Introduction to Replication

Motivations for Replication:

1. **Performance Enhancement**
2. **Enhanced Availability**
3. **Fault Tolerance**

Main requirements:

1. **Replication Transparency**
2. **Consistency**
3. **Low Response Time**

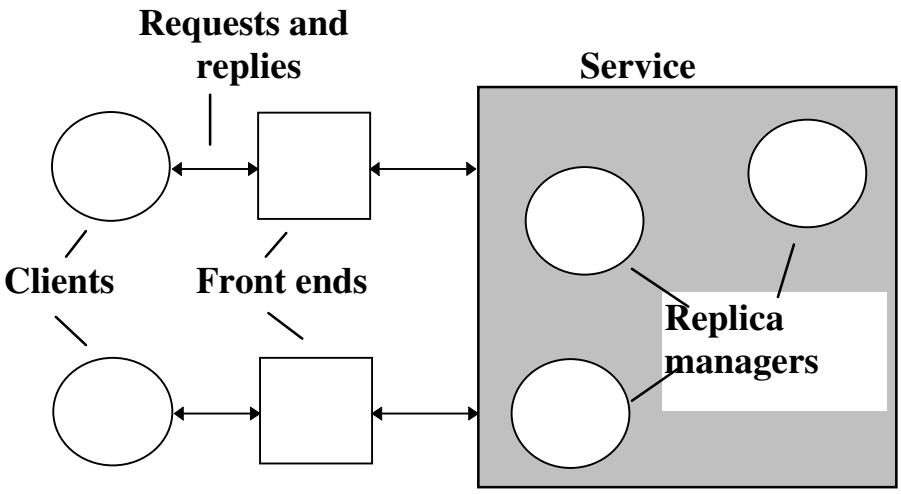
Consistency Models:

1. **Asynchronous Model**
2. **Totally Synchronous Model**
3. **Quorum-based Scheme**
4. **Causality-based Schemes**

In each consistency model, satisfactory trade-offs are necessary among:

1. **consistency**
2. **availability**
3. **response time**

Basic Architectural Model



Components:

- **Replica managers:**

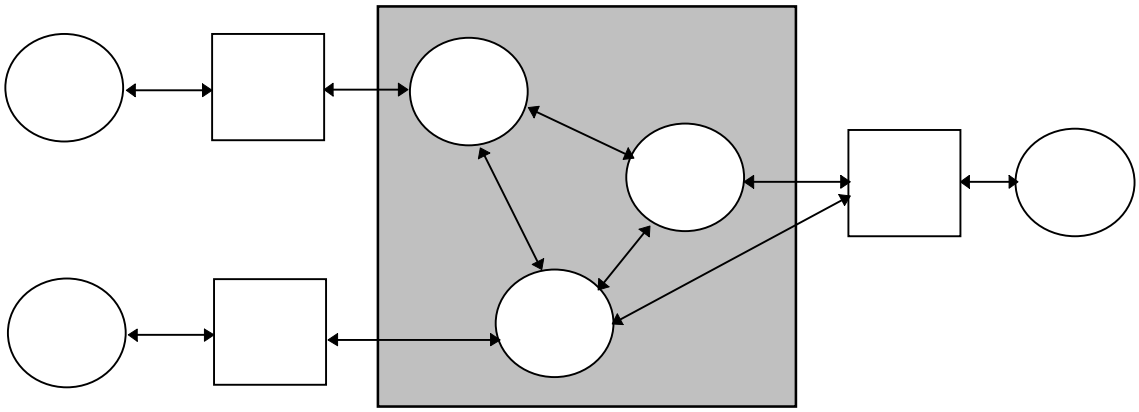
  Processes that contain replicas of data items and perform operations upon them directly

- **Front ends:**

  To communicate with one or more of the replica managers by messages passing

  It is the vehicle of implementing *replication transparency*

The Gossip Architecture



Main Features:

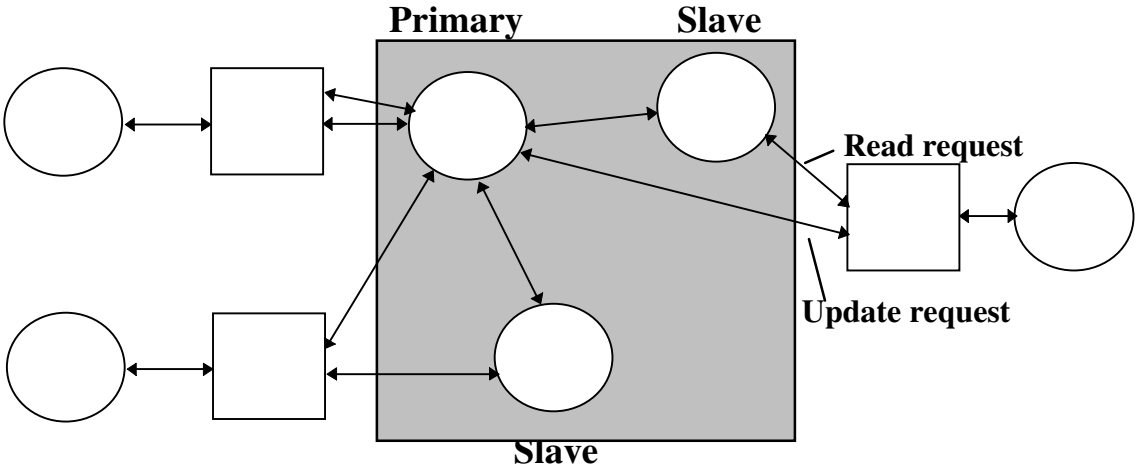
1. Replica managers and clients are separate processes

2. It does not define if front ends are separate to client processes

3. The front end normally communicates with an individual replica manager (but communication with several replica managers is not prohibited)

4. Replica managers exchange 'gossip' messages periodically in order to convey the updates

The Primary Copy Model Architecture



Main Features:

1. All front ends communicate with the same 'primary' replica manager when updating a data item

2. The primary replica manager propagates the updates to the other servers called 'slaves'

3. Front ends may read the data item from a slave

4. When the primary server fails, one of the slaves can be promoted to act as the primary

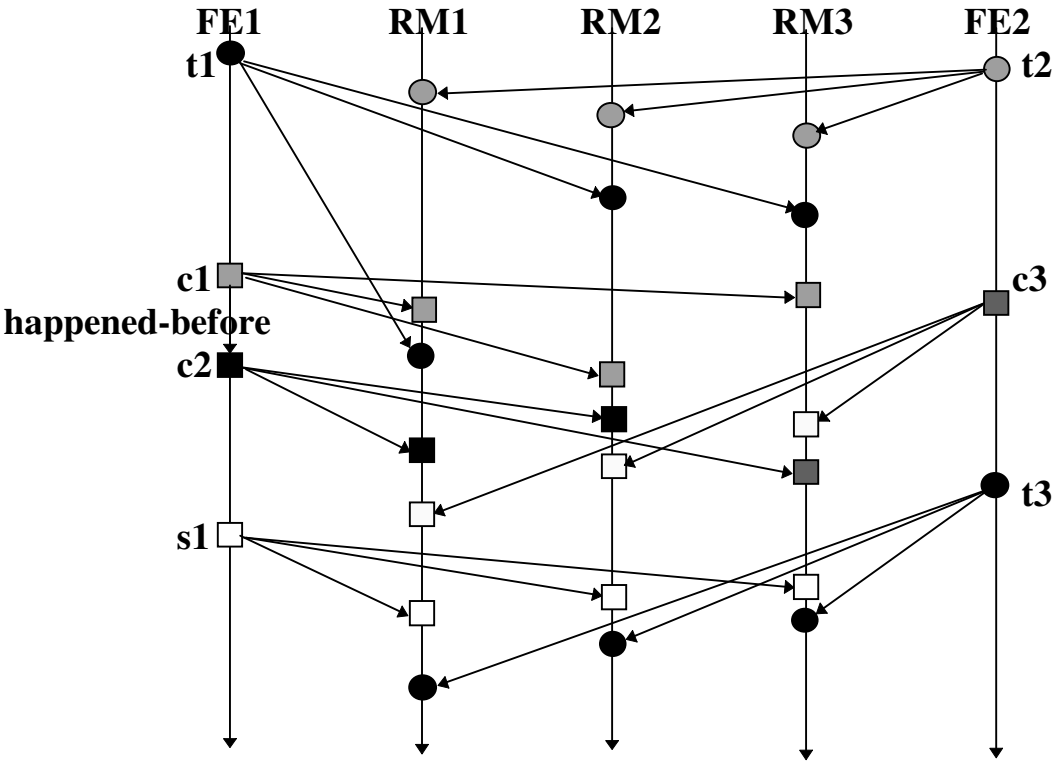
Consistency and Request Ordering

Three Models:

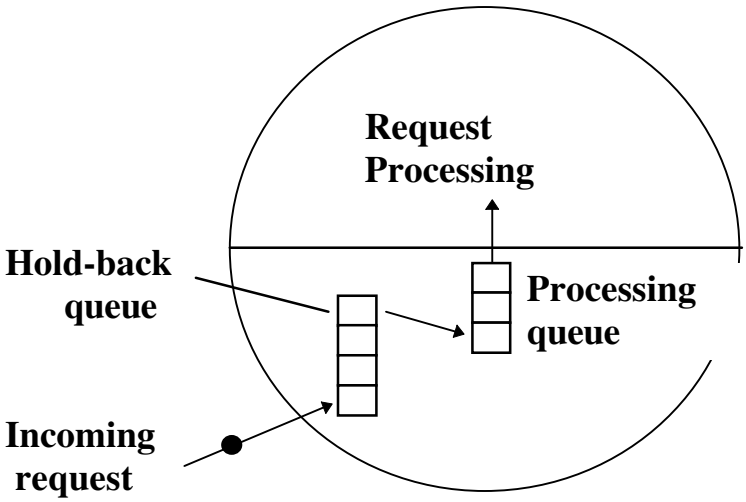
1. Total ordering
If r1 and r2 are requests, then either r1 is processed before r2 at all replica managers or r2 is processed before r1 at all replica managers (see t1 and t2 in Fig.)

2. Causal ordering
If r1 and r2 are requests and r1 *happened-before* r2, then r1 is processed before r2 at all replica managers (see c1, c2 and c3 in Fig.)

3. sync-ordering
A sync-ordered request forces the order of other requests in the sense that every other request is consistently processed before the sync-ordered request or after it at all replica managers (see c3, t3 and s1 in Fig.)



Implementing Request Ordering



Basic ideas:

1. ***Hold-back***
A received request is not processed by a replica manager until ordering constraints can be met. During this stage they are stored in the ***hold-back queue***.

2. A request message is said to be ***stable*** at a replica manager if all prior requests have been processed. They are moved to the ***processing queue***.

Properties to be proved:

1. ***Safety property***
Once a request has been processed it is impossible for a 'prior' request to arrive

2. ***Liveness property***
No request should wait on the hold-back queue indefinitely

Implementing Total Ordering

Basic idea:

To assign totally ordered identifiers to requests so that each RM site makes the same ordering decision based upon these identifiers.

Two main methods:

1. Centralised method based on a *sequencer* process

2. Fully distributed method

Centralised method:

1. All requests are sent to the sequencer as well as to the RM sites

2. The sequencer assigns consecutive increasing identifiers to requests and forwards the identifiers to the RM sites

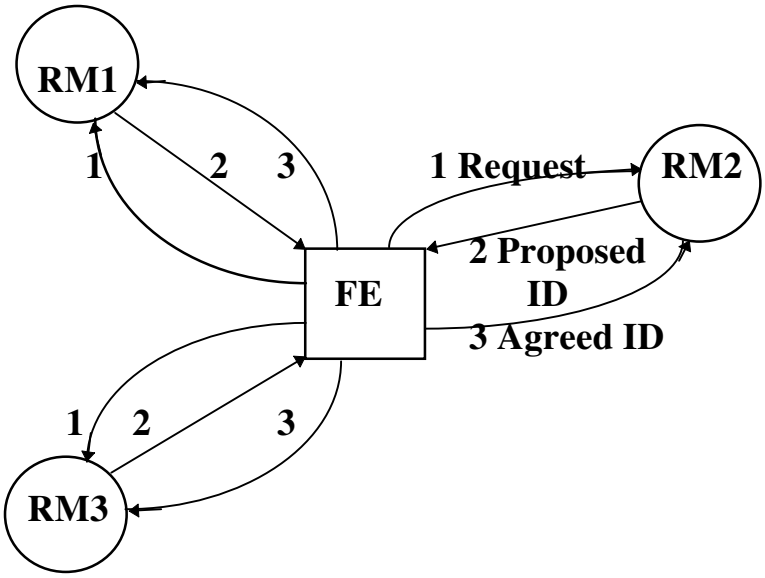
3. Requests arriving at an RM site are held back until they are next in sequence

Drawbacks:

1. The sequencer may become a bottleneck

2. The sequencer is a critical point of failure

Distributed Total Ordering



The algorithm:

1. The FE sends the request bearing a temporary identifier to all Rms. (The temporary ID is larger than any IDs used by the FE.)

2. Each RM replies. RMi responds with a proposed ID:

$$\mathbf{Max(F_{max}, P_{max}) + 1 + i/N}$$

where F_{max} is the largest final ID agreed so far and P_{max} is its own largest proposed ID. N is the number of RM sites.

Each RM provisionally assigns the proposed ID to the request and places it on its hold-back queue which is ordered with the *smallest* request ID at the front.

3. The FE collects all the proposed IDs and selects the largest one as the next agreed ID and sends this final ID to all RMs**.**

4. The RMs attach the final ID to the request which is then reordered on the hold-back queue. When the request at the front of the hold-back queuehas been assigned, it is stable and is transferred to the tail of the processing queue.

Distributed Total Ordering (Cont.)

Proof of properties:

1. ***Liveness***
Every request will eventualy be assigned a final ID and reach the front of the hold-back queue. (Since new requests are assigned increasing proposed IDs, a finalized request cannot continually be pre-empted by candidate requests with smaller IDs.

2. ***Safety property***
Assume that a request r1 has been assigned a final ID and has reached the front of the hold-back queue. Let r2 be any other request on the same queue without a final ID. Because of the algorithm:

final_ID(r2) ≤ proposed_ID(r2)

Since r1 is at the front of the hold-back queue:

proposed_ID(r2) > final_ID(r1)

Therefore:

final_ID(r2) > final_ID(r1)

and the safety property is assured.

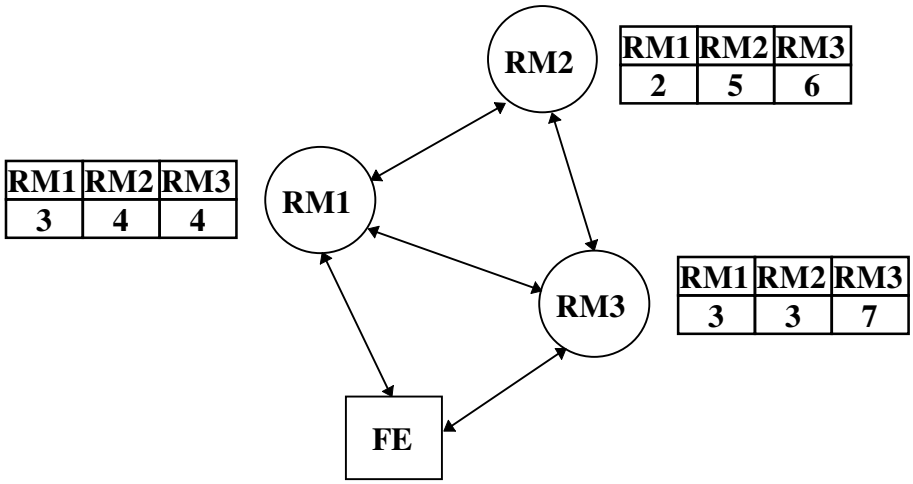
Advantages:

1. Simple to implement
2. There is no bottleneck or single point of failure

Drawback:

1. Expensive algorithm
Three messages are sent between the FE and each RM before a request can become stable.

Implementing Clausal Ordering with Vector Timestamps



Notations:

- For each process pi there is a vector clock whose timestamp is denoted by Vti. Each Vti is a vector of integer values and its length is n where n is the number of processes (replica managers).

- The value $Vt_i[k]$ represents a count of events that have occurred at pk and that are known at pi.

The algorithm:

1. All processes Pi initialize VT_i to zeroes.

2. When Pi generates a new event, it increments $VT_i[i]$ by 1; it piggybacks the value $vt = VT_i$ on the messages it sends.

3. When P_k processes a request bearing a timestamp vt, it updates its vector clock as follows:

VT_k := merge(VT_k, vt)

The merge operation is defined for any pair of vector timestamps u, v as:

merge(u, v)[k] = max(u[k], v[k]) for k=1, 2, ..., n

If e, f are events and u, v are their timestamps, $u \leq v$ iff e ***happened-before*** f.