Teaching material based on Distributed Systems: Concepts and Design, Edition 3, Addison-Wesley 2001.

Distributed Systems

Replication

George Coulouris
and
Jean Dollimore

Modified for
TIE370 Distributed Systems
by
Mikko Vapa

Contents

- Introduction to replication
- System model
  - logical objects, physical copies (replicas), replica managers, front ends
- Group communication
  - membership management and view delivery
- Fault-tolerant services
  - linearizability, sequential consistency
- Passive (primary-backup) replication model
  - primary and backup replica managers
- Active replication model
  - equal replica managers and operations delivered using multicast

Introduction to replication

Replication of data = the maintenance of copies of data at multiple computers

- replication can provide the following
- performance enhancement
  - e.g. several web servers can have the same DNS name and the servers are selected in turn for load-balancing or web browser caches recently retrieved data to avoid delay of refetching the same document
  - replication of read-only data is simple, but replication of changing data has overheads
- fault-tolerant service
  - guarantees correct behaviour in spite of certain faults
  - if *f* of *f*+1 servers crash then 1 remains to supply the service
  - if *f* of 2*f*+1 servers have byzantine/arbitrary faults then they can supply a correct service

3

Introduction to replication

- availability is hindered by
  - server failures
    - replicate data at failure-independent servers and when one fails, client may use another
    - note that caches do not help with availability (they are incomplete)
  - network partitions and disconnected operation
    - users of mobile computers deliberately disconnect, and then on re-connection, resolve conflicts

4

Requirements for replicated data

- replication transparency
  - clients see logical objects (not several physical copies)
    - they access one logical item and receive a single result
- consistency
  - specified to suit the application,
    - e.g. when a user of a diary disconnects, their local copy may be inconsistent with the others and will need to be reconciled when they connect again
    - but connected clients using different copies should get consistent results
    - these issues are addressed for example in Bayou and Coda file systems

5

System model for replication

- each *logical* object is implemented by a collection of *physical* copies called *replicas*
  - the replicas are not necessarily consistent all the time (some may have received updates, not yet conveyed to the others)
- we assume an asynchronous system where processes fail only by crashing and generally assume no network partitions
- replica managers
  - an RM contains replicas on a computer and access them directly
  - RMs apply operations to replicas recoverably
    - i.e. they do not leave inconsistent results if they crash
  - objects are copied at all RMs unless stated otherwise
  - static systems are based on a fixed set of RMs
  - in a dynamic system: RMs may join or leave (e.g. when they crash)

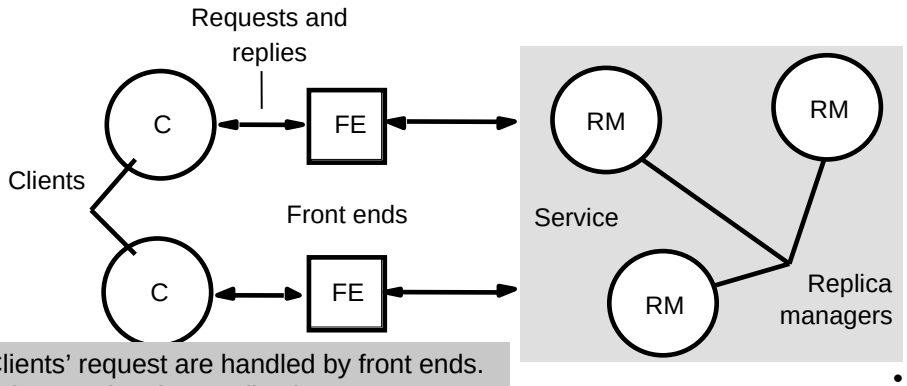
6

A basic architectural model for the management of replicated data

A collection of RMs provides a service to clients

Clients see a service that gives them access to logical objects, which are in fact replicated at the RMs

Clients request operations: those without updates are called *read-only* requests the others are called *update* requests (they may include reads)



Clients' request are handled by front ends. A front end makes replication transparent.

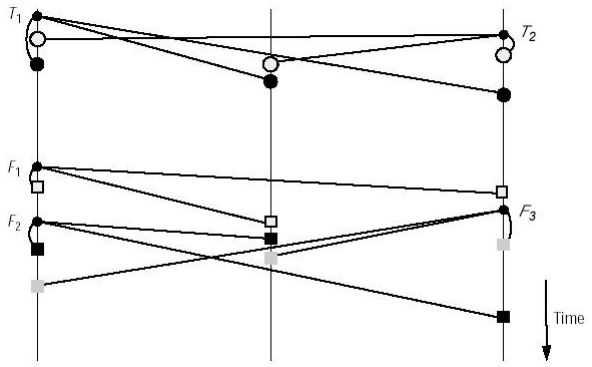
Five phases in performing a request

- issue request
  - the FE either
    - sends the request to a single RM that passes it on to the others
    - or multicasts the request to all of the RMs
- coordination
  - the RMs decide whether to apply the request; and decide on its ordering relative to other requests (according to FIFO or total ordering)
- execution
  - the RMs execute the request (sometimes tentatively waiting for commit)
- agreement
  - RMs *agree* on the effect of the request, e.g., perform 'lazily' or immediately
- response
  - one or more RMs reply to FE. e.g.
    - for high availability give first response to client
    - to tolerate byzantine faults, take a vote

8

FIFO and Total Ordering

- T_1 and T_2 are totally ordered
- F_1, F_2 and F_3 are FIFO ordered



9

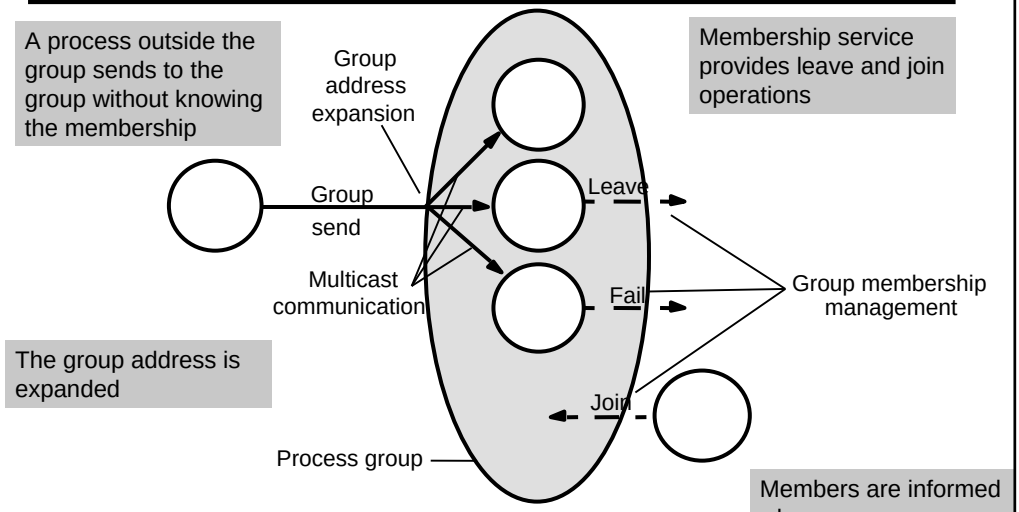
Group communication

Membership service is required to allow dynamic membership of groups

- process groups are useful for managing replicated data
  - but replication systems need to be able to add/remove RMs
- group membership service provides:
  - interface for adding/removing members
    - create, destroy process groups, add/remove members
    - a process can generally belong to several groups
  - implements a failure detector
    - which monitors members for failures (crashes/communication),
    - and excludes them when unreachable
  - notifies members of changes in membership
  - expands group addresses
    - multicasts addressed to group identifiers
    - and group identifier expanded to a list of delivery addresses
    - coordinates delivery when membership is changing
- e.g. IP multicast allows members to join/leave and performs address expansion, but not the other features

10

Services provided for process groups



Group views and network partitions

- Group views
  - a full membership service maintains group views
  - lists of current group members generated each time a process joins or leaves the group
  - if process "suspected" to have failed
    - exclusion from group view
    - if process not failed, or recovered, it needs to re-join group
    - false suspicion reduces effectiveness of group
- Network partitions
  - occur when routers or links fail such that two subnets emerge that are no longer connected to each other
  - group management in the presence of partitions
    - *primary-partition*: at most one sub-group survives, remaining processes told to suspend
    - *partitionable*: subgroups survive as independent multicast groups

12

View delivery

- the idea is that processes can ‘deliver views’
- each member of a group is treated in a consistent way when group membership changes
- necessary to relieve programmer from querying state of all other group members before making a send decision
- group management service delivers sequence of views to members, e.g.
  - $v_0(g) = \{p\}$, $v_1(g) = \{p, p'\}$, $v_2(g) = \{p\}$, ...
- system imposes an ordering on the possibly concurrent view changes
- receiving/delivering a view
  - queue in hold-back queue as for multicast until all members agree to deliver the view

13

View delivery

- properties of view delivery
  - order: if some process delivers two views in some order, then all processes in the group will do so
  - integrity: if some p delivers view of group g, then $p \in g$
  - non-triviality:
    - if q joins group and becomes indefinitely reachable, then q will eventually be included in all views
    - if the group partitions, then eventually views delivered in one partition will exclude views delivered in other partitions

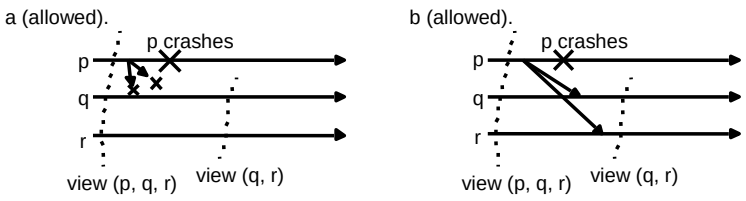
14

View-synchronous group communication

- extends reliable multicast semantics to view delivery
  - guarantees not only the above properties for view delivery, but also includes guarantees on the delivery of multicast messages
- for simplicity the possibility of network partitioning is excluded
- guarantees that
  - all processes agree on the ordering of messages and membership changes (the same set of messages is delivered in any given view),
  - a joining process can safely get state from another member,
  - if system fails to deliver a message to any process q
    - immediately notifies surviving processes by delivering view that excludes q
    - hence, if for the next view $q \notin view'(g)$, then p knows that q has failed
- implemented in the ISIS system [Birman 1993]

View-synchronous group communication

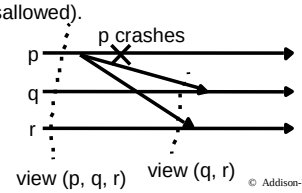
- example: p sends m while in view {p, q, r}, p crashes soon after sending m
  - a) If p crashes before m reaches any of q and r, then q and r each deliver new view {q, r} and neither delivers m
  - b) m has reached at least one of q and r before p crashes, then q and r deliver first m, and then view {q, r}



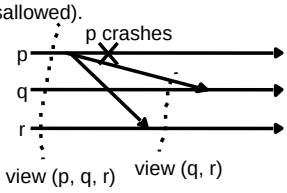
View-synchronous group communication

- example: p sends m while in view {p, q, r}, p crashes soon after sending m
  - c) not allowed for q and r to first deliver view {q, r} and then m, since this would mean delivering a message from a failed process
  - d) not allowed to deliver the message and then the view in opposite order, for the same reason

c (disallowed).



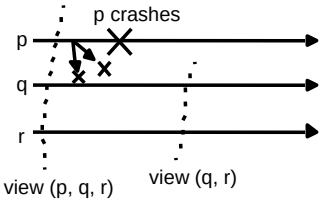
d (disallowed).



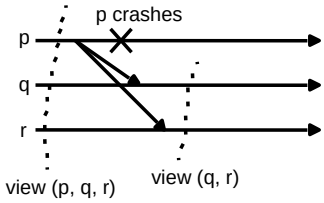
17

View-synchronous group communication

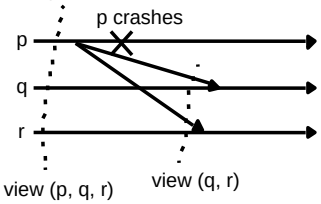
a (allowed).



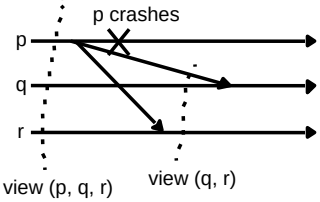
b (allowed).



c (disallowed).



d (disallowed).



18

Fault-tolerant services

- provision of a service that is correct even if *f* processes fail
  - by replicating data and functionality at RMs
  - assume communication reliable and no partitions
  - RMs are assumed to behave according to specification or to crash (so no byzantine failures are allowed)
  - intuitively, a service is correct if it responds despite failures and clients can't tell the difference between replicated data and a single copy
  - but care is needed to ensure that a set of replicas produce the same result as a single one would

Example of a naive replication system

Client 1:	Client 2:
$setBalance_B$(x,1)	
$setBalance_A$(y,2)	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

RMs at *A* and *B* maintain copies of *x* and *y*
- clients use local RM when available, otherwise the other one
- RMs propagate updates to one another after replying to client

- initial balance of x and y is $0
  - client 1 updates X at B (local) then finds B has failed, so uses A
  - client 2 reads balances at A (local)
    - as client 1 updates y after x, client 2 should see $1 for x
  - not the behaviour that would occur if A and B were implemented at a single server
- systems can be constructed to replicate objects without producing this anomalous behaviour
- we now discuss what counts as correct behaviour in a replication system

Linearizability - the strictest criterion for a replication system

- the correctness criteria for replicated objects are defined by referring to a virtual interleaving which would be correct

Consider a replicated service with two clients, that perform read and update operations. A client waits for one operation to complete before doing another. Client operations o_{10}, o_{11}, o_{12} and o_{20}, o_{21}, o_{22} at a single server are interleaved in some order e.g. o_{20}, o_{21}, o_{10}, o_{22} , o_{11}, o_{12} (client 1 does o_{10} etc.)

- for any set of client operations there is a virtual interleaving (which would be correct for a set of single objects)
- each client sees a view of the objects that is consistent with this, that is, the results of clients operations make sense within the interleaving
  - the bank example did not make sense: if the second update is observed, the first update should be observed too

21

Linearizability - the strictest criterion for a replication system

- a replicated object service is *linearizable* if for any execution there is some interleaving of clients' operations such that:
  - the interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - the order of operations in the interleaving is consistent with the real time at which they occurred
- the real-time requirement means clients should receive up-to-date information
  - but may not be practical due to difficulties of synchronizing clocks
  - a weaker criterion is sequential consistency

22

Sequential consistency

- a replicated shared object service is *sequentially consistent* if for any execution there is some interleaving of clients' operations such that:
  - the interleaved sequence of operations meets the specification of a (single) correct copy of the objects
  - the order of operations in the interleaving is consistent with the program order in which each client executed them

The following execution is sequentially consistent but not linearizable

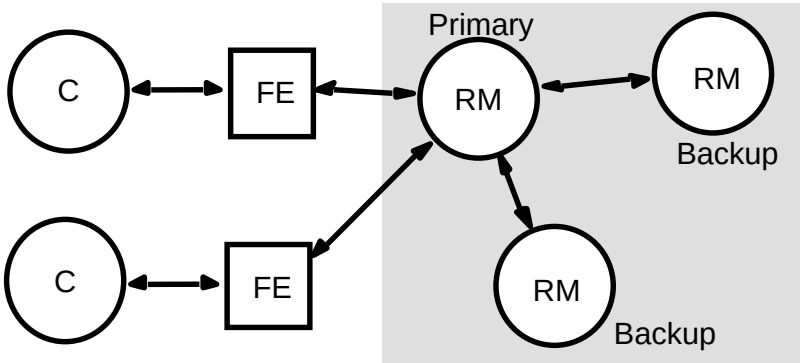
Client 1:	Client 2:
$setBalance_B(x,1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y,2)$	

this is possible under a naive replication strategy, even if neither *A* or *B* fails - the update at *B* has not yet been propagated to *A* when Client 2 reads it

it is not linearizable because Client 2's *getBalance* is after Client 1's *setBalance* in real time.

23

The passive replication model for fault tolerance



- there is at any time a single primary RM and one or more secondary (backup, slave) RMs
- FEs communicate with the primary which executes the operation and sends copies of the updated data to the result to backups
- if the primary fails, one of the backups is promoted to act as the primary

the FE has to find the primary, e.g. after the primary crashes and another takes over

Passive (primary-backup) replication

- the five phases in performing a client request are as follows:
- 1. Request:
  - a FE issues the request, containing a unique identifier, to the primary RM
- 2. Coordination:
  - the primary takes each request atomically, in the order in which it receives it relative to other requests
  - it checks the unique id; if it has already done the request it re-sends the response
- 3. Execution:
  - the primary executes the request and stores the response
- 4. Agreement:
  - if the request is an update the primary sends the updated state, the response and the unique identifier to all the backups and the backups send an acknowledgement
- 5. Response:
  - the primary responds to the FE, which hands the response back to the client

25

Passive (primary-backup) replication

- this system implements linearizability, since the primary sequences all the operations on the shared objects
- if the primary fails, the system is linearizable, if a single backup takes over exactly where the primary left off, i.e.:
  - the primary is replaced by a unique backup
  - surviving RMs agree which operations had been performed at take over
- view-synchronous group communication can achieve this
  - when surviving backups receive a view without the primary, they use an agreed function to calculate which is the new primary
  - the new primary registers with name service
  - view synchrony also allows the processes to agree which operations were performed before the primary failed
  - e.g., when a FE does not get a response, it retransmits it to the new primary
  - the new primary continues from phase 2 (coordination) by using the unique identifier to discover whether the request has already been performed

26

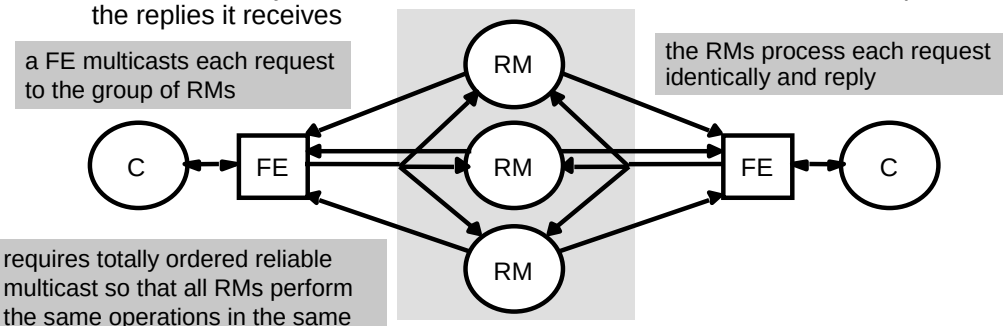
Passive (primary-backup) replication

- to survive *f* process crashes, *f*+1 RMs are required
  - it cannot deal with byzantine failures because the client can't get replies from the backup RMs
- to design passive replication that is linearizable
  - view-synchronous communication has relatively large overheads
  - several rounds of messages per multicast
  - after failure of primary, there is latency due to delivery of group view
- variant in which clients can read from backups
  - which reduces the work for the primary
  - get sequential consistency but not linearizability
- Sun NIS uses passive replication with weaker guarantees
  - weaker than sequential consistency, but adequate to the type of data stored
  - achieves high availability and good performance
  - master receives updates and propagates them to slaves communication
  - for reading clients can use either master or slave
  - updates are not done via RMs - they are made on the files at the master

27

The active replication model for fault tolerance

- the RMs are *state machines* all playing the same role and organised as a group
  - all start in the same state and perform the same operations in the same order so that their state remains identical
- if RM crashes it has no effect on performance of the service because the others continue as normal
- it can tolerate byzantine failures because the FE can collect and compare the replies it receives



28

Active replication – five phases for a client request

- Request
  - FE attaches a unique *id* and uses *totally ordered reliable multicast* to send request to RMs
  - FE can at worst, crash and it does not issue requests in parallel
- Coordination
  - the multicast delivers requests to all the RMs in the same (total) order.
- Execution
  - every RM executes the request and the *id* is put in the response
  - they are state machines and receive requests in the same order, so the effects are identical
- Agreement
  - no agreement is required because all RMs execute the same operations in the same order, due to the properties of the totally ordered multicast
- Response
  - FEs collect responses from RMs and they may use one or more responses: if it is only trying to tolerate crash failures, it gives the client the first response

29

Active replication

- as RMs are state machines we have sequential consistency
  - due to reliable totally ordered multicast, the RMs collectively do the same as a single copy would do
  - it works in a synchronous system
  - in an asynchronous system reliable totally ordered multicast is impossible – but failure detectors can be used to work around this problem (how to do that is beyond the scope of this course)
- this replication scheme is not linearizable
  - because total order is not necessarily the same as real-time order
- to deal with byzantine failures
  - for up to *f* byzantine failures, use 2*f*+1 RMs
  - FE collects f+1 identical responses
- to improve performance,
  - FEs send read-only requests to just one RM

30

Summary for Replication

- replicating objects helps services to provide good performance, high availability and fault tolerance
- system model - each logical object is implemented by a set of physical replicas
- linearizability and sequential consistency can be used as correctness criteria
  - sequential consistency is less strict and more practical to use
- fault tolerance can be provided by:
  - passive replication - using a primary RM and backups,
    - but to achieve linearizability when the primary crashes, view-synchronous communication is used, which is expensive
    - less strict variants can be useful
  - active replication - in which all RMs process all requests identically
    - needs totally ordered and reliable multicast, which can be achieved in a synchronous system