

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



Distributed Systems Course

Replication

Copyright © George
Coulouris, Jean Dollimore,
Tim Kindberg 2001
email: authors@cdk2.net

This material is made
available for private study
and for direct use by
individual teachers.

It may not be included in any
product or employed in any
service without the written
permission of the authors.

**Viewing: These slides
must be viewed in
slide show mode.**

14.1 Introduction to replication

14.2 System model and group
communication

14.3 Fault-tolerant services

14.4 Highly available services

14.4.1 Gossip architecture

14.5 Transactions with replicated data

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. To share the load.
- replication of data to multiple servers can reduce the latency of data access. e.g. a user on a train with a laptop with no access to a network will prepare by copying data to the laptop, e.g. a shared diary. If they update the diary they risk missing updates by other people.
- replication of data to multiple servers can reduce the overheads of data access.

✍ fault-tolerant

- guarantees correct data
- if f of $f+1$ servers crash then f remains to supply the service
- if f of $2f+1$ servers have byzantine faults then they can supply a correct service


✍ 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


Requirements for replicated data

What is replication transparency?

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 in Bayou and Coda.

14.2.1

State machine

- ✍ applies operations atomically
- ✍ its state is a deterministic function of its initial state and the operations applied
- ✍ all replicas start identical and carry out the same operations
- ✍ its operations must not be affected by clock readings etc.

✍ 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

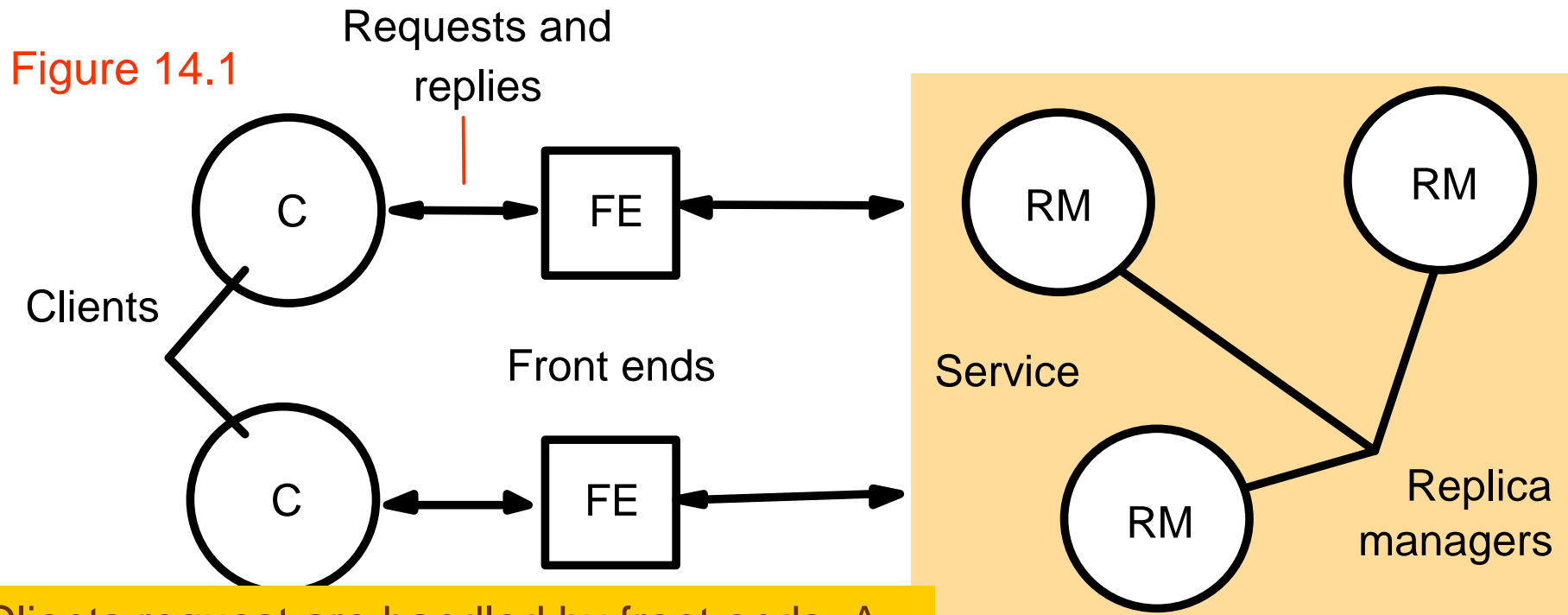
- an RM contains replicas on a computer and access them directly
- RMs apply operations to replicas recoverably
 - ✍ i.e. they do not leave inconsistent results if they crash
- objects are copied at all RMs unless we state otherwise
- static systems are based on a fixed set of RMs
- in a dynamic system: RMs may join or leave (e.g. when they crash)
- an RM can be a *state machine*, which has the following properties:

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.

What can the FE hide from a client?

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 (in state machine approach)

✍ coordination

- the RMs decide whether to apply the request; and decide on its ordering relative to other requests (according to FIFO, causal or total ordering)

Total ordering: if a correct RM handles r before r' , then any correct RM handles r before r'

Bayou sometimes executes responses tentatively so as to be able to reorder them

- RMs *agree* on the effect of the request, .e.g perform lazily or immediately

✍ response

RMs agree - I.e. reach a consensus as to effect of the request. In Gossip, all RMs eventually receive updates.

- ✍ to tolerate byzantine faults, take a vote

14.2.2 Group communication

We require a membership service 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 (section 11.1 - not studied in this course)

- ✍ which monitors members for failures (crashes/communication),

- ✍ and excludes them when unreachable

- notifies members of changes in membership

- expands group addresses

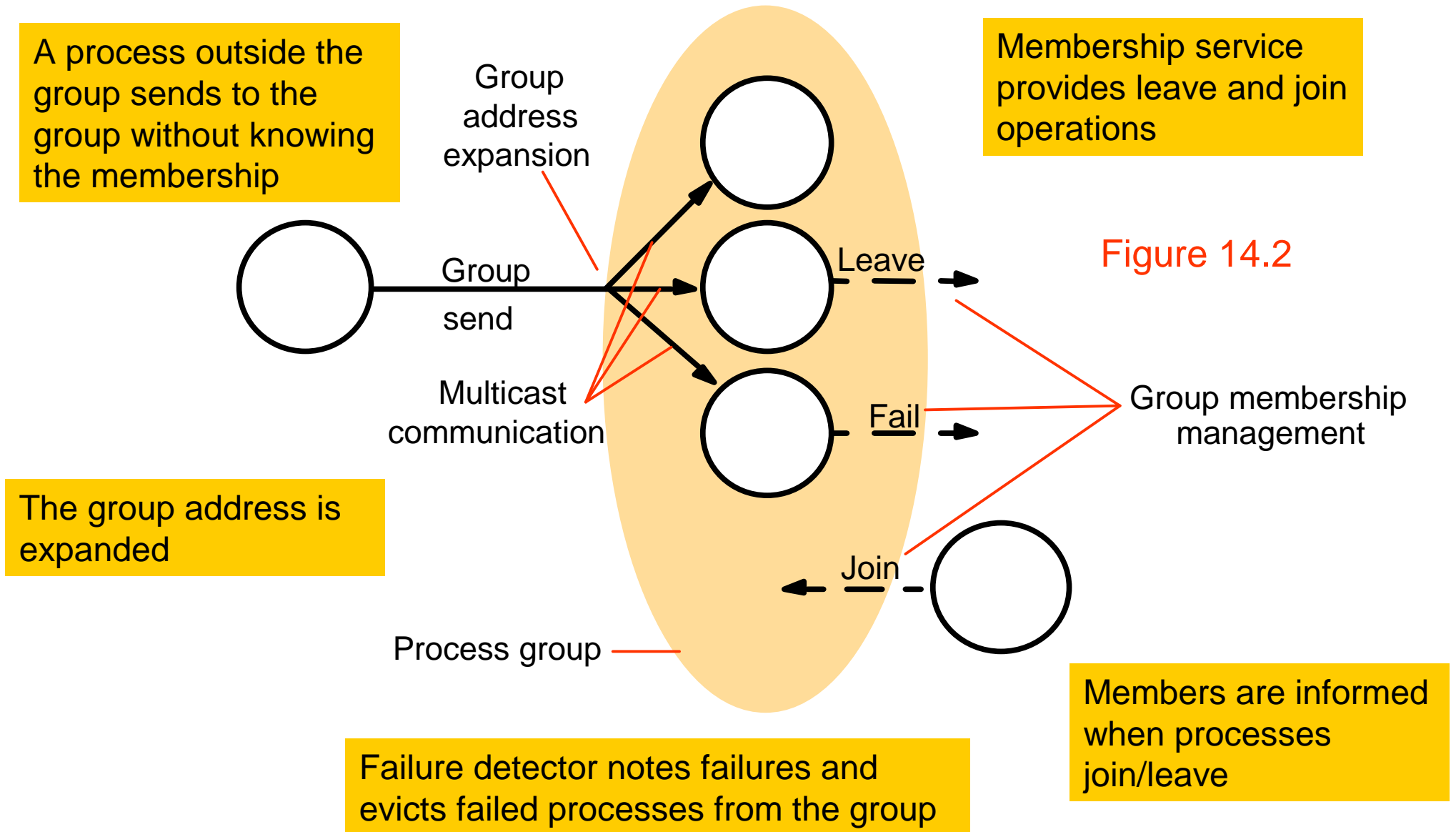
- ✍ multicasts addressed to group identifiers,

- ✍ coordinates delivery when membership is changing

✍ e.g. IP multicast allows members to join/leave and performs address expansion, but not the other features

Section 11.4 discussed multicast communication (also known as group communication there we took group membership to be static (although members may crash))

Services provided for process groups



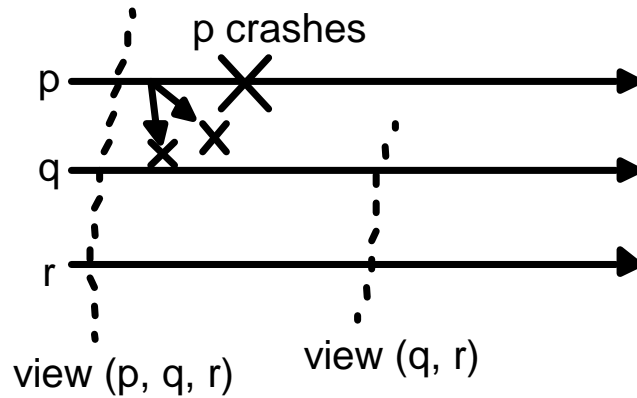
We will leave out the details of view delivery and view synchronous group communication

- ✍ A full membership service maintains *group views*, which are lists of group members, ordered e.g. as members join group.
- ✍ A new group view is generated each time a process joins or leaves the group.
- ✍ *View delivery* p 561. The idea is that processes can 'deliver views' (like delivering multicast messages).
 - ideally we would like all processes to get the same information in the same order relative to the messages.
- ✍ *view synchronous group communication* (p562) with reliability.
 - Illustrated in Fig 14.3
 - all processes agree on the ordering of messages and membership changes,
 - a joining process can safely get state from another member.
 - or if one crashes, another will know which operations it had already performed
 - This work was done in the ISIS system (Birman)

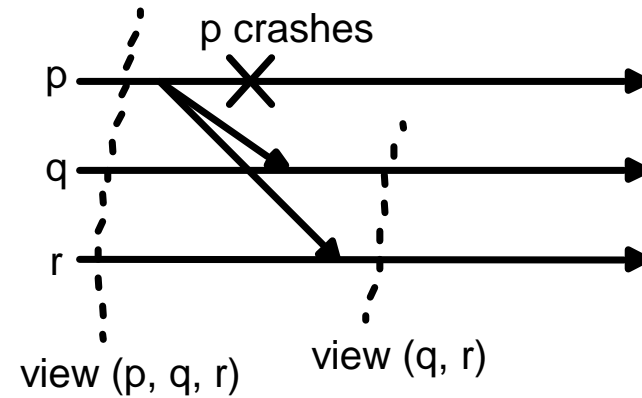
Figure 14.3

View-synchronous group communication

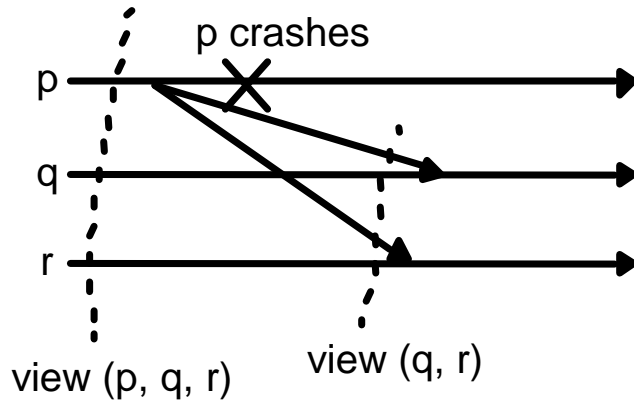
a (allowed).



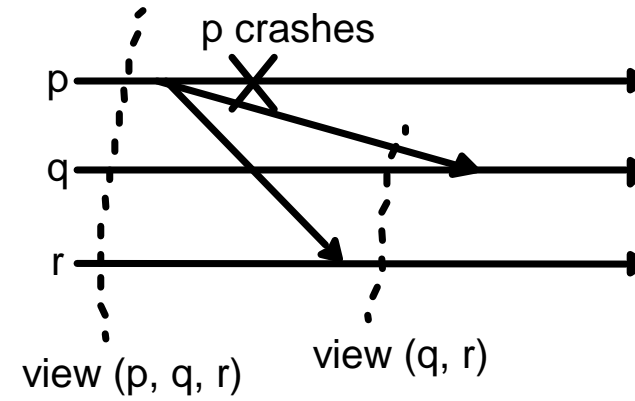
b (allowed).



c (disallowed).



d (disallowed).



14.3 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
- 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.

- e.g (next slide).

Example of a naive replication system

Client 1:	Client 2:
$setBalance_B(x,1)$	
$setBalance_A(y,2)$	
	$getBalance_A(y) ? ?$
	$getBalance_A(x) ? ?$

RM's at A and B maintain copies of x and y
clients use local RM when available, otherwise the other one
RM's 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 (p566) the strictest criterion for a

linearizability is not intended to be used with transactional replication systems

- 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

Consider a replicated service with two clients that perform read and update

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

- 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.



it is not linearizable because client2's *getBalance* is after client 1's *setBalance* in real time.

- ✍ 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 is sequentially consistent but not linearizable

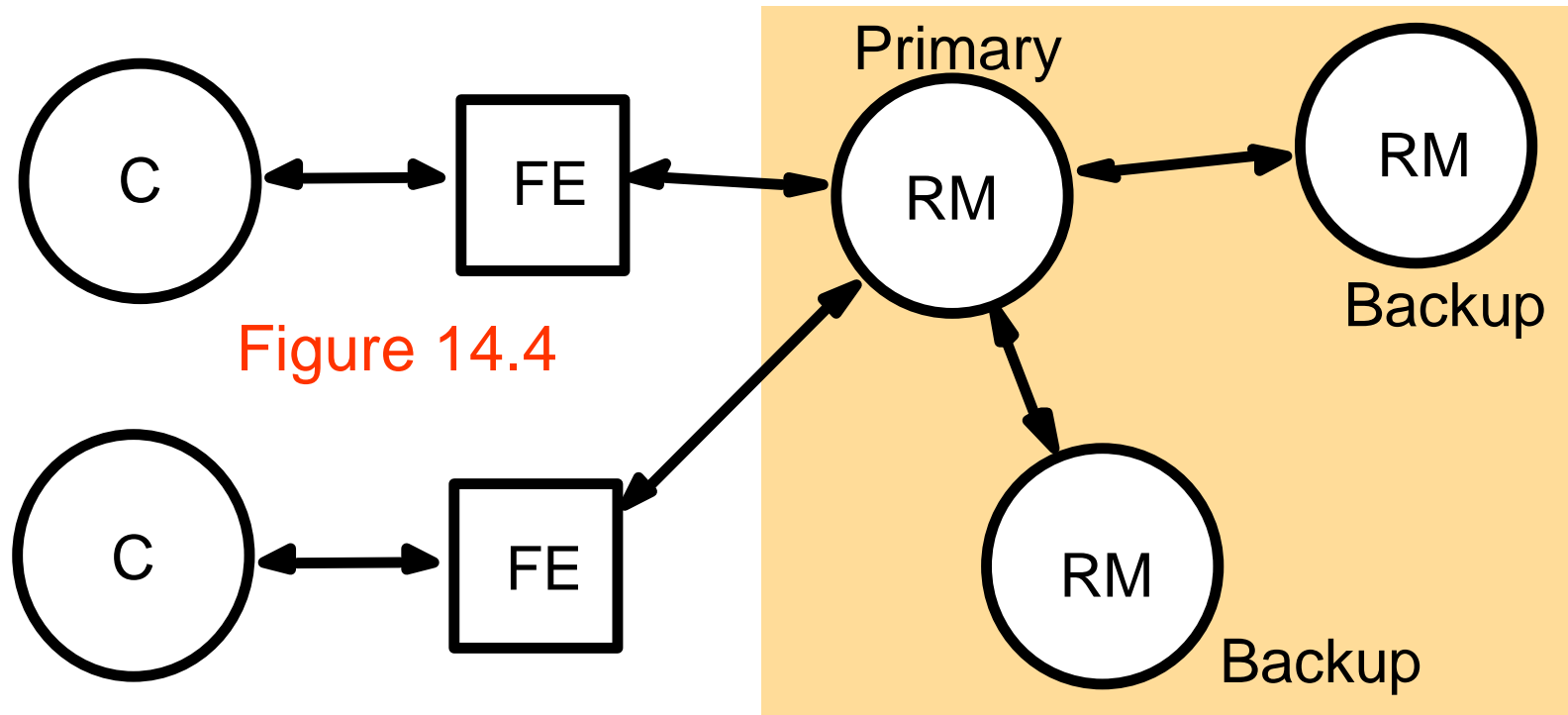
Client 1:	Client 2:
<i>setBalance_B</i> (x,1)	
	<i>getBalance_A</i> (y) ? ?
	<i>getBalance_A</i> (x) ? ?
<i>setBalance_A</i> (y,2)	

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

but the following interleaving satisfies both criteria for sequential consistency :
getBalance_A(y) ? ?0; *getBalance_A*(x) ? ??0; *setBalance_B*(x,1); *setBalance_A*(y,2)

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

The passive (primary-backup) 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

Passive (primary-backup) replication. Five phases.

✍ 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 performs 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. The backups send an acknowledgement.

✍ 5. Response:

- The primary responds to the FE, which hands the response back to the client.

Passive (primary-backup) replication (discussion)

- ✍ 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 -uses the unique identifier to discover whether the request has already been performed.

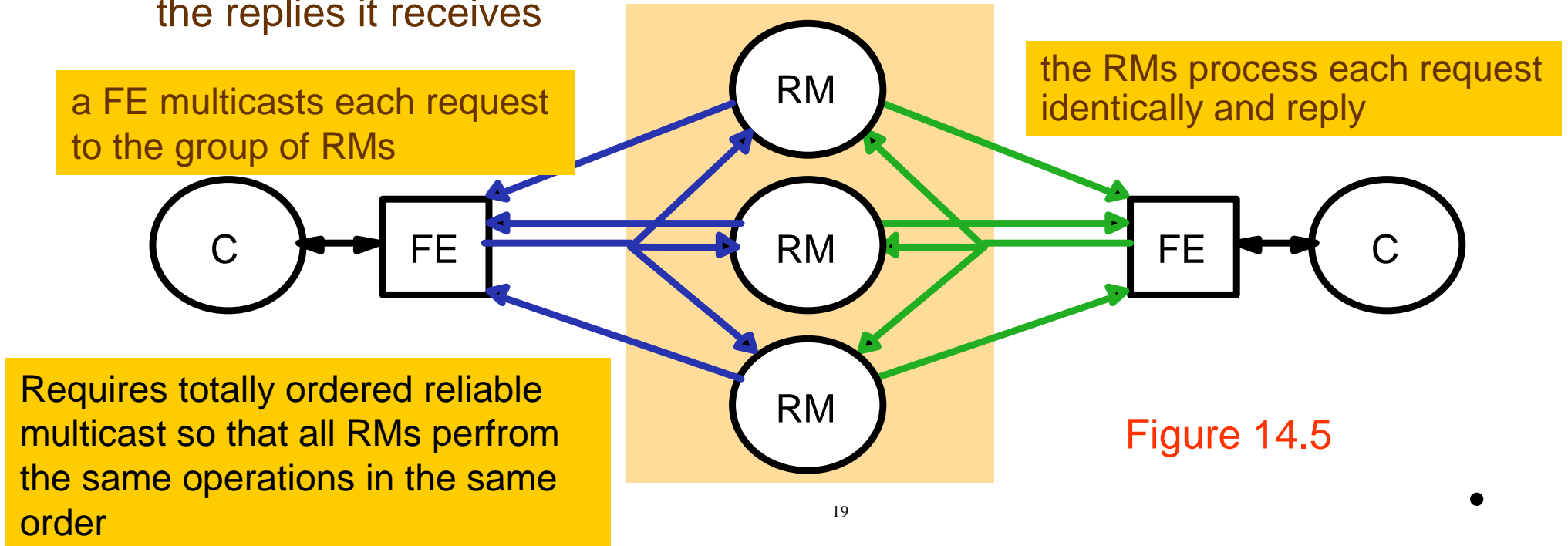
Discussion of passive replication

- ✍ 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 using 1-1 communication. Clients can use either master or slave
 - updates are not done via RMs - they are made on the files at the master

What sort of system do we need to perform totally ordered reliable multicast?

13.3.2. Active replication 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 an 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



Active replication - five phases in performing 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. 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. They are state machines and receive requests in the same order, so the effects are identical. The *id* is put in the response

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. FE may just use one or more responses. If it is only trying to tolerate crash failures, it gives the client the first response.

Active replication - discussion

- ✍ 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 $2f+1$ RMs
 - FE collects $f+1$ identical responses
- ✍ To improve performance,
 - FEs send read-only requests to just one RM

Summary for Sections 14.1-14.3

- ✍ 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

Highly available services

✍ we discuss the application of replication techniques to make services highly available.

– we aim to give clients access to the service with:

✍ reasonable response times for as much of the time as possible

✍ even if some results do not conform to sequential consistency

✍ e.g. a disconnected user may accept temporarily inconsistent results if they can continue to work and fix inconsistencies later

✍ eager versus lazy updates

– fault-tolerant systems send updates to RMs in an ‘eager’ fashion (as soon as possible) and reach agreement before replying to the client

– for high availability, clients should:

✍ only need to contact a minimum number of RMs and

✍ be tied up for a minimum time while RMs coordinate their actions

– weaker consistency generally requires less agreement and makes data more available. Updates are propagated 'lazily'.

14.4.1 The gossip architecture

- ✍ the gossip architecture is a framework for implementing highly available services
 - data is replicated close to the location of clients
 - RMs periodically exchange ‘gossip’ messages containing updates
- ✍ gossip service provides two types of operations
 - queries - read only operations
 - updates - modify (but do not read) the state
- ✍ FE sends queries and updates to any chosen RM
 - one that is available and gives reasonable response times
- ✍ Two guarantees (even if RMs are temporarily unable to communicate)
 - *each client gets a consistent service over time* (i.e. data reflects the updates seen by client, even if the use different RMs). Vector timestamps are used – with one entry per RM.
 - *relaxed consistency between replicas*. All RMs eventually receive all updates. RMs use ordering guarantees to suit the needs of the application (generally causal ordering). Client may observe stale data.

Query and update operations in a gossip service

- ✍ The service consists of a collection of RMs that exchange gossip messages
- ✍ Queries and updates are sent by a client via an FE to an RM

prev is a vector timestamp for the latest version seen by the FE (and client)

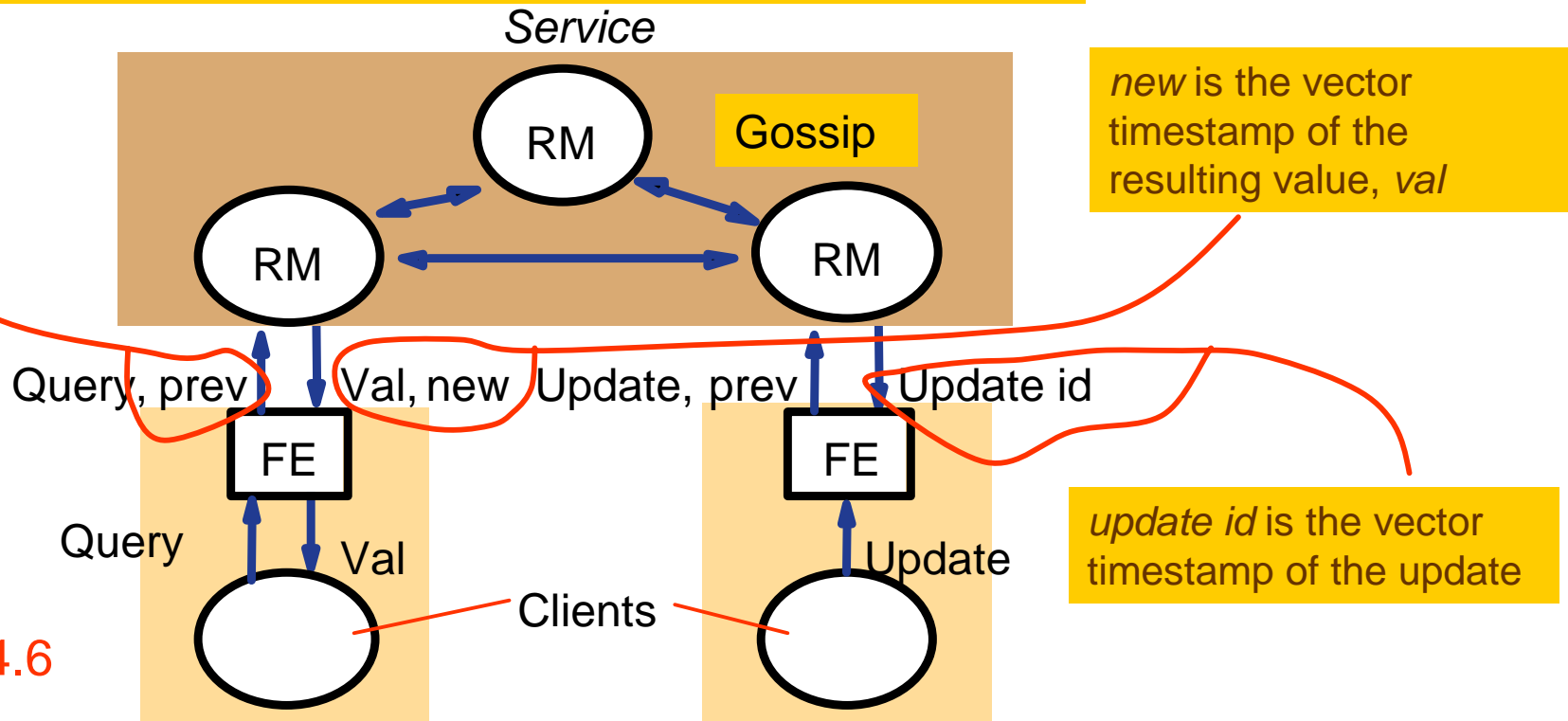


Figure 14.6

Causal ordering

Gossip processing of queries and updates

- ✍ The five phases in performing a client request are:
 - request
 - ✍ FEs normally use the same RM and may be blocked on queries
 - ✍ update operations return to the client as soon as the operation is passed to the FE
 - update response - the RM replies as soon as it has seen the update
 - coordination
 - ✍ the RM waits to apply the request until the ordering constraints apply.
 - ✍ this may involve receiving updates from other RMs in gossip messages
 - execution - the RM executes the request
 - query response - if the request is a query the RM now replies:
 - agreement
 - ✍ RMs update one another by *exchanging* gossip messages (lazily)
 - e.g. when several updates have been collected
 - or when an RM discovers it is missing an update

Front ends propagate their timestamps whenever clients communicate directly

- ✍ each FE keeps a vector timestamp of the latest value seen (*prev*)
 - which it sends in every request
 - clients communicate with one another via FEs which pass vector timestamps

client-to-client communication can lead to causal relationships between operations.

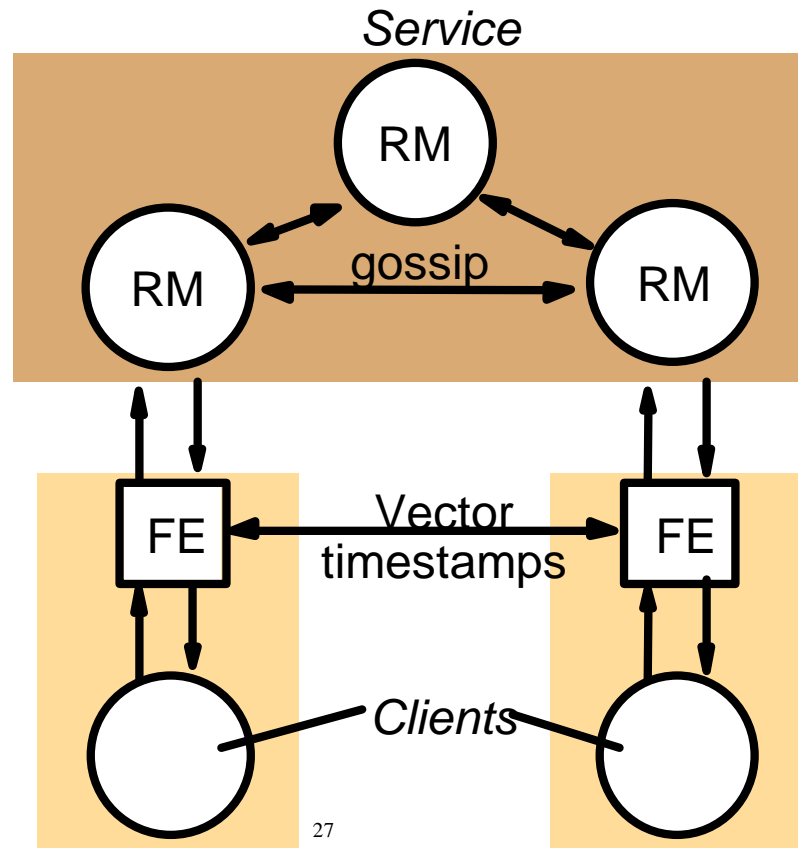


Figure 14.7

A gossip replica manager, showing its main state components

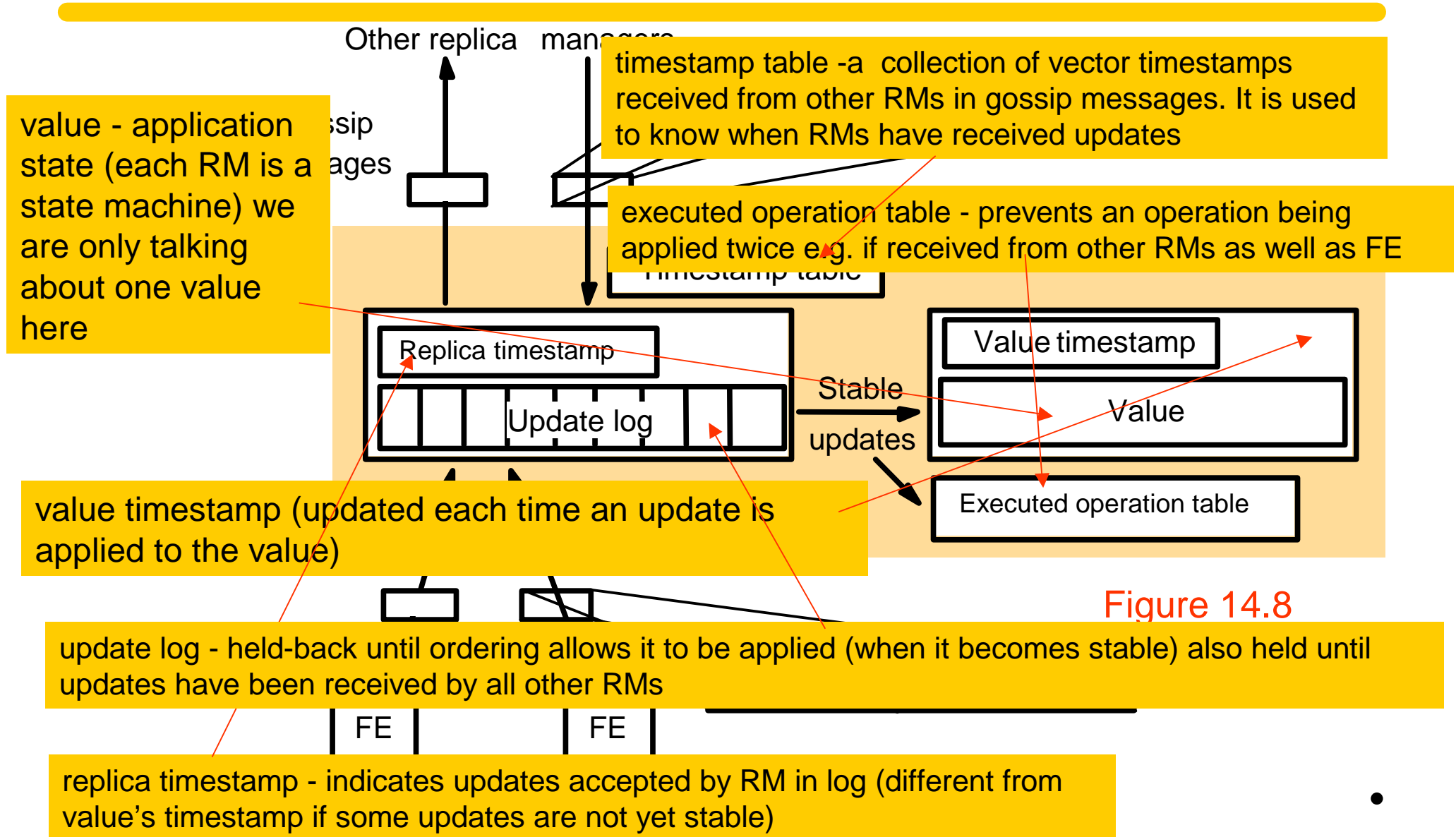


Figure 14.8


Processing of query and update operations

e.g. in a gossip system with 3 RM's a value of (2,4,5) at RM 0 means that the value there reflects the first 2 updates accepted from FEs at RM 0, the first 4 at RM 1 and the first 5 at RM 2.

 Vector timestamp held by RM i consists of:

- i th element holds updates received from FEs by that RM
- j th element holds updates received by RM j and propagated to RM i

 Query operations contain $q.prev$

- they can be applied if $q.prev = valueTS$ (value timestamp)
- failing this, the RM can wait for gossip message or initiate them
 -  e.g. if $valueTS = (2,5,5)$ and $q.prev = (2,4,6)$ - RM 0 has missed an update from RM 2
- Once the query can be applied, the RM returns $valueTS (new)$ to the FE. The FE merges new with its vector timestamp

Gossip update operations

- ✍ Update operations are processed in causal order
 - A FE sends update operation $u.op$, $u.prev$, $u.id$ to RM i
 - ✍ A FE can send a request to several RMs, using same id
 - When RM i receives an update request, it checks whether it is new, by looking for the id in its executed ops table and its log
 - if it is new, the RM
 - ✍ increments by 1 the i th element of its replica timestamp,
 - ✍ assigns a unique vector timestamp ts to the update
 - ✍ and stores the update in its log $logRecord = \langle i, ts, u.op, u.prev, u.id \rangle$
 - The timestamp ts is calculated from $u.prev$ by replacing its i th element by the i th element of the replica timestamp.
 - The RM returns ts to the FE, which merges it with its vector timestamp
 - For stability $u.prev = valueTS$
 - That is, the valueTS reflects all updates seen by the FE.
 - When stable, the RM applies the operation $u.op$ to the $value$, updates $valueTS$ and adds $u.id$ to the executed operation table.

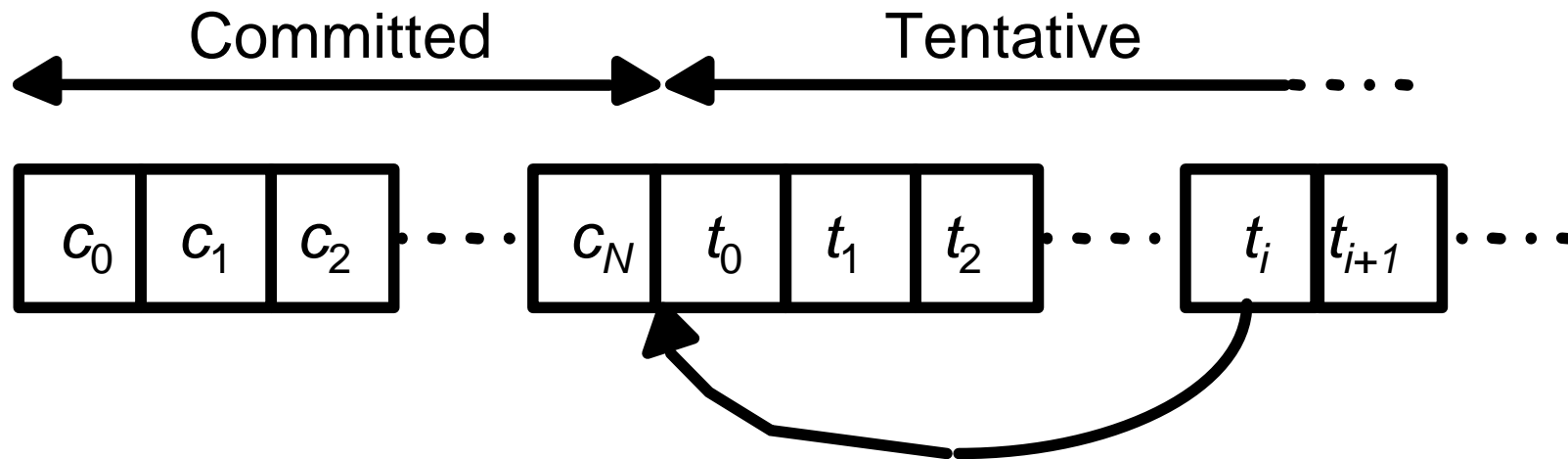
Gossip messages

- ✍ an RM uses entries in its timestamp table to estimate which updates another RM has not yet received
 - The timestamp table contains a vector timestamp for each other replica, collected from gossip messages
- ✍ gossip message, m contains log $m.log$ and replica timestamp $m.ts$
- ✍ an RM receiving gossip message m has the following main tasks
 - merge the arriving log with its own (omit those with $ts = replicaTS$)
 - apply in causal order updates that are new and have become stable
 - remove redundant entries from the log and executed operation table when it is known that they have been applied by all RMs
 - merge its replica timestamp with $m.ts$, so that it corresponds to the additions in the log

Discussion of Gossip architecture

- ✍ the gossip architecture is designed to provide a highly available service
- ✍ clients with access to a single RM can work when other RMs are inaccessible
 - but it is not suitable for data such as bank accounts
 - it is inappropriate for updating replicas in real time (e.g. a conference)
- ✍ scalability
 - as the number of RMs grow, so does the number of gossip messages
 - for R RMs, the number of messages per request (2 for the request and the rest for gossip) = $2 + (R-1)/G$
 - ✍ G is the number of updates per gossip message
 - ✍ increase G and improve number of gossip messages, but make latency worse
 - ✍ for applications where queries are more frequent than updates, use some read-only replicas, which are updated only by gossip messages

Figure 14.9 Committed and tentative updates in Bayou



Tentative update t_i becomes the next committed update and is inserted after the last committed update c_N .

14.5 Transactions with replicated data

- ✍ objects in transactional systems are replicated to enhance availability and performance
 - the effect of transactions on replicated objects should be the same as if they had been performed one at a time on a single set of objects.
 - this property is called *one-copy serializability*.
 - it is similar to, but not to be confused with, sequential consistency.
 - ✍ sequential consistency does not take transactions into account
 - each RM provides concurrency control and recovery of its own objects
 - ✍ we assume two-phase locking in this section
 - replication makes recovery more complicated
 - ✍ when an RM recovers, it restores its objects with information from other RMs

14.5.1 Architectures for replicated transactions

- ✍ We assume that an FE sends requests to one of a group of RMs
 - in the primary copy approach, all FEs communicate with a single RM which propagates updates to back-ups.
 - In other schemes, FEs may communicate with any RM and coordination between RMs is more complex
 - an RM that receives a request is responsible for getting cooperation from the other RMs
 - ✍ rules as to how many RMs are involved vary with the replication scheme
 - e.g. in the read one/write all scheme, one RM is required for a *read* request and all RMs for a *write* request
- ✍ propagate requests immediately or at the end of a transaction?
 - in the primary copy scheme, we can wait until end of transaction (concurrency control is applied at the primary)
 - but if transactions access the same objects at different RMs, we need to propagate the requests so that concurrency control can be applied
- ✍ two-phase commit protocol
 - becomes a two-level nested 2PC. If a coordinator or worker is an RM it will communicate with other RMs that it passed requests to during the transaction

Consider pairs of operations by different transactions on the same object.
 Any pair of write operations will require conflicting locks at all of the RMs
 a read operation and a write operation will require conflicting locks at a single RM.
 This one-copy serializability is achieved

a

one RM is required for a
 write request

each read operation is performed by a single RM, which sets a read lock

every write operation must be performed at all RMs, each of which applies a write lock

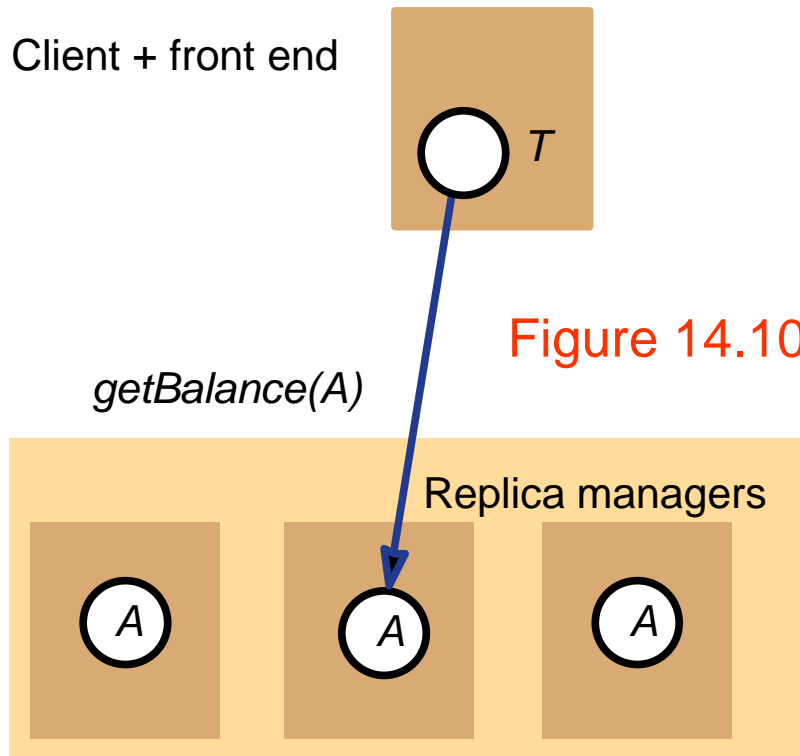
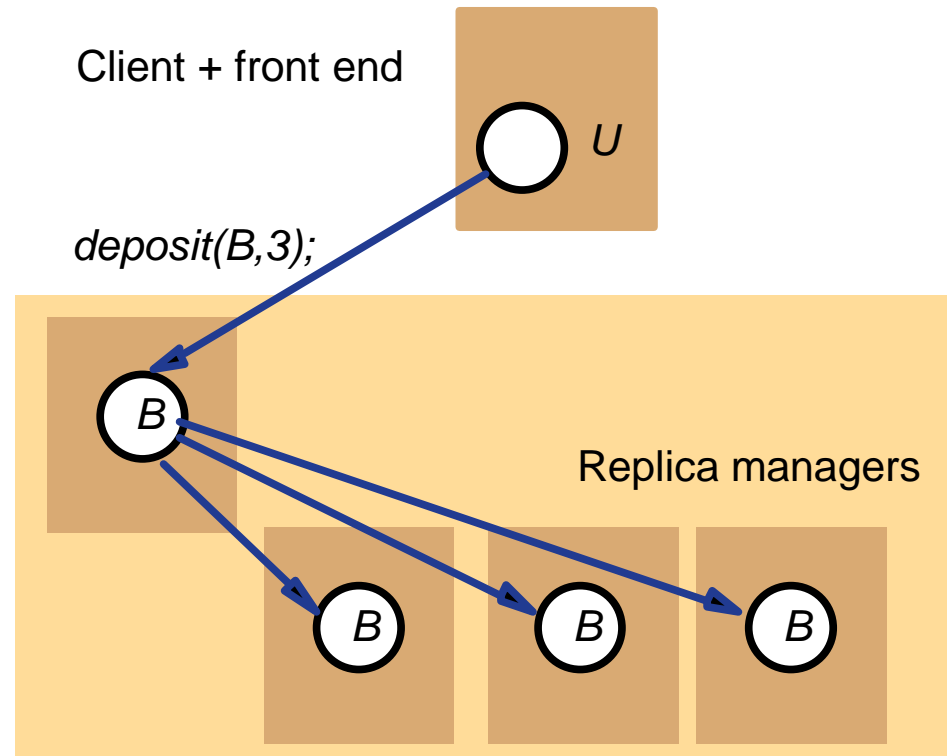


Figure 14.10



14.5.2 Available copies replication

- ✍ the simple read one/write all scheme is not realistic
 - because it cannot be carried out if some of the RMs are unavailable,
 - either because they have crashed or because of a communication failure
- ✍ the available copies replication scheme is designed to allow some RMs to be temporarily unavailable
 - a read request can be performed by any available RM
 - write requests are performed by the receiving RM and all other available RMs in the group

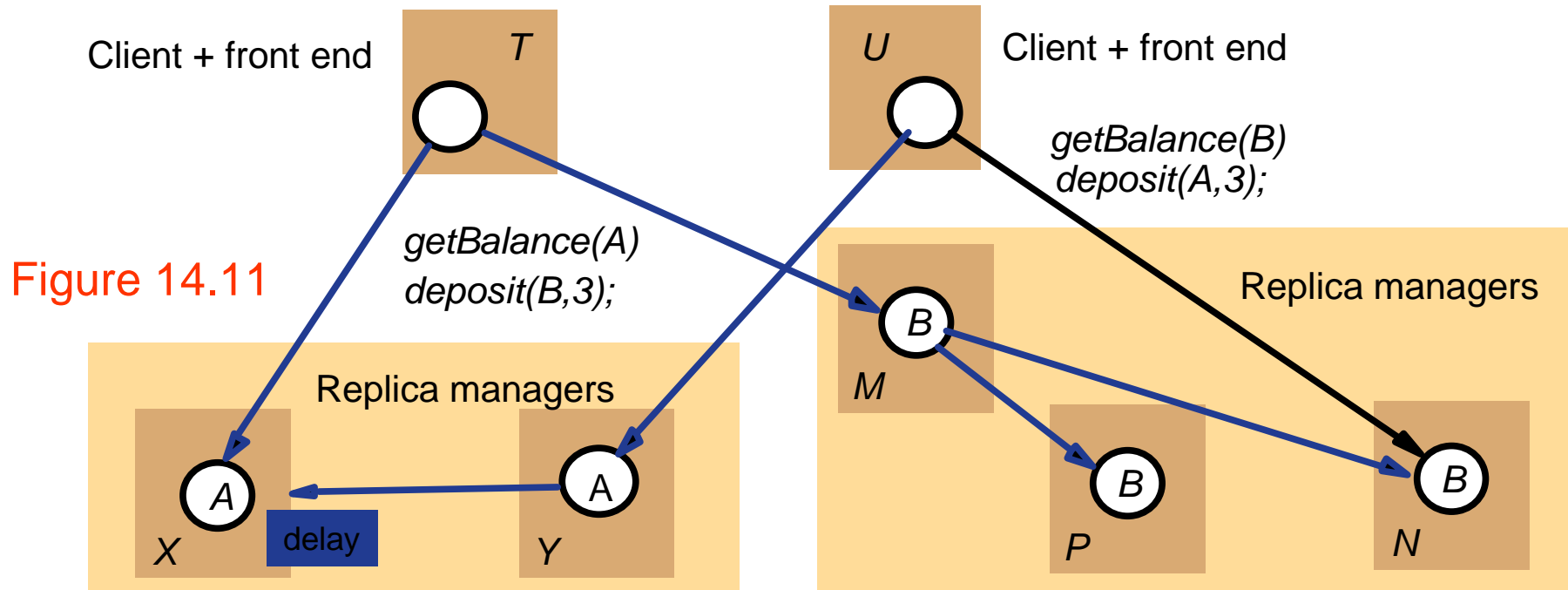
Available copies – read one/ write all available

- ✍ local concurrency control achieves one-copy serializability provided the set of RMs does not change.
- ✍ but we have RMs failing and recovering

T's *getBalance* is performed by X

whereas T's *deposit* is performed by M, N and P.

At X T has read A and has locked it. Therefore U's *deposit* is delayed until T finishes



Available copies

✍ Replica manager failure

- An RM can fail by crashing and is replaced by a new process
 - ✍ the new process restores its state from its recovery file
- FEs use timeouts in case an RM fails
 - ✍ then try the request at another RM
 - ✍ in the case of a *write*, the RM passing it on may observe failures
- If an RM is doing recovery, it rejects requests (& FE tries another RM)
- For one-copy serializability, failures and recoveries are serialized with respect to transactions
 - ✍ that is, if a transaction observes that a failure occurs, it must be observed before it started or after it finished
 - ✍ one-copy serializability is not achieved if different transactions make conflicting failure observations

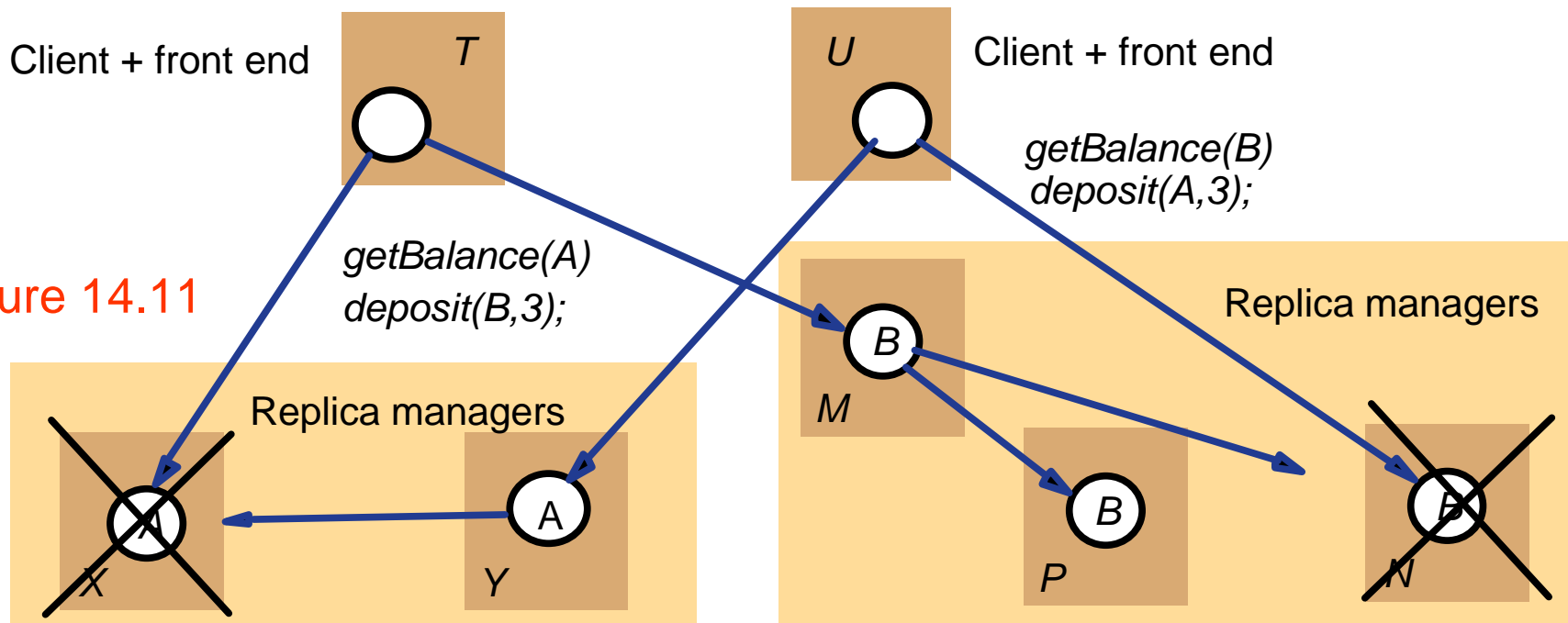
therefore additional concurrency control is required to prevent inconsistent results between a *read* in one transaction and a *write* in another transaction

- both RMs fail before *T* and *U* have performed their *deposit* operations
 - Therefore *T*'s deposit will be performed at RMs *M* and *P* (all available)
 - and *U*'s deposit will be performed at RM *Y*. (all available).

concurrency control at *X* does not prevent *U* from updating *A* at *Y*

concurrency control at *M* does not prevent *T* from updating *B* at *M* & *P*

Figure 14.11



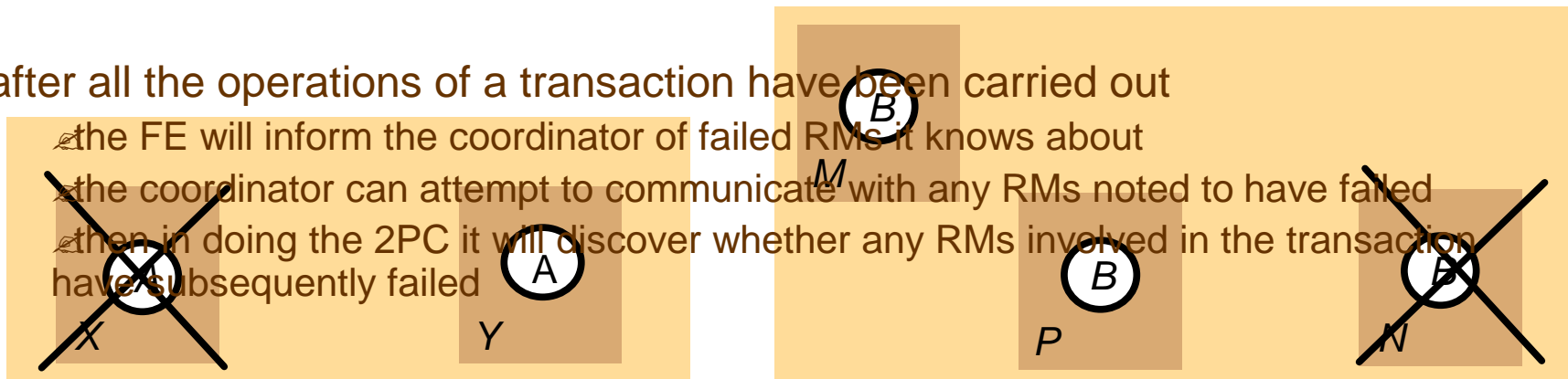
Available copies replication

Local validation (the additional concurrency control)

- before a transaction commits, it checks for failures and recoveries of the RMs it has contacted
 - ✍ e.g. T would check if N is still unavailable and that X , M and P are still available.
 - ✍ If this is the case, T can commit.
 - this implies that X failed after T validated and before U validated
 - i.e. we have N fails ? T commits ? X fails ? U validates
 - (above, we said X fails before T 's *deposit*, in which case, T would have to abort)
 - ✍ U checks if N is still available (no) and X still unavailable
 - therefore U must abort

–after all the operations of a transaction have been carried out

- ✍ the FE will inform the coordinator of failed RMs it knows about
- ✍ the coordinator can attempt to communicate with any RMs noted to have failed
- ✍ then in doing the 2PC it will discover whether any RMs involved in the transaction have subsequently failed



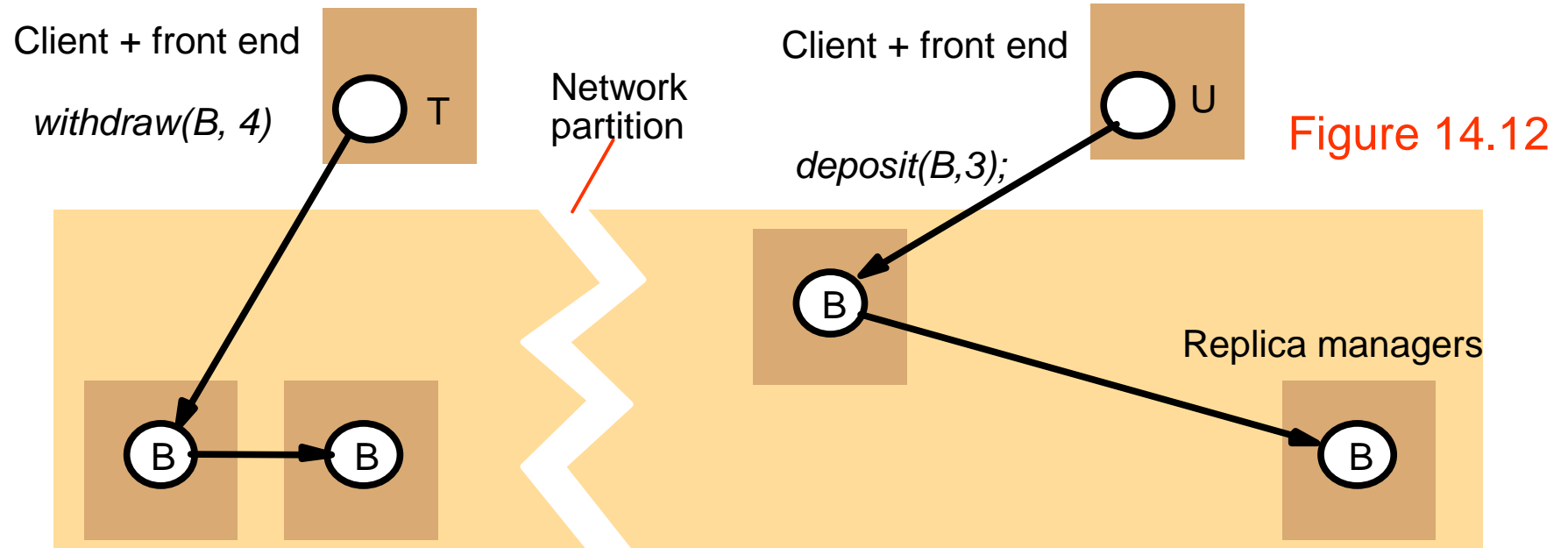
Network partitions divide RMs into subgroups

e.g. the RMs doing the *deposit* can't communicate with those doing the *withdraw*

Reading during a partition would not cause inconsistency, writing might.

Optimistic schemes e.g. available copies with validation - resolve consistencies when a partition is repaired. We have to be able to do compensating actions, otherwise the scheme is unsuitable

e.g. unsuitable for banking. We are not studying this. See section 14.5.4



14.5.5 Quorum consensus methods

- ✍ To prevent transactions in different partitions from producing inconsistent results
 - make a rule that operations can be performed in only one of the partitions.
- ✍ RMs in different partitions cannot communicate:
 - each subgroup decides independently whether they can perform operations.
- ✍ A *quorum* is a subgroup of RMs whose size gives it the right to perform operations.
 - e.g. if having the majority of the RMs could be the criterion
- ✍ in quorum consensus schemes
 - update operations may be performed by a subset of the RMs
 - ✍ and the other RMs have out-of-date copies
 - ✍ version numbers or timestamps are used to determine which copies are up-to-date
 - ✍ operations are applied only to copies with the current version number

Gifford's quorum consensus file replication scheme

- ✍ a number of 'votes' is assigned to each physical copy of a logical file at an RM
 - a vote is a weighting giving the desirability of using a particular copy.
 - each *read* operation must obtain a read quorum of R votes before it can read from any up-to-date copy
 - each *write* operation must obtain a write quorum of W votes before it can do an update operation.
 - R and W are set for a group of replica managers such that
 - ✍ $W >$ half the total votes
 - ✍ $R + W >$ total number of votes for the group
 - ensuring that any pair contain common copies (i.e. a read quorum and a write quorum or two write quora)
 - therefore in a partition it is not possible to perform conflicting operations on the same file, but in different partitions.



Gifford's quorum consensus - performing *read* and *write* operations

- ✍ before a *read* operation, a read quorum is collected
 - by making version number enquiries at RMs to find a set of copies, the sum of whose votes is not less than R (not all of these copies need be up to date).
 - as each read quorum overlaps with every write quorum, every read quorum is certain to include at least one current copy.
 - the *read* operation may be applied to any up-to-date copy.
- ✍ before a *write* operation, a write quorum is collected
 - by making version number enquiries at RMs to find a set with up-to-date copies, the sum of whose votes is not less than W .
 - if there are insufficient up-to-date copies, then an out-of-date file is replaced with a current one, to enable the quorum to be established.
 - the *write* operation is then applied by each RM in the write quorum, the version number is incremented and completion is reported to the client.
 - the files at the remaining available RMs are then updated in the background.
- ✍ Two-phase read/write locking is used for concurrency control
 - the version number enquiry sets read locks (read and write quora overlap)

Gifford's quorum consensus: configurability of groups of replica managers

- ✍ groups of RMs can be configured to give different performance or reliability characteristics
 - once the R and W have been chosen for a set of RMs:
 - the reliability and performance of *write* operations may be increased by decreasing W
 - and similarly for reads by decreasing R
- ✍ the performance of read operations is degraded by the need to collect a read consensus
- ✍ examples from Gifford
 - three examples show the range of properties that can be achieved by allocating weights to the various RMs in a group and assigning R and W appropriately
 - weak representatives (on local disk) have zero votes, get a read quorum from RMs with votes and then read from the local copy

Gifford's quorum consensus examples (1979)

		<i>Example 1</i>	<i>Example 2</i>	<i>Example 3</i>
<i>Latency</i> <i>(milliseconds)</i>	Replica 1	75	75	75
	Replica 2	65	100	750
	Replica 3	65	750	750
<i>Voting</i> <i>configuration</i>	Replica 1	1	2	1
	Replica 2	0	1	1
	Replica 3	0	1	1
<i>Quorum</i> <i>sizes</i>	<i>R</i>	1	2	1
	<i>W</i>	1	3	3

Derived performance
latency
blocking probability - probability that a quorum cannot be obtained, assuming probability of 0.01 that any single RM is unavailable

Example 1 is configured for a file with high read to write ratio

Example 2 is configured for a file with a moderate read to write ratio

Example 3 is configured for a file with a very high read to write ratio.

Reads can be done at any RM and the probability of the file being unavailable is small. But *writes* must access all RMs.

Each RM and one remote RM. If the local RM fails only *reads* are allowed

Figure 14.13

Two network partitions

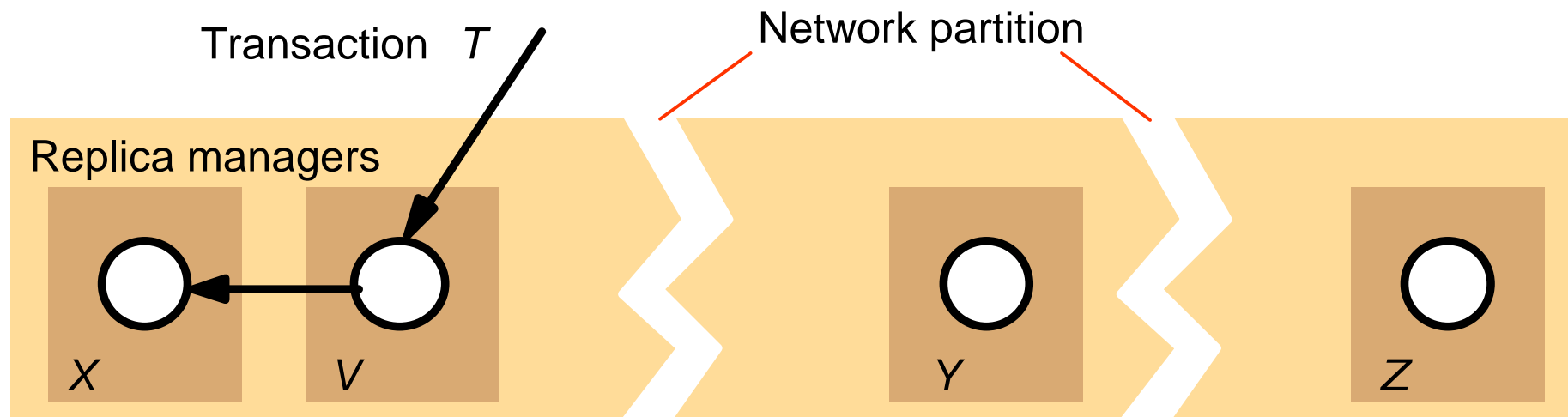


Figure 14.14 Virtual partition

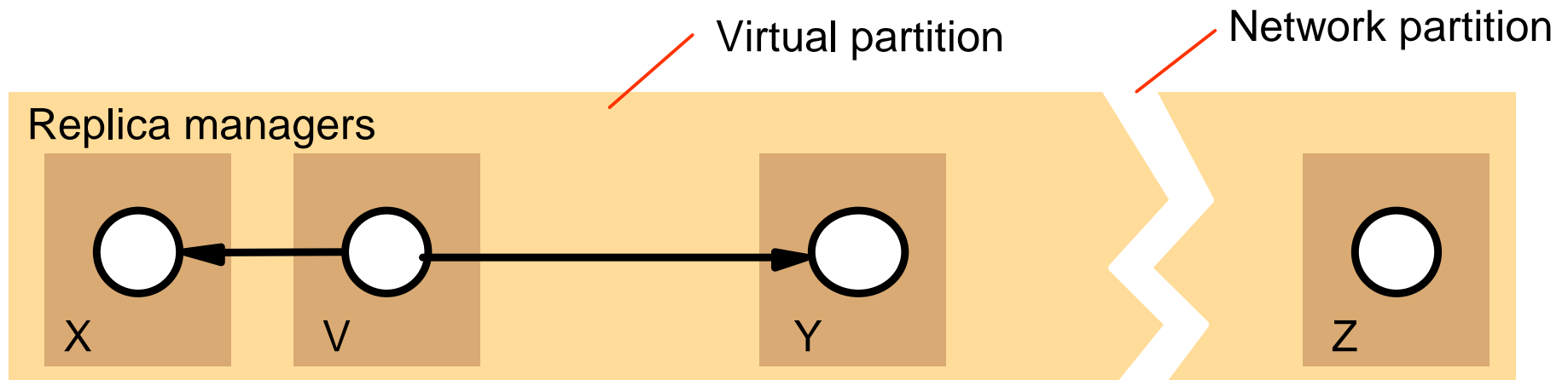


Figure 14.15

Two overlapping virtual partitions

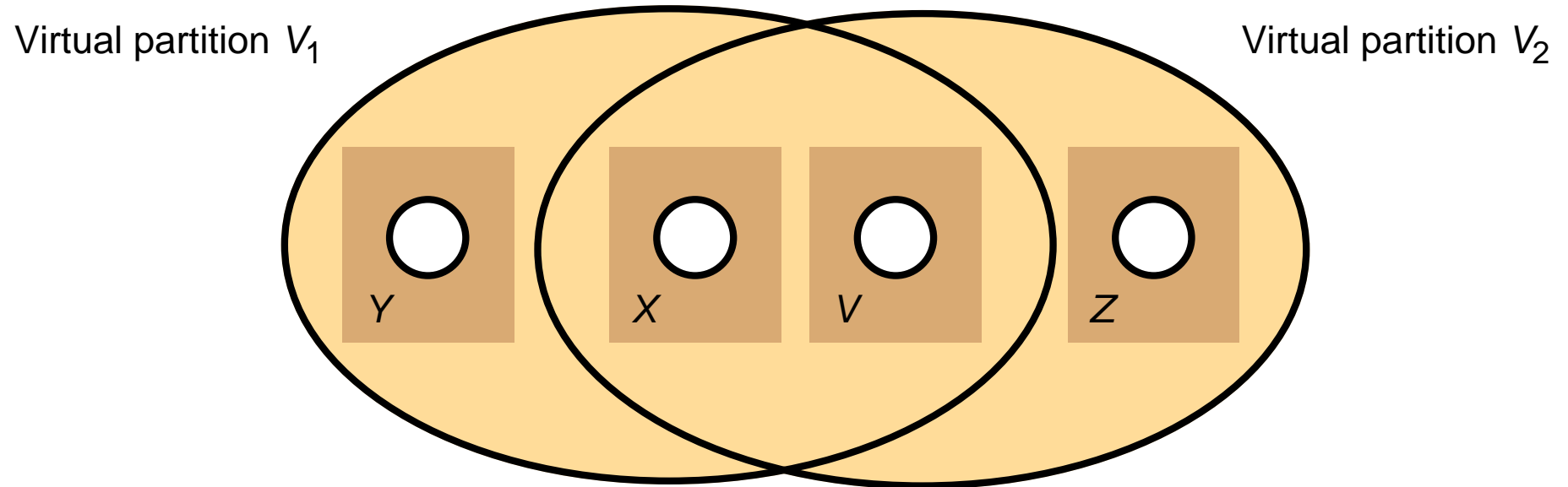


Figure 14.16

Creating a virtual partition

Phase 1:

- The initiator sends a *Join* request to each potential member. The argument of *Join* is a proposed logical timestamp for the new virtual partition.
- When a replica manager receives a *Join* request, it compares the proposed logical timestamp with that of its current virtual partition.
 - If the proposed logical timestamp is greater it agrees to join and replies *Yes*;
 - If it is less, it refuses to join and replies *No*.

Phase 2:



- If the initiator has received sufficient *Yes* replies to have read and write quora, it may complete the creation of the new virtual partition by sending a *Confirmation* message to the sites that agreed to join. The creation timestamp and list of actual members are sent as arguments.
- Replica managers receiving the *Confirmation* message join the new virtual partition and record its creation timestamp and list of actual members.

Summary for Gossip and replication in transactions

the Gossip architecture is designed for highly available services

- it uses a lazy form of replication in which RMs update one another from time to time by means of gossip messages
- it allows clients to make updates to local replicas while partitioned
- RMs exchange updates with one another when reconnected

replication in transactions

- primary-backup architectures can be used
- other architectures allow FMs to use any RM
 -  available copies allows RMs to fail, but cannot deal with partitions
 -  quorum consensus does allow transactions to progress in the presence of partitions, but the performance of read operations is degraded by the need to collect a read consensus