# 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 the user should be served with *correct* data (regardless of the replica manager he directly has access to). What *correct* means, depends on the particular application:

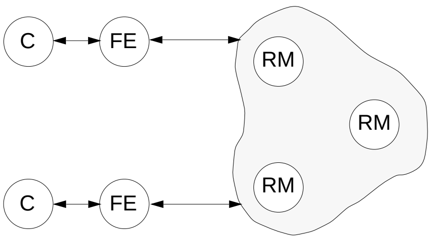
- In some situations it is enough that all operations are *eventually* performed on all copies; it is acceptable that at certain moments different users read different versions of the replicated data. Question: how different?
- Very often, any user access has to provide *the most recent* version of the 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 last version?
3. The effect of replication on performance: strong requirements on consistency can lead to important overheads.

## Architectural Model

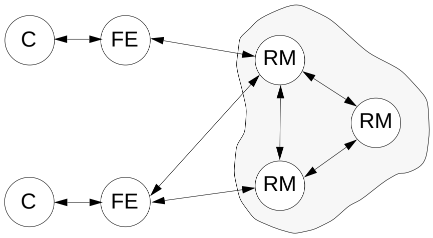
- Client (C): makes the request (reads and updates).
- Front end (FE): communicates with one or more replica manager (provides replication transparency).
- Replica managers (RM): contain the replicas and perform operations upon them.



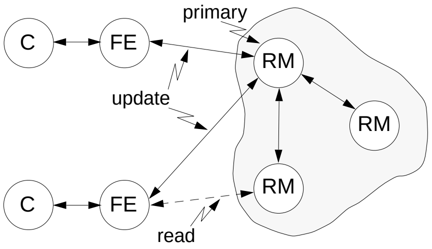
☞ Different alternative models are possible, depending on the particular communication pattern between FEs and RMs, and between the different RMs.

## Architectural Model (cont’d)

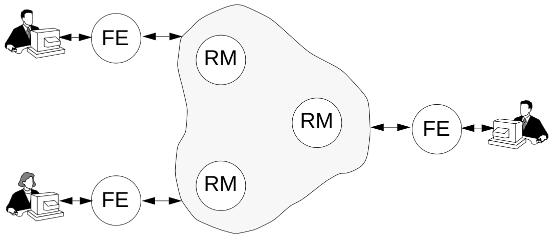
- All RMs communicate with each other in order to agree on operations so that coherence is preserved between copies.
  - Alternatively, FEs can communicate with only one or with several of the RMs.



- There exists a "primary" RM which coordinates the other RMs managing copies of the same data. All updates are directed from the FEs to that primary RM which then propagates them to the other RMs. Requests for reading 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, can select a certain item to visualise, and can respond to a given message.

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 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.

- 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.

## Request Ordering (cont’d)

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 he has seen the initial message.

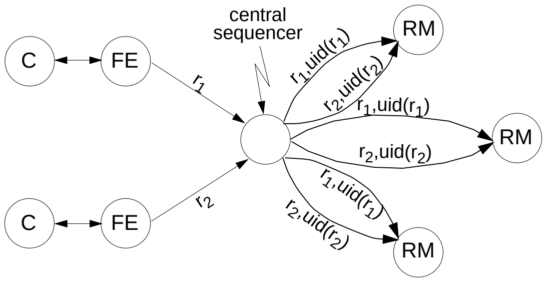
## Implementing Total Ordering

☞ The basic idea: assign totally ordered identifiers to requests, so that each replica manager makes the same ordering decision based upon 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*($r_1$).

- 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.

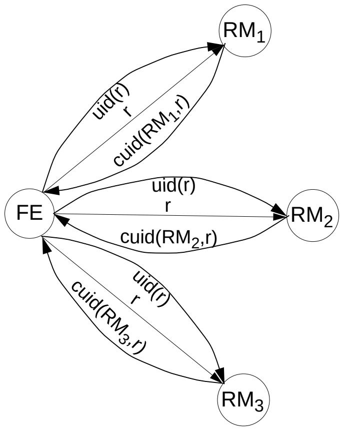
- One of the RMs, appointed as result of an election, can act as central sequencer.
- The sequencer becomes a performance bottleneck and a critical point of failure.

## Total Ordering Based on Distributed Agreement

☞ This method avoids the need for a centralized sequencer; identifiers are assigned to requests as result of 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 which 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 (cont'd)

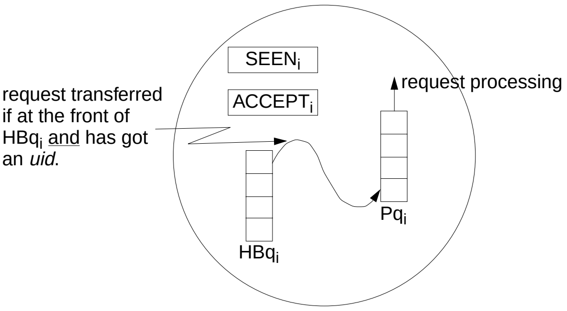
☞ A replica manager $RM_i$ has *seen* a request *r* once $RM_i$ has received *r* and has proposed a *cuid*($RM_i$,*r*) to be forwarded to the respective FE.

☞ 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.

Notice

- At delivery from an FE, a request *r* is labelled with an identifier which is used to uniquely identify it during the agreement process; this can be a pair (*k*,*i*), where *k* is an integer counting successive requests made by $FE_i$, and *i* is an identifier of $FE_i$. Although unique, this identifier cannot be used for total ordering because it does not satisfy the condition in the *Notice* on slide 9.

## Total Ordering Based on Distributed Agreement (cont’d)


SEEN$_i$
ACCEPT$_i$
request transferred if at the front of HBq$_i$ and has got an *uid*.
request processing
Pq$_i$
HBq$_i$

☞ 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 to 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 has got an* uid*, it is moved to* Pq$_i$.
- Processing queue (*Pq$_i$*): *Pq$_i$* holds accepted requests which before have been placed at the front of *HBq$_i$*; these requests are processed in order of their *uid*.

## Total Ordering Based on Distributed Agreement (cont’d)

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

$$cuid(RM_i, r) = \underbrace{\underbrace{max(\lfloor SEEN_i \rfloor, \lfloor ACCEPT_i \rfloor) + 1}_{\text{the identifier is unique per RM}_i} + i/N}_{\text{the identifier is unique in the system}}$$

☞ Once an FE has received, for a certain request *r*, the *cuid*(RM$_i$,*r*) from all RM$_i$, it decides on the *uid* for *r*:

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

## Total Ordering Based on Distributed Agreement (cont’d)

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*($r_1$)?

☞ In order to be moved to *Pq* the request has
- to be at the front of *HBq*.
- to have got an *uid*.

- $r_2$ has already got an *uid* when $r_1$ is moved ⇒
  *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) ⇒
  *uid*($r_2$) ≥ *cuid*(RM,$r_2$)   (see previous slide)
  *cuid*(RM,$r_2$) > *uid*($r_1$)   ($r_1$ is in front of *HBq*)
  ⇩
  *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 ≥ *uid*($r_1$)
  *cuid*(RM,$r_2$) > ACCEPT   (see previous slide)
  *uid*($r_2$) ≥ *cuid*(RM,$r_2$)   (see previous slide)
  ⇩
  *uid*($r_2$) > *uid*($r_1$)

## Total Ordering Based on Distributed Agreement (cont’d)

The Algorithm

☞ Rule for initialization

/* performed by each RM$_i$ at initialization */

[RI1]: SEEN$_i$ := 0, ACCEPT$_i$ := 0.
*HBq$_i$* := ∅, *Pq$_i$* := ∅.

☞ Rule for handling incoming requests at an RM

/* performed whenever a request *r* is received by a replica manager RM$_i$ */

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

[RC2]: SEEN$_i$ := *cuid*(RM$_i$,*r*).

[RC3]: Introduce *r* in *HBq$_i$*, ordered according to its *cuid*.

[RC3]: RM$_i$ sends *cuid*(RM$_i$,*r*) to the FE which issued *r*.

☞ 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$_i$ := *uid*(*r*).

[RU2]: If *uid*(*r*) ≠ *cuid*(RM$_i$,*r*) then *HBq$_i$* is reordered so that *r* is placed according to its *uid*.

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

## Total Ordering Based on Distributed Agreement (cont'd)

☞ Rule for issuing requests at an FE

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

[RF1]: FE sends request *r* to all RM$_i$, $i \in \{1 .. N\}$.

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

[RF3]: FE sends the final *uid* for *r* to all RM$_i$.

- 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. The algorithm has been discussed at Fö 5, slide 16.

## Update Protocols

Problem

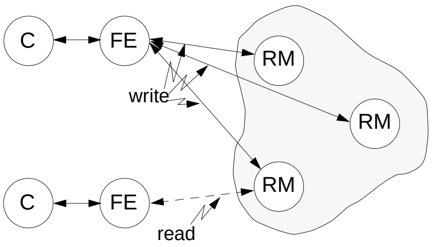
We have a replicated file; how do we solve that a user request is always provided with *the most recent* version of the file?

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 read, this protocol performs poorly.

## Available-Copies Protocol

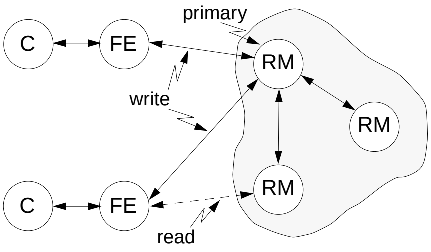
☞ This protocol tries to improve on the previous one ⇒ not all RMs, but only those which are not down, must be available in order to perform a write.

- A read operation is performed by reading *any available copy* of the file.
- A write operation is performed by writing to *all available copies* of the file.
- When an 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*.
- Each RM having a secondary copy updates its copy either by receiving notification of change from the RM having the primary copy or by requesting the update 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.

⇩

This protocol is only another method to implement the *read-any - write-all* protocol

☞ 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?

You have to read several copies, not only one, in order to be sure you get the most recent version.

- 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 (cont’d)

☞ 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 "aquired" and written; $w$ is the *write quorum*.

☞ In order to avoid two writes updating the same data at the same time: $w > n/2$.

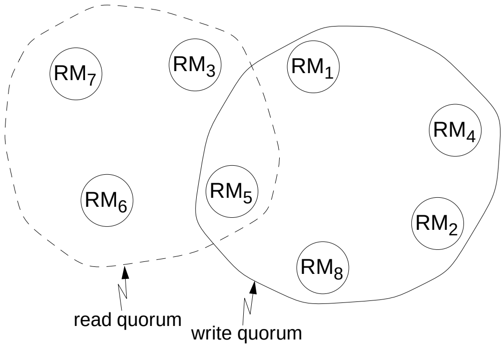
In this case we are also sure that each write quorum includes at least one copy which is up to date and has the largest version number.

☞ In order to ensure that each read gets the latest copy: $r + w > n$

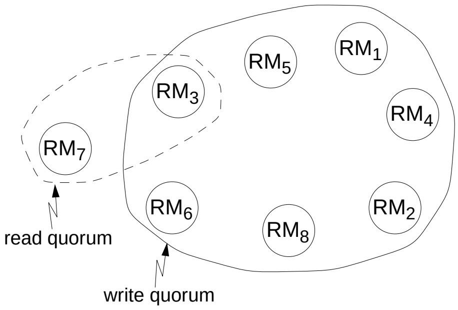
It is guaranteed that there is a non-null intersection between every read quorum and every write quorum.

## Voting Protocols (cont'd)

- $n = 8$, $w = 5$, $r = 4$.



- $n = 8$, $w = 7$, $r = 2$.



## Voting Protocols (cont'd)

☞ Rule for executing a read

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

☞ Rule for executing the write

- Retrieve a write quorum (any *w* copies).
- Of the *w* copies retrieved, select the copy 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 (cont'd)

☞ 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.

☞ Read-any - Write-all protocol is a particular case of a voting protocol, with $r = 1$ and $w = n$.

## Summary

- Replication is an important vehicle for increased performance, availability, and fault tolerance in distributed systems.
- Several aspects are essential for keeping the requested consistency of replicas:
  - the order in which successive operations are performed on the replicas;
  - how many and which replicas to update/read for a certain write/read operation;
- Ordering of requests can be *total* or *causal*.
- Total ordering can be performed using a central sequencer or based on distributed agreement.
- Causal ordering can be implemented using vector clocks (see Fö. 5).
- Update protocols have to be designed in order to guarantee that a user always reads the most recent version of a replicated file.
- The most simple update protocols are based on a *write-all* approach; this means that all replicas are updated for a write operation.
- With voting protocols both availability and performance can be improved. Only a part of the replicas have to be written for a write operation; on the other side, a certain number of replicas have to be read in order to guarantee that the most recent version of the file is accessed.