Unit-6 Transactions & Replications

Syllabus: Introduction, System Model and Group Communication, Concurrency Control in Distributed Transactions, Distributed Dead Locks, Transaction Recovery; Replication-Introduction, Passive (Primary) Replication, Active Replication

Topic 01: INTRODUCTION

Introduction to replication

Replication of data: - the maintenance of copies of data at multiple computers

- 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 read-only data is simple, but replication of changing data has overheads
- fault-tolerant service
  - guarantees correct behavior in spite of certain faults (can include timeliness)
  - if *f* of *f*+1 servers crash then 1 remains to supply the service
  - if *f* of 2*f*+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

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.

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 in Bayou and Coda.

Topic 02: System model:

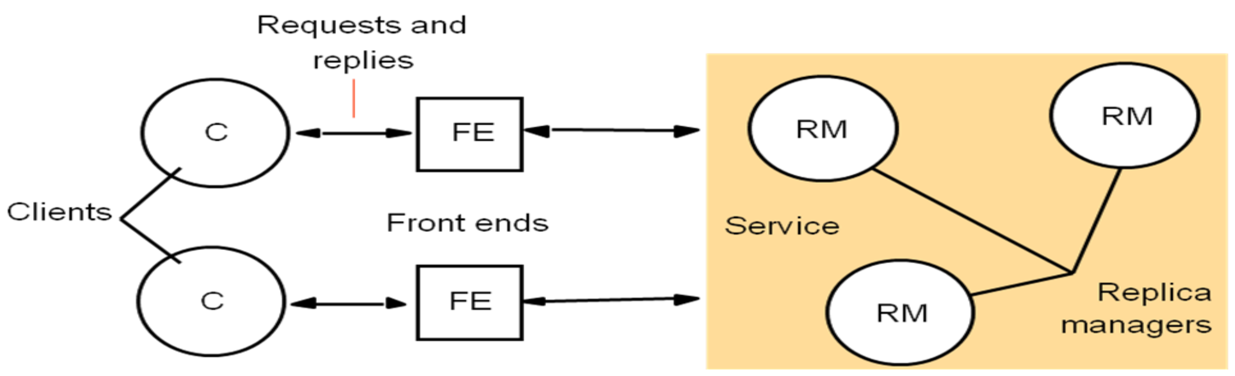
- each *logical* object is implemented by a collection of *physical* copies called *replicas*
  - the replicas are not necessarily consistent all the time (some may have received updates, not yet conveyed to the others)
- we assume an asynchronous system where processes fail only by crashing and generally 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:

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.

<u>A basic architectural model for the management of replicated data</u>



- 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 (What can the FE hide from a client?)

- <u>issue request</u>
    - <u>the FE either</u>
        - 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)
- <u>coordination</u>
    - the RMs decide whether to apply the request; and decide on its ordering relative to other requests (according to FIFO, causal or total ordering)
- <u>execution</u>
    - the RMs execute the request (sometimes tentatively)
- <u>agreement</u>
    - RMs *agree* on the effect of the request, .e.g perform 'lazily' or immediately
- <u>response</u>
    - one or more RMs reply to FE. e.g.

- for high availability give first response to client.
- to tolerate byzantine faults, take a vote

FIFO ordering: if a FE issues r then *r*', then any correct RM handles *r* before *r*'

Causal ordering: if r ® *r*', then any correct RM handles *r* before *r*'

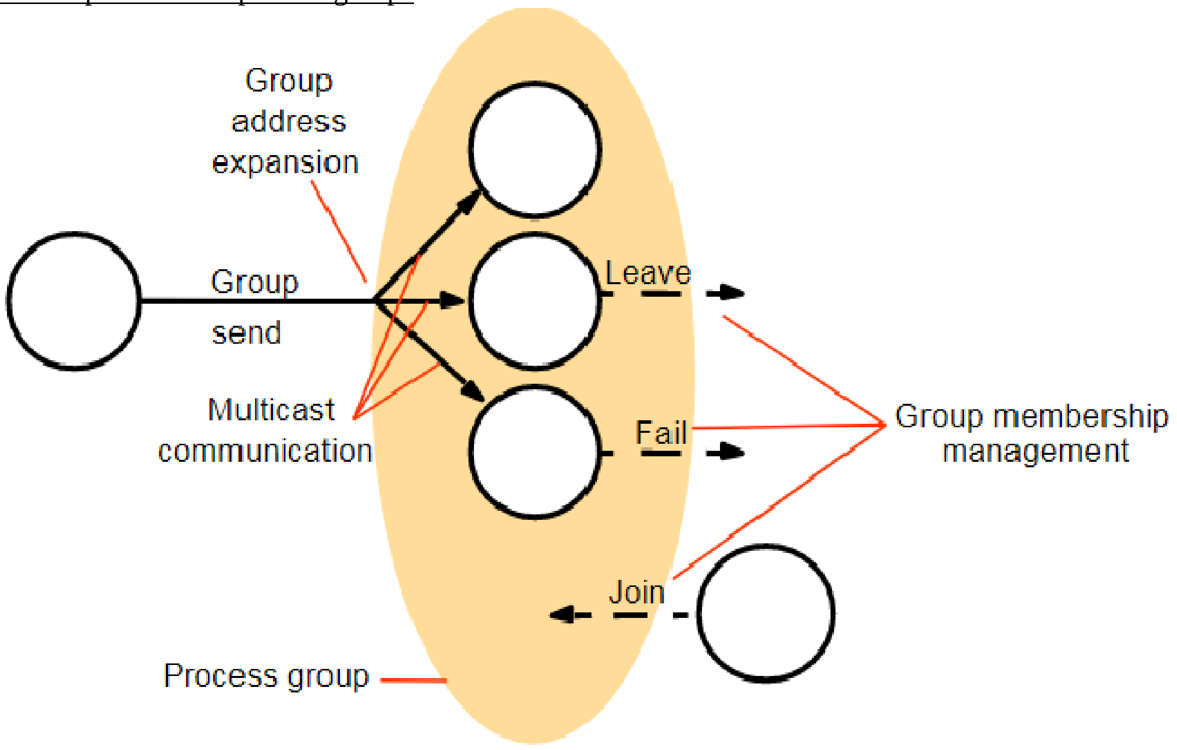
Total ordering: if a correct RM handles r before *r*', then any correct RM handles *r* before *r*' Bayou sometimes executes responses tentatively so as to be able to reorder them

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

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

<u>Services provided for process groups</u>



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 below
  – 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)

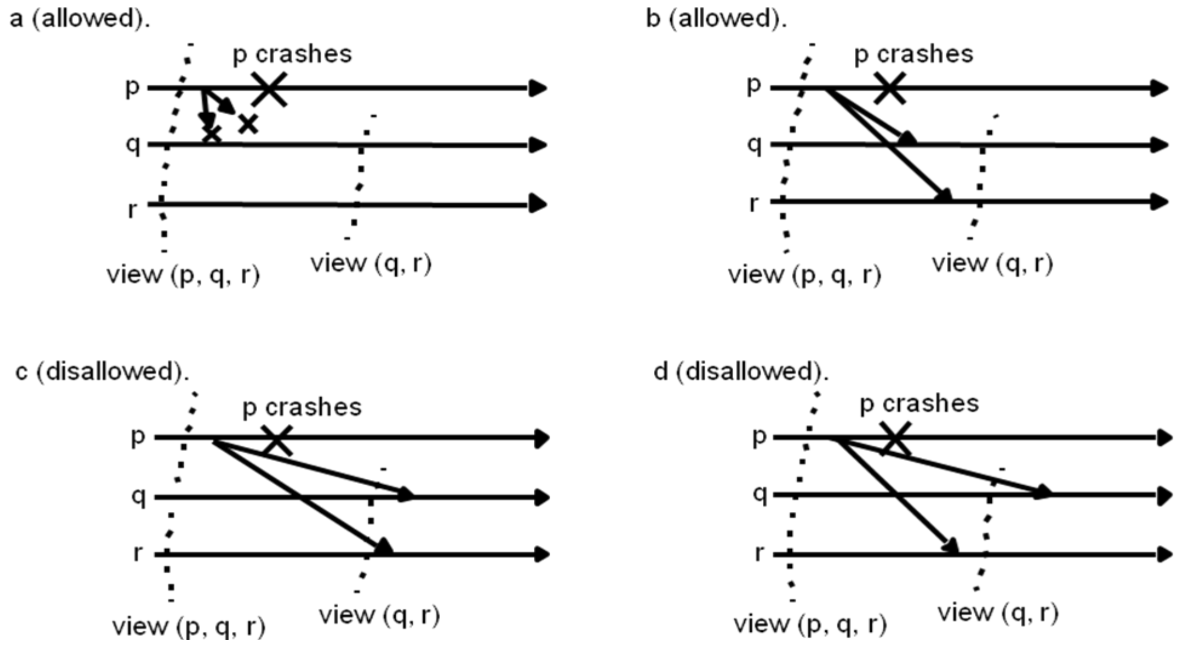


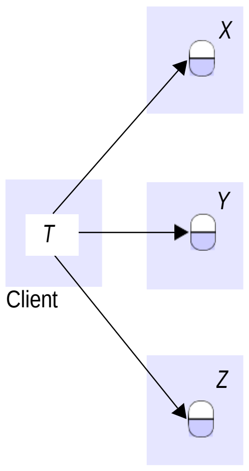
Figure : View-synchronous group communication

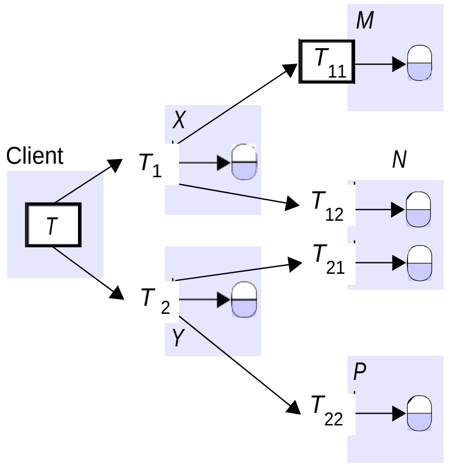
Topic 03: Distributed transactions – introduction

- a *distributed transaction* refers to a flat or nested transaction that accesses objects managed by multiple servers
- When a distributed transaction comes to an end
  – the either all of the servers commit the transaction
  – or all of them abort the transaction.
- one of the servers is *coordinator*, it must ensure the same outcome at all of the servers.
- the ‘two-phase commit protocol’ is the most commonly used protocol for achieving this

Flat and nested distributed transaction: The distributed transactions are two types:

1. Flat Transaction
2. Nested Transaction



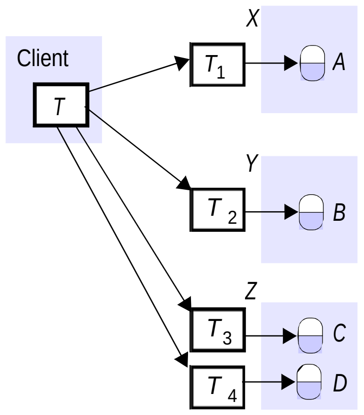


Flat Transaction **Nested Transaction**

- In a flat transaction, each transaction is decoupled from and independent of other transactions in the system. Another transaction cannot start in the same thread until the current transaction ends. Flat transactions are the most prevalent model and are supported by most commercial database systems.
- A nested transaction is a database transaction that is started by an instruction within the scope of an already started transaction.This means that a commit in an inner transaction does not necessarily persist updates to the system.

Nested Banking Transaction:

T = openTransaction
openSubTransaction
a.withdraw(10);
openSubTransaction
b.withdraw(20);
openSubTransaction
c.deposit(10);
openSubTransaction
d.deposit(20);
closeTransaction

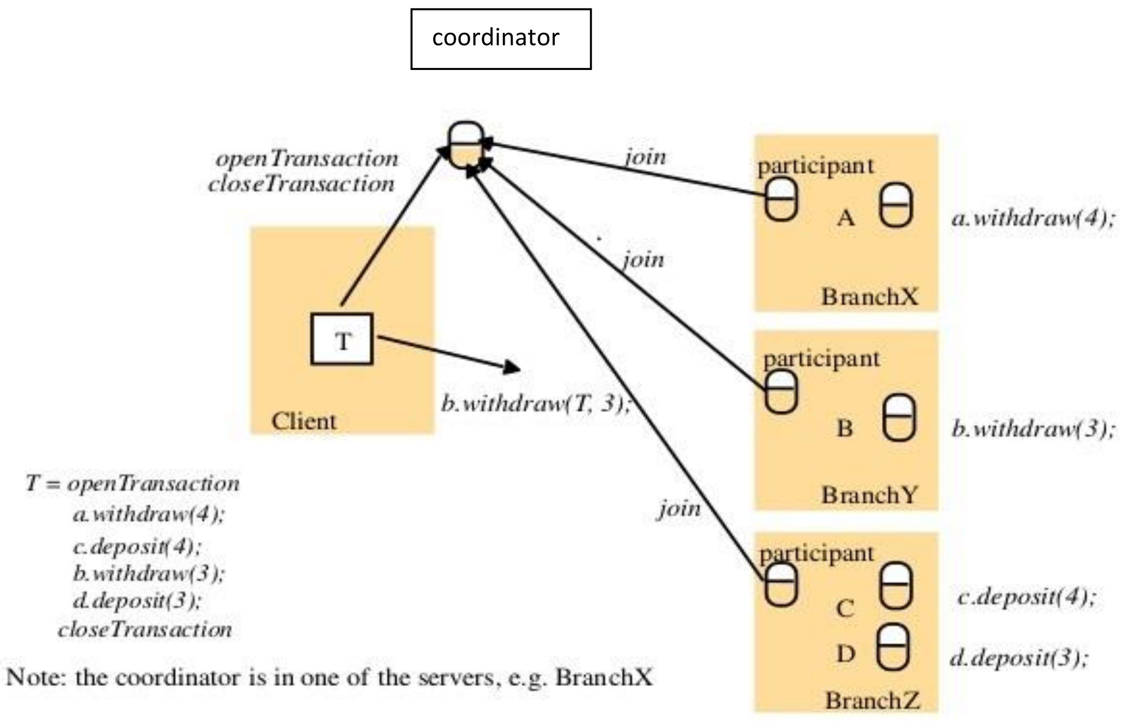


a.withdraw(10)

b.withdraw(20)

c.deposit(10)
d.deposit(20)

Distributed Banking Transaction:



Concurrency control in distributed transactions:

- Each server manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions
  - – therefore, each server is responsible for applying concurrency control to its own objects.
  - – the members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner
  - – therefore if transaction *T* is before transaction *U* in their conflicting access to objects at one of the servers then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both *T* and *U*

Sub Topic 4.1 : Locking

- In a distributed transaction, the locks on an object are held by the server that manages it.
  - – The local lock manager decides whether to grant a lock or make the requesting transaction wait.
  - – it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.
  - – the objects remain locked and are unavailable for other transactions during the atomic commit protocol
    - an aborted transaction releases its locks after phase 1 of the protocol.

<u>Interleaving of transactions T and U at servers X and Y</u>

- in the example on page 529, we have
  - – *T* before *U* at server X and *U* before *T* at server Y
- different orderings lead to cyclic dependencies and distributed deadlock

– detection and resolution of distributed deadlock in next section

T			U		
Write(A)	**at X**	**Locks A**			
			Write(B)	**at Y**	**Locks B**
Read(B)	**at Y**	**Wait for U**			
			Read(A)	**at X**	**Wait for T**

Sub topic 4.2 :Timestamp ordering concurrency control

- Single server transactions
  - – coordinator issues a unique timestamp to each transaction before it starts
  - – serial equivalence ensured by committing objects in order of timestamps
- Distributed transactions
  - – the first coordinator accessed by a transaction issues a globally unique timestamp
  - – as before the timestamp is passed with each object access
  - – the servers are jointly responsible for ensuring serial equivalence
    - that is if T access an object before U, then T is before U at all objects
  - – coordinators agree on timestamp ordering
    - a timestamp consists of a pair <*local timestamp, server-id*>.
    - the agreed ordering of pairs of timestamps is based on a comparison in

which the server-id part is less significant – they should relate to time

- The same ordering can be achieved at all servers even if their clocks are not synchronized
  - – for efficiency it is better if local clocks are roughly synchronized
  - – then the ordering of transactions corresponds roughly to the real time order in which they were started
- Timestamp ordering
  - – conflicts are resolved as each operation is performed
  - – if this leads to an abort, the coordinator will be informed
    - it will abort the transaction at the participants
  - – any transaction that reaches the client request to commit should always be able to do so
    - participant will normally vote *yes*
    - unless it has crashed and recovered during the transaction

Optimistic concurrency control

- Each transaction is validated before it is allowed to commit
  - I. Transaction numbers assigned at start of validation
  - II. Transactions serialized according to transaction numbers
  - III. Validation takes place in phase 1 of 2PC protocol
- Consider the following interleaving of T and U
  - I. T before U at X and U before T at Y

Suppose T & U start validation at about the same time
X does T first
Y does U first

No parallel Validation – Commitment deadlock.

T	U
Read (A) at X	Read (B) at Y
Write (A)	Write (B)
Read (B) at Y	Read (A) at X
Write (B)	Write (A)

<u>Commitment deadlock in optimistic concurrency control</u>

- servers of distributed transactions do parallel validation
  – therefore rule 3 must be validated as well as rule 2
    - the write set of *Tv* is checked for overlaps with write sets of earlier transactions
  – this prevents commitment deadlock
  – it also avoids delaying the 2PC protocol
- another problem - independent servers may schedule transactions in different orders
  – e.g. T before U at X and U before T at Y
  - this must be prevented

Topic 05: Distributed deadlocks

- Single server transactions can experience deadlocks
  – prevent or detect and resolve
  – use of timeouts is clumsy, detection is preferable.
    - it uses wait-for graphs.
- Distributed transactions lead to distributed deadlocks
  – in theory can construct global wait-for graph from local ones

– a cycle in a global wait-for graph that is not in local ones is a distributed deadlock

Sub topic 5.1:Interleavings of transactions U,V and W

Objects A,B managed by X and Y; C and D by Z – next slide has global wait - for graph

U		***V***		***W***	
D .deposit(10)	***Lock D***				
		b.deposit(10)	***Lock B at Y***		
U->V at Y				***c.deposit(30)***	***Lock C at Z***
b.withdraw(30)	***Wait at Y***	***V->W at Z***			
		c.withdraw(20)	***Wait at Z***	***W->U at X***	
				a.withdraw(20)	***Wait at X***

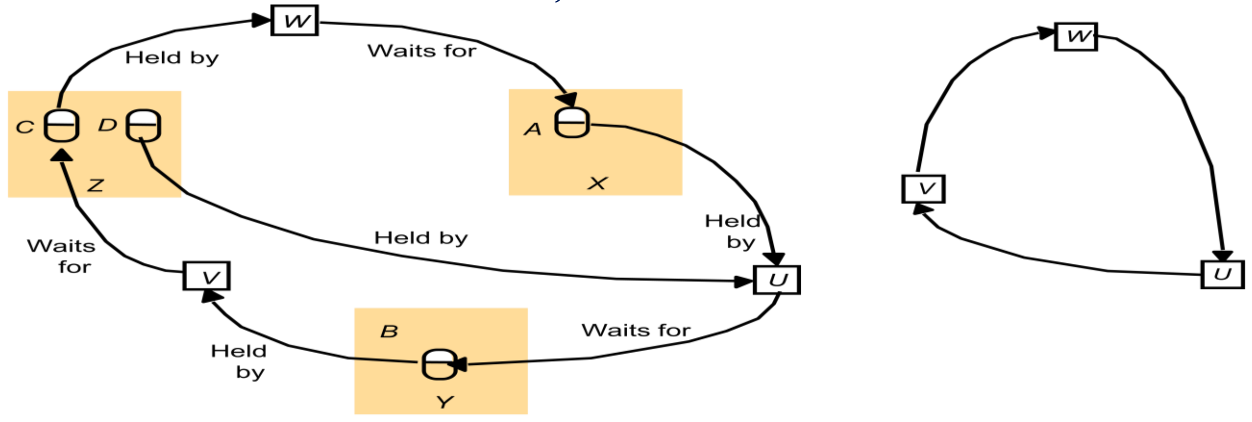
Figure: Interleavings of transactions U,V and W

Sub topic 5.2 Distributed deadlock

Deadlock detection - local wait-for graphs

- Local wait-for graphs can be built, e.g.
  – server Y: U ® V added when U requests b.withdraw(30)
  – server Z: V ® W added when V requests c.withdraw(20)
  – server X: W ® U added when W requests a.withdraw(20)
- to find a global cycle, communication between the servers is needed

- centralized deadlock detection
  - one server takes on role of global deadlock detector
  - the other servers send it their local graphs from time to time
  - it detects deadlocks, makes decisions about which transactions to abort and informs the other servers
  - usual problems of a centralized service - poor availability, lack of fault tolerance and no ability to scale
- **A dead lock cycle has alternate edges showing wait for and held by**
- **wait for added in order: U - > V at Y; V - > W at Z and W - > U at X**



Figure; Distributed deadlock

Local and global wait-for graphs

- **Phantom deadlocks**
  - a 'deadlock' that is detected, but is not really one
  - happens when there appears to be a cycle, but one of the transactions has released a lock, due to time lags in distributing graphs
  - in the figure suppose U releases the object at X then waits for V at Y
    - and the global detector gets Y's graph before X's (T ® U ® V ® T)

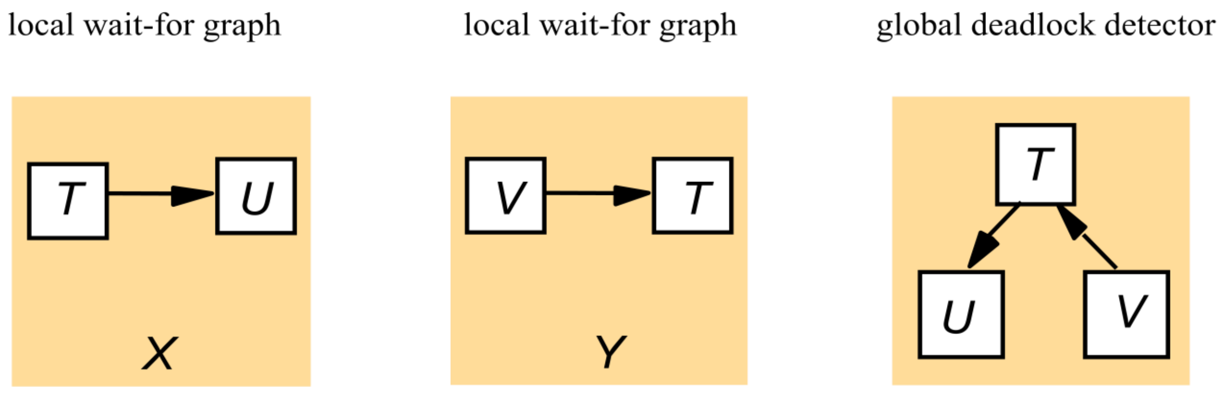


Figure: Local and global wait-for graphs

<u>**Edge chasing -** a distributed approach to deadlock detection</u>

- a global graph is not constructed, but each server knows about some of the edges
  - servers try to find cycles by sending *probes* which follow the edges of the graph through the distributed system

- when should a server send a probe (go back to Fig 13.13)
- edges were added in order *U* ® *V* at *Y; V* ® *W* at Z and *W* ® *U* at *X*
  - when *W* ® *U* at *X* was added, U was waiting, but
  - when *V* ® *W* at *Z*, *W* was not waiting

send a probe when an edge *T*1 ® *T*2 when *T*2 is waiting

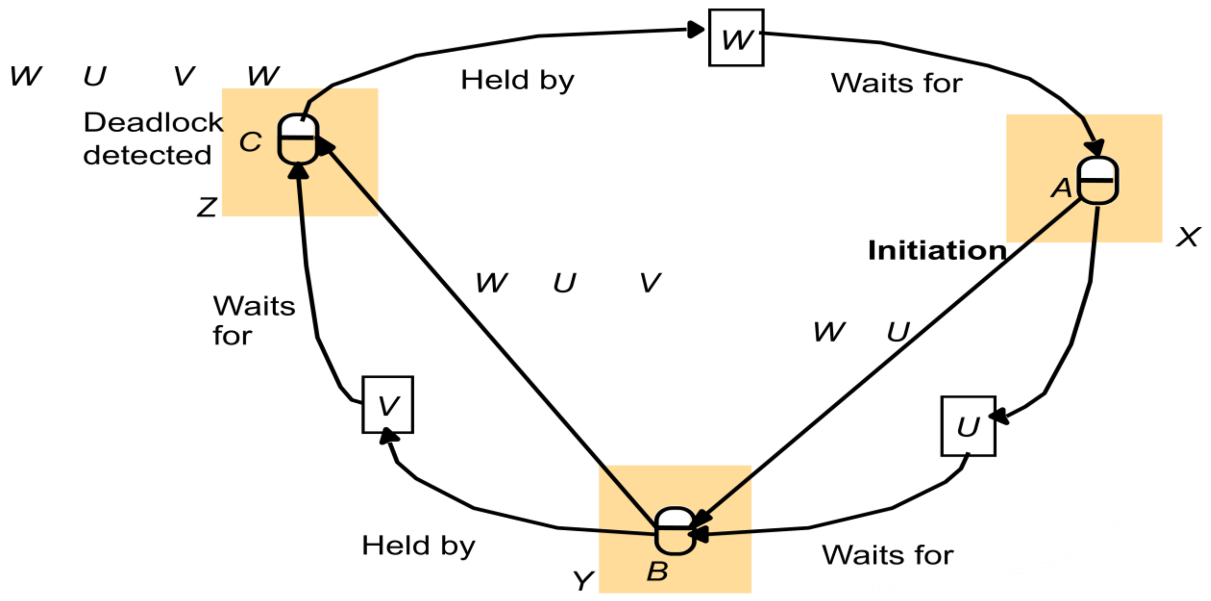
- each coordinator records whether its transactions are active or waiting
  - the local lock manager tells coordinators if transactions start/stop waiting
  - when a transaction is aborted to break a deadlock, the coordinator tells the participants, locks are removed and edges taken from wait-for graphs

<u>Edge-chasing algorithms</u>

- Three steps
  - Initiation:
    - When a server notes that T starts waiting for U, where U is waiting at another server, it initiates detection by sending a probe containing the edge < T ® U > to the server where U is blocked.
    - If U is sharing a lock, probes are sent to all the holders of the lock.
  - Detection:
    - Detection consists of receiving probes and deciding whether deadlock has occurred and whether to forward the probes.
      - e.g. when server receives probe < T ® U > it checks if U is waiting, e.g. U ® V, if so it forwards < T ® U ® V > to server where V waits
      - when a server adds a new edge, it checks whether a cycle is there
  - Resolution:
    - When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

<u>Probes transmitted to detect deadlock</u>

Example of edge chasing starts with X sending <W→ U>, then Y sends <W→ U →V>,then Z sends <W-> U-> V-> W>



<u>Edge chasing conclusion</u>

- probe to detect a cycle with *N* transactions will require 2(*N*-1) messages.
  - Studies of databases show that the average deadlock involves 2 transactions.
- the above algorithm detects deadlock provided that
  waiting transactions do not abort
  - no process crashes, no lost messages
  - to be realistic it would need to allow for the above failures
- refinements of the algorithm
  - to avoid more than one transaction causing detection to start and then more than one being aborted
  - not time to study these now

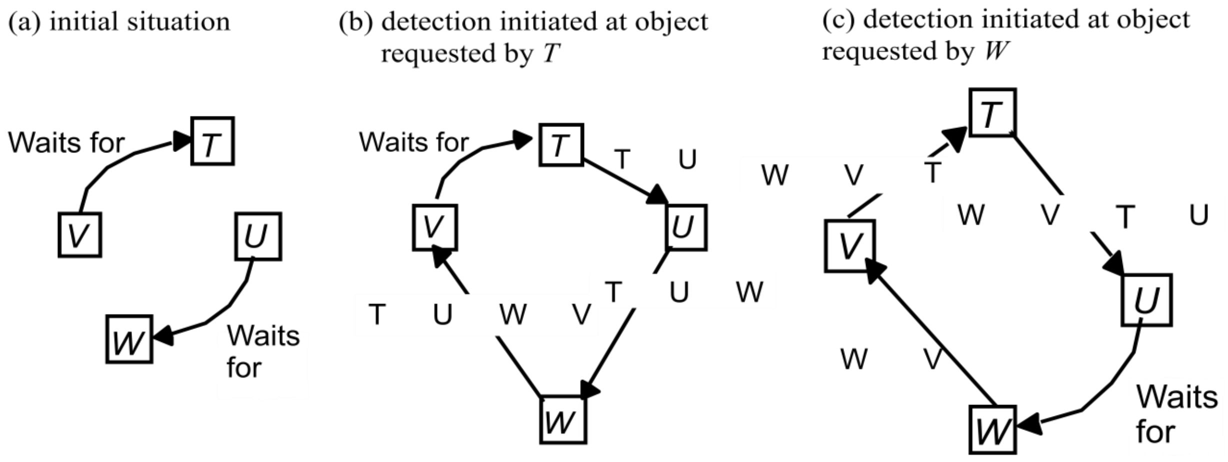

(a) initial situation
(b) detection initiated at object requested by T
(c) detection initiated at object requested by W
Waits for
T
V
U
W
Waits for
T U
W V T
T U W V
T U W
W V T U
W V
Waits for

Figure: Two probes initiated

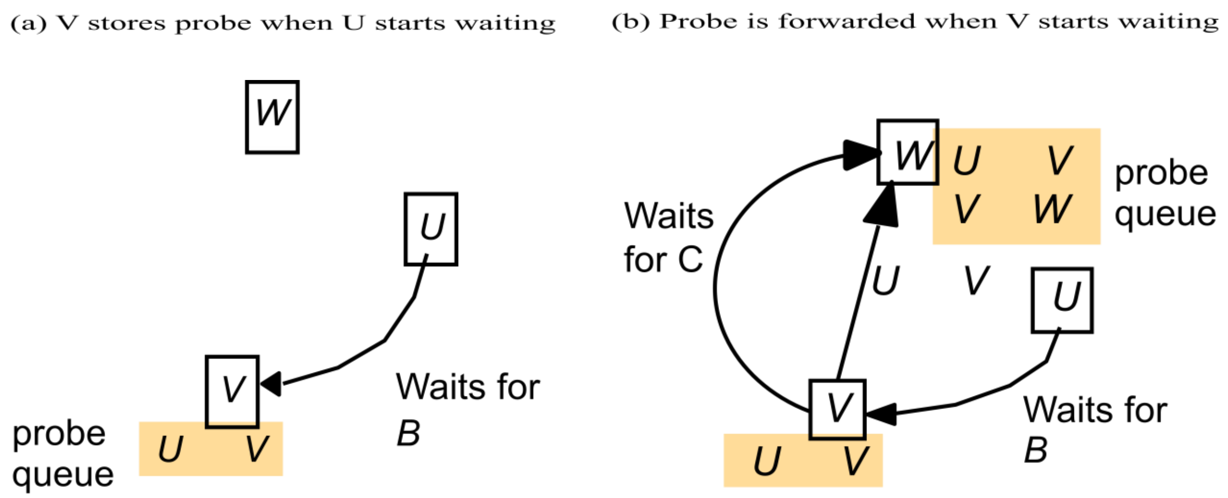

(a) V stores probe when U starts waiting
(b) Probe is forwarded when V starts waiting
W
U
V
probe queue
U V
Waits for B
W
U V
V W
probe queue
Waits for C
U V
U
V
U V
Waits for B

Figure: Probes travel downhill

Topic 06: Transaction recovery

- Atomicity property of transactions
  - durability and failure atomicity

  – durability requires that objects are saved in permanent storage and will be available indefinitely
  – failure atomicity requires that effects of transactions are atomic even when the server crashes

- Recovery is concerned with
  - ensuring that a server's objects are durable and
  - that the service provides failure atomicity.
  - for simplicity we assume that when a server is running, all of its objects are in volatile memory
  - and all of its committed objects are in a *recovery file* in permanent storage
  - recovery consists of restoring the server with the latest committed versions of all of its objects from its recovery file

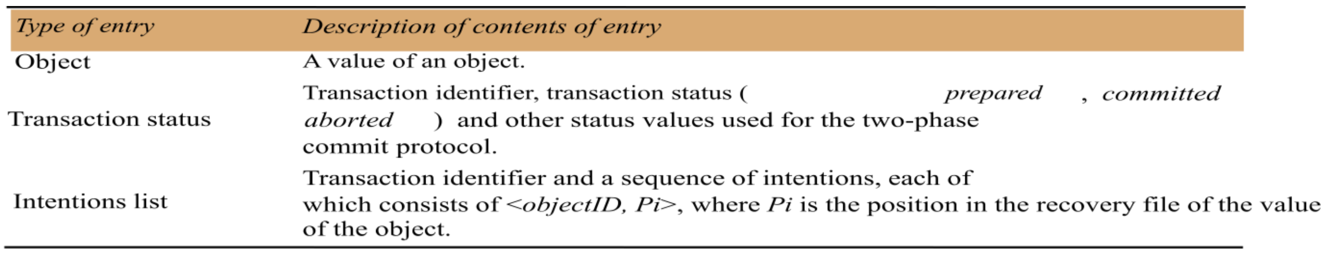
Recovery manager

- The task of the Recovery Manager (RM) is:
  - to save objects in permanent storage (in a recovery file) for committed transactions;
  - to restore the server's objects after a crash;
  - to reorganize the recovery file to improve the performance of recovery;
  - to reclaim storage space (in the recovery file).
- media failures
  - i.e. disk failures affecting the recovery file
  - need another copy of the recovery file on an independent disk. e.g. implemented as stable storage or using mirrored disks

Recovery - intentions lists

- Each server records an intentions list for each of its currently active transactions
  - an intentions list contains a list of the object references and the values of all the objects that are altered by a transaction
  - when a transaction commits, the intentions list is used to identify the objects affected
    - the committed version of each object is replaced by the tentative one
    - the new value is written to the server's recovery file
  - in 2PC, when a participant says it is ready to commit, its RM must record its intentions list and its objects in the recovery file
    - it will be able to commit later on even if it crashes
    - when a client has been told a transaction has committed, the recovery files of all participating servers must show that the transaction is committed,
      - even if they crash between *prepare* to commit and *commit*

Types of entry in a recovery file



Type of entry	*Description of contents of entry*
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared* , *committed* *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <*objectID, Pi*>, where *Pi* is the position in the recovery file of the value of the object.

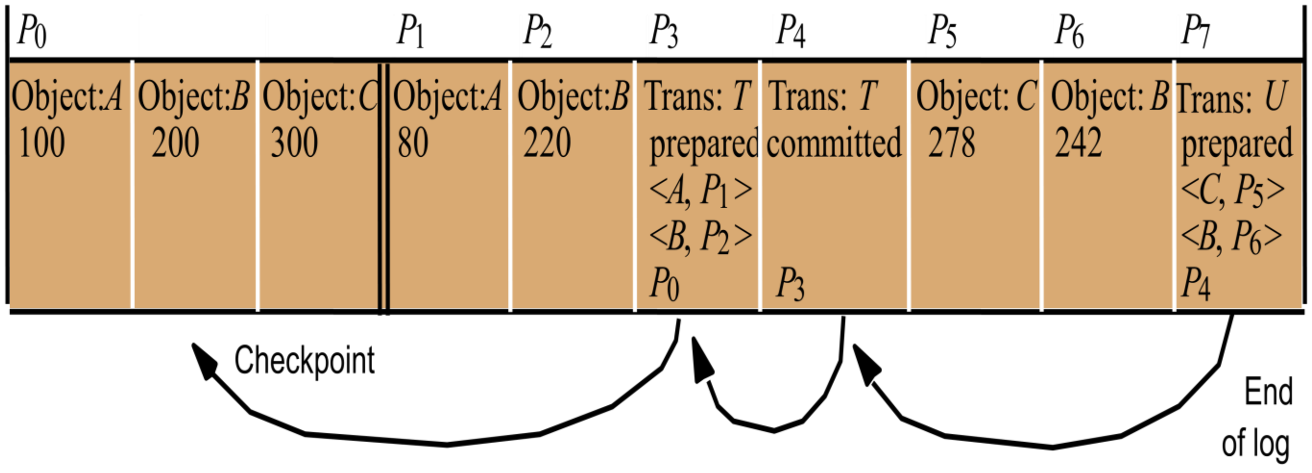
For distributed transactions we need information relating to the 2pc as well as object values, that is:

- Transaction status(committed, prepared or aborted)
- Intentions List

Logging - a technique for the recovery file

- the recovery file represents a log of the history of all the transactions at a server
  - it includes objects, intentions lists and transaction status
  - in the order that transactions prepared, committed and aborted
  - a recent snapshot + a history of transactions after the snapshot
  - during normal operation the RM is called whenever a transaction prepares, commits or aborts
    - prepare - RM appends to recovery file all the objects in the intentions list followed by status (prepared) and the intentions list
    - commit/abort - RM appends to recovery file the corresponding status
    - assume *append* operation is atomic, if server fails only the last write will be incomplete
    - to make efficient use of disk, buffer *writes*. Note: sequential *writes* are more efficient than those to random locations
    - committed status is forced to the log - in case server crashes

Log for banking service



- Logging mechanism (there would really be other objects in log file)
  - initial balances of *A*, *B* and *C* \$100, \$200, \$300
  - *T* sets *A* and *B* to \$80 and \$220. *U* sets *B* and *C* to \$242 and \$278
  - entries to left of line represent a snapshot (checkpoint) of values of *A*, *B* and *C* before *T* started. *T* has committed, but *U* is prepared.
  - the RM gives each object a unique identifier (*A*, *B*, *C* in diagram)
  - each status entry contains a pointer to the previous status entry, then the checkpoint can follow transactions backwards through the file

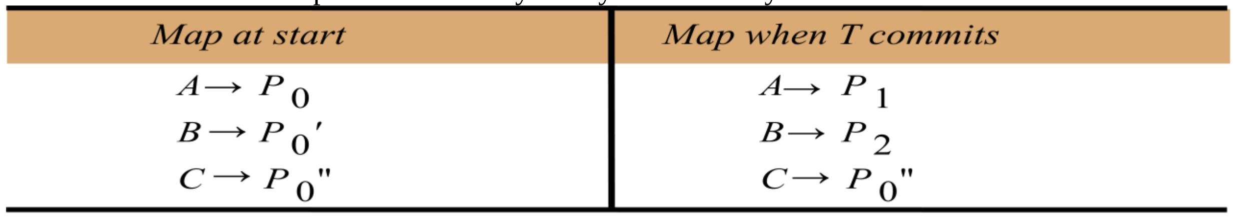
Recovery of objects – with logging

- When a server is replaced after a crash

  - it first sets default initial values for its objects
  - and then hands over to its recovery manager.
- The RM restores the server's objects to include
  - all the effects of all the committed transactions in the correct order and
  - none of the effects of incomplete or aborted transactions
  - it 'reads the recovery file backwards' (by following the pointers)
    - restores values of objects with values from committed transactions
    - continuing until all of the objects have been restored
  - if it started at the beginning, there would generally be more work to do
  - to recover the effects of a transaction use the intentions list to find the value of the objects
    - e.g. look at previous slide (assuming the server crashed before T committed)
  - the recovery procedure must be idempotent

<u>Logging – Reorganizing the recovery file</u>

- RM is responsible for reorganizing its recovery file
  - so as to make the process of recovery faster and
  - to reduce its use of space
- checkpointing
  - the process of writing the following to a new recovery file
    - the current committed values of a server's objects,
    - transaction status entries and intentions lists of transactions that have not yet been fully resolved
    - including information related to the two-phase commit protocol (see later)
  - checkpointing makes recovery faster and saves disk space
    - done after recovery and from time to time
    - can use old recovery file until new one is ready, add a 'mark' to old file
    - do as above and then copy items after the mark to new recovery file
    - replace old recovery file by new recovery file



Map at start	*Map when T commits*
$A \rightarrow P_0$	$A \rightarrow P_1$
$B \rightarrow P_0'$	$B \rightarrow P_2$
$C \rightarrow P_0''$	$C \rightarrow P_0''$

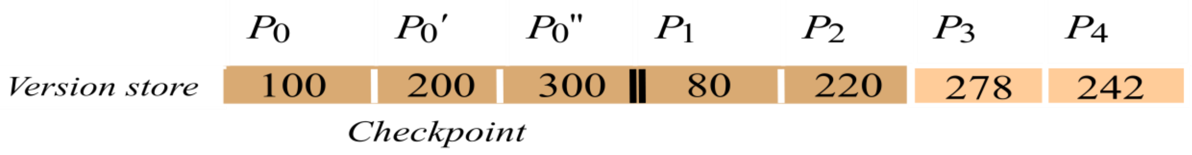


Figure: Shadow versions

<u>Recovery of the two-phase commit protocol</u>

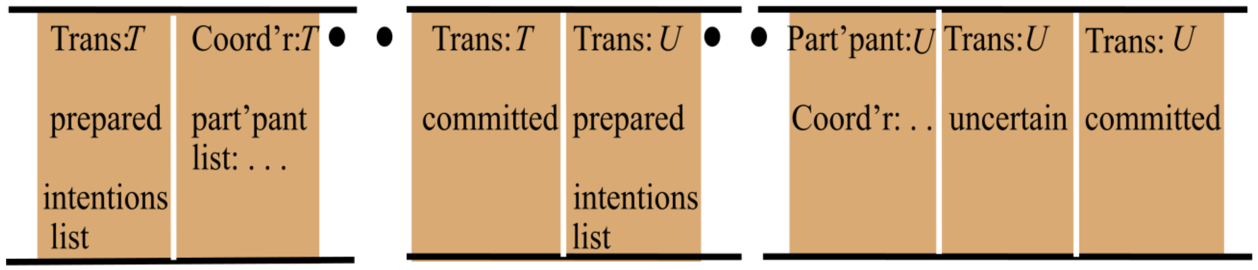
- The above recovery scheme is extended to deal with transactions doing the 2PC protocol when a server fails

- it uses new transaction status values *done, uncertain*
- the coordinator uses *committed* when result is *Yes;*
- *done* when 2PC complete (if a transaction is *done* its information may be removed when reorganising the recovery file)
- the participant uses *uncertain* when it has voted *Yes; committed* when told the result (*uncertain* entries must not be removed from recovery file)

– It also requires two additional types of entry:

Type OF Entry	Description of contents of entry
Coordinator	Transaction identifier, list of participants added by RM when coordinator prepared
Participant	Transaction identifier, coordinator added by RM when participant votes yes

Log with entries relating to two-phase commit protocol



Trans:T prepared intentions list
Coord'r:T part'pant list: . . .
Trans: T committed
Trans: U prepared intentions list
Part'pant:U Coord'r: . .
Trans:U uncertain
Trans: U committed

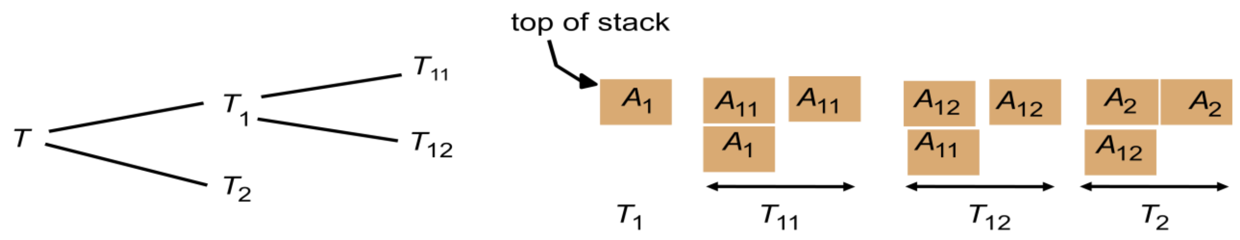
- entries in log for
  - T where server is coordinator (prepared comes first, followed by the coordinator entry, then committed – done is not shown)
  - And U where server is participant (prepared comes first followed by the participant entry, then uncertain and finally committed)
  - These entries will be interspersed with values of objects
- Recovery must deal with 2PC entries as well as restoring objects
  - Where server was coordinator find coordinator entry and status entries.
  - Where server was participant find participant entry and status entries.

Start at end, for U find it is committed and a participant , We have T committed and coordinator, But if the server has crashed before the last entry we have U *uncertain* and participant, or if the server crashed earlier we have U *prepared* and participant

Recovery of the two-phase commit protocol

Role	*Status*	*Action of recovery manager*
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *abortTransaction* to all the servers in the participant list and adds the transaction status*aborted* in its recovery file. Same action for state *aborted*. If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. It sends a *doCommit* to all the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (Fig 13.5).
Participant	*committed*	The participant sends a *haveCommitted* message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	*uncertain*	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a *getDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	*prepared*	The participant has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

Nested transactions:



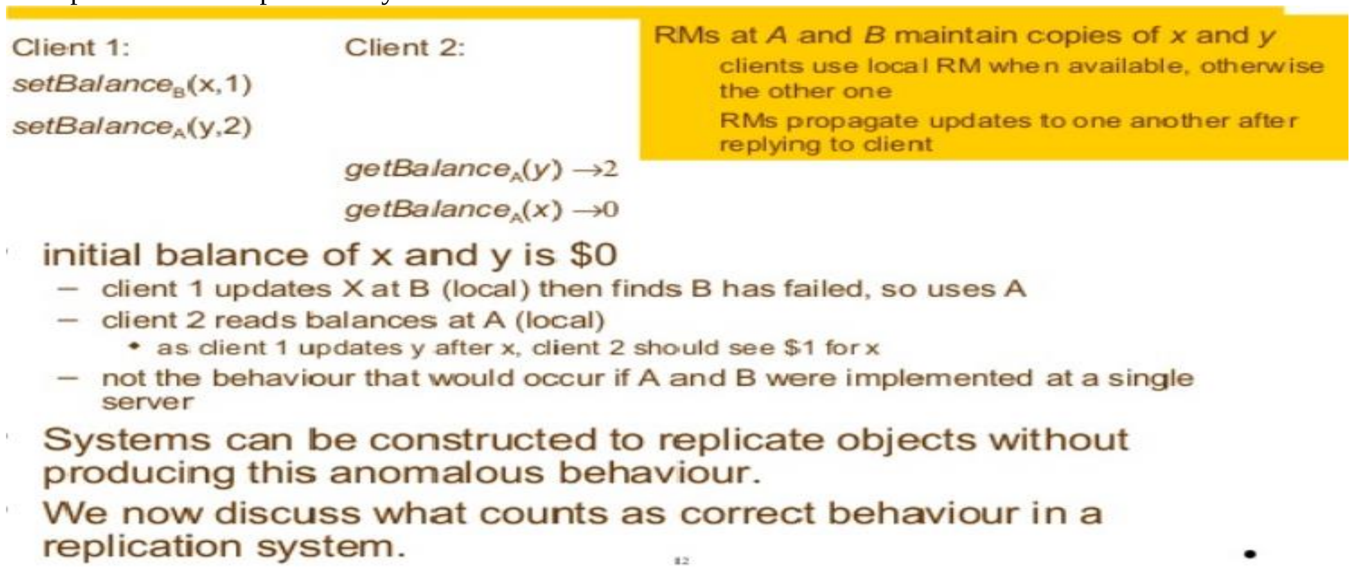
Summary of transaction recovery

- Transaction-based applications have strong requirements for the long life and integrity of the information stored.
- Transactions are made durable by performing checkpoints and logging in a recovery file, which is used for recovery when a server is replaced after a crash.
- Users of a transaction service would experience some delay during recovery.
- It is assumed that the servers of distributed transactions exhibit crash failures and run in an asynchronous system,
  - but they can reach consensus about the outcome of transactions because crashed servers are replaced with new processes that can acquire all the relevant information from permanent storage or from other servers

Topic 07: 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.

Example of a naive replication system



Client 1:

setBalance$_B$(x,1)

setBalance$_A$(y,2)

Client 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.

Figure: Native Replication System

Linearizability the strictest criterion for a replication system

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 o10, o11, o12 and o20, o21, o22 at a single server are interleaved in some order e.g. o20, o21, o10, o22 , o11, o12 (client 1 does o10 etc)

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

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

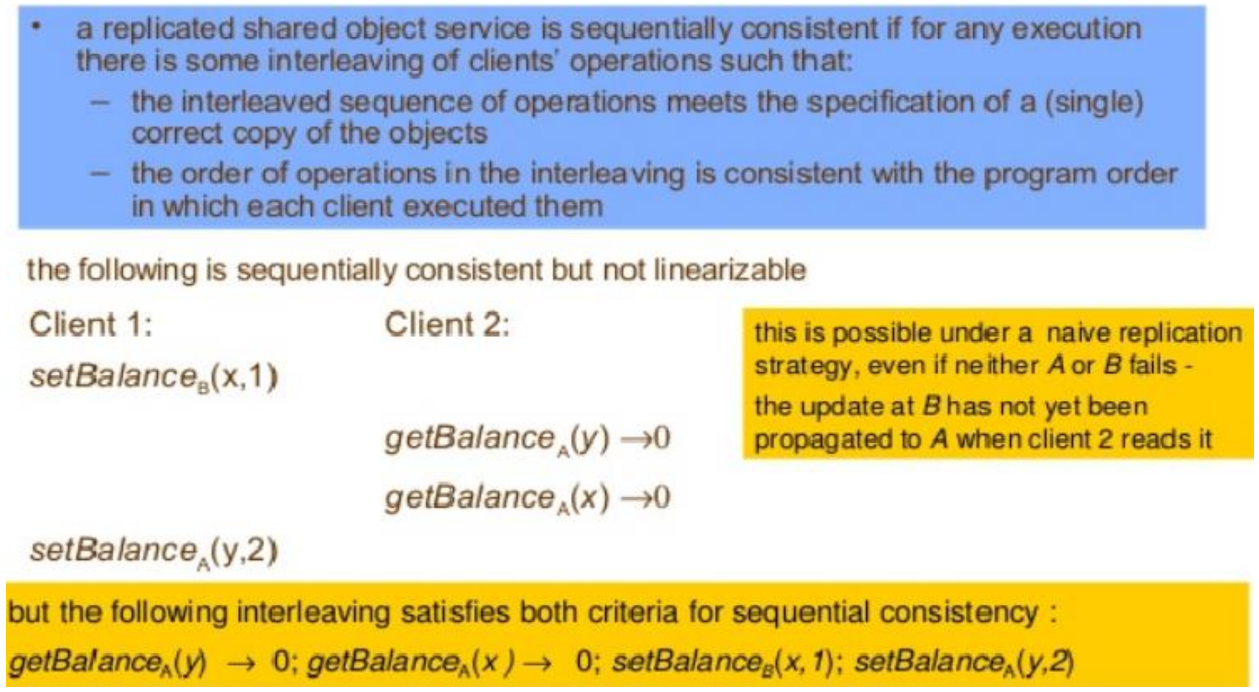
- the bank example did not make sense: if the second update is observed,the first update should be observed too.

– 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

<u>Sequential consistency</u>



- 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:
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

but the following interleaving satisfies both criteria for sequential consistency :
getBalance$_A$(*y*) $\rightarrow$ 0; *getBalance*$_A$(*x*) $\rightarrow$ 0; *setBalance*$_B$(*x*, 1); *setBalance*$_A$(*y*,2)

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

<u>The passive (primary-backup) model for fault tolerance</u>

- 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

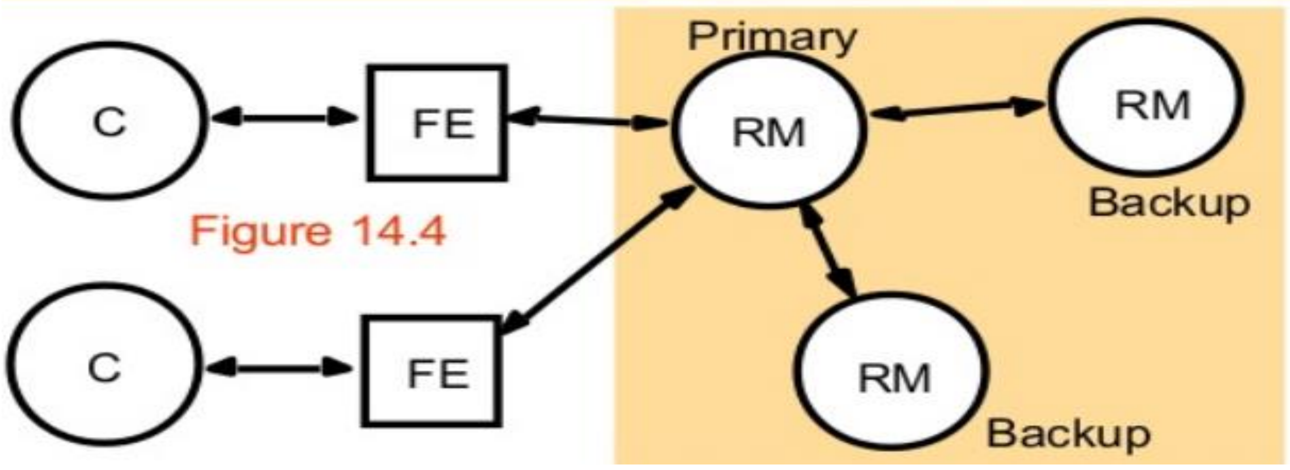


Figure 14.4

The FE has to find the primary, e.g. after it crashes and another takes over
<u>Passive (primary-backup) replication. Five phases.</u>

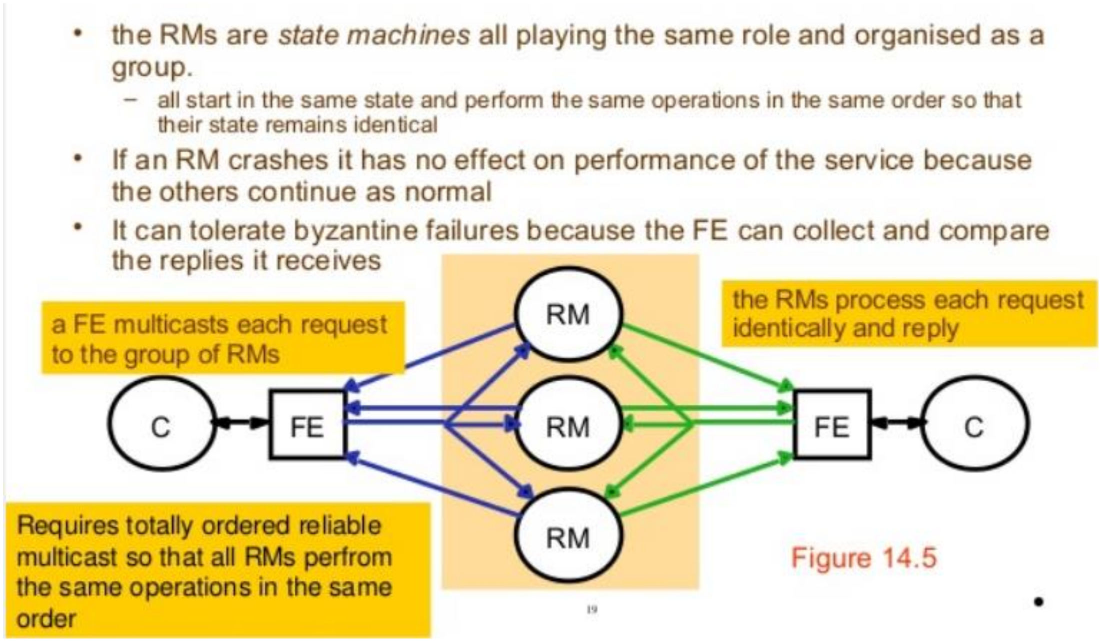
- The five phases in performing a client request are as follows:
- 1. Request:
  - a FE issues the request, containing a unique identifier, to the primary RM
- 2. Coordination:
  - the primary performs each request atomically, in the order in which it receives it relative to other requests
  - it checks 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.

<u>Discussion of passive replication</u>

- 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 uses either master or slave
  - updates are not done via RMs - they are made on the files at the master

<u>Active replication for fault tolerance</u>



- 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

a FE multicasts each request to the group of RMs

the RMs process each request identically and reply

Requires totally ordered reliable multicast so that all RMs perfrom the same operations in the same order

Figure 14.5

<u>Active replication - five phases in performing a client request</u>

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