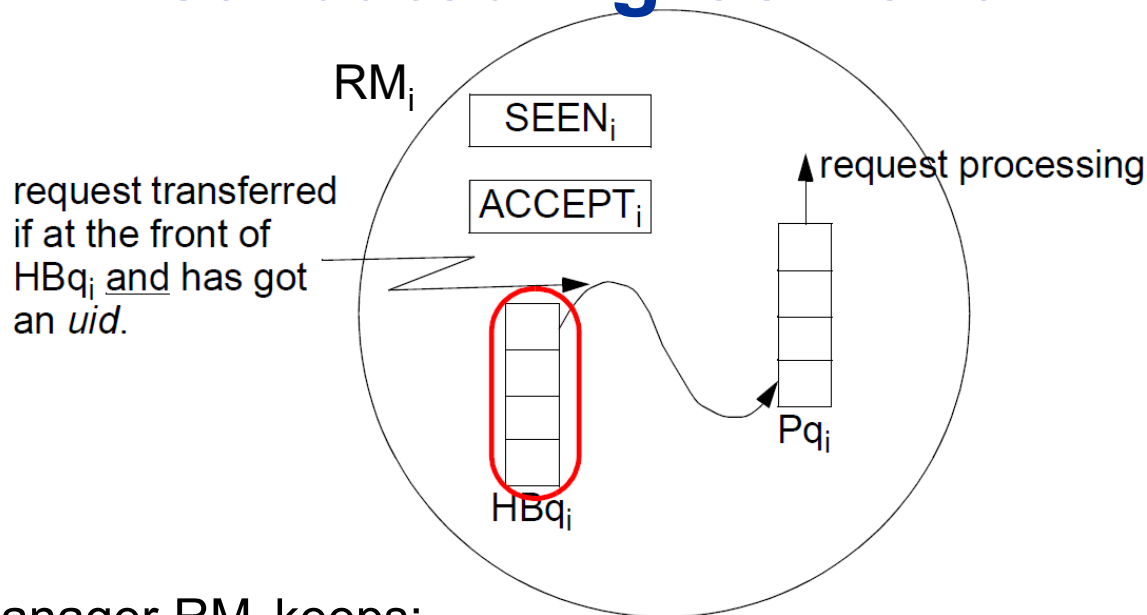
# Total Ordering Based on Distributed Agreement



Each replica manager  $RM_i$  keeps:

- $SEEN_i$ : the largest  $cuid(RM_i, r)$  assigned to any request  $r$  so far seen by  $RM_i$
- $ACCEPT_i$ : the largest  $uid(r)$  assigned to any request  $r$  so far accepted by  $RM_i$

# Total Ordering Based on Distributed Agreement

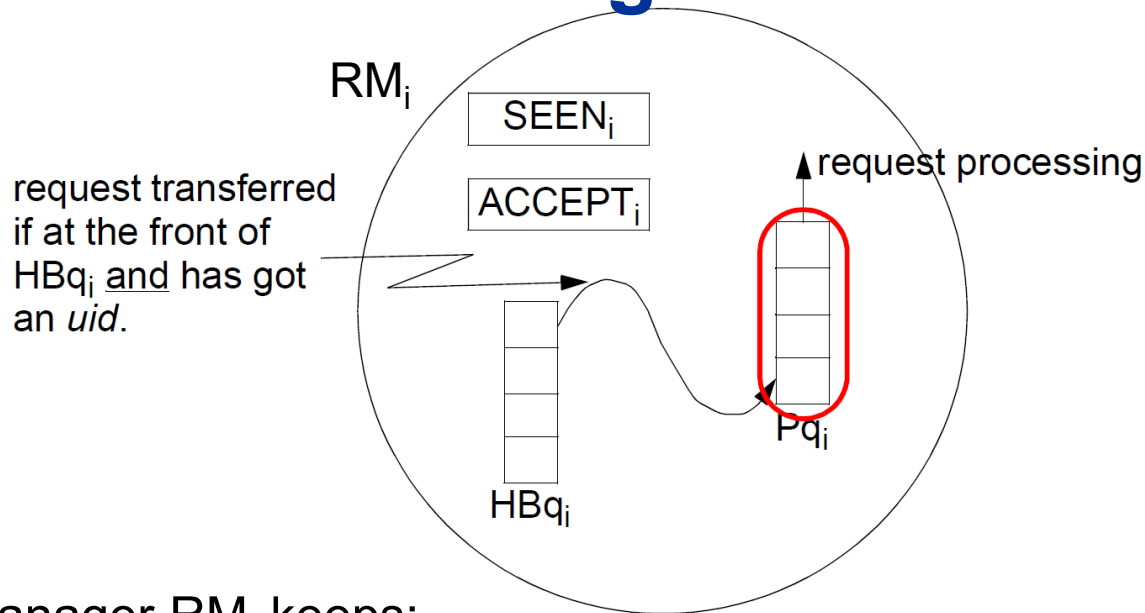


Each replica manager  $RM_i$  keeps:

- $SEEN_i$ : the largest  $cuid(RM_i, r)$  assigned to any request  $r$  so far seen by  $RM_i$
- $ACCEPT_i$ : the largest  $uid(r)$  assigned to any request  $r$  so far accepted by  $RM_i$
- **Hold-back queue** ( $HBq_i$ ): When arrived at  $RM_i$ , a request  $r$  is kept on the  $HBq_i$ , ordered according to its  $cuid(RM_i, r)$ .
  - When the final  $uid(r)$  is received,  $HBq_i$  is **reordered** so that  $r$  is placed according to its  $uid$ .
  - When a request is at the front of  $HBq_i$  and got an  $uid$ , it is **moved** to  $Pq_i$ .



# Total Ordering Based on Distributed Agreement



Each replica manager  $RM_i$  keeps:

- $SEEN_i$ : the largest  $cuid(RM_i, r)$  assigned to any request  $r$  so far seen by  $RM_i$
- $ACCEPT_i$ : the largest  $uid(r)$  assigned to any request  $r$  so far accepted by  $RM_i$
- **Hold-back queue** ( $HBq_i$ ): When arrived at  $RM_i$ , a request  $r$  is kept on the  $HBq_i$ , ordered according to its  $cuid(RM_i, r)$ .
  - When the final  $uid(r)$  is received,  $HBq_i$  is **reordered** so that  $r$  is placed according to its  $uid$ .
  - When a request is at the front of  $HBq_i$  and got an  $uid$ , it is **moved** to  $Pq_i$ .
- **Processing queue** ( $Pq_i$ ):  $Pq_i$  holds accepted requests which before had been placed at the front of  $HBq_i$ ; these requests are processed in order of their  $uid$ .

# Total Ordering

## Based on Distributed Agreement

- The *cuid* proposed by  $RM_i$  for a certain request  $r$  is:  
( $N$  is the number of RMs)

$$cuid(RM_i, r) = \max( SEEN_i, ACCEPT_i ) + 1 + i / N$$

the identifier is unique per  $RM_i$

the identifier is unique in the system

- Once a FE has received, for a certain request  $r$ , the  $cuid(RM_i, r)$  from all  $RM_i$ , it decides on the *uid* for  $r$ :

$$uid(r) = \max_{i=1 \dots N} ( cuid(RM_i, r) )$$

**Question:** Once a request  $r_1$  with  $uid(r_1)$  has been moved to  $Pq$ , is it possible that another request  $r_2$  will be moved later and  $uid(r_2) < uid(r_1)$ ?

# Total Ordering

## Based on Distributed Agreement

In order to be moved to  $Pq$ , the request has

- to be at the front of  $HBq$ , and
- to have got an  $uid$ .

Possible alternatives:

- $r_2$  has already got an  $uid$  when  $r_1$  is moved  
 $\rightarrow uid(r_2) > uid(r_1)$  ( $r_1$  is in front of  $HBq$ )
- $r_2$  has no  $uid$  yet, but has already got a  $cuid$  when  $r_1$  is moved  
 ( $r_2$  has been seen, but not accepted)  
 $\rightarrow uid(r_2) \geq cuid(RM, r_2)$   
 $cuid(RM, r_2) > uid(r_1)$  ( $r_1$  is in front of  $HBq$ )  
 $\rightarrow uid(r_2) > uid(r_1)$
- $r_2$  has no  $cuid$  yet when  $r_1$  is moved ( $r_2$  has not been seen yet) .  
 $ACCEPT \geq uid(r_1)$   
 $cuid(RM, r_2) > ACCEPT$   
 $uid(r_2) \geq cuid(RM, r_2)$   
 $\rightarrow uid(r_2) > uid(r_1)$

# Total Ordering Based on Distributed Agreement

## Rule for initialization:

*/\* performed by each  $RM_i$  at initialization \*/*

[RI1]:  $SEEN_i := 0$   
 $ACCEPT_i := 0$   
 $HBq_i := \emptyset$   
 $Pq_i := \emptyset$

## Rule for handling incoming requests at an RM:

*/\* performed whenever a request  $r$  is received  
by a replica manager  $RM_i$  \*/*

[RC1]:  $cuid(RM_i, r) = \max( SEEN_i, ACCEPT_i ) + 1 + i / N$

[RC2]:  $SEEN_i := cuid(RM_i, r)$

[RC3]: Introduce  $r$  in  $HBq_i$ , ordered according to its *cuid*

[RC4]:  $RM_i$  sends  $cuid(RM_i, r)$  to the FE which issued  $r$ .

# Total Ordering

## Based on Distributed Agreement

### Rule for handling incoming *uid*'s at an RM:

*/\* performed whenever a decision concerning the *uid* of a request *r* is received by a replica manager  $RM_i$  \*/*

[RU1]:  $ACCEPT_i := \max ( ACCEPT_i , uid(r) )$

[RU2]: **if**  $uid(r) \neq cuid(RM_i, r)$  **then**

$HBq_i$  is **reordered** so that  $r$  is placed according to its *uid*  
**end if**

[RU3]: **If** the request at the front of  $HBq_i$  has an *uid*,  
**it is moved to**  $Pq_i$  **in order to be processed.**

### Rule for issuing requests at an FE:

*/\* performed by FE when it issues request *r* and assigns the corresponding *uid* \*/*

[RF1]: FE sends request  $r$  to all  $RM_i$ ,  $i \in \{1, \dots, N\}$

[RF2]: After  $cuid(RM_i, r)$  has been received from all  $RM_i$ ,  
 $uid(r) := \max_{i \in \{1, \dots, N\}} cuid(RM_i, r)$

[RF3]: FE **sends** the final *uid* for  $r$  to all  $RM_i$

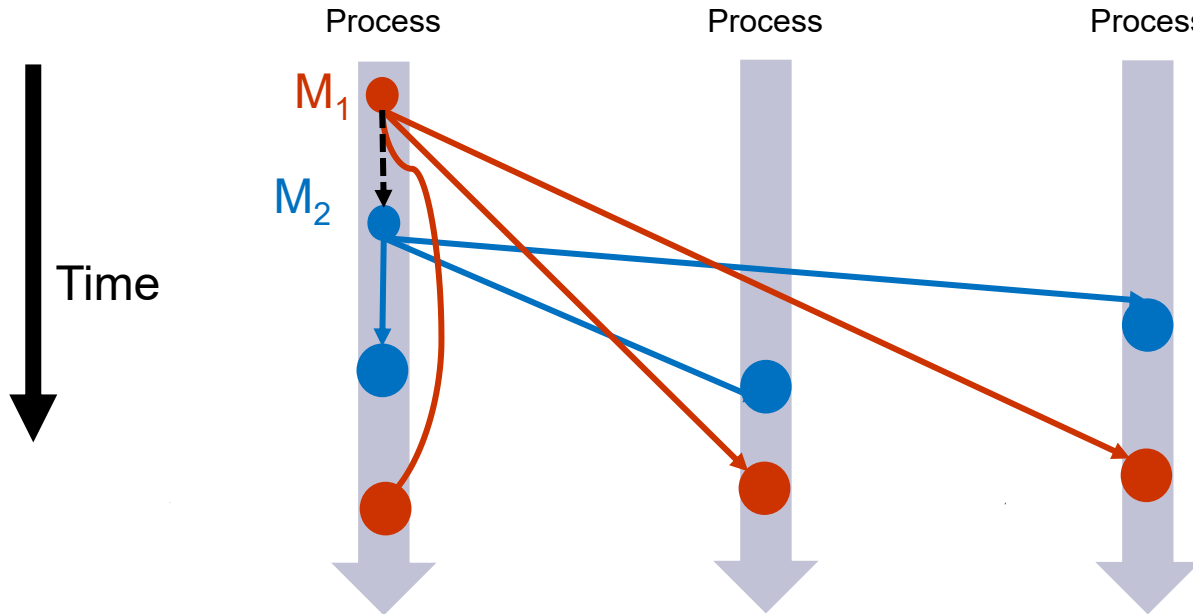
# Total Ordering Based on Distributed Agreement

- Compared to the central sequencer approach, there is no performance bottleneck and unique point of failure.
- If the FE fails before sending out the final *uid*, an RM can take over after an election process.
- If an RM fails before sending its *cuid*, the FE can detect this after a time-out, and ignore the RM.

# Implementing Causal Ordering

- The *total ordering* implemented by the previous algorithm is not causal:  
if we have two requests  $r_1 \rightarrow r_2$ , it is possible that they will be processed on *all* RMs in the order  $r_2, r_1$ .
- For *causal ordering*, if two requests  $r_1$  and  $r_2$  are in a happened-before relation  $r_1 \rightarrow r_2$ , then  $r_1$  should be processed before  $r_2$  at all replica managers.
- Causal ordering of requests can be implemented using vector clocks.  
(See also Lecture 6, slides on causality with vector clocks)  
Details and pseudocode in the book, page 657.

# Total vs. Causal Ordering of Multicast Messages / Requests



**Totally ordered,  
but not causally  
ordered**



# Update Protocols

# Update Protocols

## Problem:

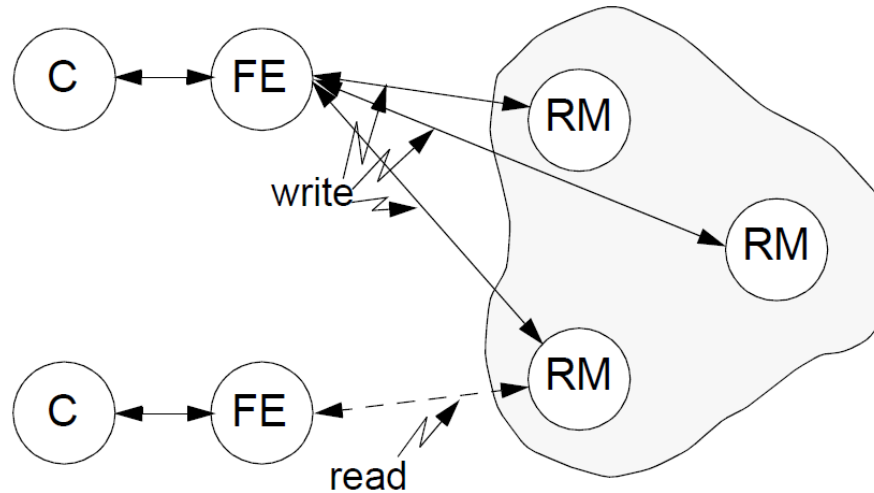
- We have a **replicated file**;  
how do we solve that a user request is always provided with the **most recent version** of the file?

## Some approaches:

- Read-any - Write-all protocol
- Available-copies protocol
- Primary-copy protocol
- Voting protocols

# Read-any - Write-all Protocol

A **read** operation is performed by reading **any** available copy of the file.  
 A **write** operation is performed by writing to **all** copies of the file.



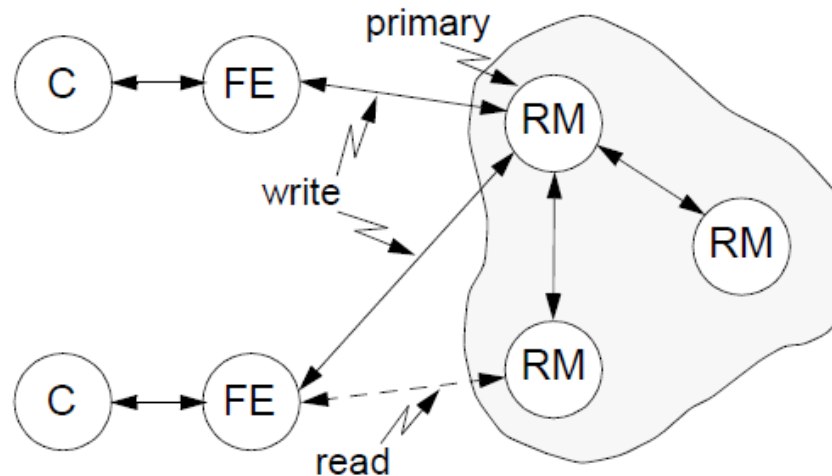
- Some simple kind of **locking** is required: before updating, all copies are locked, and after all have been updated, the lock is released.
- For write operations to succeed, **all** RMs must be **available**; for read operations, only one RM must be available.
- If write operations are frequent compared to reads, this protocol performs poorly.

# Available-Copies Protocol

- This protocol is just a practical variant of read-any – write-all:
  - not *all* RMs, but only those which are *not down*, must be available to perform a write.
  - A **read** operation is performed by **reading any** available copy of the file.
  - A **write** operation is performed by **writing to all available** copies.
  - When a RM **recovers** after a failure, it brings itself up to date by copying from another server, **before** accepting any user request.
- Failed RMs have to be detected and configured out of the system; recovered RMs have to be configured back.

# Primary-Copy Protocol

- A **read** operation is performed by **reading any** available copy of the file.
- A **write** operation is performed by **writing to the primary copy**.



- If **consistency** requirements are **strong** (any read should get the most recent version):
  - When the primary copy gets an update, it immediately locks the secondary copies and updates them.
- If **consistency** requirements are **looser**:
  - updating secondary copies can be performed in the background
  - all the secondary copies will ultimately get updated.

# Voting Protocols

- With voting protocols, the requirement of writing to *all* copies can be softened, without giving up strong consistency.

## The price?

- One has to read *several* copies, not only one, in order to be sure to get the most recent version.

## The benefit?

- Write-performance can be improved: updating becomes more efficient.
- Availability can be improved: RMs can fail and updating/reading can still go on (as long as quorums can be obtained).

# Voting Protocols

Suppose there are  $n$  copies of the file ( $n$  RMs):

- To read the file, a minimum of  $r$  copies have to be consulted
  - $r$  is the **read quorum**.
- To perform a write operation, a minimum of  $w$  copies have to be "acquired" and written
  - $w$  is the **write quorum**.

## The rules for $r$ and $w$ :

- In order to avoid two writes updating the same data at the same time:

$$w > n / 2$$

→ We are also sure that each write quorum includes at least one copy that is up-to-date and has the largest version number.

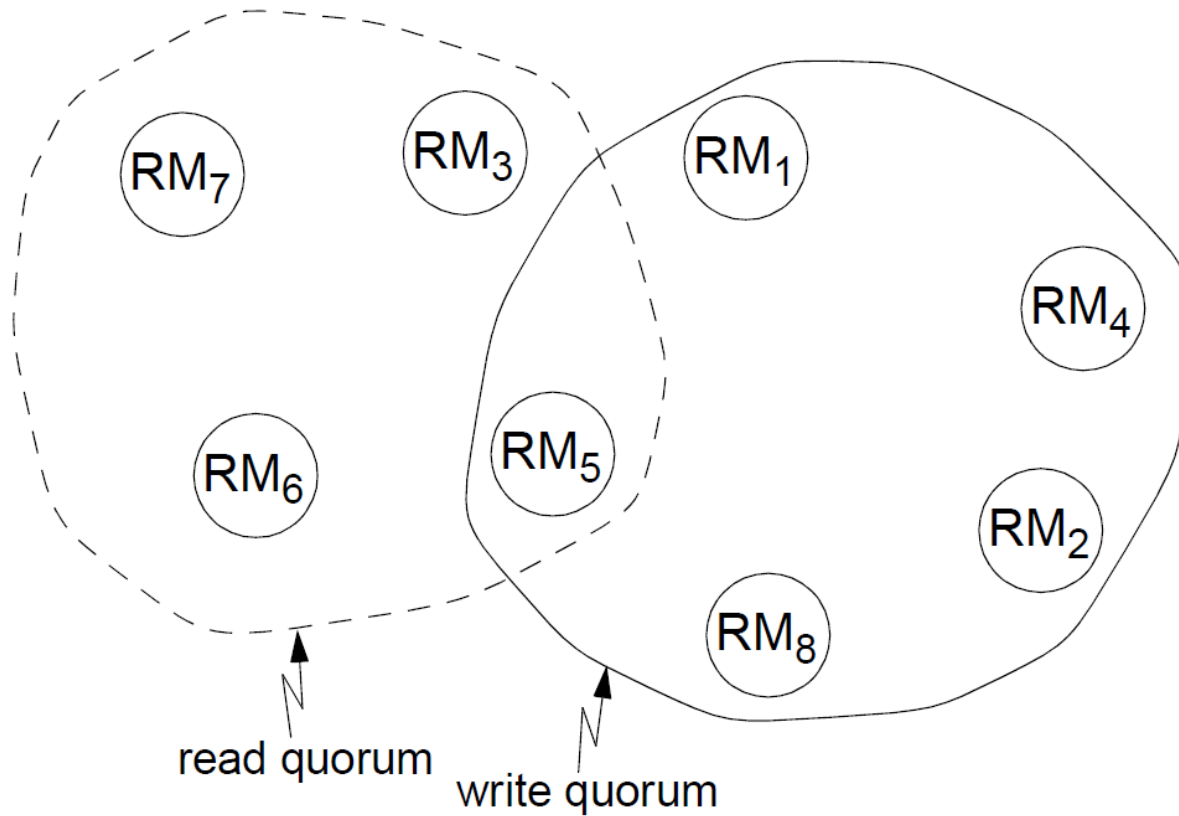
- In order to ensure that each read gets the latest copy:

$$r + w > n$$

→ It is guaranteed that there is a non-null intersection between every read quorum and every write quorum.

# Voting Protocols

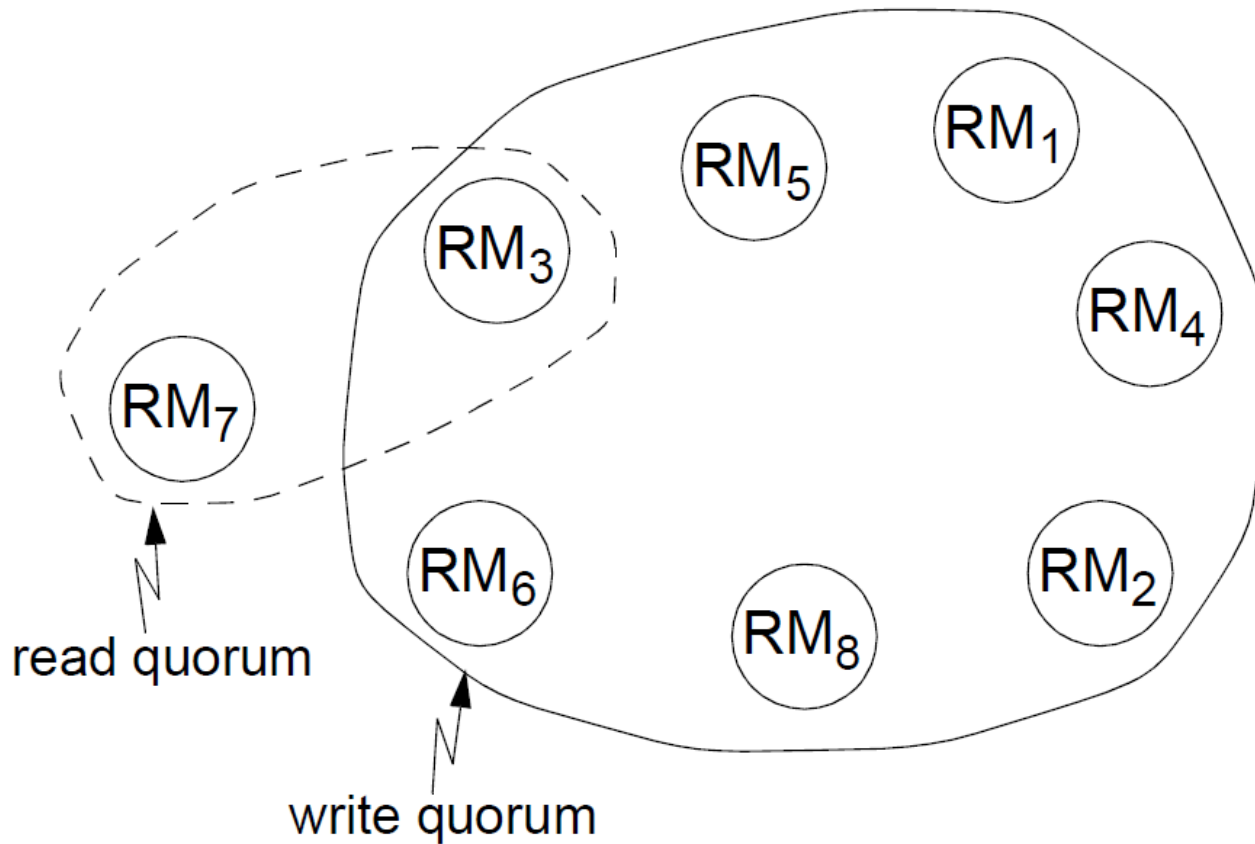
- Example 1:  $n = 8$ ,  $w = 5$ ,  $r = 4$





# Voting Protocols

- Example 2:  $n = 8$ ,  $w = 7$ ,  $r = 2$



# Voting Protocols

## Rule for executing a read:

- Retrieve a read quorum (any  $r$  copies).
- Of the  $r$  copies retrieved, select the one with the largest version number.
- Perform the read operation on the selected copy.

## Rule for executing a write:

- Retrieve a write quorum (any  $w$  copies).
- Of the  $w$  copies retrieved, select the one with the largest version number.
- Increment the version number.
- Perform the update and write the new version with the new version number into all the  $w$  copies of the write quorum.

# Voting Protocols

- The constraints given above allow several possible selections of  $r$  and  $w$ .

This depends on required performance and reliability characteristics.

- A large  $w$  with small  $r$  is suitable for systems with a large ratio of read operations relative to the writes.
  - A small  $w$  with large  $r$  performs well if the ratio of writes is large relative to the reads.
- The *Read-any - Write-all* protocol is a particular case of a voting protocol, with  $r = 1$  and  $w = n$ .

# Acknowledgments

- Most of the slide contents is based on a previous version by Petru Eles, IDA, Linköping University.