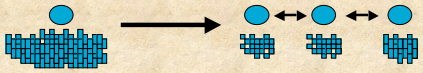
University of Dublin
Trinity College

Replication

Replication

Key to providing good performance, high availability and fault tolerance in distributed systems

- Data shared between a large community should not be held on a single server
- Distribute data copies to several servers
- Arrange that each server provides the data to a smaller community of users



Replication

- Principle goal: Replication Transparency
  - *Clients should not normally be aware that multiple copies of the data exist*
  - *Logical Data Item: set of replicas of data item*
- Secondary goal: Consistency
  - *It is not normally acceptable for different clients to have different views of the same logical data items*
- Two general models
  - *Asynchronous*
  - *Synchronous*
- Two general actions
  - *Read*
  - *Update*

Asynchronous Model

- All client requests are processed by a local replica server
- Local replica server communicates updates to all other servers that manage replicas of the same data
- Client updates can return as soon as the update has been performed locally
  - *Update can be communicated by server to its peers whenever it is convenient*
- Weak consistency
  - *does not guarantee strong ordering of events across replicas*
  - *copies of data will rarely be in-sync*

Synchronous Model

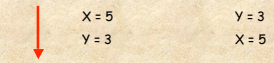
- All update requests are totally ordered
  - *Each update is processed at all replicas in the same order*
- Client 'blocks and locks'
  - *client update request returns only after update has been performed at all replica servers*
- Maintains strict consistency
- Cost of synchronous propogation gives response times and throughput that are worse than that of any single server

Replication in Practice

- Real systems implement an appropriate compromise between simple asynchronous and totally synchronous models
- Totally synchronous model is attractive, but is too expensive
- Employ various techniques and mechanism to achieve a satisfactory trade-off between
  - *consistency*
  - *availability*
  - *response time*

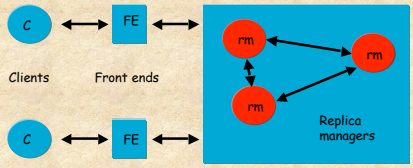
Trade-off techniques and mechanisms

- Quorum based schemes
  - *require some, but not all replicas to be contacted before a read of update request is allowed to complete*
- Causality
  - *allow requests to be processed in different orders at different replicas provided that this does not produce inconsistency*



  - *I.e. Causal ordering - Lamport*
- Multicast
  - *requires hardware support*
  - *reduces cost of maintaining consistency*
  - *updates are sent in messages delivered to a set of processes*

Basic Architectural Model



Basic Architectural Model

- Replicas of data items held by distinct replica managers
- Each replica manager may maintain a replica of every, or just <u>some</u> logical data items
- Client requests are handled by a <u>Front End</u>
  - *Front End communicates with at least one or more replica managers*
  - *Implements replication transparency*
- Front End may be implemented as a user package which executes as a part of each client, or as a separate process

Basic Architectural Model

- Depending on the replication model, the front end will communicate
  - *with only a single replica manager per operation,*
  - *or with several,*
  - *or with all replica managers.*

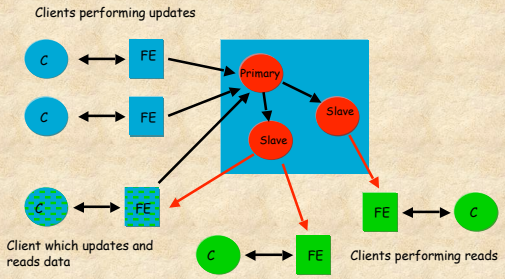
Gossip Architecture

- Client and replica managers are separate processes
- Replica managers periodically exchange 'gossip' messages to convey updates
- Front end normally communicates with an individual replica manager

Primary Copy

- All front ends communicate with the same 'primary' server when <u>updating</u> a particular data item
- Primary server propogates the updates to all other replica managers, called 'slaves'
- Front Ends may <u>read</u> data items from slaves
- If primary server fails, an election algorithm can elect a new primary
- e.g.
  - *In the Sun Network Information Service, information such as password entries (which change relatively infrequently) is updated at a master server and propogated from there to all slave servers.*

Primary Copy



13

Consistency and Request Ordering

- Replicated data must in general be kept consistent
  - *implies that requests should generally be performed in the same order at each site*

X = 5	x = x + 2
x = x + 2	x = 5
7	5 ordering matters!!

- Protocols which guarantee particular ordering can be expensive to implement
  - *generally, a large number of messages have to be transmitted*
- Avoid request ordering wherever possible
- The nature of the replicated data determines whether this is posible

14

Consistency and Request Ordering

- e.g. File System
  - *For read requests r1 and r2 the following result in the same net effect*

    r1 ; r2 r2 ; r1 r1 will be the same in each case

    but writes do not commute

    r1 ; w1 w1 ; r1 r1 will be different in each case
  - *Reads need not be ordered with respect to each other at all sites, but writes must be ordered with respect to reads and with respect to each other*

15

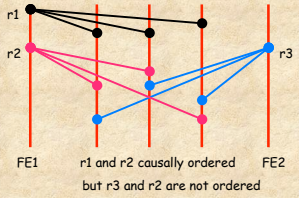
Forms of Ordering

- Total Ordering
  - *for any two events r1 and r2, either r1 is processed before r2 at all sites, or r2 is processed before r1 at all sites*

16

Forms of Ordering

- Causal Ordering
  - *for two requests r1 and r2, if r1 <u>happened-before</u> r2, then r1 is processed before r2 at all sites*

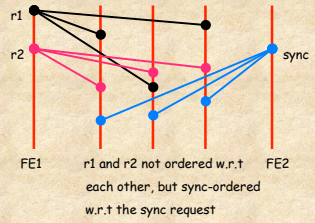


- NB: total ordering is not necessarily causal

17

Forms of Ordering

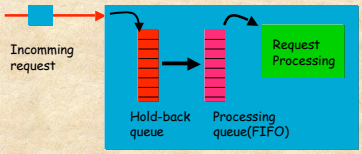
- Sync-Ordering
  - *ensures that every other request is consistently processed at each site either before or after the sync-ordered request.*
  - *Effectively flushes outstanding requests*



18

Implementing Request Ordering

- Hold-Back
  - *A received request is not processed by a replica manager until ordering constraints can be met (put in the holdback queue)*
  - *A request is stable at a replica manager if all prior requests (defined according to the type of ordering in use) have been processed and if the request is ready to be processed next*



Implementing Request Ordering

- Safety
  - *no message will be delivered out of order by being prematurely transferred from the hold-back queue to the processing queue*
- Liveness
  - *no message should wait on the queue indefinitely*

Implementing Request Ordering

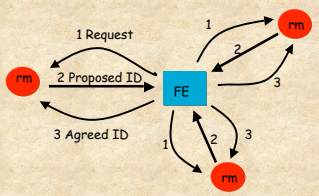
- Sequencer
  - *used to implement total ordering.*
  - *Sequencer assigns identifiers to requests so that each RM makes the same ordering decisions, based upon these identifiers*
  - *All requests are sent to the sequencer in addition to the replica manager*
  - *The sequencer assigns consecutive increasing identifiers to requests as it recieves them and forwards the assigned identifiers to the replica managers*
  - *Requests arriving at a replica manager are held back until they are next in sequence*

Implementing Request Ordering

- *Ideally, generation of identifiers is integrated with the transmission of requests to replica managers*
  - minimise the total number of messages
  - e.g. send request to sequencer, which assigns identifier and multicasts request to replica managers
- Sequencer is a bottleneck/ single failure point!!
- Can we design a distributed sequencer model?

Distributed Model for Sequencing

- Replica Managers propose identifiers for requests as they arrive and return these to Front Ends, which use them to generate final identifiers.



Distributed Model for Sequencing

- Each Replica Manager stores two variables
  - *Fmax : the largest final identifier yet agreed so far*
  - *Pmax : its own largest proposed identifier*
- Algorithm
  - *The Front End sends the request bearing a temporary identifier to all RMs*
    - Temporary identifier chosen to be larger than any identifier used previously by the FE
  - *Each RM replies with a proposed identifier i*
    - i = Max(Fmax, Pmax) + 1 + 1/n, where n is the number of RMs
    - The term 1/n guarantees the value generated for i at each RM to be unique
    - Each RM provisionally assigns its proposed identifier to the request and places it on its hold-back queue
  - *The FE site collects all proposed identifiers and selects the largest one as the next agreed identifier.*

Distributed Model for Sequencing

- *The FE notifies all RMs of the final identifier.*
  - Each RM attaches the final identifier to the request and re-orders the request in the hold-back queue
- *When the request at the front of the hold-back queue has been assigned its final identifier, it is stable*
  - It is then transferred to the tail of the delivery queue
- This algorithm is both safe and live (if adapted to handle failure of FE sites)
- Expensive
  - *2 rounds of messages needed before a request can become stable*
- Not Causal
  - *messages are delivered in an essentially arbitrary order*

Implementing Causal Ordering

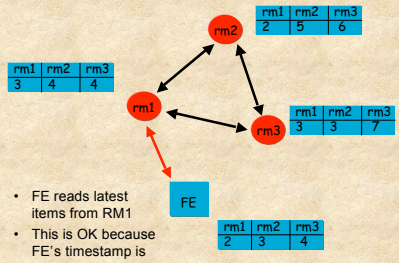
- Associate a timestamp with each replica manager which represents its state
  - *the timestamp can be represented by counts of the update events that lead to the current state*
- Update events can originate
  - *locally (forwarded by FE)*
  - *at another RM*
- We use an array of counts, one for each update source, to represent a replica managers timestamp
  - *called a vector timestamp*

RM1	RM2	RM3
2	5	7

Implementing Causal Ordering

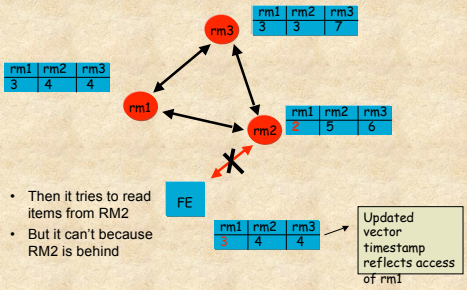
- Front ends maintain a similar vector timestamp reflecting the latest version of the data they have read
- In order to maintain causal ordering
  - *each front end must only read from a version of the data that is at least as advanced as the version from which it last read*
  - *Any updates should be performed only at replicas that already reflect all causally prior updates*

Implementing Causal Ordering



- FE reads latest items from RM1
- This is OK because FE's timestamp is behind that of RM1

Implementing Causal Ordering

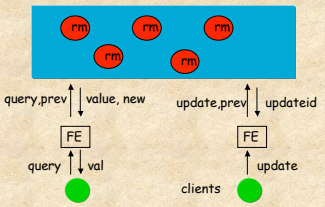


- Then it tries to read items from RM2
- But it can't because RM2 is behind

The Gossip Architecture

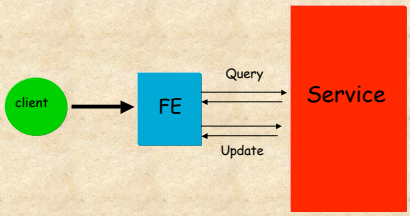
- Based on lazy updates between replication managers
  - *Gossip messages*
- Supports 3 different strengths of update ordering
  - *Causal*
  - *Forced (ie. total + causal)*
  - *Immediate (ie. sync-ordered)*
- Application designer can choose which strength to use
  - *Trade-off between consistency and cost*
- Update ordering is specified for each type of update operation.
  - *The system enforces each ordering*

The Gossip Architecture



- The front-end of a replicated service is application-specific, since it has to handle client operations
  - *Client operations either read , update or both read and update the replicated state*
  - *Inside Gossip, there are only two basic operations (update and read)*

The Gossip Architecture



The Gossip Architecture

- Clients block on query operations
- By default, clients do not block on update operations
  - *FE propogates update as a background task*
- FE keeps a vector timestamp
  - *Reflects the version of the latest values accessed by the front-end.*
  - *Sends this timestamp in every request message to replica managers*
- When a 'query' or 'update' operation returns, a new timestamp vaue is returned (new) to the front-end
- Each returned timestamp is merged with the front-end's current timestamp

The Gossip Architecture

- Clients exchange information in two ways
  - *Accessing the same replicated services*
  - *Directly (or via other services)*
- For causal ordering to be preserved, direct communication between clients must occur via the front ends.
  - *Vector timestamps should be piggybacked on comunication and merged to preserve correct causal relationship inferrence.*

Replica Manager State

- The following state is maintained by the replica manager
- Value: the application data (state)
- Value Timestamp: reflects all updates in data.
  - *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.
- Replica Timestamp: represents those updates that have been accepted by the replica manager (i.e. in the log)
- Executed Call Identifiers: List of identifiers that have been applied to the value (necessary to preserve idempotency).

Replica Manager State

- Note: A replica manager keeps updates in the log for one of two reasons…
- Update is not stable
  - *Log acts as a hold-back queue*
- Replica Manager has not received confirmation that update has been received by all other replica managers.
  - *Some replica managers may ask for the update from this replica.*

Vector Timestamps

- Vector Timestamps are used throughout

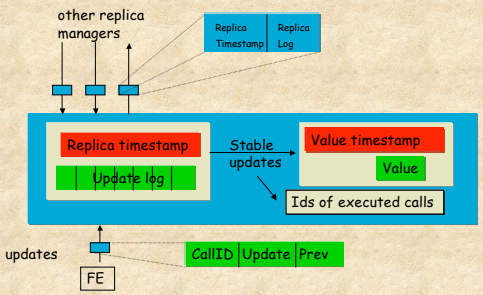
RM1	RM2	RM3	RM4
2	5	4	4

- On RM1
  - *The replica manager received 2 updates from front-ends, and 5 updates from RM2, 4 from RM3 and 4 from RM4*
- If RM1 is passed a new timestamp from a front-end…

RM1	RM2	RM3	RM4
2	5	3	6

…this implies that RM1 has to get 2 missing updates from RM4

Gossip Replica Manager



Gossip Messages

- Normally contain information concerning several updates…
  - *Replica timestamp*
  - *Log of updates*
- Receiving replica manager has 4 tasks
  - *Merge the arriving log with its own*
    - Add each log entry (r) unless its already in the log, or if the log entries timestamp, r.ts <= value timestamp
  - *Apply any updates that have become stable*
    - This may lead to an avalance effect
  - *Eliminate log entries and entries in the list of executed call identifiers when it is known that updates have been applied everywhere*
  - *Merge timestamp of incomming gossip message with replica managers own replica timestamp*

Processing Query Operations

- A query q contains a description of the operation, and a timestamp q.prev sent by the front-end
  - *Q.prev reflects the latest version of the value that the front-end as read or submitted an update to*
- The replica manager must return data at least as up to date as indicated by q.prev
  - *Q.prev <= Value Timestamp*
- The replica manager keeps q on the hold-back queue until this condition is satisfied
  - *It can await the missing updates or it can explicitly request updates from replica managers*
- Eventually (liveness), the replica manager's value will satisfy the condition and the query can complete

Processing Update Operations (causal)

- On receiving an update request, the replica manager increments its own element in its Replica Timestamp.
  - *This keeps count of the number of updates it has received directly from front-ends*
- The update is assigned a unique identifierand a record for the update is placed in the replica manager's log

<TS, u.op, u.prev>

unique identifier

update operation

front-end timestamp

- TS is derived from u.prev by replacing its i th element by that of the replica manager timestamp, assuming replica manager *i* received the update from the front-end
  - *TS is passed back to the front-end immediately, and is merged with the front-ends timestamp*

Processing Update Operations (causal)

- The update *u* is stable when…

u.prev <= value timestamp

ie. Stable when all updates on which this update depends have already been applied

- If u.prev > value timestamp, update is not performed
  - *When new gossip messages arrive, and value timestamp changes, the test will be performed again*
- Eventually, the update will be performed
  - *Its call identifier will be added to the lst of executed calls*
  - *The value timestamp will be updated to reflect this new update*

Causal Ordering: an aside

- Consider the following scenario where two front-ends perform an update in parallel to two different replica managers
  - *Initial timstamps at replica managers are illustrated below*

RM1	RM1	RM2	RM3
	2	2	2

RM timestamps

RM2	RM1	RM2	RM3
	2	2	2

… and following the updates their state will look like so…

RM1	**RM1**	RM2	RM3
	3	2	2

RM timestamps

RM2	RM1	**RM2**	RM3
	2	**3**	2

Causal Ordering: an aside

- Causal ordering: front-ends only read from replica managers at least as advanced as themselves, and new items should be added to a value only when all causally prior updates are stable and previously executed
- Question: Which update (that performed on RM1 or RM2) was causally prior?

Causal Ordering: an aside

- At some point, the update performed at RM2 will be transmitted to RM1 in a gossip message.
- This update is added to the log at RM1 iff…
  - *It isn't already in the log (we know this to be the case)*
  - *R.ts > value timestamp*

r.ts	RM1	**RM2**	RM3
	2	**3**	2

RM1	**RM1**	RM2	RM3
	3	2	2

- So we add the update!
  - *Note: The update received from the front-end at RM1 was perfomed there before the update that was received in the gossip message*

Causal Ordering: an aside

- At some point, the update performed at RM1 will be transmitted to RM2 in a gossip message.
- This update is added to the log at RM2 iff…
  - *It isn't already in the log (we know this to be the case)*
  - *R.ts > value timestamp*

r.ts	**RM1**	RM2	RM3
	3	2	2

RM2	RM1	**RM2**	RM3
	2	**3**	2

- So we add the update!
  - *Note: The update received from the front-end at RM2 was perfomed there before the update that was received in the gossip message*

Causal Ordering: an aside

- At RM1, UFE1 precedes UFE2
- At RM2 UFE2 precedes UFE1
- Why?
  - *Two events are not causally related*
- If we wish such events to be ordered consistently we must enforce a stronger ordering policy

Forced Updates

- Total ordering is not necessarily also causal
  - *But Gossip implements causal ordering as a minimum*
  - *To implement total ordering we must implement total + causal ordering*
- Forced ordering guarantees an ordering of events, even if they are not causally related
- Append unique sequence number to timestamps associated with forced updates
  - *Process the updates in order of this sequence*

Distributed Sequencer Model

- Centralised Sequencer server is inappropriate
- Replica manager is designated as a sequencer
  - *Called the primary*
- Ensure new sequencer can be elected if necessary
- If majority of replica managers record which update is next in sequence, then the operation can be applied
- As long as the majority of replica managers survive, this ordering decision will be honoured by a new primary elected from the surviving replicas

49

Immediate Updates

- Sync-ordered updates
- Ordering w.r.t forced updates by using the primary replica manager to order them
- Primary must also determine which causal updates are deemed to have preceded an immediate update
  - *Must communicate and synchronise with other replica manager to come to a decision*
- EXPENSIVE!!!

50