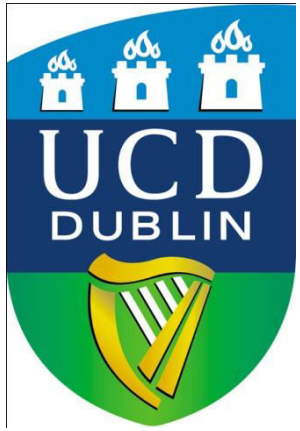
Distributed Systems:
Replication Systems

Anca Jurcut
E-mail: anca.jurcut@ucd.ie

School of Computer Science and Informatics
University College Dublin
Ireland



From Previous Lecture…

- **Replication Systems**
  - **Replication:**"the maintenance of multiple copies of data at multiple computers"
  - Key issue in the design of **effective** distributed systems
  - Provide: Enhanced Performance, High Availability, Fault Tolerance
  - **Key Features:** Replication Transparency, Consistency
  - **Replication System** is the sub-system (service) of a distributed system that is concerned with replication of data
  - Implemented as a set of **Replica Managers (RM)**
  - **5 STEPS** in Handling a request to perform an operation on a logical object
    - **Request:** FE issues the request to one or more RMs
    - **Coordination**: RMs coordinate to execute the request consistently
    - **Execution**: The RMs execute the request
    - **Agreement**: The RMs reach consensus on the effect of the request
    - **Response**: One or more RMs respond to the FE

From Previous Lecture…

- Replication Systems
  - Coordination Techniques
  - Recoverability
  - State-Based Replication
  - Static versus Dynamic Systems
  - Example: Diary System

- Group Communication
  - Multicast communication
  - IP multicast
  - Multicast System Model
  - Basic Multicast
  - Reliable Multicast
  - Group Membership Service (GMS)

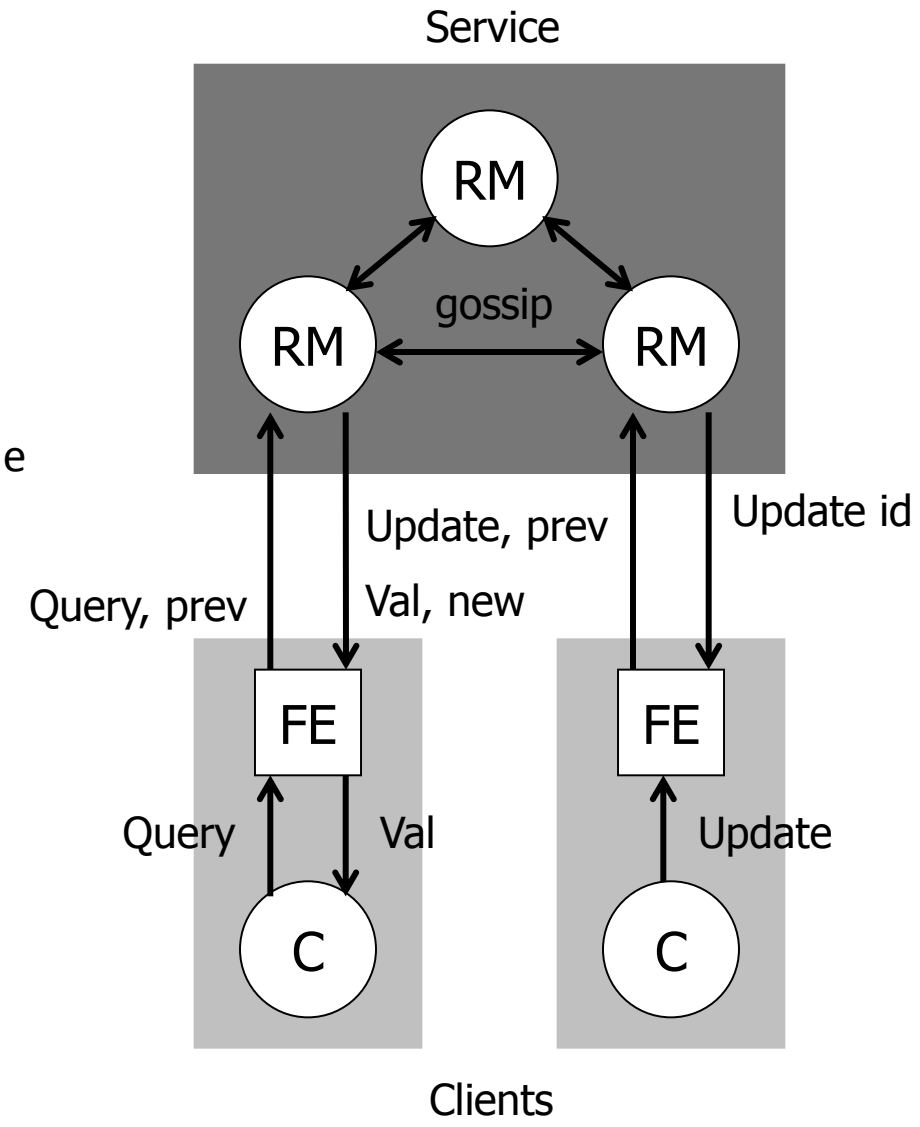
From Previous Lecture…

- **Active and Passive Replication**
  - **Consistency:** how to keep replicas consistent
  - **Linearisability:** The sequence of interleaved operations that is totally ordered based on real time.
  - **Sequential Consistency**: Any sequence of interleaved operations that satisfies the local ordering of operations carried out by the processes.
  - **Passive Replication:** A single RM and one ore more secondary (backup) RMs
    - Primary RM handles all request and sends updates to the backup RMs
    - Linearisable
  - **Active Replication:** RMs are state machines that play equivalent roles and are organised as a group
    - All RMs handle all requests concurrently
    - Sequentially Consistent

Distributed Systems:
Case Study: The Gossip Architecture

Introduction

- The **Gossip architecture** is a framework for implementing highly available services.
  - Data is replicated close to the groups of clients that need it.
  - Replica Managers then exchange "gossip" messages periodically in order to convey the updates they have each received from clients.
- The service supports two operations:
  - Query: read-only operations
  - Update: modify-only operations



Introduction

- The Front End is able to send queries and updates to any replica manager they choose.
  - Any that is available and can provide reasonable response times.

- The system makes two guarantees:
  - Each client obtains a consistent service over time:
    - Replica Managers respond to a query with data that reflects at least the updates that the client has observed so far.
    - This occurs even when the client communicates with a replica manager that is “less advanced” that the one it used before.
  - Relaxed consistency between replicas:
    - All replicas eventually receive all updates.
    - Updates are applied with ordering guarantees that make the replicas sufficiently similar to suit the needs of the applications.

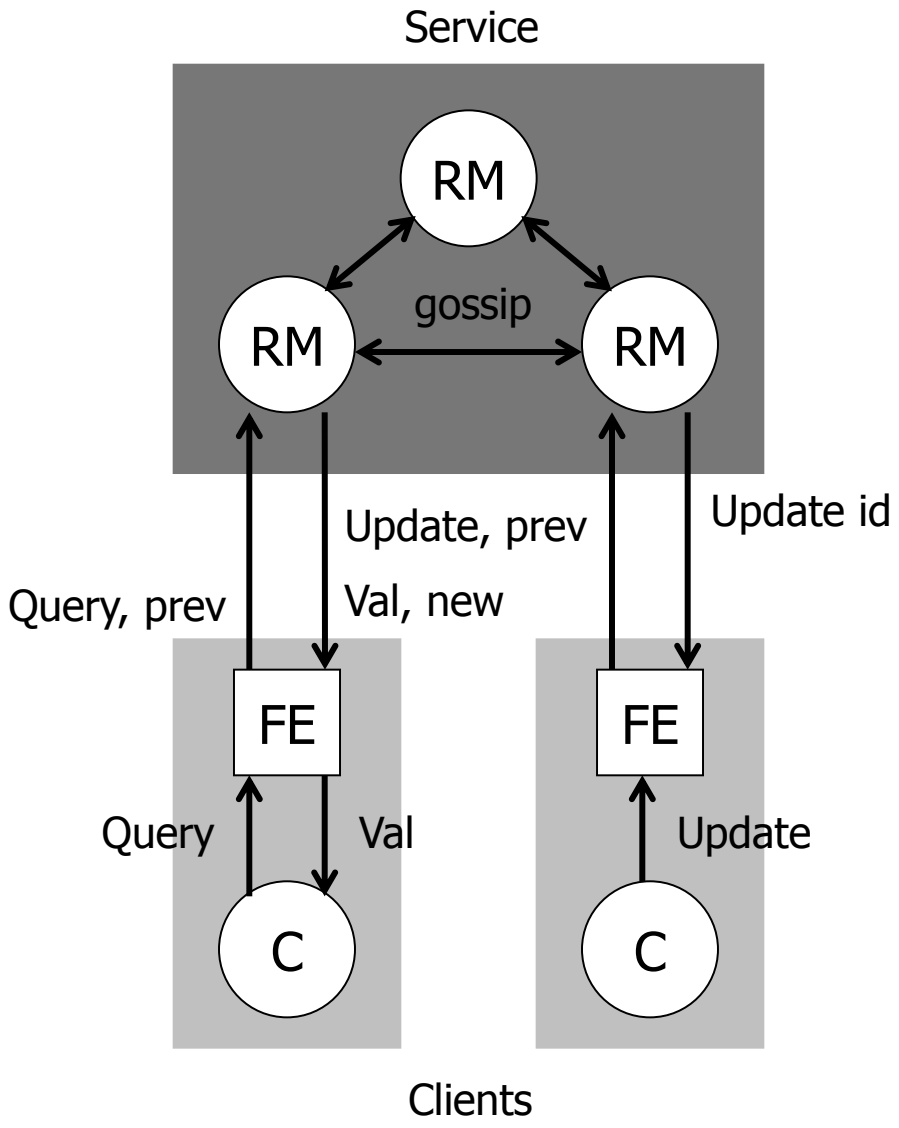
- Gossip supports many different consistency models:
  - From causal update ordering to total and causal (forced) ordering and immediate orderings.

Request Processing

- When a FE issues a request, the following steps are executed:
  - **Request**: The FE sends request to only a single replica manager at a time.
    - FE communicates with different RMs whenever the one it normally uses fails, becomes unreachable, or is overloaded.
    - The FE blocks for queries, but either returns immediately after an update or alternatively, it can be configured to return after f+1 updates.
  - **Update Response:** If the request is an update then the RM replies as soon as it has received the update.
  - **Coordination**: The RM that receives the request does not apply it until the ordering constraints are satisfied.
  - **Execution**: The RM executes the request.
  - **Query Response**: If the request is a query, then the RM replies at this point.
  - **Agreement**: The RM's update each other by exchanging gossip messages, which contain the most recent updates.
    - Regularity of updates can be from every few updates to whenever it finds out that it is missing an update.

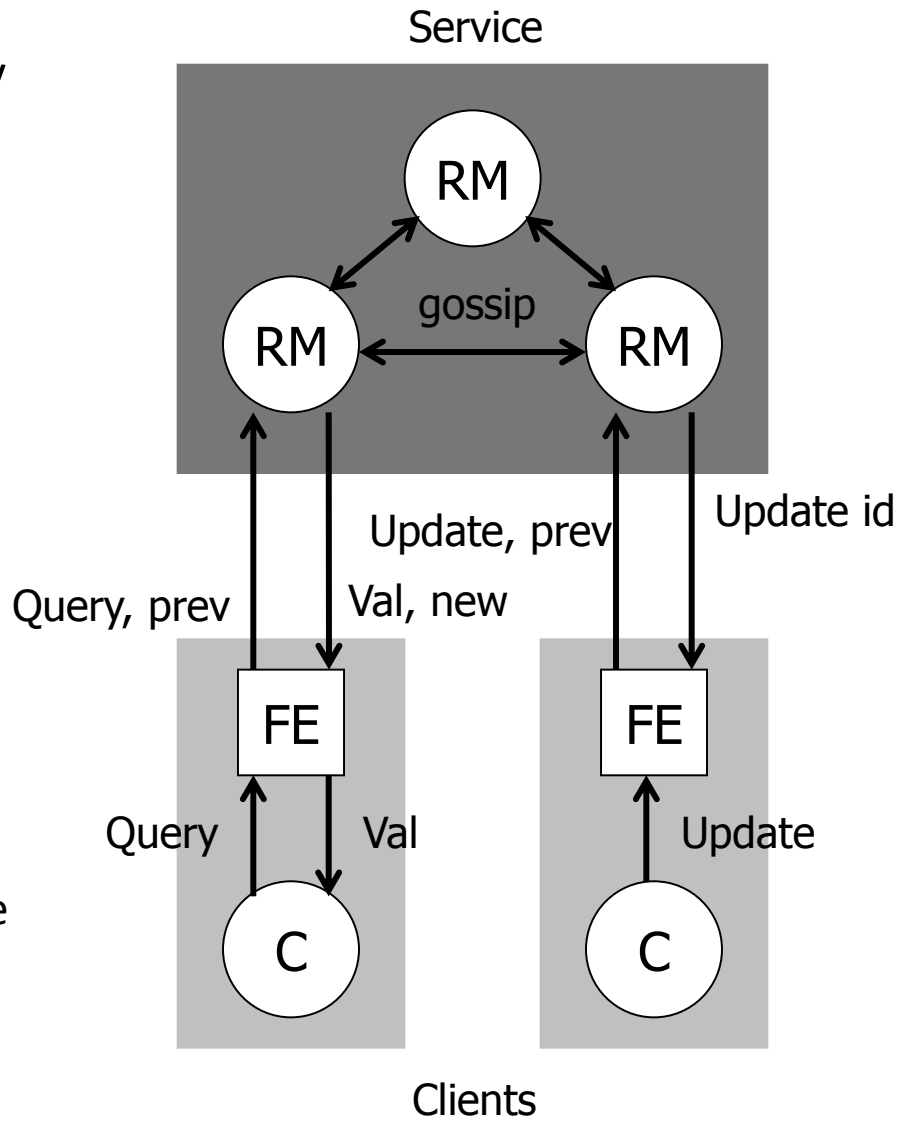
FE Timestamps

- In order to control the ordering of operation processing, the FE keeps a **vector timestamp**:
  - Each RM maintains a last update timestamp
    - Can be the actual time of the last update or linear time
  - Each FE maintains a list of the latest timestamps for each RM.
  - This list is sent in every request that the FE makes.
    - This is **prev** in the diagram



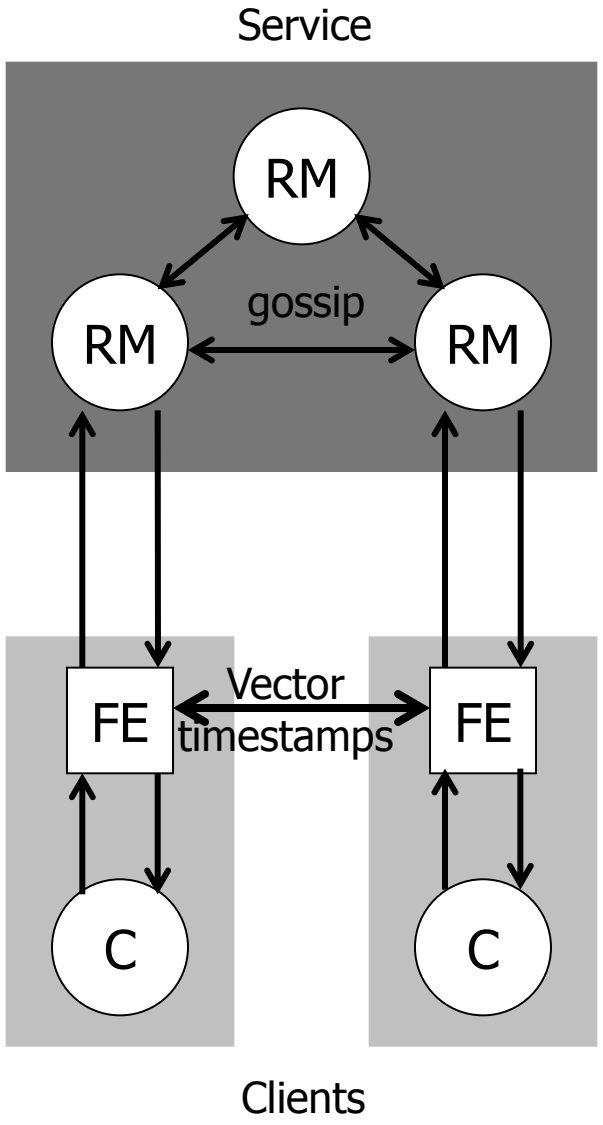
FE Timestamps

- When the FE receives the result of a query operation:
  - The responding RM returns a new vector timestamp.
    - This is **new** in the diagram.
  - Similarly, an update operation returns a vector timestamp that is unique to the update.
    - This is the **update id** in the diagram
- Each returned timestamp is merged with the FE's previous timestamp.
  - This allows the FE to record what version of the replicated data has been observed by the client.



FE Timestamps

- Clients exchange data by:
  - accessing the same gossip service, and
  - communicating directly with one another.
- Since client-to-client communication can also lead to causal relationships between operations it also occurs via the clients FEs
  - This allows FEs to piggyback their vector timestamps on messages to other clients.
- The recipients merge them with their own timestamps in order that causal relationships can be inferred correctly.



Replica Manager State

- Every Replica Manager maintains the following state components:
  - **Value**: The value of the application state as maintained by the RM.
    - Each RM is a state machine
    - Begins with an initial value
    - Subsequent values are derived from the application of update operations to that state.
  - **Value timestamp**: Vector timestamp that represents the updates that are reflected in the value
    - One for each RM
    - Updated whenever an update operation is applied to the value
  - **Update Log**: All update operations are recorded in this log as soon as they are received.
    - Only **stable updates** are applied to the value.
    - Stability of the update depends on the ordering guarantees (causal, forces, immediate)
    - The RM uses the log to ensure that updates are sent to all other RMs.

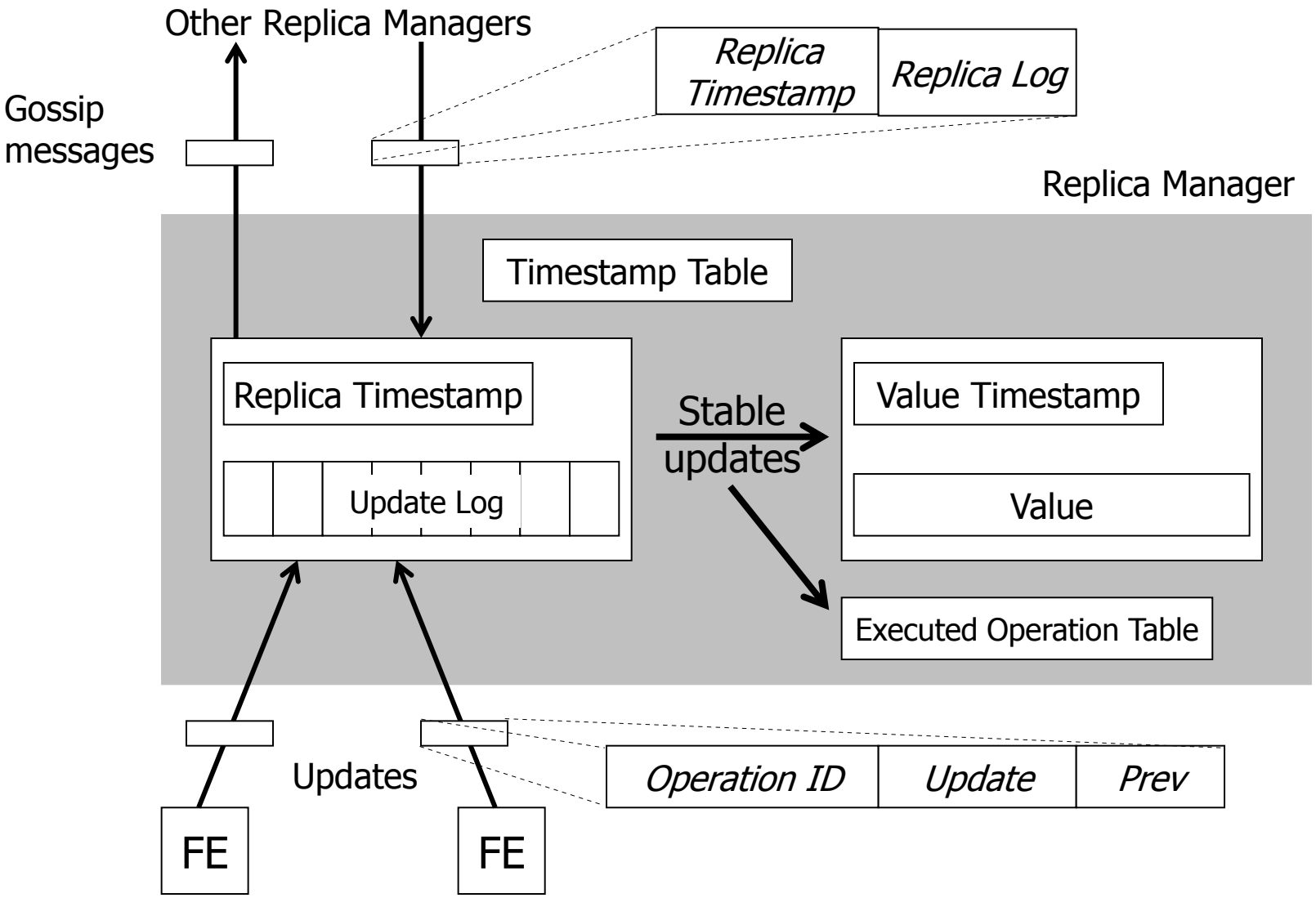
Replica Manager State

- **Replica Timestamp**: Vector timestamp that represents those updates that have been accepted by the RM.
  - That is, those updates that have placed in the managers log.
  - It differs from the value timestamp because not all updates in the log are stable.

- **Executed Operation Table**: A table containing the unique FE supplied identifiers of updates that have been applied to the value.
  - The same update may arrive at a given RM from a FE and in gossip messages from other RM's.
  - To prevent an update being applied twice the RM checks this table before adding an update to the log.

- **Timestamp Table**: Contains a vector timestamp for each other RM.
  - It is filled with timestamps that arrive from them in gossip messages.
  - RMs use the table to establish when an update has been applied to all RMs

Replica Manager State



Query Operations

- The simplest operation to consider is a query.
- A query request contains:
  - A description of the operation
  - A timestamp q.prev sent by the FE
- The latter reflects the latest version of the value that the FE has read or submitted has an update.
- The task of the RM is to return a value that is at least as recent as this.
  - If **valueTS** is the replicas timestamp, then q can be applied to the replica's value if:

$$\mathbf{q.prev} \leq \mathbf{valueTS}$$

Query Operations

- Otherwise, the RM keeps q on a list of pending query operations (a hold-back queue) until this condition is fulfilled.

- Example: Let valueTS be (2, 5, 5) and q.prev be (2, 4, 6)
  - One update is missing (from RM 3)

- It can either:
  - Wait for the missing updates, which should eventually arrive as gossip message, or
  - It can request updates from the RMs concerned.

- Once the query has been applied, the RM returns valueTS to the FE as the timestamp ***new***.
  - The FE merges this with its existing timestamp
  - So, in the above example, the FE will also receive the most recent update from RM2.

Causal Order Update Operations

- FE submits an update request to one or more replica managers.
- Each request u contains:
  - a specification of the update **u.op**,
  - the FE's timestamp **u.prev**, and
  - a unique identifier that the FE generates, **u.id**.
- When a RM, i, receives an update request from a FE:
  - It checks that it has not already processed the request by looking up its identifier in the executed operation table.
  - If not, then it records the update in its log and increments the i^{th} element in its replica timestamp by one.
  - The RM then assigns a unique vector timestamp to u, and a record of the update is placed in the RM's log.

Causal Order Update Operations

- If **ts** is the unique timestamp that the RM assigns to the update, then the update record is the following tuple:

  logRecord := <i, ts, u.op, u.prev, u.id>

- RM i derives the timestamp **ts** from u.prev by replacing u.prev's i^{th} element by the i^{th} element of its replica timestamp.
  - This action makes **ts** unique.
  - The RM immediately returns **ts** to the FE, which it merges with its existing timestamp.

- The FE can send the request to multiple RMs, requiring multiple merges to be carried out on the FE's timestamp.

Causal Order Update Operations

- The stability condition for an update u is:

  u.prev ≤ valueTS

- That is:
  - All the updates upon which this update depends have already been applied to the value.
- If this condition is not met when the update request is submitted, then it is rechecked every time a gossip message is received.
- When the stability condition for an update r has been met, the RM applies the update to the value, updates the value timestamp, and adds the operation to the executed operation table:

  value = apply(value, r.u.op)

  valueTS = merge(valueTS, r.ts)

  executed = executed ∪ {r,.u.id}

Gossip Messages

- RMs send gossip messages to other RMs.
  - Each message contains information concerning one or more updates so that the other RMs can bring their state up to date.
  - Each RM uses the entries in its timestamp table to estimate what updates the other RMs have not yet received.

- A gossip message **m** sent by a RM contains two items:
  - its log, **m.log**, and
  - its replica timestamp **m.ts**

- When an RM receives a message it does three things:
  - Merges the arriving log with its own
  - Applies any updates that have become stable but have not yet been executed
  - Remove records from the log and entries from the executed operation table when it is known that the updates have been applied everywhere and there is no danger of repeats.

Gossip Messages

- The RM's log is updated by merging it with the log in the message.
- Once merged, the RM generates a set S of updates, available in the log, that are now stable.
  - The RM sorts the entries in S using a partial order '≤' relation between timestamps.
  - The updates are then applied in order, with the smallest first.
- Once the updates have been carried out, we set the timestamp for RM j (sender of the gossip message) to be the timestamp of the last message:

  tableTS[j] = m.ts
- We can then check the update log and remove all records that do not satisfy the following constraint for all RMs i:

  tableTS[i][c] ≥ r.ts[c]

Summary

- So, Gossip is a general framework for implementing highly available services.
  - Clients interact with Gossip RMs via a FE
  - The FE chooses a single RM with which to communicate until:
    - RM crashes
    - Communication link failure
    - Overloading.
  - RMs are treated as state machines that can execute two basic operations:
    - Query
    - Update
  - Updates are passed to other RMs through **Gossip Messages**.

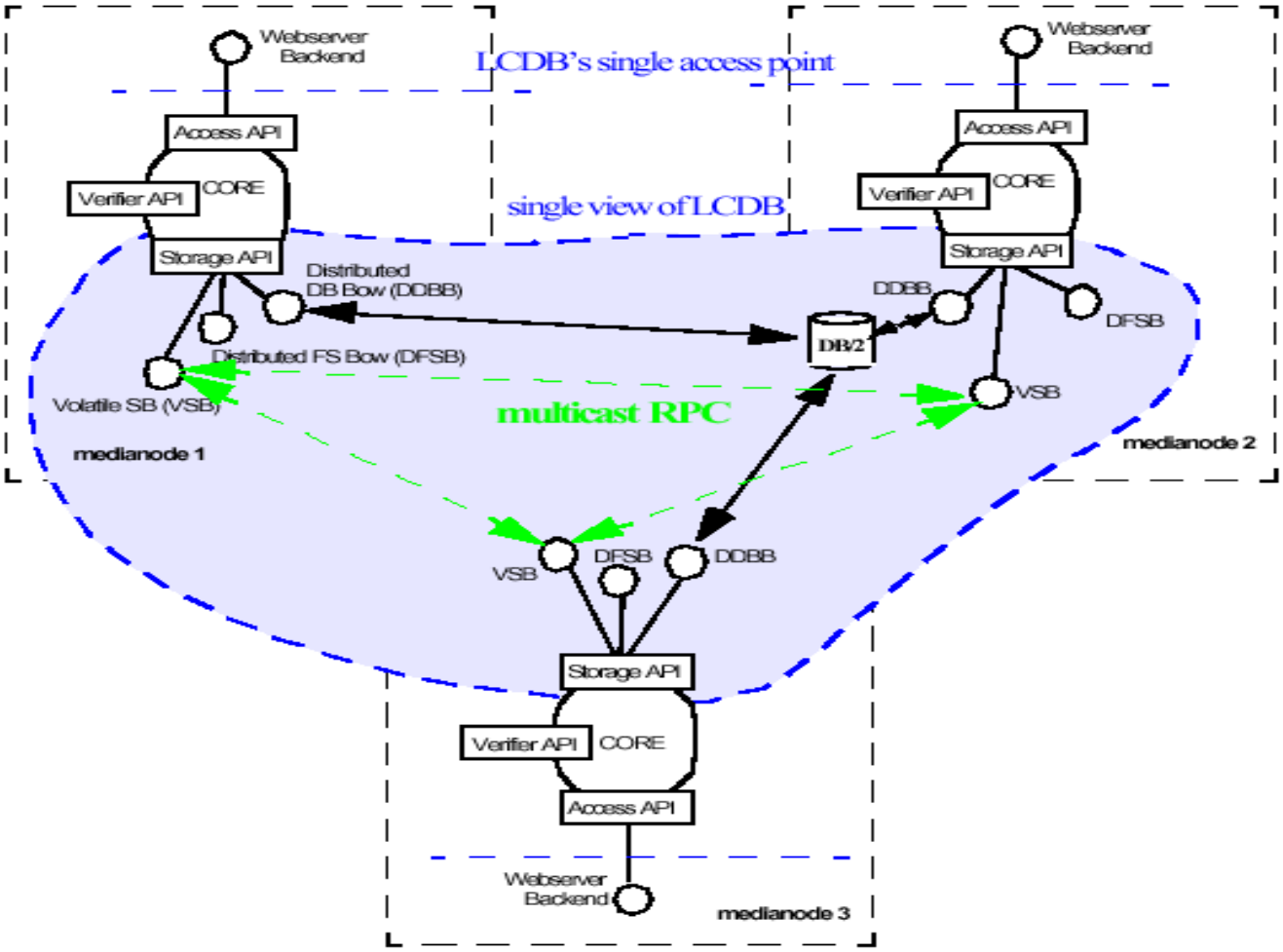
- Issues not covered by the Gossip Architecture are:
  - When to exchange gossip messages
  - How a RM picks who to send messages to.

Distributed Systems:
Case Study: Medianode

Example: Medianode

- An open infrastructure for sharing multimedia-enhanced teaching materials amongst lecture groups.
  - On et al., “Replication for a Distributed Multimedia System”, 2001
- Designed For:
  - High Availability:
    - Connected and Disconnected Modes
    - geographically distributed replicas
  - Consistency: Concurrent updates and failures
  - Location and Access Transparency
  - Cost Efficient Update Transport: Multicast based update mechanism
  - QoS Support

Example: Medianode



Presentation Data Types

- Presentation Contents
- Presentation Description Data (XML)
- Metadata:
  - User, System, Domain and Organisation
  - System resource usage information
  - User session information

Replica Classification

- Metareplicas:
  - Replicated meta-data objects
  - Eg. a list of medianodes that contain up-to-date replicas of a given file
- Softreplicas:
  - Small non-persistent meta-data objects
  - Eg. system resource information, user session information, …
- Truereplicas:
  - Large persistent objects
  - Eg. Media content files

Replication Mechanism

- Replica Manager keep track of a local file table that includes replicas.
- Information about whether and how many replicas are created is contained in every file table.
- Read access is allowed for local replicas.
  - Local cached replica is deemed valid until notified otherwise.
- Updates cause the RM to send a multicast-based update signal to RMs that have replicas.
- Multicast addresses are associated with replica sub-groups.
  - Sub group examples: file directory, or set of presentations about the same topic