UNIT -04
DISTRIBUTED TRANSACTIONS
UNIT -04/LECTURE- 01

Introduction

We use the term distributed transaction to refer to a flat or nested transaction that accesses objects managed by multiple servers. When a distributed transaction comes to an end, the atomicity property of\ transactions requires that either all of the servers involved commit the transaction or all of them abort the transaction. To achieve this, one of the servers takes on a coordinator\ role, which involves ensuring the same outcome at all of the servers. The manner in which the coordinator achieves this depends on the protocol chosen. A protocol known as the 'two-phase commit protocol' is the most commonly used. This protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort.

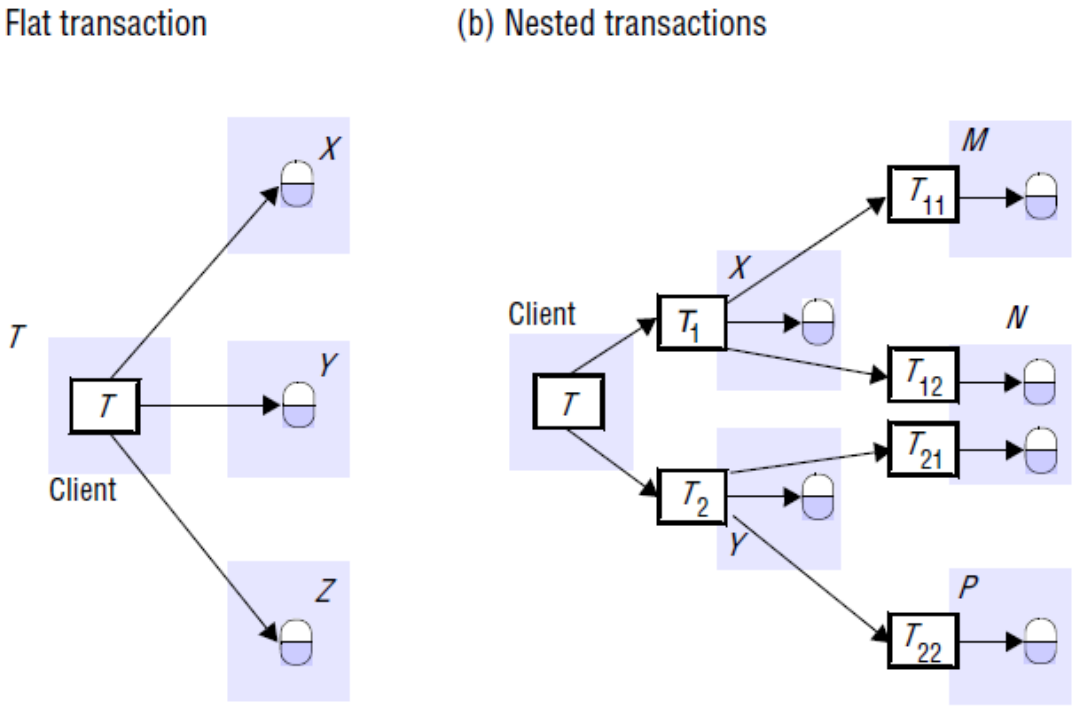
Flat and nested distributed transactions ([RGPV/ Dec 2013 (7)]

A client transaction becomes distributed if it invokes operations in several different servers. There are two different ways that distributed transactions can be structured: as flat transactions and as nested transactions.

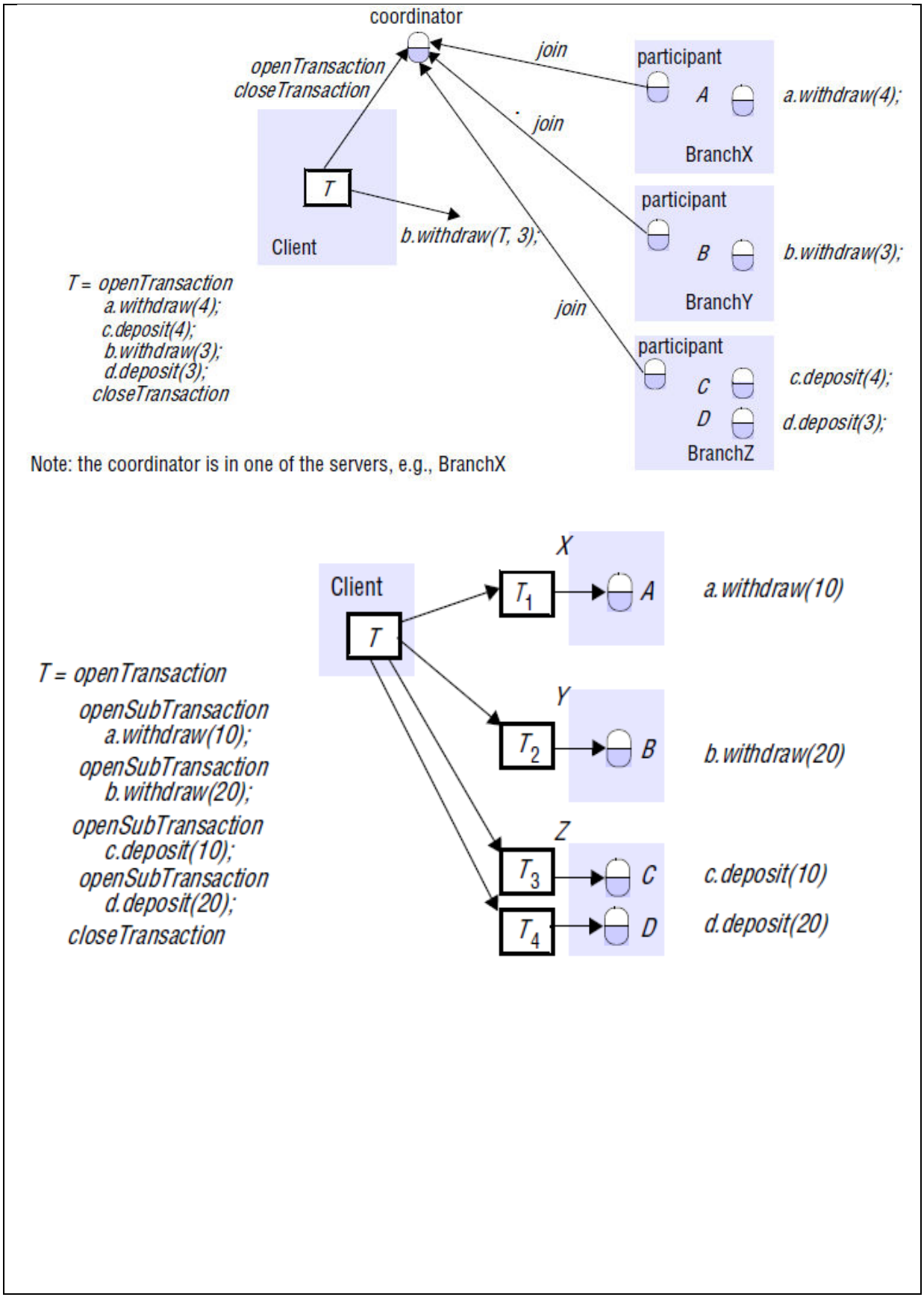
In a flat transaction, a client makes requests to more than one server. For example, transaction T is a flat transaction that invokes operations on objects in servers X, Y and Z. A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction accesses servers' objects sequentially. When servers use locking, a transaction can only be waiting for one object at a time. In a nested transaction, the top-level transaction can open subtransactions, and\ each subtransaction can open further subtransactions down to any depth of nesting.

A client transaction T that opens two subtransactions, T1 and T2, which access objects at servers X and Y. The subtransactions T1 and T2 open further subtransactions T11, T12, T21, and T22, which access objects at servers M, N and P. In the nested case, subtransactions at the same level can run concurrently, so T1 and T2 are concurrent, and as they invoke

objects in different servers, they can run in parallel. The four subtransactions T11, T12, T21 and T22 also run concurrently.



Consider a distributed transaction in which a client transfers $10 from account A to C and then transfers $20 from B to D. Accounts A and B are at separate servers X and Y and accounts C and D are at server Z. If this transaction is structured as a set of four nested transactions, the four requests (two deposits and two withdraws) can run in parallel and the overall effect can be achieved with better performance than a simple transaction in which the four operations are invoked sequentially.


coordinator
openTransaction
closeTransaction
join
join
join
participant
A
BranchX
a.withdraw(4);
participant
B
BranchY
b.withdraw(3);
participant
C
D
BranchZ
c.deposit(4);
d.deposit(3);
T
Client
b.withdraw(T, 3);
T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction
Note: the coordinator is in one of the servers, e.g., BranchX
Client
T
X
T1
A
a.withdraw(10)
Y
T2
B
b.withdraw(20)
Z
T3
C
c.deposit(10)
T4
D
d.deposit(20)
T = openTransaction
openSubTransaction
a.withdraw(10);
openSubTransaction
b.withdraw(20);
openSubTransaction
c.deposit(10);
openSubTransaction
d.deposit(20);
closeTransaction

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are nested transactions and what is their role in a distributed system.	Dec 2013	7

UNIT -04/LECTURE -02

The coordinator of a distributed transaction

Servers that execute requests as part of a distributed transaction need to be able to communicate with one another to coordinate their actions when the transaction commits. A client starts a transaction by sending an openTransaction request to a coordinator in any server. The coordinator that is contacted carries out the openTransaction and returns the resulting transaction identifier (TID) to the client. Transaction identifiers for distributed transactions must be unique within a distributed system. A simple way to achieve this is for a TID to contain two parts: the identifier (for example, an IP address) of the server that created it and a number unique to the server.

The coordinator that opened the transaction becomes the coordinator for the distributed transaction and at the end is responsible for committing or aborting it. Each of the servers that manages an object accessed by a transaction is a participant in the transaction and provides an object we call the participant. Each participant is responsible for keeping track of all of the recoverable objects at that server that are involved, in the transaction. The participants are responsible for cooperating with the coordinator in carrying out the commit protocol.

During the progress of the transaction, the coordinator records a list of references to the participants, and each participant records a reference to the coordinator. The interface for Coordinator shown provides an additional method, join, which is used whenever a new participant joins the transaction:

join(Trans, reference to participant)

Informs a coordinator that a new participant has joined the transaction Trans. The coordinator records the new participant in its participant list. The fact that the coordinator

knows all the participants and each participant knows the coordinator will enable them to collect the information that will be needed at commit time.

A client whose (flat) banking transaction involves accounts A, B, C and D at servers BranchX, BranchY and BranchZ. The client's transaction, T, transfers $4 from account A to account C and then transfers $3 from account B to account D. The transaction described on the left is expanded to show that openTransaction and closeTransaction are directed to the coordinator, which would be situated in one of the servers involved in the transaction. Each server is shown with a participant, which joins the transaction by invoking the join method in the coordinator.

When the client invokes one of the methods in the transaction, for example b.withdraw(T, 3), the object receiving the invocation (B at BranchY, in this case) informs its participant object that the object belongs to the transaction T. If it has not already informed the coordinator, the participant object uses the join operation to do so. In this example, we show the transaction identifier being passed as an additional argument so that the recipient can pass it on to the coordinator. By the time the client calls closeTransaction, the coordinator has references to all of the participants. Note that it is possible for a participant to call abortTransaction in the coordinator if for some reason it is unable to continue with the transaction.

Atomic Commit Protocols ([RGPV/ June 2014 (7)]

Transaction commit protocols were devised in the early 1970s, and the two-phase commit protocol appeared in Gray. The atomicity property of transactions requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them. In the case of a distributed transaction, the client has requested operations at more than one server. A transaction comes to an end when the client requests that it be committed or aborted. A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the commit or abort request to all of the participants in

the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out. This is an example of a onephase atomic commit protocol.

This simple one-phase atomic commit protocol is inadequate, though, because it does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit. Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control. For example, if locking is in use, the resolution of a deadlock can lead to the aborting of a transaction without the client being aware unless it makes another request to the server. Also if optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction. Finally, the coordinator may not know if a server has crashed and been replaced during the progress of a distributed transaction – such a server will need to abort the transaction.

The two-phase commit protocol is designed to allow any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must be aborted. In the first phase of the protocol, each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a prepared state for a transaction if it will eventually be able to commit it. To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its status – prepared.

In the second phase of the protocol, every participant in the transaction carries out the joint decision. If any one participant votes to abort, then the decision must be to abort the transaction. If all the participants vote to commit, then the decision is to commit the transaction. The problem is to ensure that all of the participants vote and that they all reach

the same decision. This is fairly simple if no errors occur, but the protocol must work correctly even when some of the servers fail, messages are lost or servers are temporarily unable to communicate with one another.

Failure model for the commit protocols

A failure model for transactions that applies equally to the two-phase (or any other) commit protocol. Commit protocols are designed to work in an asynchronous system in which servers may crash and messages may be lost. It is assumed that an underlying request-reply protocol removes corrupt and duplicated messages. There are no Byzantine faults – servers either crash or obey the messages they are sent.

The two-phase commit protocol is an example of a protocol for reaching a consensus. However, the two-phase commit protocol does reach consensus under those conditions. This is because crash failures of processes are masked by replacing a crashed process with a new process whose state is set from information saved in permanent storage and information held by other processes.

The two-phase commit protocol

During the progress of a transaction, there is no communication between the coordinator and the participants apart from the participants informing the coordinator when they join the transaction. A client's request to commit (or abort) a transaction is directed to the coordinator. If the client requests abortTransaction, or if the transaction is aborted by one of the participants, the coordinator informs all participants immediately. It is when the client asks the coordinator to commit the transaction that the two-phase commit protocol comes into use.

In the first phase of the two-phase commit protocol the coordinator asks all the participants if they are prepared to commit; in the second, it tells them to commit (or abort) the transaction. If a participant can commit its part of a transaction, it will agree as soon as it

has recorded the changes it has made (to the objects) and its status in permanent storage and is therefore prepared to commit. The coordinator in a distributed transaction communicates with the participants to carry out the two-phase commit protocol by means of the operations summarized. The methods canCommit, doCommit and doAbort are methods in the interface of the participant. The methods haveCommitted and getDecision are in the coordinator interface.

The two-phase commit protocol consists of a voting phase and a completion phase. By the end of step 2, the coordinator and all the participants that voted Yes are prepared to commit. By the end of step 3, the transaction is effectively completed. At step 3(A) the coordinator and the participants are committed, so the coordinator can report a decision to commit to the client. At 3(B) the coordinator reports a decision to abort to the client. At step 4 participants confirm that they have committed so that the coordinator knows when the information it has recorded about the transaction is no longer needed. This apparently straightforward protocol could fail due to one or more of the servers crashing or due to a breakdown in communication between the servers.

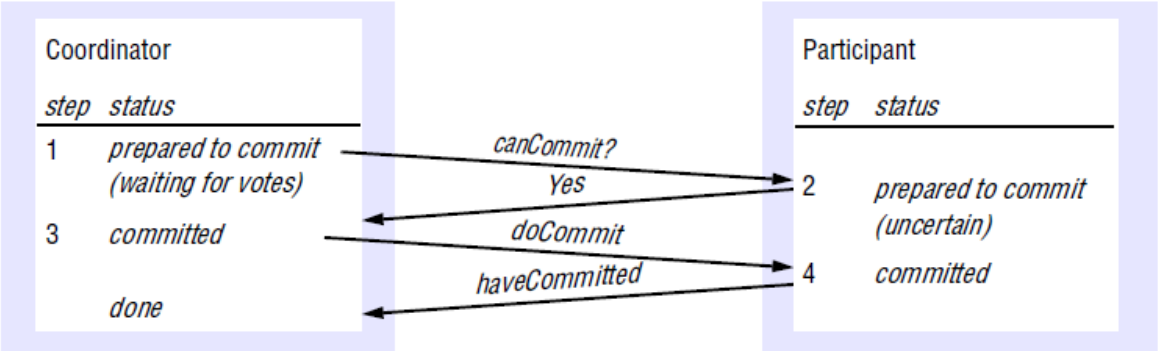
To deal with the possibility of crashing, each server saves information relating to the two-phase commit protocol in permanent storage. This information can be retrieved by a new process that is started to replace a crashed server. The exchange of information between the coordinator and participants can fail when one of the servers crashes, or when messages are lost. Timeouts are used to avoid processes blocking forever. When a timeout occurs at a process, it must take an appropriate action. To allow for this the protocol includes a timeout action for each step at which a process may block. These actions are designed to allow for the fact that in an asynchronous system, a timeout may not necessarily imply that a server has failed.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain atomic commit protocols in distributed transaction processing.	June 2014	7

UNIT 04/LECTURE 03

Timeout actions in the two-phase commit protocol

There are various stages in the protocol at which the coordinator or a participant cannot progress its part of the protocol until it receives another request or reply from one of the others. Consider first the situation where a participant has voted Yes and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort the transactions. Such a participant is uncertain of the outcome and cannot proceed any further until it gets the outcome of the vote from the coordinator. The participant cannot decide unilaterally what to do next, and meanwhile the objects.



It is used by its transaction cannot be released for use by other transactions. The participant can make a getDecision request to the coordinator to determine the outcome of the transaction. When it gets the reply, it continues the protocol at step 4 . If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the uncertain state. Alternative strategies are available for the participants to obtain a decision cooperatively instead of contacting the coordinator. These strategies have the advantage that they may be used when the coordinator has failed.

However, even with a cooperative protocol, if all the participants are in the uncertain state, they will be unable to get a decision until the coordinator or a participant with the necessary knowledge is available. Another point at which a participant may be delayed is when it has carried out all its client requests in the transaction but has not yet received a canCommit? call from the coordinator. As the client sends the closeTransaction to the coordinator, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time – for example, by the time a timeout period on a lock expires.

As no decision has been made at this stage, the participant can decide to abort unilaterally. The coordinator may be delayed when it is waiting for votes from the participants. As it has not yet decided the fate of the transaction it may decide to abort the transaction after some period of time. It must then announce doAbort to the participants who have already sent their votes. Some tardy participants may try to vote Yes after this, but their votes will be ignored and they will enter the uncertain state as described .

Performance of the two-phase commit protocol

Provided that all goes well – that is, that the coordinator, the participants and the communications between them do not fail – the two-phase commit protocol involving N participants can be completed with N canCommit? messages and replies, followed by N doCommit messages. That is, the cost in messages is proportional to 3N, and the cost in time is three rounds of messages. The have Committed messages are not counted in the estimated cost of the protocol, which can function correctly without them – their role is to enable servers to delete stale coordinator information.

In the worst case, there may be arbitrarily many server and communication failures during the two-phase commit protocol. However, the protocol is designed to tolerate a succession

of failures (server crashes or lost messages) and is guaranteed to complete eventually, although it is not possible to specify a time limit within which it will be completed.

As noted, the two-phase commit protocol can cause considerable delays to participants in the uncertain state. These delays occur when the coordinator has failed and cannot reply to getDecision requests from participants. Even if a cooperative protocol allows participants to make getDecision requests to other participants, delays will occur if all the active participants are uncertain. Three-phase commit protocols have been designed to alleviate such delays, but they are more expensive in terms of the number of messages and the number of rounds required for the normal (failure-free) case. For a description of three-phase commit.

Two-phase commit protocol for nested transactions ([RGPV/ June 2013(7), ([RGPV/June 2014(7)]

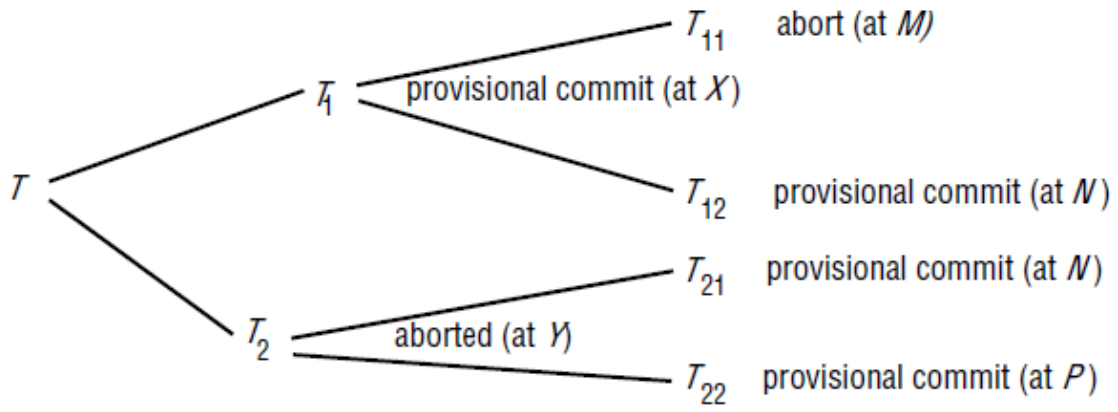
The outermost transaction in a set of nested transactions is called the top-level transaction. Transactions other than the top-level transaction are called subtransactions. In Figure T is the top-level transaction and T1, T2, T11, T12, T21 and T22 are subtransactions. T1 and T2 are child transactions of T, which is referred to as their parent. Similarly, T11 and T12 are child transactions of T1, and T21 and T22 are child transactions of T2. Each subtransaction starts after its parent and finishes before it. Thus, for example, T11 and T12 start after T1 and finish before it.

When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A provisional commit is different from being prepared to commit: nothing is backed up in permanent storage. If the server crashes subsequently, its replacement will not be able to commit. After all subtransactions have completed, the provisionally committed ones participate in a two-phase commit protocol, in which servers of provisionally committed subtransactions express their intention to commit and those with an aborted ancestor will abort. Being prepared to commit guarantees a subtransaction

will be able to commit, whereas a provisional commit only means it has finished correctly – and will probably agree to commit when it is subsequently asked to.

A coordinator for a subtransaction will provide an operation to open a subtransaction, together with an operation enabling that coordinator to enquire whether its parent has yet committed or aborted, as shown. Asks the coordinator to report on the status of the transaction trans. Returns values representing one of the following: committed, aborted or provisional. A client starts a set of nested transactions by opening a top-level transaction with an openTransaction operation, which returns a transaction identifier for the top-level transaction. The client starts a subtransaction by invoking the openSubTransaction operation, whose argument specifies its parent transaction. The new subtransaction automatically joins the parent transaction, and a transaction identifier for a subtransaction is returned.

An identifier for a subtransaction must be an extension of its parent's TID, constructed in such a way that the identifier of the parent or top-level transaction of a subtransaction can be determined from its own transaction identifier. In addition, all subtransaction identifiers should be globally unique. The client makes a set of nested transactions come to completion by invoking closeTransaction or abortTransaction on the coordinator of the top-level transaction. Meanwhile, each of the nested transactions carries out its operations. When they are finished, the server managing a subtransaction records information as to whether the subtransaction committed provisionally or aborted. Note that if its parent aborts, then the subtransaction will be forced to abort too.



A parent transaction – including a top-level transaction – can commit even if one of its child subtransactions has aborted. In such cases, the parent transaction will be programmed to take different actions according to whether a subtransaction has committed or aborted. For example, consider a banking transaction that is designed to perform all the 'standing orders' at a branch on a particular day. This transaction is expressed as several nested Transfer subtransactions, each of which consists of nested deposit and withdraw subtransactions. We assume that when an account is overdrawn, withdraw aborts and then the corresponding Transfer aborts. But there is no need to abort all the standing orders just because one Transfer subtransaction aborts.

Instead of aborting, the top-level transaction will note the Transfer subtransactions that aborted and take appropriate actions. Consider the top-level transaction T and its subtransactions shown . . Each subtransaction has either provisionally without giving any information about its descendants. Eventually, the top-level transaction receives a list of all the subtransactions in the tree, together with the status of each. Descendants of aborted subtransactions are omitted from this list.

Information held by coordinators of nested transactions

Coordinator of transaction	*Child transactions*	*Participant*	*Provisional commit list*	*Abort list*
T	T_1, T_2	yes	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	yes	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	no (aborted)		T_2
T_{11}		no (aborted)		T_{11}
T_{12}, T_{21}		T_{12} but not T_{21}*	T_{21}, T_{12}	
T_{22}		no (parent aborted)	T_{22}	

* T_{21}'s parent has aborted

When subtransaction T2 aborted, it reported the fact to its parent, T, but without passing on any information about its subtransactions T21 and T22. A subtransaction is an orphan if one of its ancestors aborts, either explicitly or because its coordinator crashes. In our example, subtransactions T21 and T22 are orphans because their parent aborted without passing information about them to the top-level transaction. Their coordinator can, however, make enquiries about the status of their parent by using the getStatus operation. A provisionally committed subtransaction of an aborted transaction should be aborted, irrespective of whether the top-level transaction eventually commits.The top-level transaction plays the role of coordinator in the two-phase commit protocol, and the participant list consists of the coordinators of all the subtransactions in the tree that have provisionally committed but do not have aborted ancestors. By this stage, the logic of the program has determined that the top-level transaction should try to commit whatever is left, in spite of some aborted subtransactions.

The coordinators of T, T1 and T12 are participants and will be asked to vote on the outcome. If they vote to commit, then they must prepare their transactions by saving the state of the objects in permanent storage. This state is recorded as belonging to the top-level transaction of which it will form a part. The two-phase commit protocol may be performed in either a hierarchic manner or a flat manner.

The second phase of the two-phase commit protocol is the same as for the nonnested case. The coordinator collects the votes and then informs the participants as to the outcome. When it is complete, coordinator and participants will have committed or aborted their transactions.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain Two-phase commit protocol for nested transactions.	June 2013, June 2014	7

UNIT- 04/LECTURE -04

Concurrency control in distributed transactions ([RGPV/June 2014 (7)])

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. This implies that 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.

Locking

In a distributed transaction, the locks on an object are held locally (in the same server). The local lock manager can decide whether to grant a lock or make the requesting transaction wait. However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction. When locking is used for concurrency control, the objects remain locked and are unavailable for other transactions during the atomic commit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.

As lock managers in different servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions. Consider the following interleaving of transactions T and U at servers X and Y:

T			*U*		
write(A)	at *X*	locks *A*			
			write(B)	at *Y*	locks *B*
read(B)	at *Y*	waits for *U*			
			read(A)	at *X*	waits for *T*

The transaction T locks object A at server X, and then transaction U locks object B at server Y. After that, T tries to access B at server Y and waits for U's lock. Similarly, transaction U tries to access A at server X and has to wait for T's lock. Therefore, we have T before U in one server and U before T in the other. These different orderings can lead to cyclic dependencies between transactions, giving rise to a distributed deadlock situation. When a deadlock is detected, a transaction is aborted to resolve the deadlock. In this case, the coordinator will be informed and will abort the transaction at the participants involved in the transaction.

Timestamp ordering concurrency control

In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them. In distributed transactions, we require that each coordinator issue globally unique timestamps. A globally unique transaction timestamp is issued to the client by the first coordinator accessed by a transaction. The transaction timestamp is passed to the coordinator at each server whose objects perform an operation in the transaction.

The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. For example, if the version of an object accessed by transaction U commits after the version accessed by T at one server, if T and U access the

same object as one another at other servers they must commit them in the same order. To achieve the same ordering at all the servers, the coordinators must agree as to the ordering of their timestamps. A timestamp consists of a <local timestamp, server-id> pair. The agreed ordering of pairs of timestamps is based on a comparison in which the server-id part is less significant.

The same ordering of transactions can be achieved at all the servers even if their local clocks are not synchronized. However, for reasons of efficiency it is required that the timestamps issued by one coordinator be roughly synchronized with those issued by the other coordinators. When this is the case, the ordering of transactions generally corresponds to the order in which they are started in real time. Timestamps can be kept roughly synchronized by the use of synchronized local physical clocks. When timestamp ordering is used for concurrency control, conflicts are resolved as each operation is performed using the rules. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants. Therefore any transaction that reaches the client request to commit should always be able to commit, and participants in the two-phase commit protocol will normally agree to commit. The only situation in which a participant will not agree to commit is if it has crashed during the transaction.

Optimistic concurrency control

Recall that with optimistic concurrency control, each transaction is validated before it is allowed to commit. Transaction numbers are assigned at the start of validation and transactions are serialized according to the order of the transaction numbers. A distributed transaction is validated by a collection of independent servers, each of which validates transactions that access its own objects. This validation takes place during the first phase of the two-phase commit protocol.

Consider the following interleaving of transactions T and U, which access objects A and B at

servers X and Y, respectively:

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)*	

The transactions access the objects in the order T before U at server X and in the order U before T at server Y. Now suppose that T and U start validation at about the same time, but server X validates T first and server Y validates U first. A simplification of the validation protocol that makes a rule that only one transaction may perform validation and update phases at a time. Therefore each server will be unable to validate the other transaction until the first one has completed. This is an example of commitment deadlock.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain concurrency control in distributed processing with suitable example.	June 2014	7

UNIT -04/LECTURE -05

Distributed deadlocks ([RGPV/Dec 2012(7)], ([RGPV/Dec 2013, (7)])

Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks. Using timeouts to resolve possible deadlocks is a clumsy approach – it is difficult to choose an appropriate timeout interval, and transactions may be aborted unnecessarily. With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock. Most deadlock detection schemes operate by finding cycles in the transaction wait-for graph. In a distributed system involving multiple servers being accessed by multiple transactions, a global.

Interleaving of transaction U,V and W

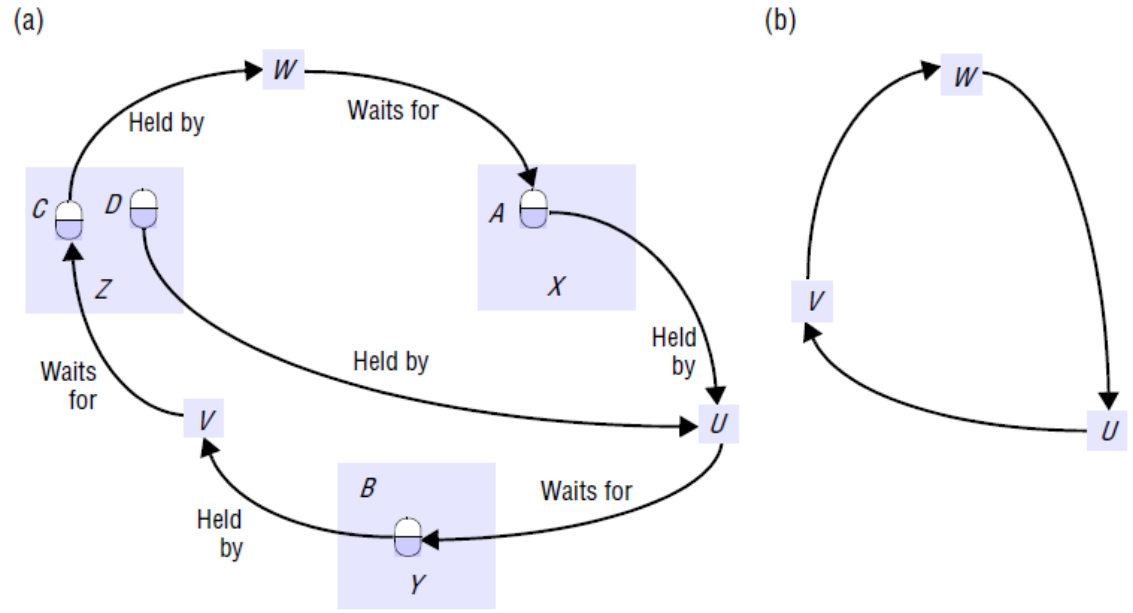
U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B*		
a.deposit(20)	lock *A*		at *Y*		
	at *X*				
				c.deposit(30)	lock *C*
b.withdraw(30)	wait at *Y*				at *Z*
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

Wait-for graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a distributed deadlock. Recall that the wait-for graph is a directed graph in which nodes represent transactions and objects, and edges represent either an object held by a transaction or a transaction waiting for an object. There is a deadlock if and only if there is a cycle in the wait-for graph the interleavings the interleavings of the transactions U, V and W involving the objects A and B managed by servers X and Y and objects C and D managed by server Z.

The complete wait-for graph shows that a deadlock cycle consists of alternate edges, which represent a transaction waiting for an object and an object held by a transaction. As any transaction can only be waiting for one object at a time, objects can be left out of wait-for graphs, as shown. Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions. Local wait-for graphs can be built by the lock manager at each server. In the above example, the local wait-for graphs of the servers are:

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))

As the global wait-for graph is held in part by each of the several servers involved, communication between these servers is required to find cycles in the graph. A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for graph. The global deadlock detector checks for cycles in the global wait-for graph.



Distributed Deadlock ([RGPV/Dec 2011 (7)])

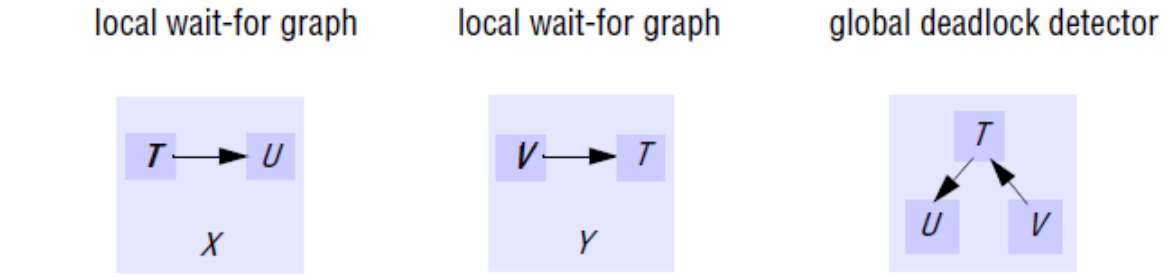
When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort. Centralized deadlock detection is not a good idea, because it depends on a single server to carry it out. It suffers from the usual problems associated with centralized solutions in distributed systems – poor availability, lack of fault tolerance and no ability to scale. In addition, the cost of the frequent transmission of local wait-for graphs is high. If the global graph is collected less frequently, deadlocks may take longer to be detected.

Phantom deadlocks

A deadlock that is 'detected' but is not really a deadlock is called a phantom deadlock. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. As this procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Consider the case of a global deadlock detector that receives local wait-for graphs from servers X and Y, as shown. Suppose that transaction U then releases an object at server X and requests the one held by V at server Y. Suppose also that the global detector receives server Y's local graph before server X's. In this case, it would detect a cycle T □ U □ V □ T, although the edge T □ U no longer exists. This is an example of a phantom deadlock. The observant reader will have realized that if transactions are using two-phase locks, they cannot release objects and then obtain more objects, and phantom deadlock.

Local and global wait-for graphs



Consider the situation in which a cycle T □ U □ V □ T is detected: either this represents a deadlock or each of the transactions T, U and V must eventually commit. It is actually impossible for any of them to commit, because each of them is waiting for an object that will never be released. A phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure. For example, if there is a cycle T □ U □ V □ T and U aborts after the information concerning U has been collected, then the cycle has been broken already and there is no deadlock.

Edge chasing

A distributed approach to deadlock detection uses a technique called edge chasing or path pushing. In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called probes, which follow the edges of the graph throughout the

distributed system. A probe message consists of transaction wait-for relationships representing a path in the global wait-for graph.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What are the methods to prevent distributed deadlock.	Dec 2013	7
Q.2	Discuss about various deadlock handling techniques.	Dec 2014	7
Q.3	What are the different deadlock handling strategies? Explain a distributed Deadlock Detection algorithm.	Dec 2011	7

UNIT -04/LECTURE -06

Transaction recovery ([RGPV/Dec 2012(7)]

The atomic property of transactions requires that all the effects of committed transactions and none of the effects of incomplete or aborted transactions are reflected in the objects they accessed. This property can be described in terms of two aspects: durability and failure atomicity. Durability requires that objects are saved in permanent storage and will be available indefinitely thereafter. Therefore an acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's (volatile) objects. 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. Although file servers and database servers maintain data in permanent storage, other kinds of servers of recoverable objects need not do so except for recovery purposes. In this chapter, we assume that when a server is running it keeps all of its objects in its volatile memory and records its committed objects in a recovery file or files. Therefore recovery consists of restoring the server with the latest committed versions of its objects from permanent storage. Databases need to deal with large volumes of data. They generally hold the objects in stable storage on disk with a cache in volatile memory.

The requirements for durability and failure atomicity are not really independent of one another and can be dealt with by a single mechanism – the recovery manager.

The tasks of a recovery manager are:

- 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).

In some cases, we require the recovery manager to be resilient to media failures. Corruption during a crash, random decay or a permanent failure can lead to failures of the recovery file, which can result in some of the data on the disk being lost. In such cases we need another copy of the recovery file. Stable storage, which is implemented so as to be very unlikely to fail by using mirrored disks or copies at a different location may be used for this purpose.

Intentions list

Any server that provides transactions needs to keep track of the objects accessed by clients' transactions. When a client opens a transaction, the server first contacted provides a new transaction identifier and returns it to the client. Each subsequent client request within a transaction up to and including the commit or abort request includes the transaction identifier as an argument. During the progress of a transaction, the update operations are applied to a private set of tentative versions of the objects belonging to the transaction. At each server, an intentions list is recorded for all of its currently active transactions – an intentions list of a particular transaction contains a list of the references and the values of all the objects that are altered by that transaction. When a transaction is committed, that transaction's intentions list is used to identify the objects it affected.

The committed version of each object is replaced by the tentative version made by that transaction, and the new value is written to the server's recovery file. When a transaction aborts, the server uses the intentions list to delete all the tentative versions of objects made by that transaction. A distributed transaction must carry out an atomic commit protocol before it can be committed or aborted. Our discussion of recovery is based on the twophase commit protocol, in which all the participants involved in a transaction first say whether they are prepared to commit and later, if all the participants agree, carry out the actual commit actions. If the participants cannot agree to commit, they must abort the transaction.

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. At the point when a participant says it is prepared to commit a transaction, its recovery manager must have saved both its intentions list for that transaction and the objects in that intentions list in its recovery file, so that it will be able to carry out the commitment later, even if it crashes in the interim. When all the participants involved in a transaction agree to commit it, the coordinator informs the client and then sends messages to the participants to commit their part of the transaction. Once the client has been informed that a transaction has committed, the recovery files of the participating servers must contain sufficient information to ensure that the transaction is committed by all of the servers, even if some of them crash between preparing to commit and committing.

Entries in recovery file • To deal with recovery of a server that can be involved in distributed transactions, further information in addition to the values of the objects is stored in the recovery file. This information concerns the status of each transaction – whether it is committed, aborted or prepared to commit. In addition, each object in the recovery file is associated with a particular transaction by saving the intentions list in the recovery file.

Recovery of objects • 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 recovery manager is responsible for restoring the server's objects so that they include all the effects of the committed transactions performed in the correct order and none of the effects of

incomplete or aborted transactions. The most recent information about transactions is at the end of the log. There are two approaches to restoring the data from the recovery file. In the first, the recovery manager starts at the beginning and restores the values of all of the objects from the most recent checkpoint (discussed in the next section). It then reads in the values of each of the objects, associates them with their transaction's intentions lists and for committed transactions replaces the values of the objects. In this approach, the transactions are replayed in the order in which they were executed and there could be a large number of them. In the second approach, the recovery manager will restore a server's objects by 'reading the recovery file backwards'. The recovery file has been structured so that there is a backwards pointer from each transaction status entry to the next. The recovery manager uses transactions with committed status to restore those objects that have not yet been restored. It continues until it has restored all of the server's objects. This has the advantage that each object is restored once only.

To recover the effects of a transaction, a recovery manager gets the corresponding intentions list from its recovery file. The intentions list contains the identifiers and positions in the recovery file of values of all the objects affected by the transaction. If the server fails at the point reached, its recovery manager will recover the objects as follows. It starts at the last transaction status entry in the log (at P7) and concludes that transaction U has not committed and its effects should be ignored. It then moves to the previous transaction status entry in the log (at P4) and concludes that transaction T has committed. To recover the objects affected by transaction T, it moves to the previous transaction status entry in the log (at P3) and finds the intentions list for T (< A, P1 >, < B, P2 >). It then restores objects A and B from the values at P1 and P2. As it has not yet restored C, it moves back to P0, which is a checkpoint, and restores C.

To help with subsequent reorganization of the recovery file, the recovery manager notes all the prepared transactions it finds during the process of restoring the server's objects. For each prepared transaction, it adds an aborted transaction status to the recovery file. This

ensures that in the recovery file, every transaction is eventually shown as either committed or aborted. The server could fail again during the recovery procedures. It is essential that recovery be idempotent, in the sense that it can be done any number of times with the same effect. This is straightforward under our assumption that all the objects are restored to volatile memory. In the case of a database, which keeps its objects in permanent storage with a cache in volatile memory, some of the objects in permanent storage will be out of date when a server is replaced after a crash. Therefore the recovery manager has to restore the objects in permanent storage. If it fails during recovery, the partially restored objects will still be there. This makes idempotence a little harder to achieve.

Reorganizing the recovery file

A recovery manager is responsible for reorganizing its recovery file so as to make the process of recovery faster and to reduce its use of space. If the recovery file is never reorganized, then the recovery process must search backwards through the recovery file until it has found a value for each of its objects. Conceptually, the only information required for recovery is a copy of the committed version of each object in the server. This would be the most compact form for the recovery file. The name check pointing is used to refer to the process of writing the current committed values of a server's objects to a new recovery file, together with transaction status entries and intentions lists of transactions that have not yet been fully resolved (including information related to the two-phase commit protocol). The term checkpoint is used to refer to the information stored by the check pointing process. The purpose of making checkpoints is to reduce the number of transactions to be dealt with during recovery and to reclaim file space. Check pointing can be done immediately after recovery but before any new transactions are started. However, recovery may not occur very often. Therefore, check pointing may need to be done from time to time during the normal activity of a server. The checkpoint is written to a future recovery file, and the current recovery file remains in use until the checkpoint is complete. Check pointing consists of 'adding a mark' to the recovery file when the check pointing starts, writing the server's objects to the future recovery file and then copying to that file (1) all entries before

the mark that relate to as-yet-unresolved transactions and (2) all entries after the mark in the recovery file. When the checkpoint is complete, the future recovery file becomes the recovery file. The recovery system can reduce its use of space by discarding the old recovery file. When the recovery manager is carrying out the recovery process, it may encounter a checkpoint in the recovery file. When this happens, it can immediately restore all outstanding objects from the checkpoint.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Write short note on Transaction recovery.	Dec 2011	7

UNIT -04/LECTURE -07

Replication ([RGPV/Dec 2013(7)], ([RGPV/June 2013(7)], ([RGPV/June 2014(7)]]

Replication is a key to the effectiveness of distributed systems in that it can provide enhanced performance, high availability and fault tolerance. Replication is used widely. For example, the caching of resources from web servers in browsers and web proxy servers is a form of replication, since the data held in caches and at servers are replicas of one another.

Replication is a technique for enhancing services. The motivations for replication include:

Performance enhancement:

The caching of data at clients and servers is by now familiar as a means of performance enhancement.

Increased availability:

Users require services to be highly available. That is, the proportion of time for which a service is accessible with reasonable response times should be close to 100%. Apart from delays due to pessimistic concurrency control conflicts (due to data locking), the factors that are relevant to high availability are:

- server failures;
- network partitions and disconnected operation (communication disconnections that are often unplanned and are a side effect of user mobility).

To take the first of these, replication is a technique for automatically maintaining the availability of data despite server failures. If data are replicated at two or more failure-independent servers, then client software may be able to access data at an alternative server should the default server fail or become unreachable.

Fault tolerance:

Highly available data is not necessarily strictly correct data. It may be out of date, for example; or two users on opposite sides of a network partition may make updates that conflict and need to be resolved. A fault-tolerant service, by contrast, always guarantees strictly correct behaviour despite a certain number and type of faults.

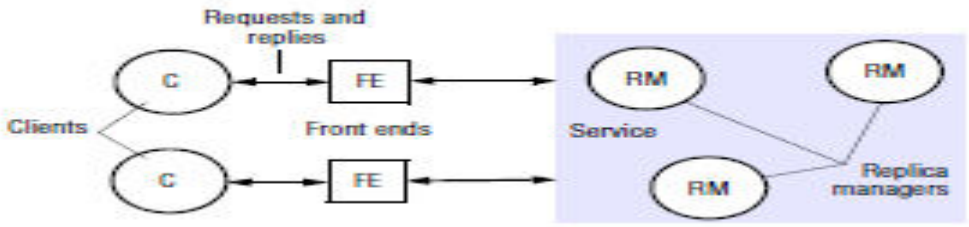
System model and the role of group communication

The data in our system consist of a collection of items that we shall call objects. An 'object' could be a file, say, or a Java object. But each such logical object is Implemented by a collection of physical copies called replicas. The replicas are physical objects, each stored at a single computer, with data and behaviour that are tied to some degree of consistency by the system's operation. The 'replicas' of a given object are not necessarily identical, at least not at any particular point in time. Some replicas may have received updates that others have not received.

System model

We assume an asynchronous system in which processes may fail only by crashing. Our default assumption is that network partitions may not occur, but we shall sometimes consider what happens if they do occur. Network partitions make it harder to build failure detectors, which we use to achieve reliable and totally ordered multicast.

A basic architectural model for the management of replicated data



For the sake of generality, we describe architectural components by their roles and do not mean to imply that they are necessarily implemented by distinct processes (or hardware).

The model involves replicas held by distinct replica managers which are components that contain the replicas on a given computer and perform operations upon them directly. This general model may be applied in a client-server environment, in which case a replica manager is a server. We shall sometimes simply call them servers instead. Equally, it may be applied to an application and application processes can in that case act as both clients and replica managers. For example, the user's laptop on a train may contain an application that acts as a replica manager for their diary.

We shall always require that a replica manager applies operations to its replicas recoverably. This allows us to assume that an operation at a replica manager does not leave inconsistent results if it fails part way through. We sometimes require each replica manager to be a state machine .Such a replica manager applies operations to its replicas atomically (indivisibly), so that its execution is equivalent to performing operations in some strict sequence. Moreover, the state of its replicas is a deterministic function of their initial states and the sequence of operations that it applies to them. Other stimuli, such as the reading on a clock or an attached sensor, have no bearing on these state values. Without this assumption, consistency guarantees between replica managers that accept update operations independently could not be made. The system can only determine which operations to apply at all replica managers and in what order – it cannot reproduce non-deterministic effects. The assumption implies that it may not be possible, depending upon the threading architecture, for the servers to be multi-threaded.

Often each replica manager maintains a replica of every object, and we assume this is so unless we state otherwise. However, the replicas of different objects may be maintained by different sets of replica managers. For example, one object may be needed mostly by clients on one network and another by clients on another network. There is little to be gained by replicating them at managers on the other network. The set of replica managers may be static or dynamic. In a dynamic system, new replica managers may appear (for example, if a second secretary copies a diary onto their laptop); this is not allowed in a static system. In a

dynamic system, replica managers.

In a static system, replica managers do not crash (crashing implies never executing another step), but they may cease operating for an indefinite period. We return to the issue of failure. A collection of replica managers provides a service to clients. The clients see a service that gives them access to objects (for example, diaries or bank accounts), which in fact are replicated at the managers. Each client requests a series of operations – invocations upon one or more of the objects. An operation may involve a combination of reads of objects and updates to objects. Requested operations that involve no updates are called read only
requests; requested operations that update an object are called update requests (these may also involve reads).

Each client's requests are first handled by a component called a front end. The role of the front end is to communicate by message passing with one or more of the replica managers, rather than forcing the client to do this itself explicitly. It is the vehicle for making replication transparent. A front end may be implemented in the client's address space, or it may be a separate process. In general, five phases are involved in the performance of a single request upon the replicated objects. The actions in each phase vary according to the type of system, as will become clear in the next two sections. For example, a service that supports disconnected operation behaves differently from one that provides a fault-tolerant service. The phases are as follows:

Request: The front end issues the request to one or more replica managers:
– either the front end communicates with a single replica manager, which in turn communicates with other replica managers;
– or the front end multicasts the request to the replica managers.

Coordination: The replica managers coordinate in preparation for executing the request

consistently. They agree, if necessary at this stage, on whether the request is to be applied (it might not be applied at all if failures occur at this stage). They also decide on the ordering of this request relative to others. All of the types of ordering defined for multicast also apply to request handling and we define those orders again for this context:

FIFO ordering: If a front end issues request r and then request r, any correct replica manager that handles r handles r before it.

Causal ordering:

If the issue of request r happened-before the issue of request r, then any correct replica manager that handles r handles r before it.

Total ordering:

If a correct replica manager handles r before request r, then any correct replica manager that handles r handles r before it. Most applications require FIFO ordering. We discuss the requirements for causal and total ordering – and the hybrid orderings that are both FIFO and total, or both causal and total – in the next two sections.

Execution:

The replica managers execute the request – perhaps tentatively: that is, in such a way that they can undo its effects later. Agreement: The replica managers reach consensus on the effect of the request – if any – that will be committed. For example, in a transactional system the replica managers may collectively agree to abort or commit the transaction at this stage.

Response:

One or more replica managers responds to the front end. In some systems, one replica manager sends the response. In others, the front end receives responses from a collection of replica managers and selects or synthesizes a single response to pass back to the client.

For example, it could pass back the first response to arrive, if high availability is the goal. If tolerance of Byzantine failures is the goal, then it could give the client the response that a majority of the replica managers provides.

Different systems may make different choices about the ordering of the phases, as well as their contents. For example, in a system that supports disconnected operation, it is important to give the client (the application on the user's laptop, say) as early a response as possible. The user does not want to wait until the replica manager on the laptop and the replica manager back in the office can coordinate. By contrast, in a fault-tolerant system the client is not given the response until the end, when the correctness of the result can be guaranteed.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	What is replication in distributed system?	Dec 2013	7
Q.2	What do you mean by replication? Explain with proper example.	June 2014,June 2013	7

UNIT -04/LECTURE- 08

The role of group communication ([RGPV/Dec 2012(7)]

In replication, and indeed in many other practical circumstances, there is a strong requirement for dynamic membership, in which processes join and leave the group as the system executes. In a service that manages replicated data, for example, users may add or withdraw a replica manager, or a replica manager may crash and thus need to be withdrawn from the system's operation. Group membership management, which was introduced , is therefore particularly important in this context. Systems that can adapt as processes join, leave and crash – fault-tolerant systems, in particular – require the more advanced features of failure detection and notification of membership changes. A full group membership service maintains group views, which are lists of the current group members, identified by their unique process identifiers. The list is ordered, for example, according to the sequence in which the members joined the group. A new group view is generated each time that a process is added or excluded.

It is important to understand that a group membership service may exclude a process from a group because it is Suspected, even though it may not have crashed. A communication failure may have made the process unreachable, while it continues to execute normally. A membership service is always free to exclude such a process. The effect of exclusion is that no messages will be delivered to that process henceforth. Moreover, in the case of a closed group, if that process becomes connected again, any messages it attempts to send will not be delivered to the group members. That process will have to rejoin the group (as a 'reincarnation' of itself, with a new identifier), or abort its operations.

A false suspicion of a process and the consequent exclusion of the process from the group may reduce the group's effectiveness. The group has to manage without the extra reliability or performance that the withdrawn process could potentially have provided. The design challenge, apart from designing failure detectors to be as accurate as possible, is to ensure

that a system based on group communication does not behave incorrectly if a process is falsely suspected. An important consideration is how a group management service treats network partitions. Disconnection or the failure of components such as a router in a network may split a group of processes into two or more subgroups, with communication between the subgroups impossible. Group management services differ in whether they are primary partition or partitionable. In the first case, the management service allows at most one subgroup (a majority) to survive a partition; the remaining processes are informed that they should suspend operations. This arrangement is appropriate for cases where the processes manage important data and the costs of inconsistencies between two or more subgroups outweigh any advantage of disconnected working.

On the other hand, in some circumstances it is acceptable for two or more subgroups to continue to operate – a partitionable group membership service allows this. For example, in an application in which users hold an audio or video conference to discuss some issues, it may be acceptable for two or more subgroups of users to continue their discussions independently despite a partition. They can merge their results when the partition heals and the subgroups are connected again.

Fault-tolerant services

We examine how to provide a service that is correct despite up to f process failures, by replicating data and functionality at replica managers. For the sake of simplicity, we assume that communication remains reliable and that no partitions occur. Each replica manager is assumed to behave according to a specification of the semantics of the objects it manages, when they have not crashed. For example, a specification of bank accounts would include an assurance that funds transferred between bank accounts can never disappear, and that only deposits and withdrawals affect the balance of any particular account. Intuitively, a service based on replication is correct if it keeps responding despite failures and if clients cannot tell the difference between the service they obtain from an implementation with replicated data and one provided by a single correct replica manager. Care is needed in

meeting these criteria. If precautions are not taken, then anomalies can arise when there are several replica managers – even bearing in mind that we are considering the effects of individual operations, not transactions.

Consider a naive replication system, in which a pair of replica managers at computers A and B each maintain replicas of two bank accounts, x and y. Clients read and update the accounts at their local replica manager but try another replica manager if the local one fails. Replica managers propagate updates to one another in the background after responding to the clients. Both accounts initially have a balance of $0.Client 1 updates the balance of x at its local replica manager B to be $1 and then attempts to update y's balance to be $2, but discovers that B has failed. Client 1 therefore applies the update at A instead. Now client 2 reads the balances at its local replica manager A. It finds first that y has $2 and then that x has $0 – the update to bank account x from B has not arrived, since B failed. The situation is shown below, where the operations are labelled by the computer at which they first took place and lower operations happen later:

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

This execution does not match a common-sense specification for the behaviour of bank accounts: client 2 should have read a balance of $1 for x, given that it read the balance of $2 for y, since y's balance was updated after that of x. The anomalous behaviour in the replicated case could not have occurred if the bank accounts had been implemented by a single server. We can construct systems that manage replicated objects without the anomalous behaviour produced by the naive protocol in our example. First, we need to understand what counts as correct behaviour for a replicated system.

S.NO	RGPV QUESTION	YEAR	MARKS
Q.1	Explain role of group communication.	Dec 2012	7

UNIT -04/LECTURE -09

Transaction with replicated data.

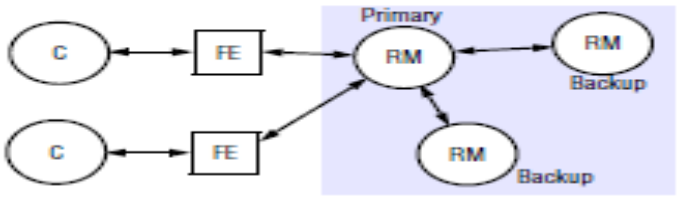
Passive (primary-backup) replication

In the passive or primary-backup model of replication for fault tolerance ,there is at any one time a single primary replica manager and one or more secondary replica managers – 'backups' or 'slaves'. In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service. The primary replica manager executes the operations and sends copies of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary.

The sequence of events when a client requests an operation to be performed is as follows:

1. Request: The front end issues the request, containing a unique identifier, to the primary replica manager.
2. Coordination: The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request, and if so it simply resends the response.
3. Execution: The primary executes the request and stores the response.
4. Agreement: If the request is an update, then 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 front end, which hands the response back to the client.

The passive (primary-backup) model for fault tolerance



This system obviously implements linearizability if the primary is correct, since the primary sequences all the operations upon the shared objects. If the primary fails, then the system retains linearizability if a single backup becomes the new primary and if the new system configuration takes over exactly where the last left off. That is if:

• The primary is replaced by a unique backup (if two clients began using two backups, then the system could perform incorrectly).

• The replica managers that survive agree on which operations had been performed at the point when the replacement primary takes over.

Both of these requirements are met if the replica managers (primary and backups) are organized as a group and if the primary uses view-synchronous group communication to send the updates to the backups. The first of the above two requirements is then easily satisfied. When the primary crashes, the communication system eventually delivers a new view to the surviving backups, one that excludes the old primary. The backup that replaces the primary can be chosen by any function of that view. For example, the backups can choose the first member in that view as the replacement. That backup can register itself as the primary with a name service that the clients consult when they suspect that the primary has failed (or when they require the service in the first place).

The second requirement is also satisfied, by the ordering property of view synchrony and the use of stored identifiers to detect repeated requests. The viewsynchronous semantics guarantee that either all the backups or none of them will deliver any given update before delivering the new view. Thus the new primary and the surviving backups all agree on

whether any particular client's update has or has not been processed. Consider a front end that has not received a response. The front end retransmits the request to whichever backup takes over as the primary. The primary may have crashed at any point during the operation. If it crashed before the agreement stage (4), then the surviving replica managers cannot have processed the request. If it crashed during the agreement stage, then they may have processed the request. If it crashed after that stage, then they have definitely processed it. But the new primary does not have toknow what stage the old primary was in when it crashed. When it receives a request, it proceeds from stage 2 above. By view-synchrony, no consultation with the backups is necessary, because they have all processed the same set of messages.

Discussion of passive replication

The primary-backup model may be used even where the primary replica manager behaves in a non-deterministic way, for example due to multi-threaded operation. Since the primary communicates the updated state from the operations rather than a specification of the operations themselves, the backups slavishly record the state determined by the primary's actions alone. To survive up to f process crashes, a passive replication system requires f +1 replica managers (such a system cannot tolerate Byzantine failures). The front end requires little functionality to achieve fault tolerance. It just needs to be able to look up the new primary when the current primary does not respond. Passive replication has the disadvantage of providing relatively large overheads. View-synchronous communication requires several rounds of communication per multicast, and if the primary fails then yet more latency is incurred while the group communication system agrees upon and delivers the new view. In a variation of the model presented here, clients may be able to submit read requests to the backups, thus offloading work from the primary. The guarantee of linearizability is thereby lost, but the clients receive a sequentially consistent service.

Passive replication is used in the Harp replicated file system. The Sun Network Information Service (NIS) uses passive replication to achieve high availability and good performance,

although with weaker guarantees than sequential consistency. The weaker consistency guarantees are still satisfactory for many purposes, such as storing certain types of system administration records. The replicated data is updated at a master server and propagated from there to slave servers using one-to-one (rather than group) communication. Clients may communicate with either a master or a slave server to retrieve information. In NIS, however, clients may not request updates: updates are made to the master's files.

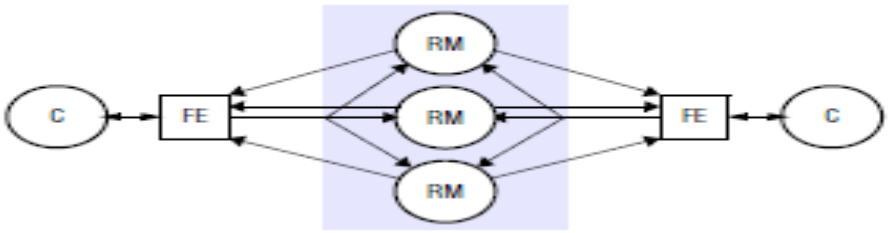
Active replication

In the active model of replication for fault tolerance , the replica managers are state machines that play equivalent roles and are organized as a group. Front ends multicast their requests to the group of replica managers and all the replica managers process the request independently but identically and reply. If any replica manager crashes, this need have no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way. We shall see that active replication can tolerate Byzantine failures, because the front end can collect and compare the replies it receives.

Under active replication, the sequence of events when a client requests an operation to be performed is as follows:

1. Request: The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst. It does not issue the next request until it has received a response.
2. Coordination: The group communication system delivers the request to every correct replica manager in the same (total) order.

Active replication



3. Execution: Every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.

4. Agreement: No agreement phase is needed, because of the multicast delivery semantics.

5. Response: Each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and the multicast algorithm. If, for example, the goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest (it can distinguish these from responses to other requests by examining the identifier in the response). This system achieves sequential consistency.

REFERENCES

BOOK	AUTHOR	PRIORITY
Distributed operating systems; Concepts and design	P K Sinha	1
Distributed systems: Principles and paradigms	Tanenbaum and Steen	2