ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Systems - Kernfach
Transactions in distributed settings

Prof. Dr. Gustavo Alonso
Institute for Pervasive Computing
Computer Science Department
ETH Zürich
alonso@inf.ethz.ch
http://www.inf.ethz.ch/~alonso

Basics of transaction processing

©Gustavo Alonso. ETH Zürich.

Transaction Processing

Why is transaction processing relevant?

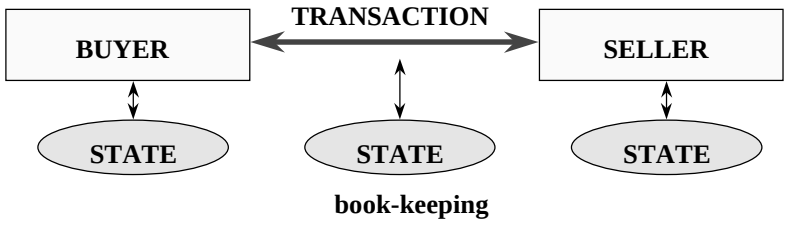
- Most of the information systems used in businesses are transaction based (either databases or TP-Monitors). The market for transaction processing is many tens billions of dollars per year
- Not long ago, transaction processing was used mostly in large companies (both users and providers). This is no longer the case (CORBA, WWW, Commodity TP-Monitors, Internet providers, distributed computing)
- Transaction processing is not just database technology, it is core distributed systems technology

Why distributed transaction processing?

- It is an accepted, proven, and tested programming model and computing paradigm for complex applications
- The convergence of many technologies (databases, networks, workflow management, ORB frameworks, clusters of workstations ...) is largely based on distributed transactional processing

From business to transactions

- A business transaction usually involves an exchange between two or more entities (selling, buying, renting, booking ...).
- When computers are considered, these business transactions become electronic transactions:



- The ideas behind a business transaction are intuitive. These same ideas are used in electronic transactions.
- Electronic transactions open up many possibilities that are unfeasible with traditional accounting systems.

The problems of electronic transactions

Transactions are a great idea:

- Hack a small, cute program and that's it.

Unfortunately, they are also a complex idea:

- From a programming point of view, one must be able to encapsulate the transaction (not everything is a transaction).
- One must be able to run high volumes of these transactions (buyers want fast response, sellers want to run many transactions cheaply).
- Transactions must be correct even if many of them are running concurrently (= at the same time over the same data).
- Transactions must be atomic. Partially executed transactions are almost always incorrect (even in business transactions).
- While the business is closed, one makes no money (in most business). Transactions are "mission critical".
- Legally, most business transactions require a written record. So do electronic transactions.

What is a transaction?

Transactions originated as "spheres of control" in which to encapsulate certain behavior of particular pieces of code.

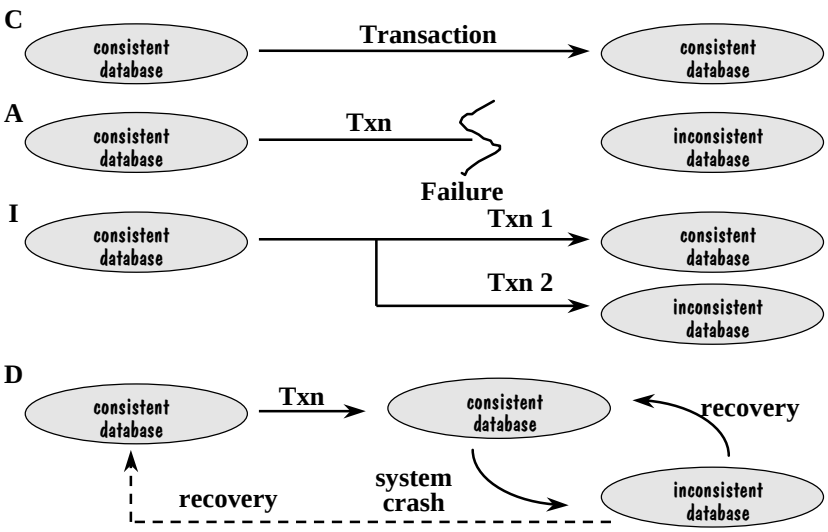
- A transaction is basically a set of service invocations, usually from a program (although it can also be interactive).
- A transaction is a way to help the programmer to indicate when the system should take over certain tasks (like semaphores in an operating system, but much more complicated).
- Transactions help to automate many tedious and complex operations:
  - Þ record keeping,
  - Þ concurrency control,
  - Þ recovery,
  - Þ durability,
  - Þ consistency.
- It is in this sense that transactions are considered ACID (Atomic, Consistent, Isolated, and Durable).

Transactional properties

These systems would have been very difficult to build without the concept of transaction. To understand why, one needs to understand the four key properties of a transaction:

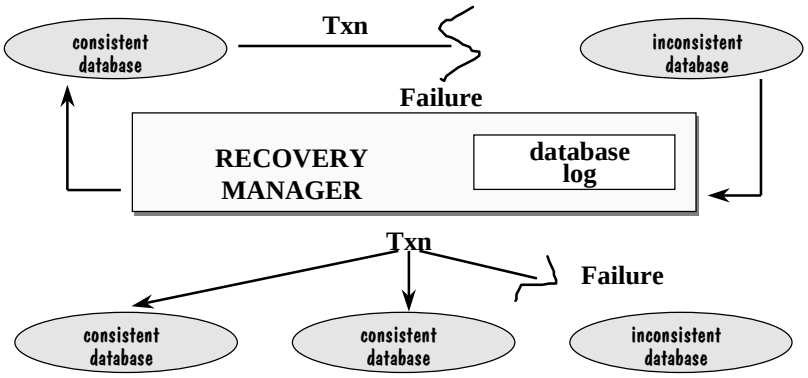
- **ATOMICITY:** necessary in any distributed system (but also in centralized ones). A transaction is atomic if it is executed in its entirety or not at all.
- **CONSISTENCY:** used in database environments. A transactions must preserve the data consistency.
- **ISOLATION:** important in multi-programming, multi-user environments. A transaction must execute as if it were the only one in the system.
- **DURABILITY:** important in all cases. The changes made by a transaction must be permanent (= they must not be lost in case of failures).

Transactional properties



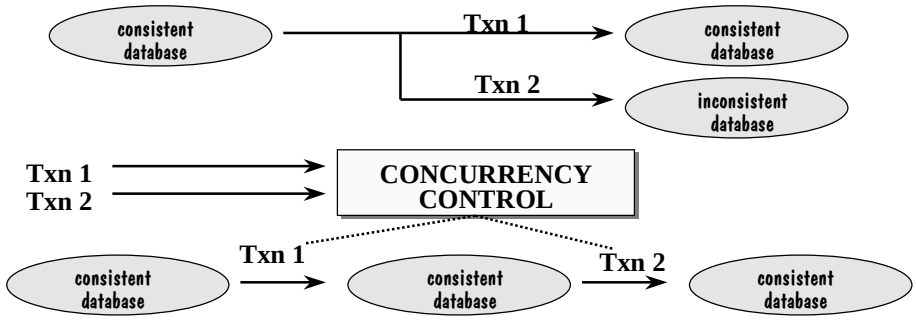
Transactional atomicity

- Transactional atomicity is an "all or nothing" property: either the entire transaction takes place or it does not take place at all.
- A transaction often involves several operations that are executed at different times (control flow dependencies). Thus, transactional atomicity requires a mechanism to eliminate partial, incomplete results (a recovery protocol).



Transactional isolation

- Isolation addresses the problem of ensuring correct results even when there are many transactions being executed concurrently over the same data.
- The goal is to make transactions believe there is no other transaction in the system (isolation).
- This is enforced by a concurrency control protocol, which aims at guaranteeing serializability.



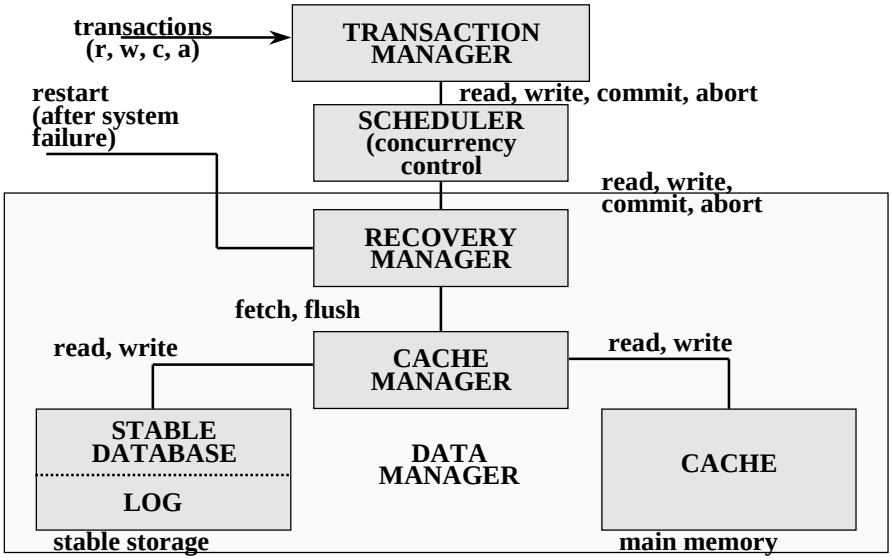
Transactional consistency

- Concurrency control and recovery protocols are based on a strong assumption: the transaction is always correct.
- In practice, transactions make mistakes (introduce negative salaries, empty social security numbers, different names for the same person ...). These mistakes violate database consistency.
- Transaction consistency is enforced through integrity constraints:
  - Null constrains: when an attribute can be left empty.
  - Foreign keys: indicating when an attribute is a key in another table.
  - Check constraints: to specify general rules ("employees must be either managers or technicians").
- Thus, integrity constraints acts as filters determining whether a transaction is acceptable or not.
- NOTE: integrity constraints are checked by the system, not by the transaction programmer.

Transactional durability

- Transactional system often deal with valuable information. There must be a guarantee that the changes introduced by a transaction will last.
- This means that the changes introduced by a transaction must survive failures (if you deposit money in your bank account, you don't want the bank to tell you they have lost all traces of the transaction because there was a disk crash).
- In practice, durability is guaranteed by using replication: database backups, mirrored disks.
- Often durability is combined with other desirable properties such as availability:
  - Availability is the percentage of time the system can be used for its intended purpose (common requirement: 99.86% or 1 hour a month of down time).
  - Availability plays an important role in many systems. Consider, for instance, the name server used in a CORBA implementation.

A Simple Transaction Manager (I)



A Simple Transaction Manager (II)

- Each one of the modules shown is a complex component that can be optimized by using clever tricks (engineering, not theory).
- In practice, these modules tend to be heavily interconnected (if one wants performance, forget about modularity and nice, clear cut interfaces).
- A crucial aspect of a transaction manager is to ensure that operations are executed in the proper order.
- When a module indicates "A should be executed before B", this should be the case at all levels, independently of the optimizations performed at each level. There are two ways to guarantee such property:
  - FIFO queues between each module: force sequential processing but have problems with threads and multi-processing.
  - Handshaking: If A must happen before B, B is not passed to a lower module until the execution of A has not been confirmed by the lower module.

Example Application (ATM)

Example 1: Automated Teller Machines (ATM)

- Tables:
  - AccountBalance (Acct#, balance): the accounts and the money in them
  - HotCard-List (Acct#): card that have been canceled/stolen/suspended.
  - AccountVelocity (Acct#,SumWithdrawals): stores the latest transactions and the accumulated amount.
  - PostingLog (Acct#,ATMid,Amount): a record of each operation.
- Typical operation (money withdrawal):
  - Get input (Acct#, ATM#, type, PIN, Txn-id, Amount).
  - Write request to PostingLog.
  - Check PIN
  - Check Acct# with HotCard-List table.
  - Check Acct# with AccountVelocity table
  - Update AccountVelocity table.
  - Update balance in AccountBalance table.
  - Write withdrawal record to PostingLog
  - Commit transaction and dispense money.

Example Application (ATM)

- Size: with several hundred ATMs and about one million customers, the database takes 25-50 MB.
- Load: the system is configured to deal with the peak load: about one transaction per minute per ATM. Under normal circumstances, one mirrored disk (two disks doing the same operations) can handle 5 transactions per second (tps). Assuming 5 I/O operations per transaction, one mirrored disk can then handle about 300 ATMs.
- The PostingLog can be updated off-line (at night).
- Before, these systems were based on snapshot replication. Today, many of these systems access on-line databases.

Example Application (Stock Exchange)

Example 2: Stock Exchange.

- Tables:
  - Users: list of traders and market watchers.
  - Stocks: list of traded stocks.
  - BuyOrders/SellOrders: all the orders entered during the day
  - Trades: all trades executed during the day.
  - Price: buy and sell total volume, and numbers of orders for each stock and price.
  - Log: all users' requests and system replies.
  - NotificationMssgs: all messages sent to the users (usually, confirmations of an operation).
- Typical operation (Execute Trade):
  - Read information about the stock from the Stocks table.
  - Get timestamp.
  - Read scheduled trading periods for the stock.
  - Check validity of operation (time, value, prices).
  - If valid, find a matching trade operation, update Trades, NotificationMssgs, Orders, Prices, Stocks.

Example Application (Stock Exchange)

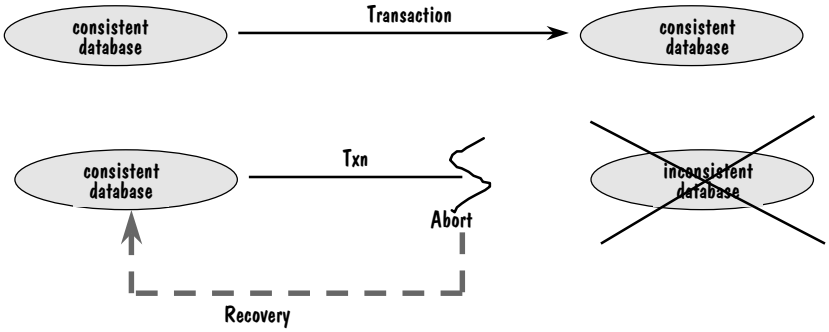
  - Write the system's response to the log.
  - Commit the transaction.
  - Broadcast the new book situation.
- Size: 10 stock exchanges connected, real-time distributed trading, total database size 2.6 GB.
- Load: Peak daily load is 140.000 orders. The peak-per-second load involves 180 disk I/Os and executing 300 million instructions per second.

Some basic advanced transaction models

Distributed Transactions

- The transaction model described so far is known as "flat model", i.e., transactions have only two levels, the parent transaction and the children transaction
- Distributed transactions are difficult to model with flat transactions (for instance, a chain of TRPCs), hence more complex models are needed
- The most common model for distributed transactions is the nested model in which operations of a transaction can be transactions themselves
- One of the most important aspects of distributed transactions is the problem of atomic commitment. Nested transactions help with this problem by indicating when transactions at different systems need to be committed

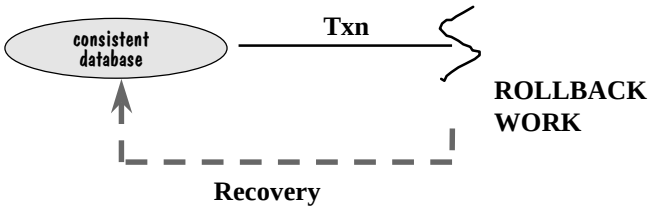
Pros and Cons of Atomicity



If a program finds there is some error, it suffices to abort the transaction. The recovery mechanism ensures the effects of the transaction will be eliminated. This is both good and bad.

Pros and Cons of Atomicity

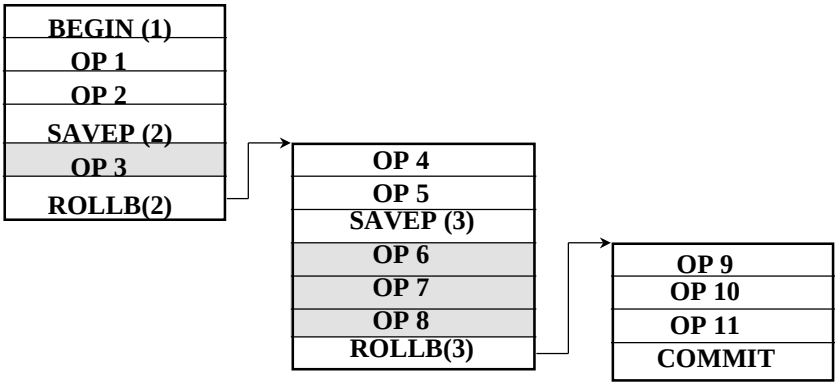
- If you are a transaction programmer, every time there is something that goes wrong (there is not enough funds, for instance), it is enough to execute ROLLBACK WORK to go back to the beginning of the transaction:



- But is transactions are long, or complex, aborting the entire transaction may be a waste of effort. In many cases, one does not want to go all the way to the beginning but to some intermediate point where one is sure things were correct, and then take again from there.

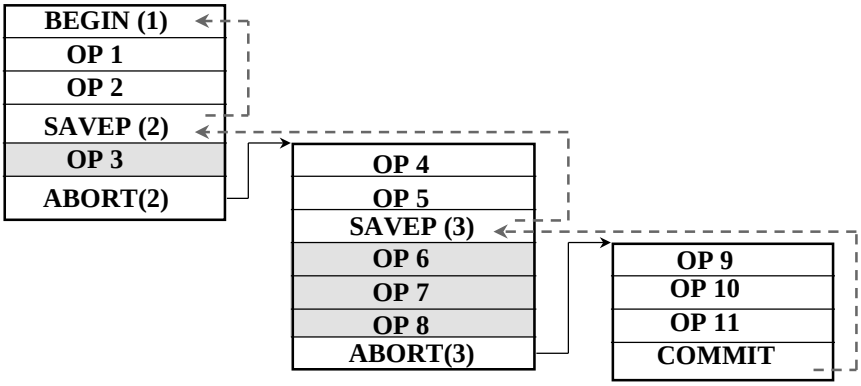
Savepoints

- To avoid this problem, savepoints are used.
- A savepoint records the current state of the execution of a transaction.
- When invoking ROLLBACK WORK, one indicates to which point one wants to rollback (to the beginning or to a savepoint).



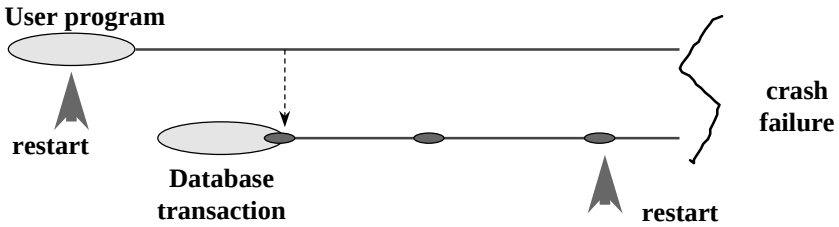
Triggered Commits

- How can savepoints be implemented ?
- Consider each set of operations between two savepoints as an atomic unit.
- A ROLLBACK(x) aborts all atomic units all the way back to savepoint x.
- A COMMIT at the end, triggers a chain of commits for each atomic unit.



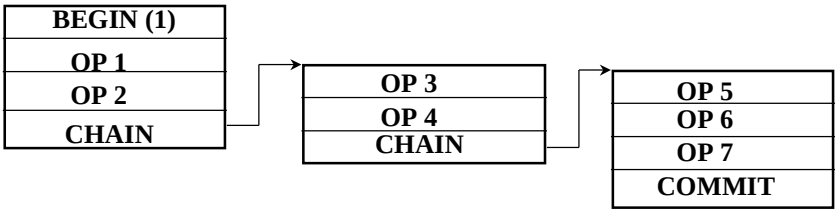
Persistent Savepoints ?

- Savepoints are a great idea:
  - Þ if something goes wrong, we can rollback to different parts of the execution and resume from there.
  - Þ rollback is performed by the system using the standard recovery mechanism.
- Can this great idea be generalized?
- If savepoints are made persistent, we may be able to resume the execution of a transaction even after crash failures!
- In principle yes, but in practice:
  - Þ The database can recover to a savepoint, but the application program may not be able to do the same.



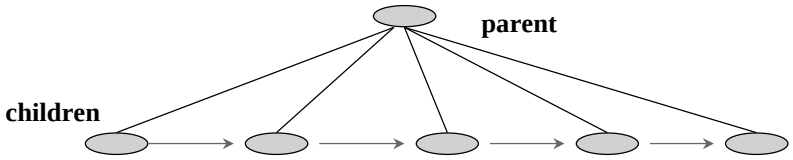
Chained Transactions

- Note that the atomic units used in savepoints are almost like a transaction.
- The important difference is that the transaction context is kept (locks are not released until commit, not needed ones can be released).
- Chained transactions allow to commit one transaction and pass its context to the next.
- If a failure occurs, committed transactions are safe, only the last active transaction will be aborted.
- However, there is no possibility of rollback to a previous transactions (now they are really committed).



Non-Flat Transaction Models

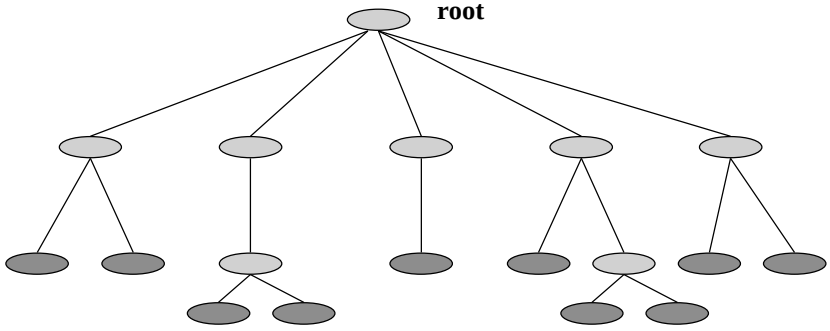
- Savepoints and chained transactions point the need to structure sequences of interactions with the database.
- The transactions we have been discussed so far are known as flat transactions.
- A chained transaction can be seen as a first step towards a non-flat transaction model:



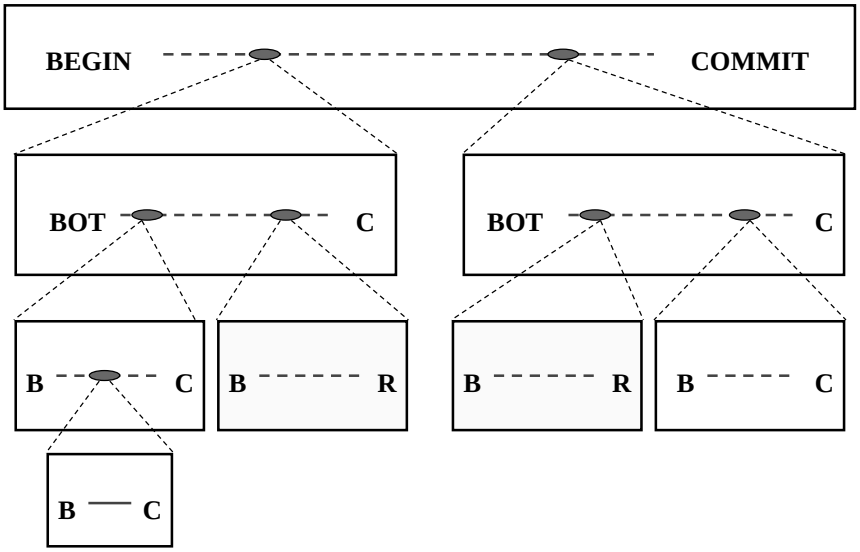
- When these ideas are generalized, one arrives at the nested transaction model.

Nested Transactions

- A nested transaction is a tree of transactions.
- Transactions at the leaves are flat transactions.
- Transactions can either commit or rollback. The commit is conditional to the parent transaction's commit (hence, transactions commit only if the root commits).
- Rollback of a transaction causes all of its children to also rollback.



Nested Transaction Structure

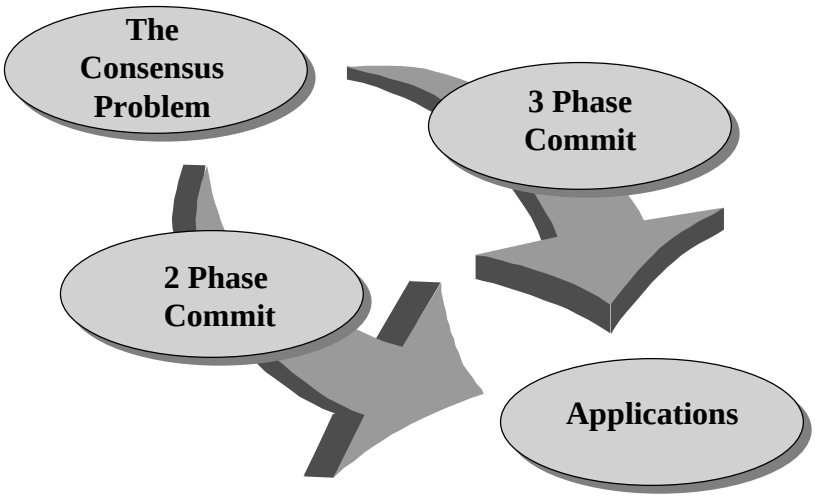


Nested Transaction Rules

- Nested Transactions are like a combination of savepoints and chained transactions.
- They follow these three rules (node = txn.):
  - Commit Rule: When a node wants to commit, it passes its context to the parent node (like in chained txns.), it will actually commit when the root node commits (like in savepoints).
  - Rollback Rule: If a node does a rollback, all of its children must also rollback.
  - Visibility Rule: When a node "commits" all of its changes become visible to the parent (because the child passes its context to the parent). Concurrent siblings are isolated from each other (they see each other as different transactions). The parent can make certain objects accessible to the children, thereby allowing the context of a child to pass to another child.
- All other notions (serializability, recoverability ...) still apply across different nested transactions
- For distributed transactions, the important aspect is how to commit all transactions, not so much the isolation aspects

Atomic commitment in practice (2PC-3PC)

Atomic Commitment



The Consensus (agreement) Problem

- Distributed consensus is the problem of reaching an agreement among all working processes on the value of a variable
- Consensus is not a difficult problem if the system is reliable (no site failures, no communication failures)
- Asynchronous = no timing assumptions can be made about the speed of processes or the network delay (it is not possible to distinguish between a failure and a slow system)

The impossibility result implies that there is always a chance to remain uncertain (unable to make a decision), hence:

- If failures may occur, then all entirely asynchronous commit protocols may block.
- No commit protocol can guarantee independent recovery (if a site fails when being uncertain, upon recovery it will have to find out from others what the decision was).
- This is a very strong result with important implications in any distributed system.

In an asynchronous environment where failures can occur reaching consensus may be impossible

Generals problem

- To succeed the generals must attack at the same time
- The generals can only communicate through messages
- The system is asynchronous: messages can be lost or delayed indefinitely

Under these circumstances, the generals will never be able to agree on a simultaneous attack, that is, they can never reach consensus

- The impossibility in the generals problem arises from the need to have complete knowledge: I need to know my state, the other's state, that the other knows my state, that the other knows that I know her state, that the other knows that I know that she knows my state ...
- If the system is entirely asynchronous, this problem cannot be solved by simply exchanging messages
- There are many forms of this problem and atomic commitment is one of them:
  - all sites must decide on whether to commit or abort a transaction and all must make the same decision

Atomic Commitment

Properties to enforce:

- AC1 = All processors that reach a decision reach the same one (agreement, consensus).
- AC2 = A processor cannot reverse its decision.
- AC3 = Commit can only be decided if all processors vote YES (no imposed decisions).
- AC4 = If there are no failures and all processors voted YES, the decision will be to commit (non triviality).
- AC5 = Consider an execution with normal failures. If all failures are repaired and no more failures occur for sufficiently long, then all processors will eventually reach a decision (liveness).

Simple 2PC Protocol and its correctness

PROTOCOL:

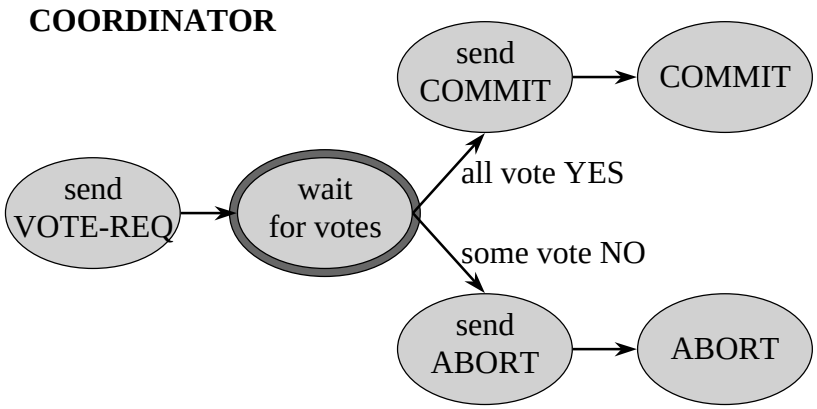
- Coordinator send VOTE-REQ to all participants.
- Upon receiving a VOTE-REQ, a participant sends a message with YES or NO (if the vote is NO, the participant aborts the transaction and stops).
- Coordinator collects all votes:
  - All YES = Commit and send COMMIT to all others.
  - Some NO = Abort and send ABORT to all which voted YES.
- A participant receiving COMMIT or ABORT messages from the coordinator decides accordingly and stops.

CORRECTNESS:

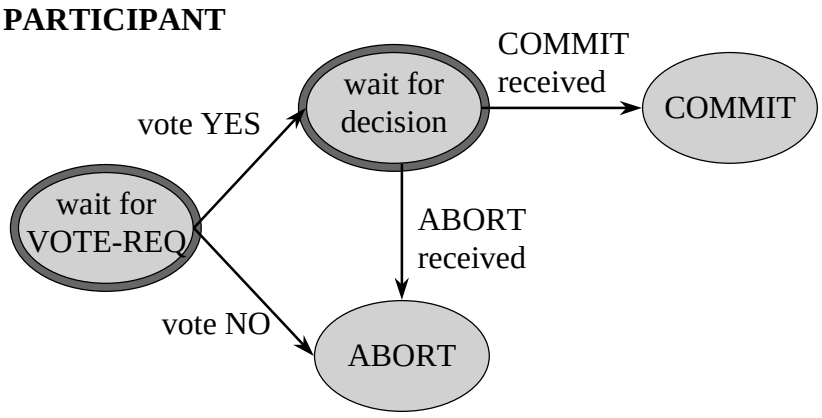
The protocol meets the 5 AC conditions (I - V):

- ACI = every processor decides what the coordinator decides (if one decides to abort, the coordinator will decide to abort).
- AC2 = any processor arriving at a decision "stops".
- AC3 = the coordinator will decide to commit if all decide to commit (all vote YES).
- AC4 = if there are no failures and everybody votes YES, the decision will be to commit.
- AC5 = the protocol needs to be extended in case of failures (in case of timeout, a site may need to "ask around").

Timeout Possibilities



Timeout Possibilities



Timeout and termination

In those three waiting periods:

- If the coordinator times-out waiting for votes: it can decide to abort (nobody has decided anything yet, or if they have, it has been to abort)
- If a participant times-out waiting for VOTE-REQ: it can decide to abort (nobody has decided anything yet, or if they have, it has been to abort)
- If a participant times-out waiting for a decision: it cannot decide anything unilaterally, it must ask (run a Cooperative Termination Protocol). If everybody is in the same situation no decision can be made: all processors will block. This state is called uncertainty period

When in doubt, ask. If anybody has decided, they will tell us what the decision was:

- There is always at least one processor that has decided or is able to decide (the coordinator has no uncertainty period). Thus, if all failures are repaired, all processors will eventually reach a decision
- If the coordinator fails after receiving all YES votes but before sending any COMMIT message: all participants are uncertain and will not be able to decide anything until the coordinator recovers. This is the blocking behavior of 2PC (compare with the impossibility result discussed previously)

Recovery and persistence

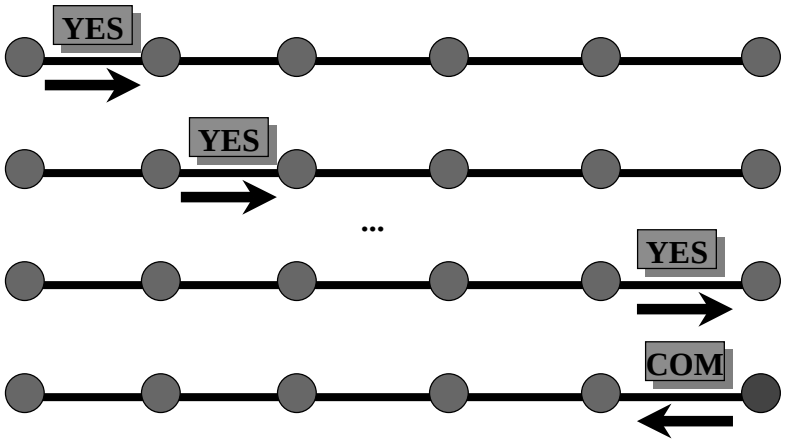
Processors must know their state to be able to tell others whether they have reached a decision. This state must be persistent:

- Persistence is achieved by writing a log record. This requires flushing the log buffer to disk (I/O).
- This is done for every state change in the protocol.
- This is done for every distributed transaction.
- This is expensive!

- When sending VOTE-REQ, the coordinator writes a START-2PC log record (to know the coordinator).
- If a participant votes YES, it writes a YES record in the log BEFORE it send its vote. If it votes NO, then it writes a NO record.
- If the coordinator decides to commit or abort, it writes a COMMIT or ABORT record before sending any message.
- After receiving the coordinator's decision, a participant writes its own decision in the log.

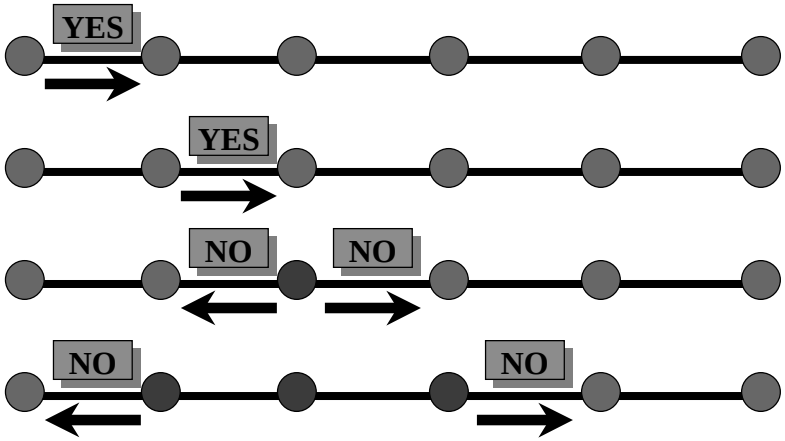
Linear 2PC

- Linear 2PC commit exploits a particular network configuration to minimize the number of messages:



Linear 2PC

- The total number of messages is 2n instead of 3n, but the number of rounds is 2n instead of 3



3 Phase Commit Protocol

2PC may block if the coordinator fails after having sent a VOTE-REQ to all processes and all processes vote YES. It is possible to reduce the window of vulnerability even further by using a slightly more complex protocol (3PC).

In practice 3PC is not used. It is too expensive (more than 2PC) and the probability of blocking is considered to be small enough to allow using 2PC instead.

But 3PC is a good way to understand better the subtleties of atomic commitment

We will consider two versions of 3PC:

- One capable of tolerating only site failures (no communication failures). Blocking occurs only when there is a total failure (every process is down). This version is useful if all participants reside in the same site.
- One capable of tolerating both site and communication failures (based on quorums). But blocking is still possible if no quorum can be formed.

Blocking in 2PC

Why does a process block in 2PC?

- If a process fails and everybody else is uncertain, there is no way to know whether this process has committed or aborted (NOTE: the coordinator has no uncertainty period. To block the coordinator must fail).
- Note, however, that the fact that everybody is uncertain implies everybody voted YES!
- Why, then, uncertain processes cannot reach a decision among themselves?

The reason why uncertain process cannot make a decision is that being uncertain does not mean all other processes are uncertain. Processes may have decided and then failed. To avoid this situation, 3PC enforces the following rule:

- NB rule: No operational process can decide to commit if there are operational processes that are uncertain.

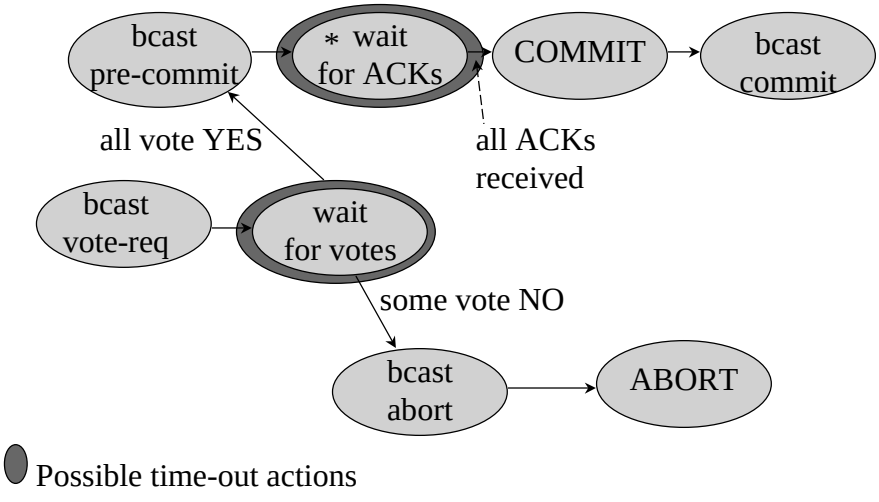
How does the NB rule prevent blocking?

Avoiding Blocking in 3PC

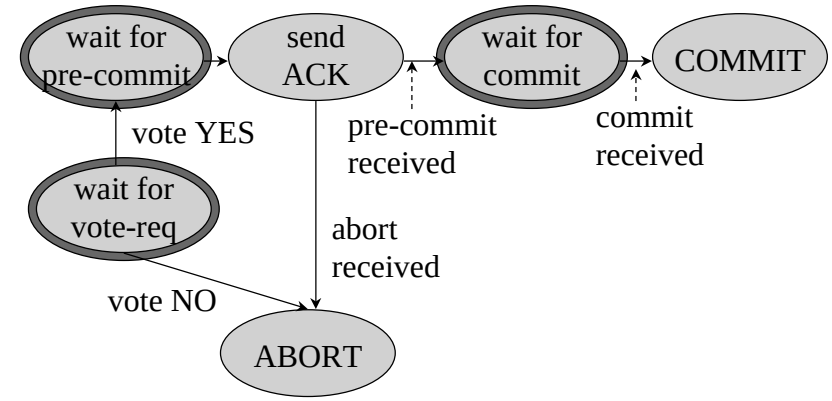
The NB rule guarantees that if anybody is uncertain, nobody can have decided to commit. Thus, when running the cooperative termination protocol, if a process finds out that everybody else is uncertain, they can all safely decide to abort.

- The consequence of the NB rule is that the coordinator cannot make a decision by itself as in 2PC. Before making a decision, it must be sure that everybody is out of the uncertainty area. Therefore, the coordinator, must first tell all processes what is going to happen: (request votes, prepare to commit, commit). This implies yet another round of messages!

3PC Coordinator



3PC Participant



Possible time-out actions

3PC and Knowledge (using the NB rule)

3PC is interesting in that the processes know what will happen before it happens:

- Once the coordinator reaches the "bcast pre-commit", it knows the decision will be to commit.
- Once a participant receives the pre-commit message from the coordinator, it knows that the decision will be to commit.

Why is the extra-round of messages useful?

- The extra round of messages is used to spread knowledge across the system. They provide information about what is going on at other processes (NB rule).

The NB rule is used when time-outs occur (remember, however, that there are no communication failures):

- If coordinator times out waiting for votes = ABORT.
- If participant times out waiting for vote-req = ABORT.
- If coordinator times out waiting for ACKs = ignore those who did not sent the ACK! (at this stage everybody has agreed to commit).
- If participant times out waiting for pre-commit = still in the uncertainty period, ask around.
- If participant times out waiting for commit message = not uncertain any more but needs to ask around!

Persistence and recovery in 3PC

Similarly to 2PC, a process has to remember its previous actions to be able to participate in any decision. This is accomplished by recording every step in the log:

- Coordinator writes "start-3PC" record before doing anything. It writes an "abort" or "commit" record before sending any abort or commit message.
- Participant writes its YES vote to the log before sending it to the coordinator. If it votes NO, it writes it to the log after sending it to the coordinator. When reaching a decision, it writes it in the log (abort or commit).

Processes in 3PC cannot independently recover unless they had already reached a decision or they have not participated at all:

- If the coordinator recovers and finds a "start 3PC" record in its log but no decision record, it needs to ask around to find out what the decision was. If it does not find a "start 3PC", it will find no records of the transaction, then it can decide to abort.
- If a participant has a YES vote in its log but no decision record, it must ask around. If it has not voted, it can decide to abort.

Termination Protocol

- Elect a new coordinator.
- New coordinator sends a "state req" to all processes. participants send their state (aborted, committed, uncertain, committable).
- TR1 = If some "aborted" received, then abort.
- TR2 = If some "committed" received, then commit.
- TR3 = If all uncertain, then abort.
- TR4 = If some "committable" but no "committed" received, then send "PRE-COMMIT" to all, wait for ACKs and send commit message.

TR4 is similar to 3PC, have we actually solved the problem?

- Yes, failures of the participants on the termination protocol can be ignored. At this stage, the coordinator knows that everybody is uncertain, those who have not sent an ACK have failed and cannot have made a decision. Therefore, all remaining can safely decide to commit after going over the pre-commit and commit phases.
- The problem is when the new coordinator fails after asking for the state but before sending any pre-commit message. In this case, we have to start all over again.

Partition and total failures

This protocol does not tolerate communication failures:

- A site decides to vote NO, but its message is lost.
- All vote YES and then a partition occurs. Assume the sides of the partition are A and B and all processes in A are uncertain and all processes in B are committable. When they run the termination protocol, those in A will decide to abort and those in B will decide to commit.
- This can be avoided is quorums are used, that is, no decision can be made without having a quorum of processes who agree (this reintroduces the possibility of blocking, all processes in A will block).

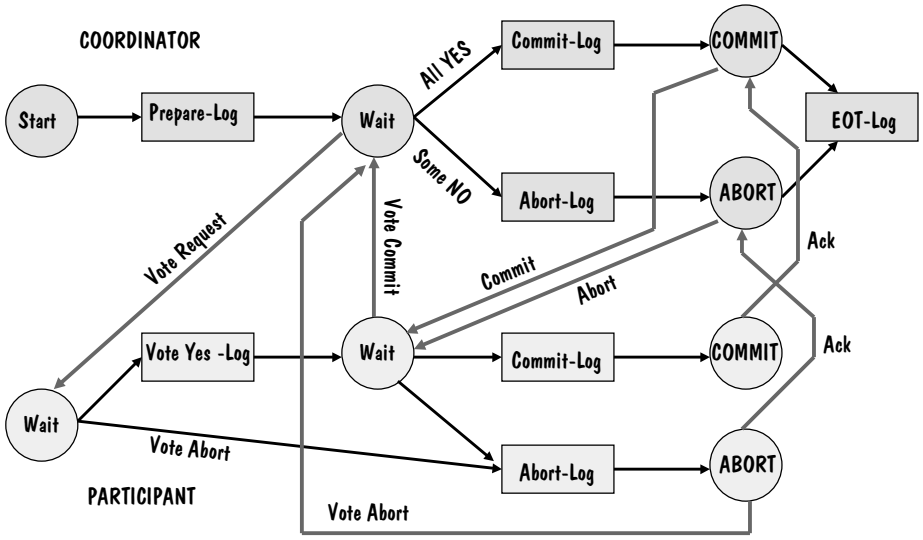
Total failures require special treatment, if after the total failure every process is still uncertain, it is necessary to find out which process was the last on to fail. If the last one to fail is found and is still uncertain, then all can decide to abort.

- Why? Because of partitions. Everybody votes YES, then all processes in A fail. Processes in B will decide to commit once the coordinator times out waiting for ACKs. Then all processes in B fail. Processes in A recover. They run the termination protocol and they are all uncertain. Following the termination protocol will lead them to abort.

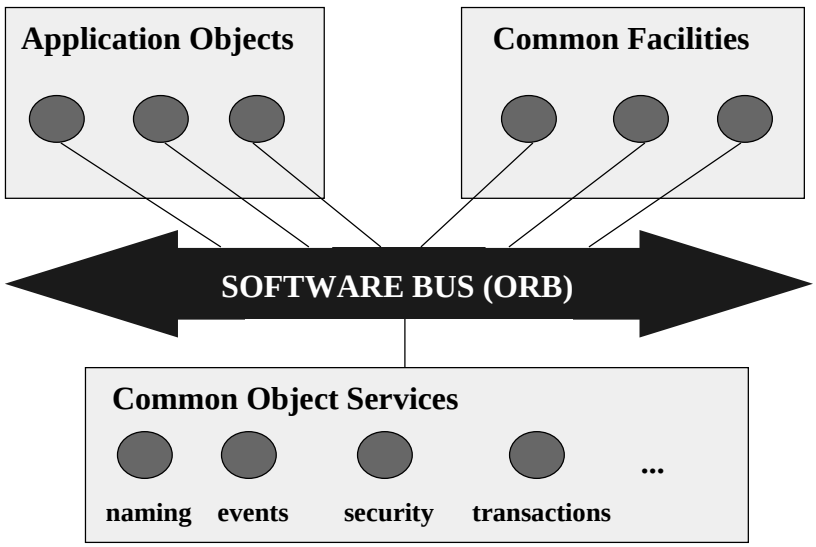
2PC in Practice

- 2PC is a protocol used in many applications from distributed systems to Internet environments
- 2PC is not only a database protocol, it is used in many systems that are not necessarily databases but, traditionally, it has been associated with transactional systems
- 2PC appears in a variety of forms: distributed transactions, transactional remote procedure calls, Object Transaction Services, Transaction Internet Protocol ...
- In any of these systems, it is important to remember the main characteristic of 2PC: if failures occur the protocol may block. In practice, in many systems, blocking does not happen but the outcome is not deterministic and requires manual intervention

2PC


COORDINATOR
Start
Prepare-Log
Wait
All YES
Some NO
Commit-Log
Abort-Log
COMMIT
ABORT
EOT-Log
Ack
Vote Request
Vote Commit
Commit
Abort
Vote Yes -Log
Wait
Commit-Log
COMMIT
Ack
Wait
Vote Abort
Abort-Log
ABORT
PARTICIPANT
Vote Abort

ORB

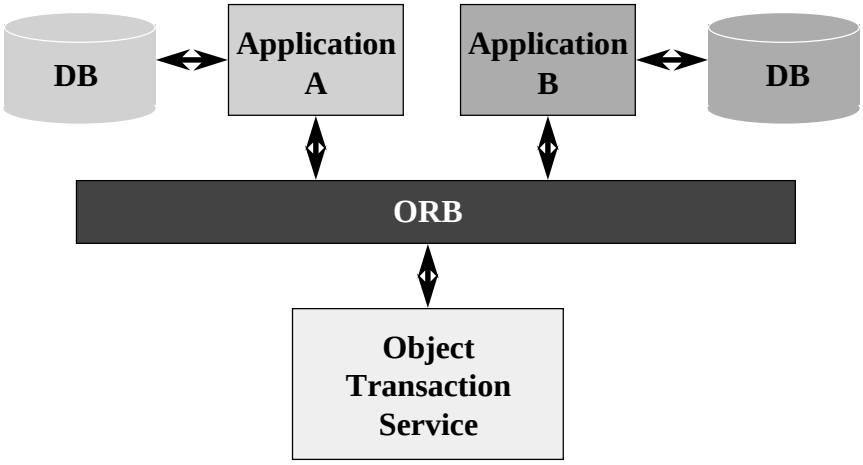

Application Objects
Common Facilities
SOFTWARE BUS (ORB)
Common Object Services
naming
events
security
transactions
...

Object Transaction Service

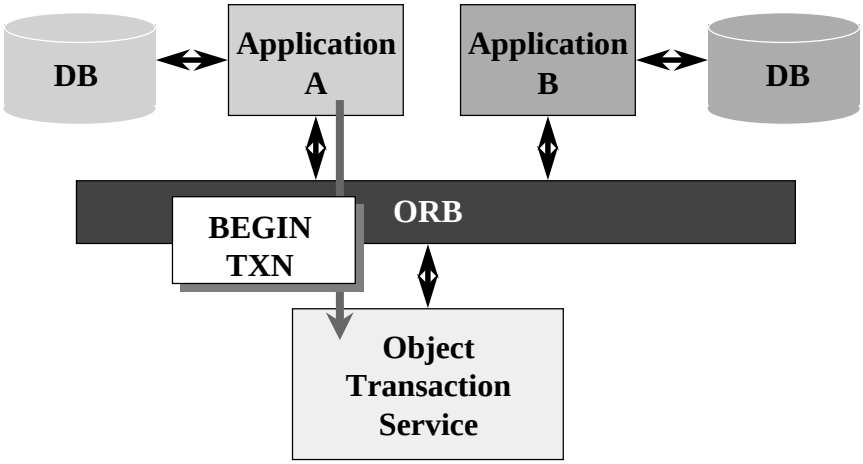
- The OTS provides transactional guarantees to the execution of invocations between different components of a distributed application built on top of the ORB
- The OTS is fairly similar to a TP-Monitor and provides much of the same functionality discussed before for RPC and TRPC, but in the context of the CORBA standard
- Regardless of whether it is a TP-monitor or an OTS, the functionality needed to support transactional interactions is the same:
  - transactional protocols (like 2PC)
  - knowing who is participating
  - knowing the interface supported by each participant

Object Transaction Service

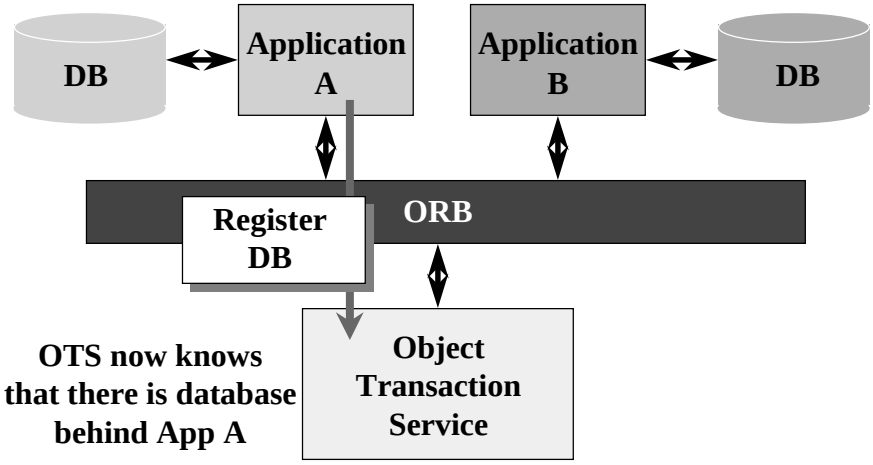
Assume App A wants to update its database and also that in B



Object Transaction Service

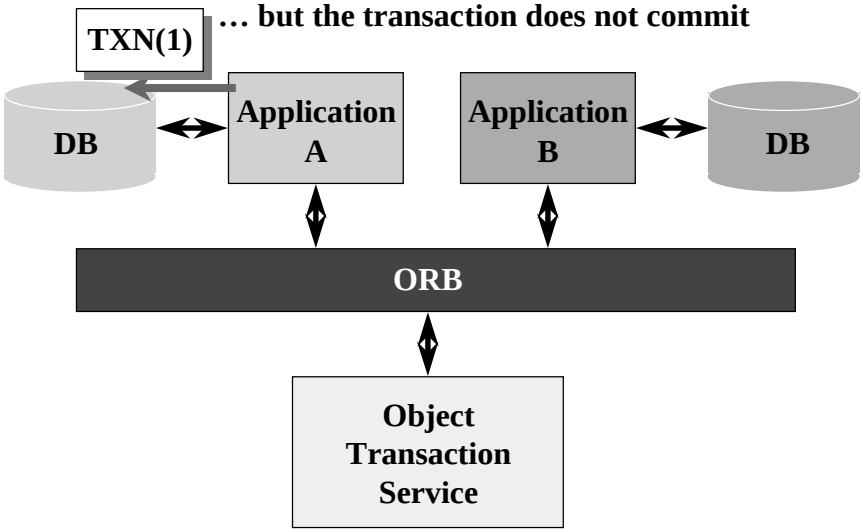


Object Transaction Service

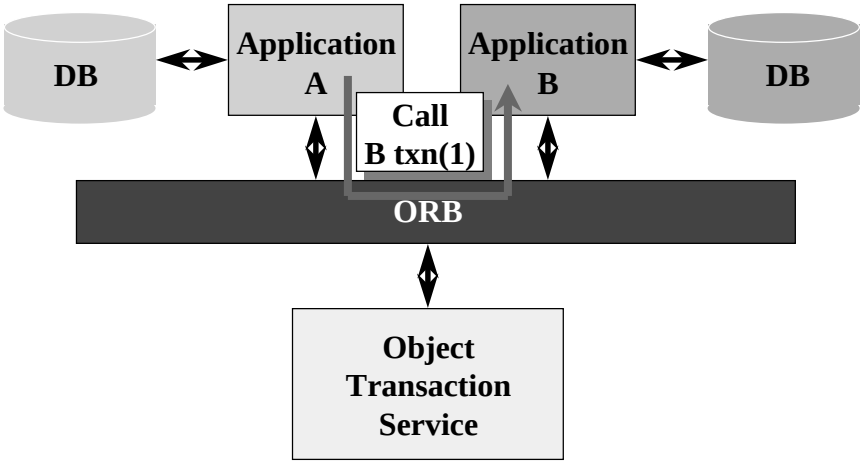


OTS now knows that there is database behind App A

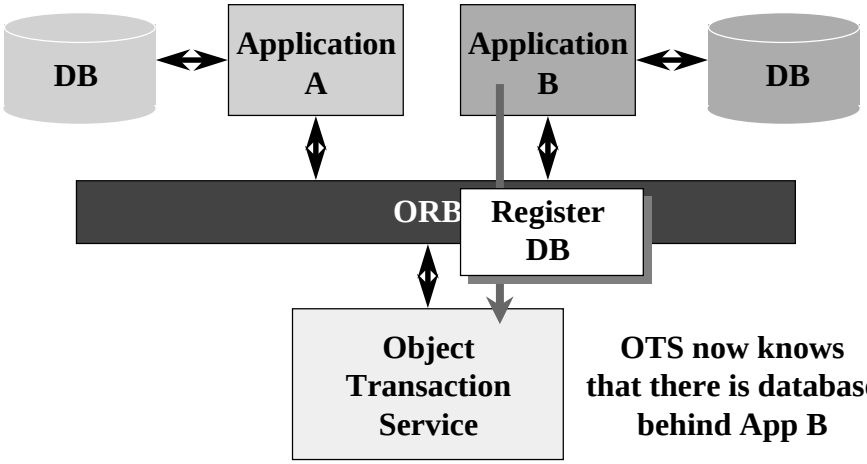
Object Transaction Service



Object Transaction Service

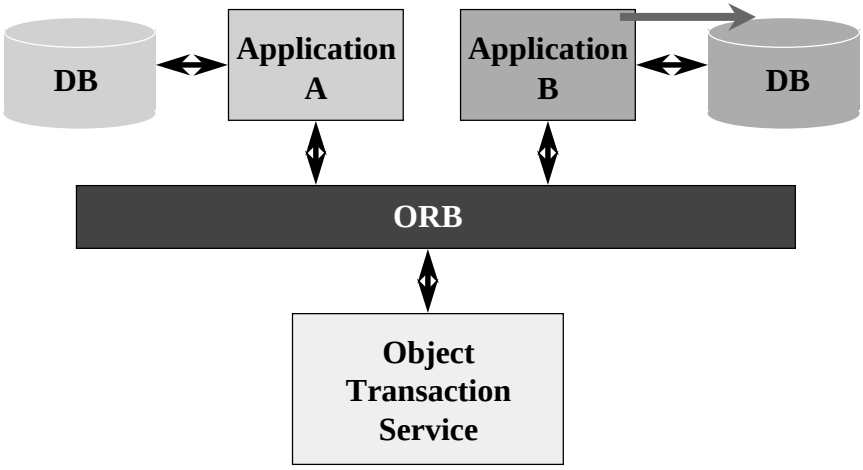


Object Transaction Service

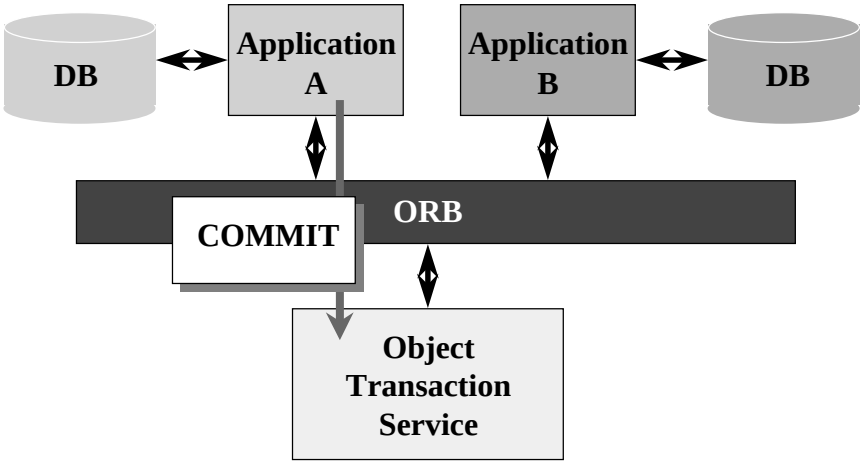


Object Transaction Service

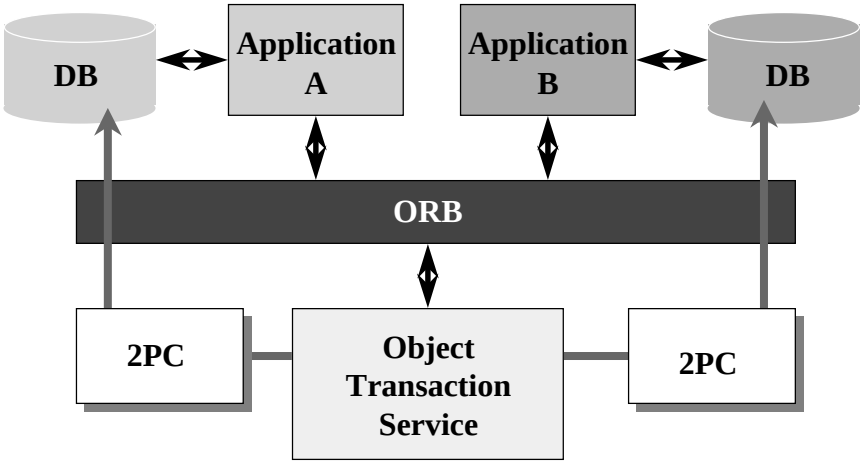
… but the transaction does not commit TXN(1)



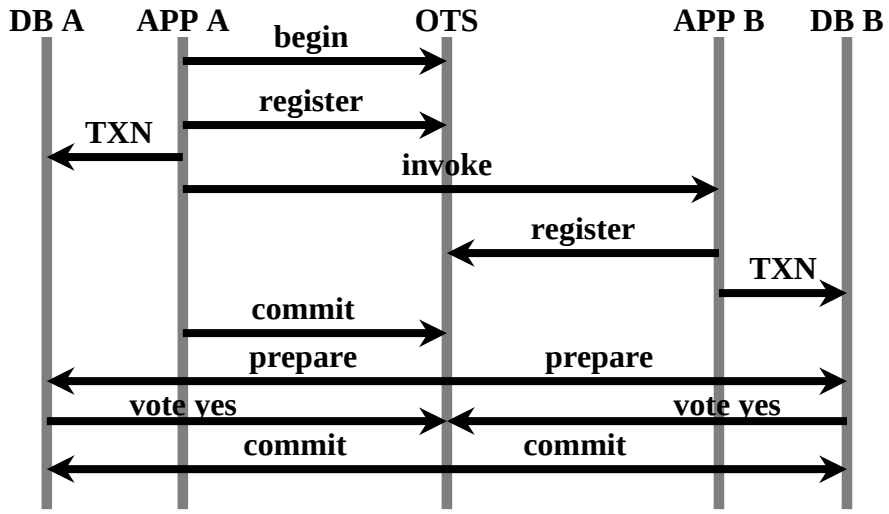
Object Transaction Service



Object Transaction Service



OTS Sequence of Messages



Transaction Propagation & Resource Registration

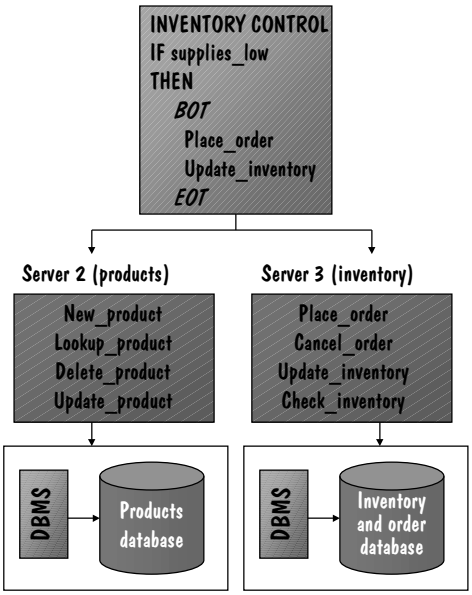
- When a call is made to another server, somebody has to know that this call belongs to a given transaction. There are two ways of doing this:
- Explicit (manual): the invocation itself contains the transaction identifier. Then, when the application registers the resource manager, it uses this transaction identifier to say to which transaction it is "subscribing"
- Implicit (automatic): the call is made through the OTS, which will forward the transaction identifier along with the invocation. This requires to link with the OTS library and to make all methods involved transactional
- Registration is necessary in order to tell the OTS who will participate in the 2PC protocol and what type of interface is supported. Registration can be manual or automatic
- Manual registration implies the the user provides an implementation of the resource. This implementation acts as an intermediary between the OTS and the actual resource manager (useful for legacy applications that need to be wrapped)
- Automatic registration is used when the resource manager understands transactions (i.e., it is a database), in which case it will support the XA interface for 2PC directly. A resource are registered only once, and implicit propagation is used to check which transactions go there

Transaction Processing Monitors (TP-monitors)

Outline

- Historical perspective:
  - The problem: synchronization and atomic interaction
  - The solution: transactional RPC and additional support
- TP Monitors
  - Example and Functionality
  - Architectures
  - Structure
  - Components
- TP Monitor functionality in CORBA

Client, server, and databases



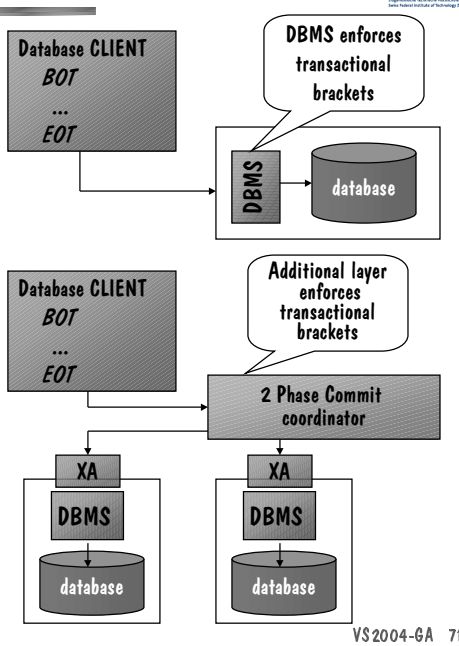
- Processing, storing, accessing and retrieving data has always been one of the key aspects of enterprise computing. Most of this data resides in relational database management systems, which have well defined interfaces and provided very clear guarantees to the operations performed over the data.
- However:
  - not all the data can reside in the same database
  - the application is built on top of the database. The guarantees provided by the database need to be understood by the application running on top

The nice thing about databases ...

- ... is that they take care of all aspects related to data management, from physical storage to concurrency control and recovery
- Using a database can reduce the amount of code necessary in a large application by about 40 %
- From a client/server perspective, the databases help in:
  - concurrency control: many servers can be connected in parallel to the same database and the database will still have correct data
  - recovery: if a server fails in the middle of an operation, the database makes sure this does not affect the data or other servers
- Unfortunately, these properties are provided only to operations performed within the database. In principle, they do not apply when:
  - An operation spawns several databases
  - the operations access data not in the database (e.g., in the server)
- To help with this problem, the Distributed Transaction processing Model was created by X/Open (a standard's body). The heart of this model is the XA interface for 2 Phase Commit, which can be used to ensure that an operation spawning several databases enjoy the same atomicity properties as if it were executed in one database.

One at a time interaction

- Databases follow a single thread execution model where a client can only have one outstanding call to one and only one server at any time. The basic idea is one call per process (thread).
- Databases provide no mechanism to bundle together several requests into a single work unit
- The XA interface solves this problem for databases by providing an interface that supports a 2 Phase Commit protocol. However, without any further support, the client becomes the one responsible for running the protocol which is highly impractical
- An intermediate layer is needed to run the 2PC protocol



2 Phase Commit

BASIC 2PC

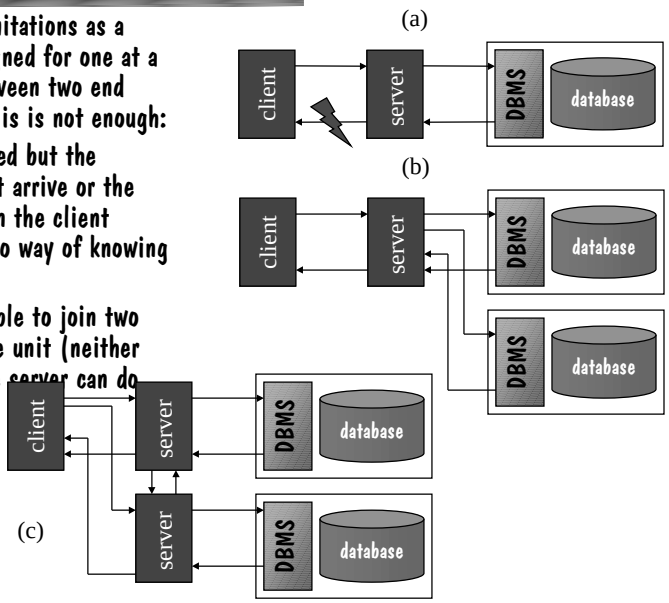
- Coordinator send PREPARE to all participants.
- Upon receiving a PREPARE message, a participant sends a message with YES or NO (if the vote is NO, the participant aborts the transaction and stops).
- Coordinator collects all votes:
  - Þ All YES = Commit and send COMMIT to all others.
  - Þ Some NO = Abort and send ABORT to all which voted YES.
- A participant receiving COMMIT or ABORT messages from the coordinator decides accordingly and stops.

What is needed to run 2PC?

- Control of Participants: A transaction may involve many resource managers, somebody has to keep track of which ones have participated in the execution
- Preserving Transactional Context: During a transaction, a participant may be invoked several times on behalf of the same transaction. The resource manager must keep track of calls and be able to identify which ones belong to the same transaction by using a transaction identifier in all invocations
- Transactional Protocols: somebody acting as the coordinator in the 2PC protocol
- Make sure the participants understand the protocol (this is what the XA interface is for)

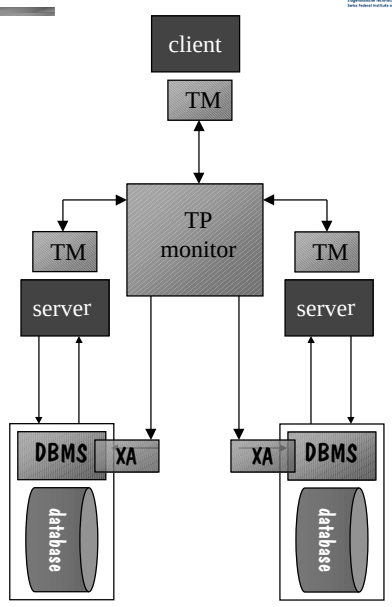
Interactions through RPC

- RPC has the same limitations as a database: it was designed for one at a time interactions between two end points. In practice, this is not enough:
  - a) the call is executed but the response does not arrive or the client fails. When the client recovers, it has no way of knowing what happened
  - b) c) it is not possible to join two calls into a single unit (neither the client nor the server can do this)

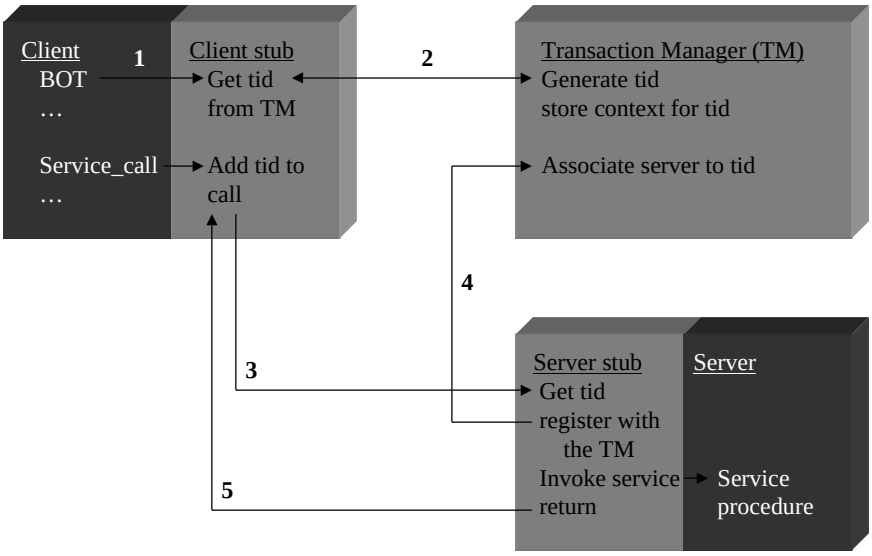


Transactional RPC

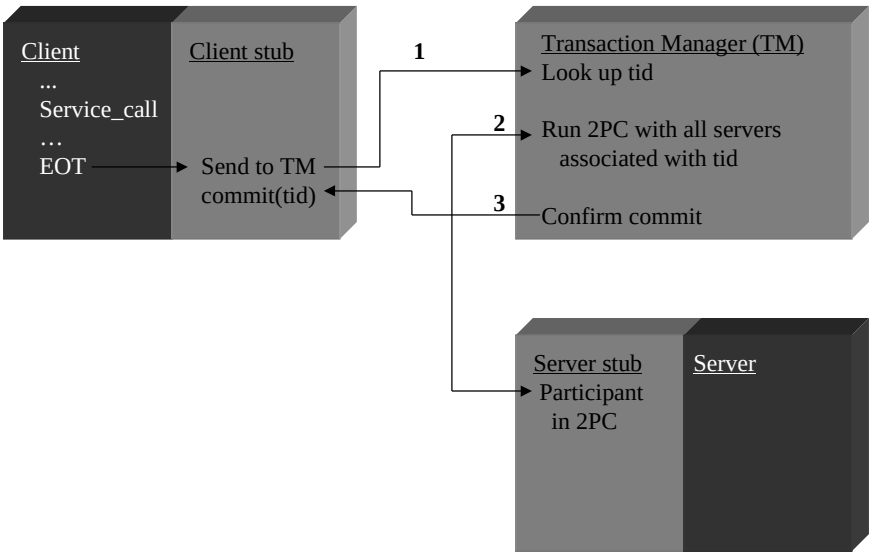
- The limitations of RPC can be resolved by making RPC calls transactional. In practice, this means that they are controlled by a 2PC protocol
- As before, an intermediate entity is needed to run 2PC (the client and server could do this themselves but it is neither practical nor generic enough)
- This intermediate entity is usually called a transaction manager (TM) and acts as intermediary in all interactions between clients, servers, and resource managers
- When all the services needed to support RPC, transactional RPC, and additional features are added to the intermediate layer, the result is a TP-Monitor



Basic TRPC (making calls)

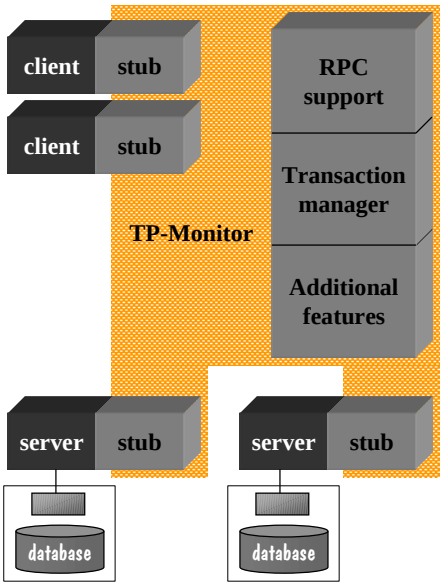


Basic TRPC (committing calls)

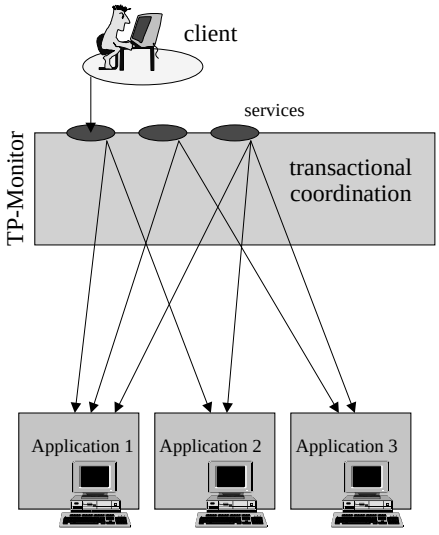


One step beyond ...

- The previous example assumes the server is transactional and can run 2PC. This could be, for instance, a stored procedure interface within a database. However, this is not the usual model
- Typically, the server invokes a resource manager (e.g., a database) that is the one actually running the transaction
- This makes the interaction more complicated as it adds more participants but the basic concept is the same:
  - the server registers the resource manager(s) it uses
  - the TM runs 2PC with those resources managers instead of with the server (see OTS at the end)



TP-Monitors = transactional RPC

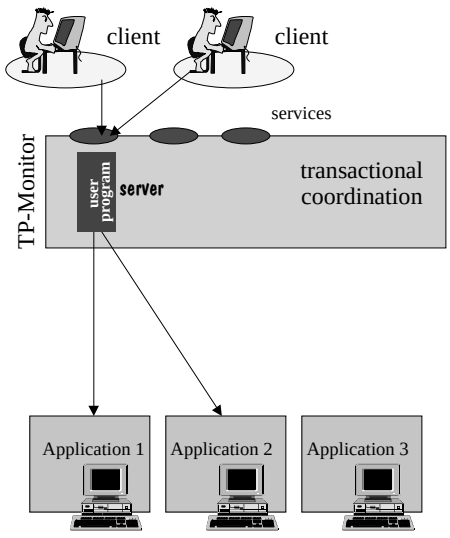


- A TP-Monitor allows building a common interface to several applications while maintaining or adding transactional properties. Examples: CICS, Tuxedo, Encina.
- A TP-Monitor extends the transactional capabilities of a database beyond the database domain. It provides the mechanisms and tools necessary to build applications in which transactional guarantees are provided.
- TP-Monitors are, perhaps, the best, oldest, and most complex example of middleware. Some even try to act as distributed operating systems providing file systems, communications, security controls, etc.
- TP-Monitors have traditionally been associated to the mainframe world. Their functionality, however, has long since migrated to other environments and has been incorporated into most middleware tools.

TP-Monitor functionality

- TP-Monitors appeared because operating systems are not suited for transactional processing. TP-Monitors are built as operating systems on top of operating systems.
- As a result, TP-Monitor functionality is not well defined and very much system dependent.
- A TP-Monitor tries to cover the deficiencies of existing "all purpose" systems. What it does is determined by the systems it tries to "improve".
- A TP-Monitor is basically an integration tool. It allows system designers to tie together heterogeneous system components using a number of utilities that can be mixed and matched depending on the particular characteristics of each case.
- Using the tools provided by the TP-Monitor, the integration effort becomes more straightforward as most of the needed functionality is directly supported by the TP-Monitor.
- A TP-Monitor addresses the problems of sharing data from heterogeneous, distributed sources, providing clean interfaces and ensuring ACID properties.
- A TP-Monitor extrapolates the functions of a transaction manager (locking, scheduling, logging, recovery) and controls the distributed execution. As such, TP-Monitor functionality is at the core of the integration efforts of many software producers (databases, workflow systems, CORBA providers, ...).
- A TP-Monitor also controls and manages distributed computations. It performs load balancing, monitoring of components, starting and finishing components as needed, routing of requests, recovery of components, logging of all operations, assignment of priorities, scheduling, etc. In many cases it has its own transactional file system, becoming almost indistinguishable from a distributed operating system.

Transactional properties



- The TP-monitor tries to encapsulate the services provided within transactional brackets. This implies RPC augmented with:
  - Þ atomicity: a service that produces modifications in several components should be executed entirely and correctly in each component or should not be executed at all (in any of the components).
  - Þ isolation: if several clients request the same service at the same time and access the same data, the overall result will be as if they were alone in the system.
  - Þ consistency: a service is correct when executed in its entirety (it does not introduce false or incorrect data into the component databases)
  - Þ durability: the system keeps track of what has been done and is capable of redoing and undoing changes in case of failures.

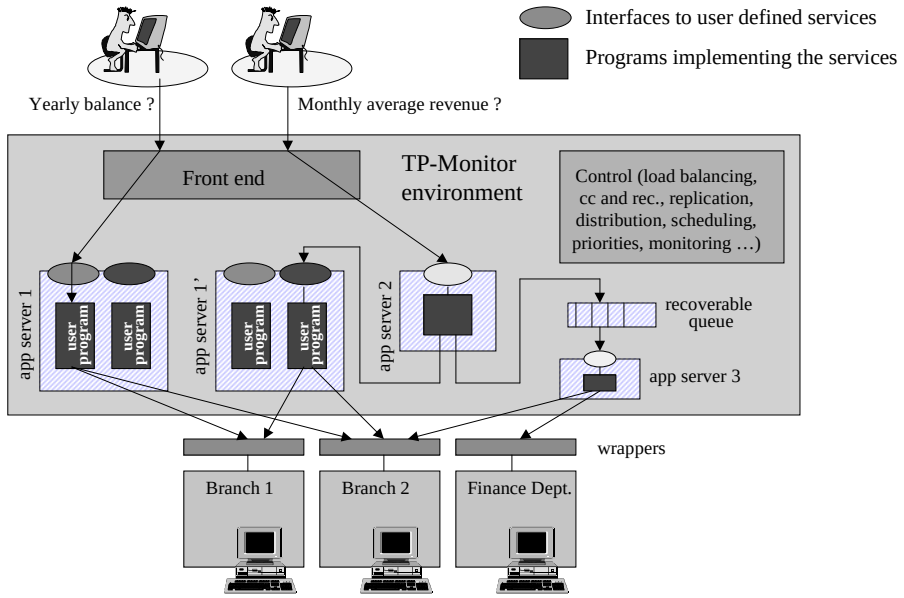
TRAN-C (Encina)

```
# include <tc/tc.h>
inModule("helloWorld");

void Main () {
   int i;
   inFunction("main");
   initTC();                    /* initializes transaction manager */

   transaction {                /* starts a transaction */
         printf("Hello World - transaction %d\n", getTid());
         if (I % 2) abort ("Odd transactions are aborted");
         }
   onCommit
         printf("Transaction Comitted");
   onAbort
         printf("Abort in module: %s\n \t %s\n", abortModuleNAme(), abortReason());
}
```


TP-Monitor, generic architecture



Tasks of a TP Monitor

Core services

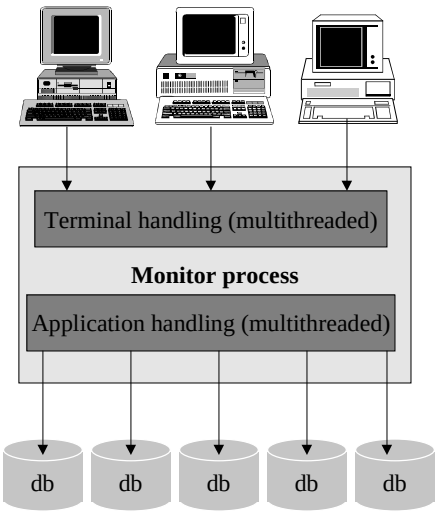
- Transactional RPC: Implements RPC and enforces transactional semantics, scheduling operations accordingly
- Transaction manager: runs 2PC and takes care of recovery operations
- Log manager: records all changes done by transactions so that a consistent version of the system can be reconstructed in case of failures
- Lock manager: a generic mechanism to regulate access to shared data outside the resource managers

Additional services

- Server monitoring and administration: starting, stopping and monitoring servers; load balancing
- Authentication and authorization: checking that a user can invoke a given service from a given terminal, at a given time, on a given object and with a given set of parameters (the OS does not do this)
- Data storage: in the form of a transactional file system
- Transactional queues: for asynchronous interaction between components
- Booting, system recovery, and other administrative chores

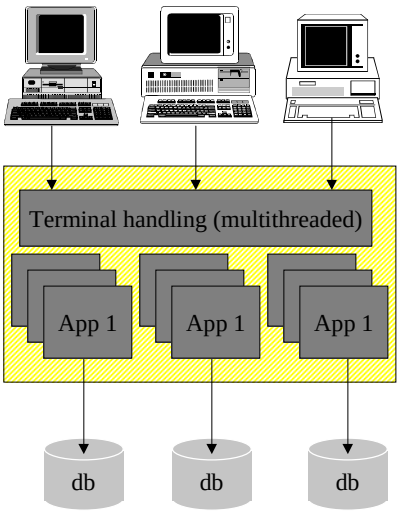
Structure of TP-Monitors (I)

- TP-Monitors try in many aspects to replace the operating system so as to provide more efficient transactional properties. Depending what type of operating system they try to replace, they have a different structure:
  - Þ Monolithic: all the functionality of the TP-Monitor is implemented within one single process. The design is simpler (the process can control everything) but restrictive (bottleneck, single point of failure, must support all possible protocols in one single place).
  - Þ Layered: the functionality is divided in two layers. One for terminal handling and several processes for interaction with the resource managers. The design is still simple but provides better performance and resilience.
  - Þ Multiprocessor: the functionality is divided among many independent, distributed processes.

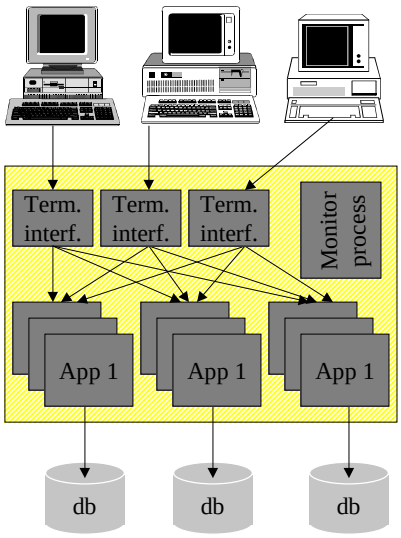


Monolithic structure

Structure of TP-Monitors (II)

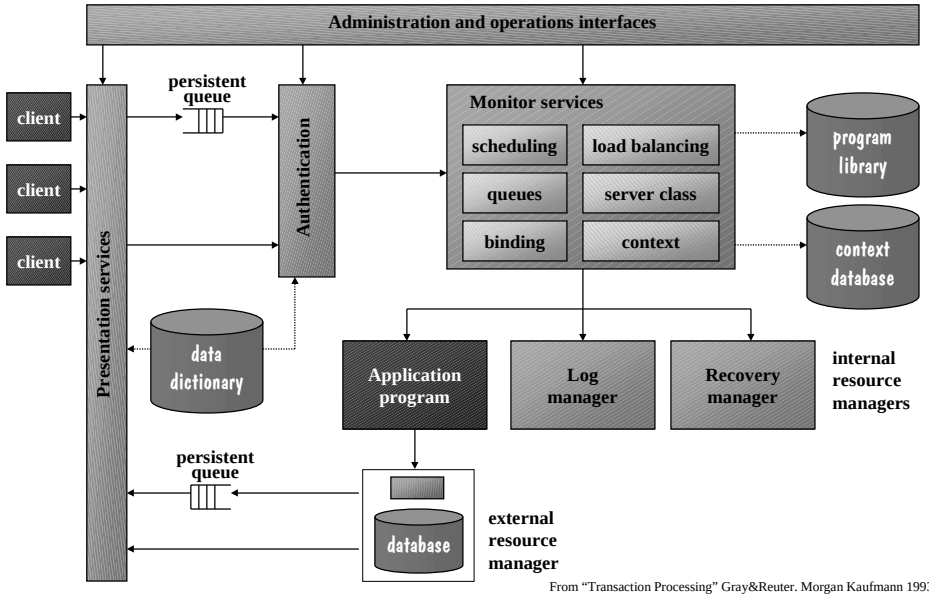


Layered structure



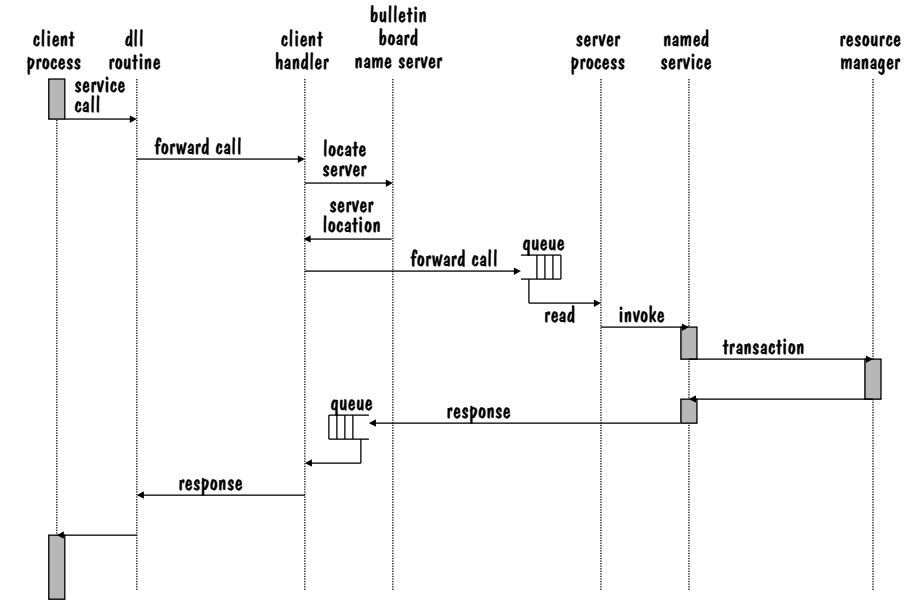
Multiprocessor structure

TP-Monitor components (generic)



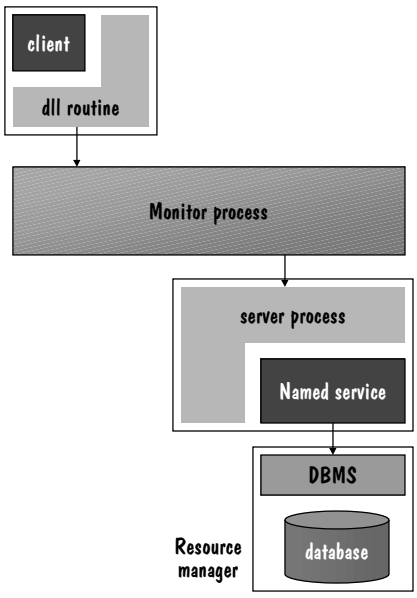
From "Transaction Processing" Gray&Reuter. Morgan Kaufmann 1993

Example: BEA Tuxedo



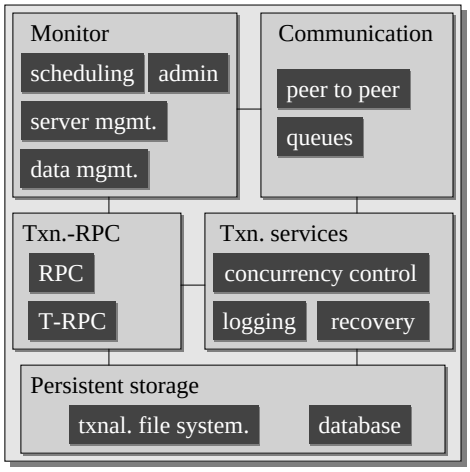
Example: BEA Tuxedo

- The client uses DLL (Dynamic Link Libraries) routines to interact with the TP-Monitor
- The Monitor Process or Tuxedo server implements all system services (name services, transaction management, load balancing, etc) and acts as the control point for all interactions
- Application services are known as named services. These named services interact with the system through a local server process
- Interaction across components is through message queues rather than direct calls (although clients and servers may interact synchronously)



TP-Monitor components (Encina)

- The current trend is towards a "family of products" instead of a single system. Each element can be used by itself (reduced footprint) and, in some cases, can be used completely independent of the TP-Monitor.
- Monitor: execution environment providing integrity, availability, security, fast response time and high throughput. It includes tools for administration and installation of components and the development environment.
- Communication services: protocols and mechanisms for persistent messages and peer to peer communication.
- Transactional RPC: basic interaction mechanism
- Transactional services: supporting concurrency control, recovery, logging and transactional programming. Behavior of the system can be tailored (advances transaction models, selective logging, ad-hoc recovery ...)
- Persistent storage



External interfaces

With clients

- The main interface is through the presentation services. In old systems, presentation services included terminal handling and format control for presentation on a screen. Today, the presentation services are mostly interfaces to other systems that take care of data presentation (mainly web servers)
- The most important part of the presentation services still in use today is the RPC (TRPC) stubs and libraries used on the client side for invoking services implemented within the TP-Monitor

With administrators

- The TP-Monitor needs to be maintained and administered like any other system. Today there are a wide variety of tools for doing so. They include:
  - Þ node monitoring
  - Þ service monitoring
  - Þ load monitoring
  - Þ configuration tools
  - Þ programming support
  - Þ ...
- Another important part of the interfaces to the system are the development environments which tend to be similar in nature to that of RPC systems

Monitor services

- Monitor services are those facilities that provide the basic functionality of the TP-Monitor. They can be implemented as part of the TP-Monitor process or as external resource managers
- Server class: each application program implementing services has a server class in the monitor. The server class starts and stops the application, creates message queues, monitors the load, etc. In general, it manages one application program
- Binding: acts as the name and directory services and offers similar functionality as the binder in RPC. It might be coupled with the load balancing service for better distribution
- Load balancing: tries to optimize the resources of the system by providing an accurate picture of the ongoing and scheduled work
- Context management: a key service in TRPC that is also used in keeping context across transaction boundaries or to store and forward data between different resource managers and servers
- Communication services (queue management and networking) are usually implemented as external resource managers. They take care of transactional queuing and any other aspect of message passing

Resource managers

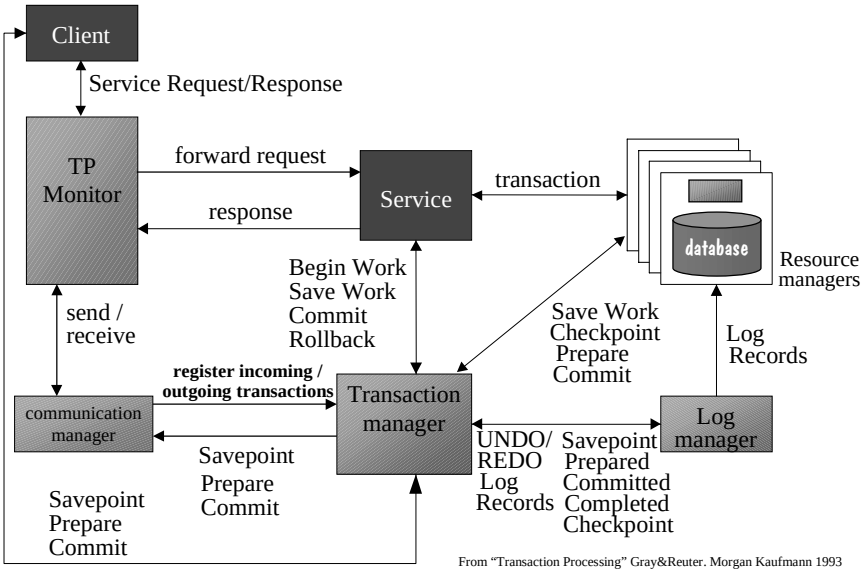
Internal Resource Managers

- These are modules that implement a particular service in the TP-Monitor. There are two kinds:
- Application programs: programs that implement a collection of services that can be invoked by the clients of the TP-Monitor. They define the application built upon the TP-Monitor
- Internal services: like logging, locking, recovery, or queuing. Implementing these services as resource managers gives more modularity to the system and even allows to use other systems for this purpose (like queue management systems)

External Resource Managers

- These are the systems the TP-Monitor has to integrate
- The typical resource manager is a database management system with an SQL/XA interface. It can also be a legacy application, in which case wrappers are needed to bridge the interface gap. A typical example are screen scraping modules that interact with mainframe based applications by posing as dumb terminals
- The number and type of external resource managers keeps growing and a resource manager can be another TP monitor.
- The WWW is slowly also becoming a resource manager

Transaction processing components



From "Transaction Processing" Gray&Reuter. Morgan Kaufmann 1993

TP-Monitors vs. OS

	processing	data	communication
TP Services	Admin interface Configuration tools Load balancing Programming tools	Databases Disaster recovery Resource managers Flow of control	Name server Server invocation Protected user interface
TP internal system services	Txn identifiers Server class Scheduling Authentication	Transaction manager Logs and context Durable queues Transactional files	Transactional RPC Transactional Sessions RPC
OS	Process – Threads Address space Scheduling Local naming protection	Repository File System Blocks, paging File security	IPC Simple sessions Naming Authentication
Hardware	CPU	Memory	Wires, switches

TP Monitor

From "Transaction Processing" Gray&Reuter. Morgan Kaufmann 1993

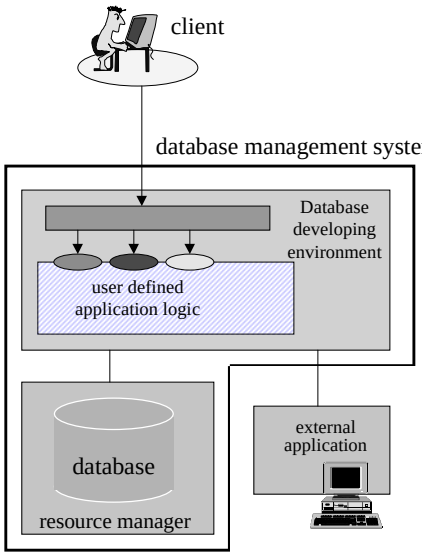
Advantages of TP-Monitors

- TP-Monitors are a development and run-time platform for distributed applications
- The separation between the monitor and the transaction manager was a practical consideration but turned out to be a significant advantage as many of the features provided by the monitor are as valuable as transactions
- The move towards more modular architectures prepared TP-Monitors for changes that had not been foreseen but turned be quite advantageous:
  - the web as the main interface to applications: the presentation services included an interface so that requests could be channeled through a web server
  - queuing as a form of middleware in itself (Message Oriented Middleware, MOM): once the queuing service was an internal resource manager, it was not too difficult to adapt the interface so that the TP-Monitor could talk with other queuing systems
  - Distributed object systems (e.g., CORBA) required only a small syntactic layer in the development tools and the presentation services so that services will appear as objects and TRPC would be come a method invocation to those objects.

TP-Heavy vs. TP-Light = 2 tier vs. 3 tier

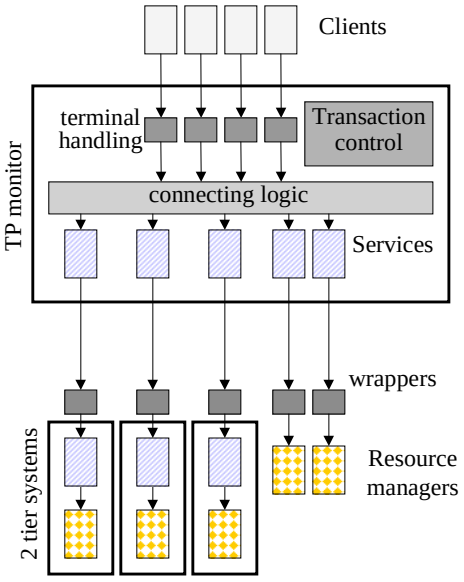
- A TP-heavy monitor provides:
  - a full development environment (programming tools, services, libraries, etc.),
  - additional services (persistent queues, communication tools, transactional services, priority scheduling, buffering),
  - support for authentication (of users and access rights to different services),
  - its own solutions for communication, replication, load balancing, storage management ... (most of the functionality of an operating system).
- Its main purpose is to provide an execution environment for resource managers (applications), and do all this with guaranteed reasonable performance (e.g., > 1000 txns. per second).
- This is the traditional monitor: CICS, Encina, Tuxedo.
- A TP-Light is an extension to a database:
  - it is implemented as threads, instead of processes,
  - it is based on stored procedures ("methods" stored in the database that perform an specific set of operations) and triggers,
  - it does not provide a development environment.
- Light Monitors are appearing as databases become more sophisticated and provide more services, such as integrating part of the functionality of a TP-Monitor within the database.
- Instead of writing a complex query, the query is implemented as a stored procedure. A client, instead of running the query, invokes the stored procedure.
- Stored procedure languages: Sybase's Transact-SQL, Oracle's PL/SQL.

TP-light: databases and the 2 tier approach



- Databases are traditionally used to manage data.
- However, simply managing data is not an end in itself. One manages data because it has some concrete application logic in mind. This is often forgotten when considering databases (specially benchmarking) and has allowed SAP to take over a significant market share before any other vendors reacted.
- But if the application logic is what matters, why not move the application logic into the database? These is what many vendors are advocating. By doing this, they propose a 2 tier model with the database providing the tools necessary to implement complex application logic.
- These tools include triggers, replication, stored procedures, queuing systems, standard access interfaces (ODBC, JDBC) .. which are already in place in many databases.

TP-Heavy: 3-tier middleware



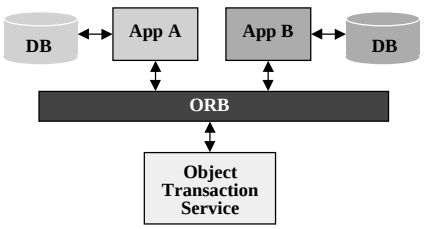
- TP-heavy are middleware platforms for developing 3-tier architectures. They provide all the functionality necessary for such an architecture to work.
- A system designer only need to program the services (which will run within the scope of the TP-Monitor; the services are linked to a number of TP libraries providing the needed functionality), the wrappers (if they are not already provided), and the clients. The TP-Monitors takes these components and embeds them within the overall system as interconnected components.
- The TP-Monitor provides the infrastructure for the components to work and the tools necessary to build services, wrappers and clients. In some cases, it provides even its own programming language (e.g., Transational-C of Encina).

Object Transaction Service

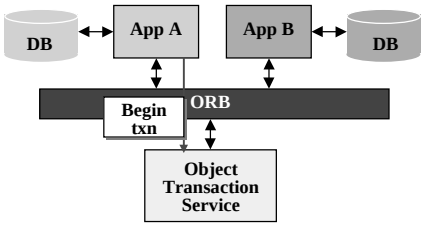
- An OTS provides transactional guarantees to the execution of invocations between different components of a distributed application built on top of an ORB. It is part of the CORBA standard It is identical to a basic TP-Monitor
- There are two ways to trace calls:
  - Explicit (manual): the invocation itself contains the transaction identifier. Then, when the application registers the resource manager, it uses this transaction identifier to say to which transaction it is "subscribing"
  - Implicit (automatic): the call is made through the OTS, which will forward the transaction identifier along with the invocation. This requires to link with the OTS library and to make all methods involved transactional
- ... and two ways to register resources (necessary in order to tell the OTS who will participate in the 2PC protocol and what type of interface is supported)
- Manual registration implies the the user provides an implementation of the resource. This implementation acts as an intermediary between the OTS and the actual resource manager (useful for legacy applications that need to be wrapped)
- Automatic registration is used when the resource manager understands transactions (i.e., it is a database), in which case it will support the XA interface for 2PC directly. A resource are registered only once, and implicit propagation is used to check which transactions go there

Running a distributed transaction (1)

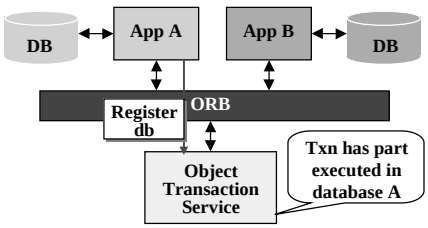
1) Assume App A wants to update databases A and B



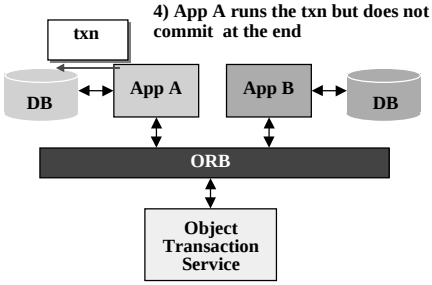
2) App A obtains a txn identifier for the operation



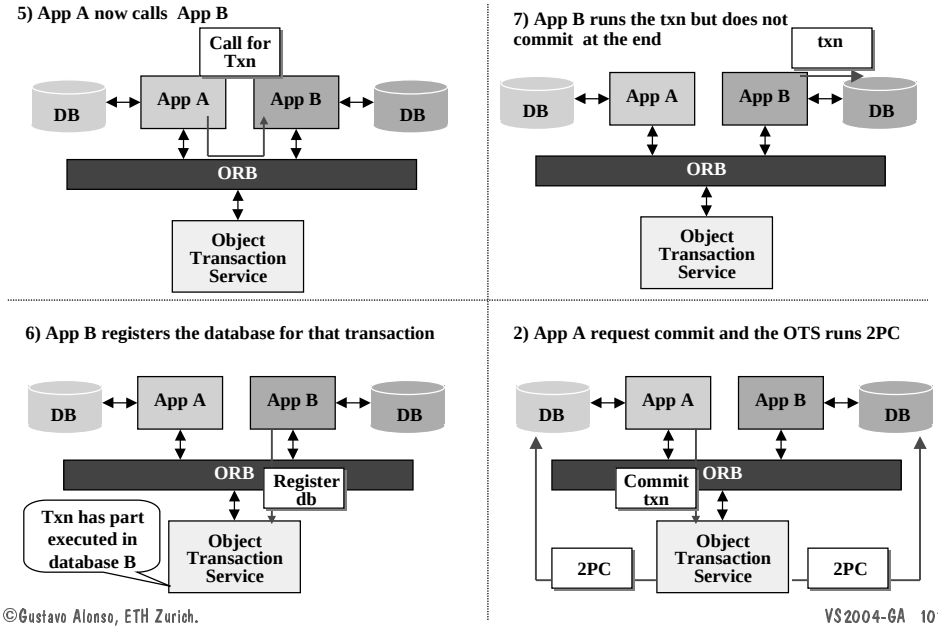
3) App A registers the database for that transaction



4) App A runs the txn but does not commit at the end



Running a distributed transaction (2)



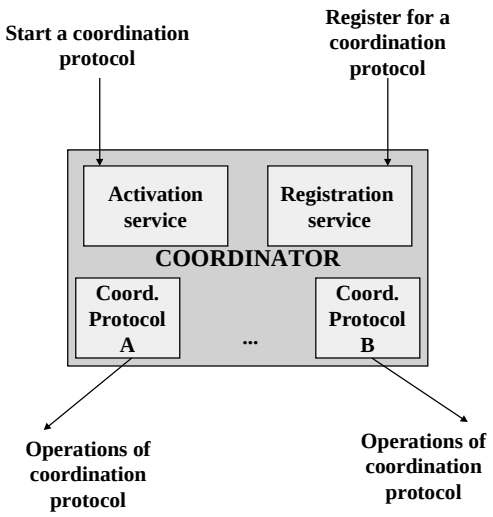
The future of TP-Monitors

- TP-Monitors are the best example of middleware and the most successful implementation both in terms of performance and functionality.
- Together with object brokers, TP-Monitors form the foundation of today's distributed data management products. Enterprise Application Integration is still largely based on TP-Monitor technology.
- TP-Monitors are the main reference for implementing middleware:
  - in terms of performance, TP-Monitors are orders of magnitude ahead of other middleware systems
  - in terms of functionality, TP-Monitors offer a quite complete, well integrated platform that can be extended to provide the functionality needed in other middleware systems
- Unlike other forms of middleware, TP-Monitors have proven to be quite resilient in time: some product lines are almost 30 years old already. Although the technology changes, the answer to fundamental design problems is well understood in TP-Monitors. These expertise will still have a significant impact on any emerging form of middleware.

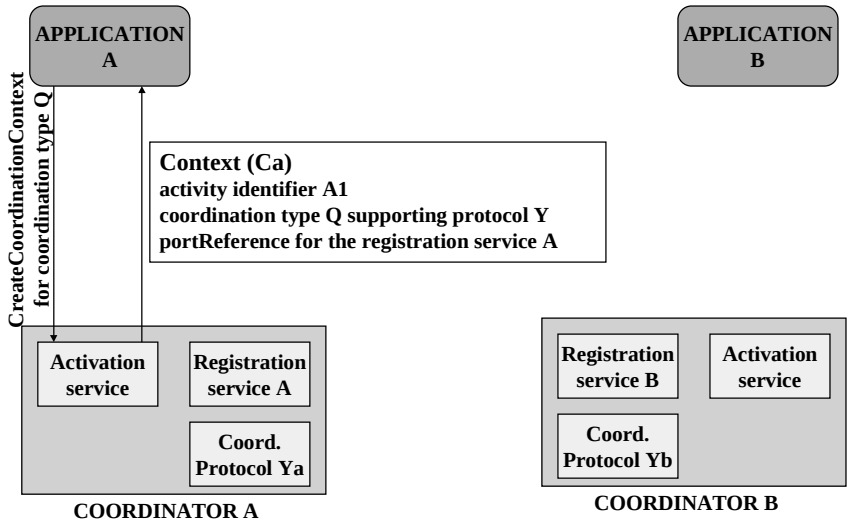
WS-Coordination

WS-Coordination

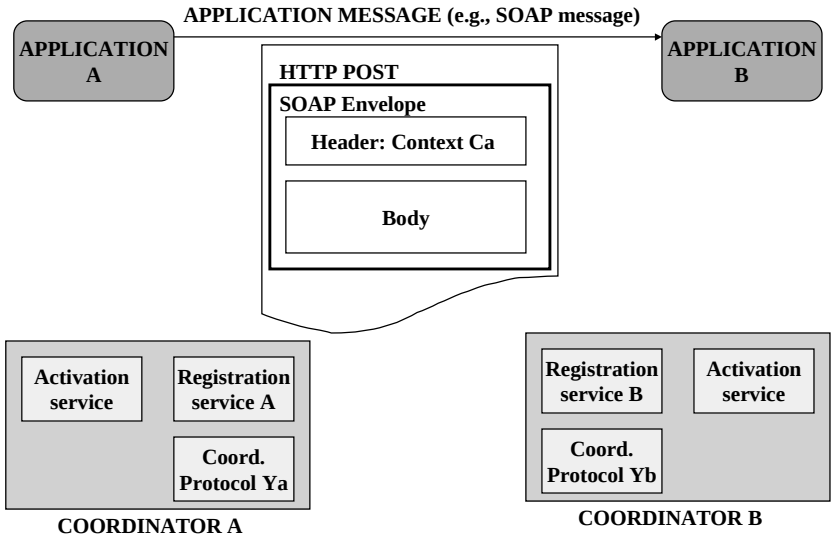
- WS-Coordination is intended as a generic infrastructure to implement coordination protocols between Web services
- Its main goal is to serve as a generic platform for implementing advanced transaction models but it can be used to implement a wide variety of coordination protocols between services (including some forms of conversations)
- WS-Coordination encompasses a set of behaviors and APIs that conform a module that will extend Web services with coordination capabilities
- It mirrors the behavior of transactional services in conventional middleware platforms



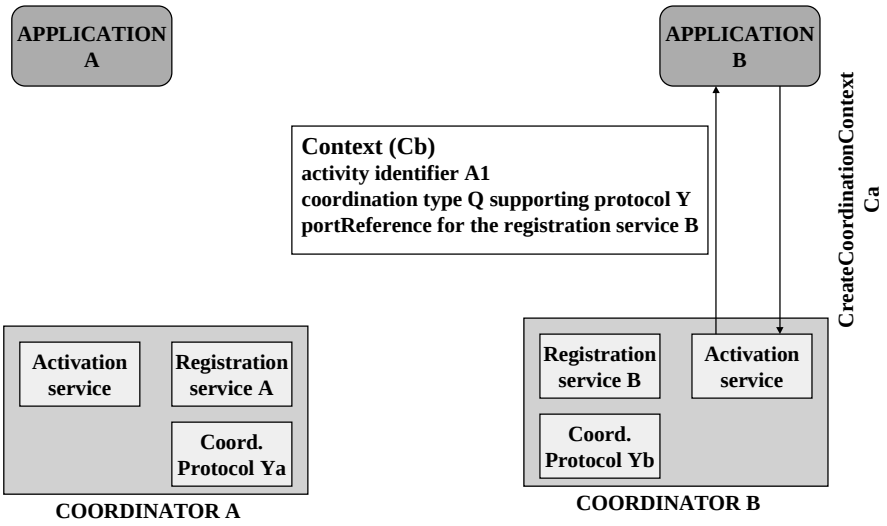
Basics of WS-Coordination (1)



Basics of WS-Coordination (2)

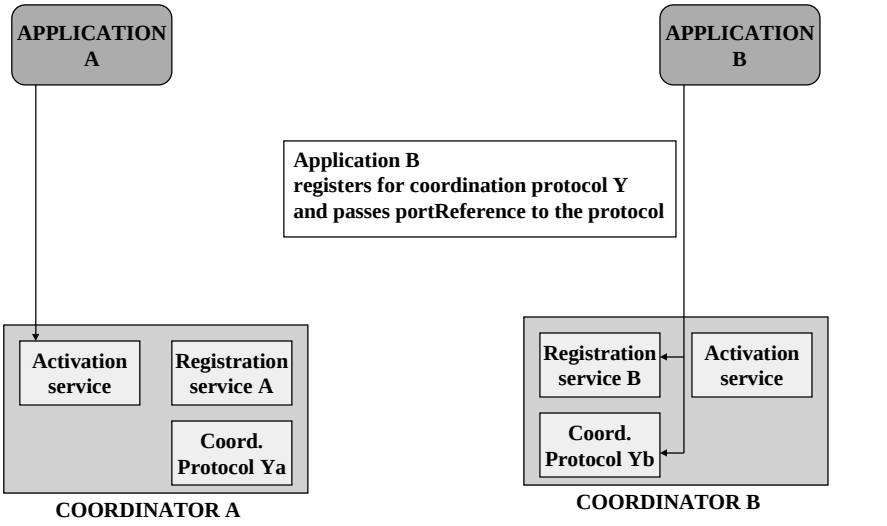


Basics of WS-Coordination (3)



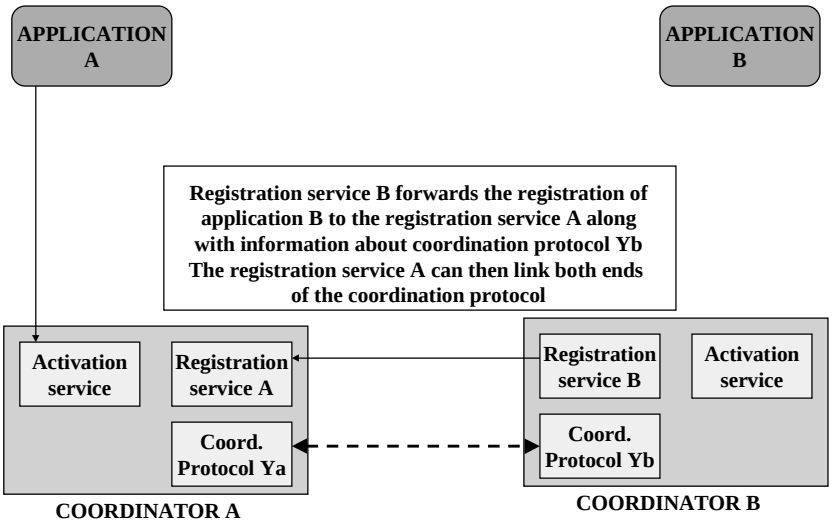
©Gustavo Alonso. ETH Zürich.

Basics of WS-Coordination (4)



©Gustavo Alonso. ETH Zürich.

Basics of WS-Coordination (5)



Messages and interfaces

- The coordinator defined by WS-Coordination is described using WSDL and offers a number of services to the application.
- The application accesses these services by sending, e.g., SOAP messages to the coordinator which then responds with new SOAP messages. Interactions with the protocol would then also be in terms of SOAP messages (but other protocols are possible, one needs only o provide alternative bindings for the coordinator services)
- The example shown considers the case where application B decides to use its own coordinator. Application B could also decide to use the same coordinator as application A but in the cases where A and B are independent services provided by different organizations a coordinator per application makes more sense
- WS-Coordination is an attempt at standardizing:
  - the use of SOAP headers for coordination protocols
  - the basic operations for most coordination protocols
  - the functionality a Web service middleware platform must support for allowing coordination protocols to be implemented

WS-Coordinator in XML

```
ACTIVATION SERVICE:
<wsdl:portType name="ActivationCoordinatorPortType">
    <wsdl:operation name="CreateCoordinationContext">
      <wsdl:input message="wscoor:CreateCoordinationContext"/>
    </wsdl:operation>
  </wsdl:portType>

RESPONSE ACTIVATION SERVICE
<wsdl:portType name="ActivationRequesterPortType">
    <wsdl:operation name="CreateCoordinationContextResponse">
      <wsdl:input message="wscoor:CreateCoordinationContextResponse"/>
    </wsdl:operation>
    <wsdl:operation name="Error">
      <wsdl:input message="wscoor:Error"/>
    </wsdl:operation>
  </wsdl:portType>
```

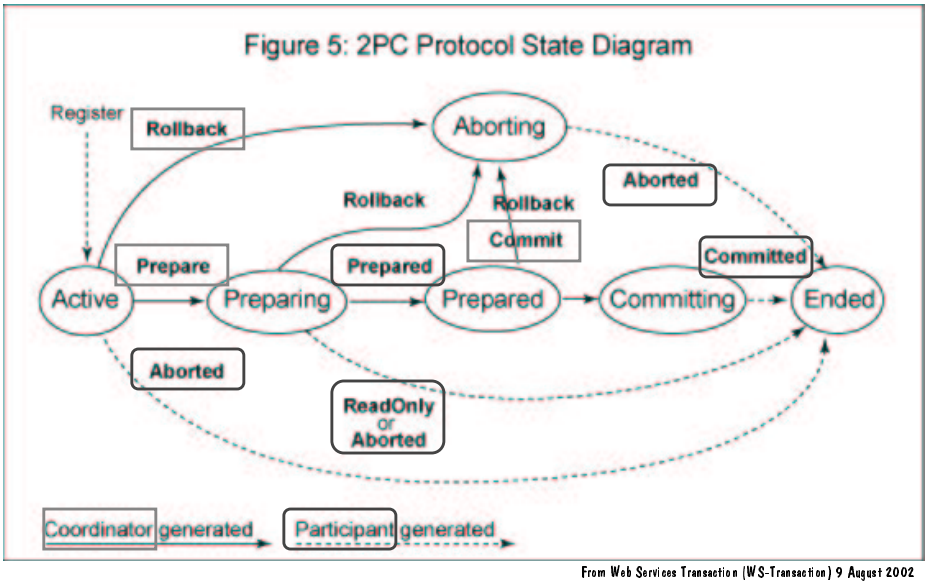
From Web Services Coordination (WS-Coordination) 9 August 2002

WS-Transactions

WS-Transactions

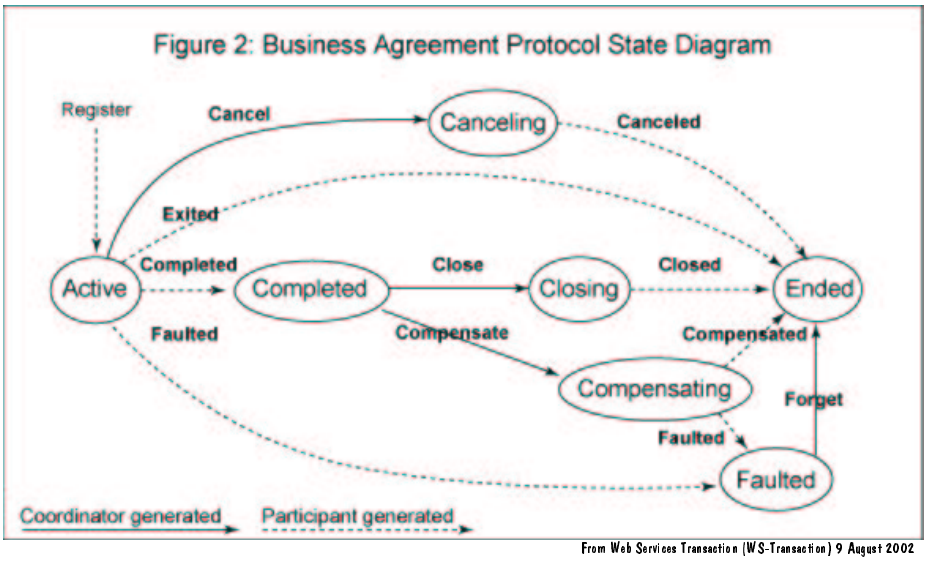
- WS-Transactions builds directly upon WS-Coordination to specify different coordination protocols related to transaction processing
  - Þ atomic transactions (governed by 2 Phase Commit)
  - Þ business activities (transactional but based on compensation activities)
    - business agreement
    - business agreement with complete
- WS-Transactions specifies the coordination protocol to be used as part of WS-Coordination. The specification deals with the nature of the interaction, the syntax and semantics of the messages to exchange as part of the coordination protocol, and the expected responses of all participants involved
- Like WS-Coordination, WS-Transactions follows very closely the transactional model found in conventional middleware platforms

Coordination protocol for 2PC



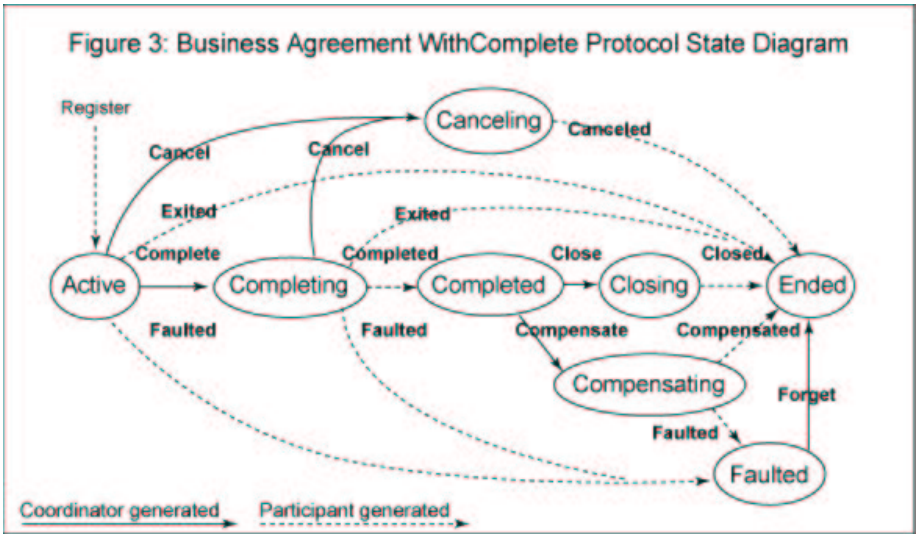
From Web Services Transaction (WS-Transaction) 9 August 2002

Business agreement



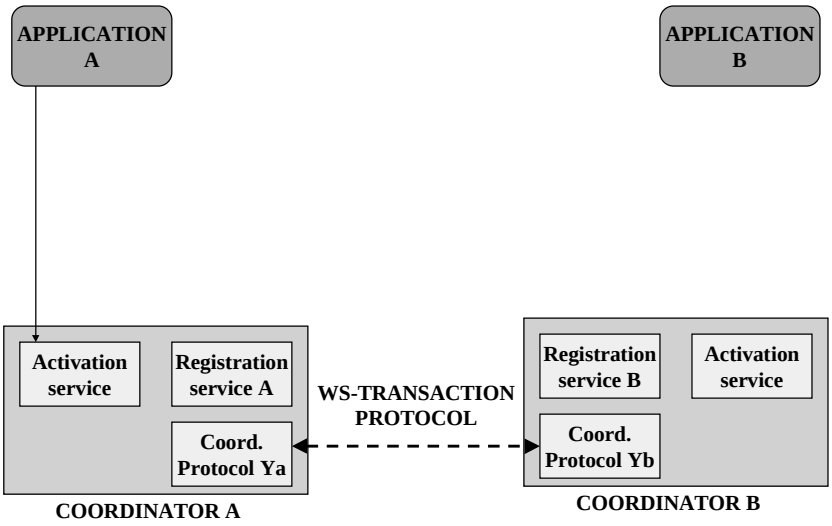
From Web Services Transaction (WS-Transaction) 9 August 2002

Business agreement with completion



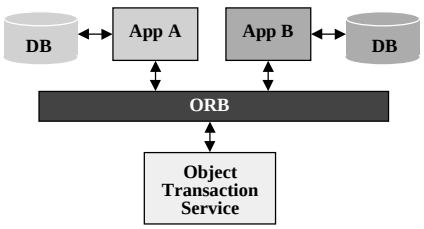
From Web Services Transaction (WS-Transaction) 9 August 2002

WS-Transactions

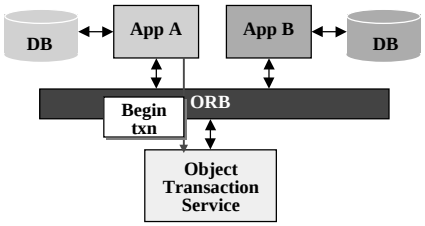


CORBA transactions (1)

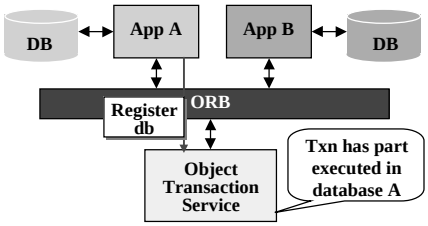
1) Assume App A wants to update databases A and B



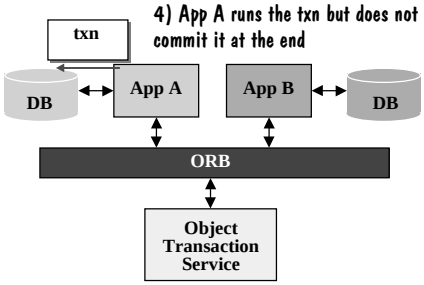
2) App A obtains a txn identifier for the operation



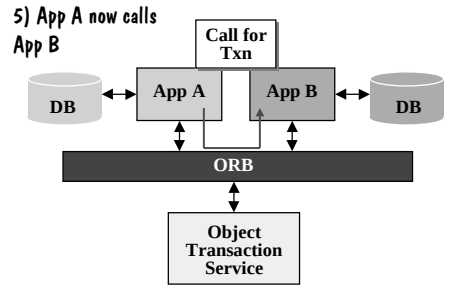
3) App A registers the database for that transaction

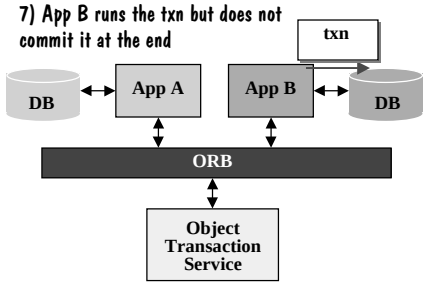


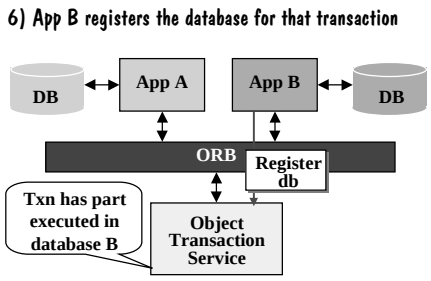
4) App A runs the txn but does not commit it at the end

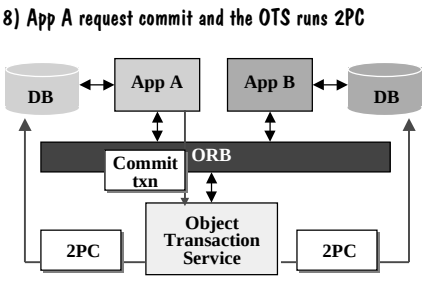


CORBA transactions (2)





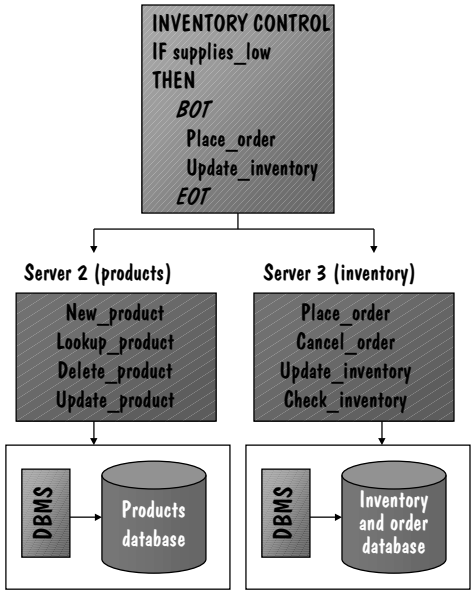




Transactional queues

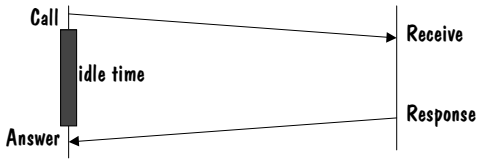
Synchronous Client/Server

- The most straightforward interaction between components is the request/response model in which the client sends a request and waits until the server provides a response:
  - Þ closely resembles the way we program (hence RPC as the basic mechanism to support this idea)
  - Þ the model is simple and intuitive
  - Þ well supported by RPC and the systems built around RPC (TRPC, TP-Monitors and even Object Monitors)
  - Þ needs additional infrastructure when interactions becomes more complex (e.g., nested) but this infrastructure is available



Disadvantages of sync C/S

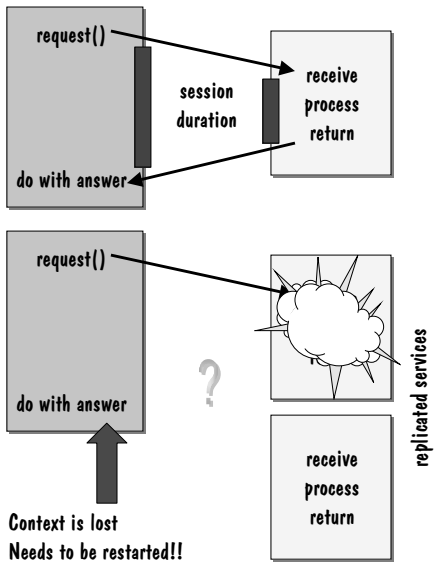
- Synchronous interaction requires both parties to be "on-line": the caller makes a request, the receiver gets the request, processes the request, sends a response, the caller receives the response.
- The caller must wait until the response comes back. The receiver does not need to exist at the time of the call (TP-Monitors, CORBA or DCOM create an instance of the service/server /object when called if it does not exist already) but the interaction requires both client and server to be "alive" at the same time



- Because it synchronizes client and server, this mode of operation has several disadvantages:
  - Þ connection overhead
  - Þ higher probability of failures
  - Þ difficult to identify and react to failures
  - Þ it is a one-to-one system; it is not really practical for nested calls and complex interactions (the problems becomes even more acute)

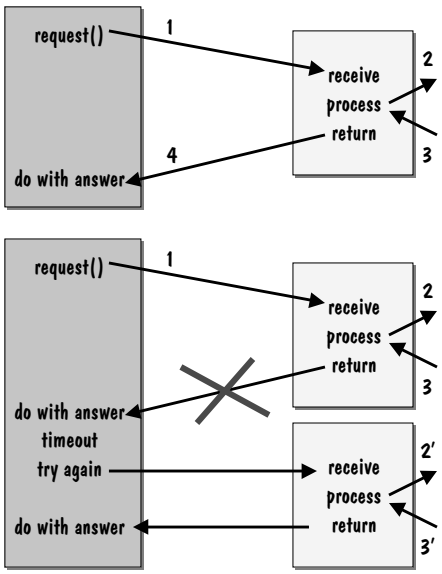
Overhead of synchronism

- Synchronous invocations require to maintain a session between the caller and the receiver.
- Maintaining sessions is expensive and consumes CPU resources. There is also a limit on how many sessions can be active at the same time (thus limiting the number of concurrent clients connected to a server)
- For this reason, client/server systems often resort to connection pooling to optimize resource utilization
  - Þ have a pool of open connections
  - Þ associate a thread with each connection
  - Þ allocate connections as needed
- When the interaction is not one-to-one, the context (the information defining a session) needs to be passed around. The context is usually volatile



Failures in synchronous calls

- If the client or the server fail, the context is lost and resynchronization might be difficult.
  - Þ If the failure occurred before 1, nothing has happened
  - Þ If the failure occurs after 1 but before 2 (receiver crashes), then the request is lost
  - Þ If the failure happens after 2 but before 3, side effects may cause inconsistencies
  - Þ If the failure occurs after 3 but before 4, the response is lost but the action has been performed (do it again?)
- Finding out when the failure took place may not be easy. Worse still, if there is a chain of invocations, the failure can occur anywhere.



Failure semantics

- A great deal of the functionality built around RPC tries to address the problem of failure semantics, i.e., determine what has happened after a failure
- Exactly-once semantics solves this problem but it has hidden costs:
  - it implies atomicity in all operations
  - the server must support some form of 2PC; if it is a database, then one can use the XA interface, otherwise one needs a TP-Monitor to make the server transactional
  - it usually requires a coordinator to oversee the interaction
- The more elements are involved in an interaction, the higher the probability that the interaction will fail (a failure in anyone of the elements results is enough)
- The more elements are required to be alive for an interaction to succeed, the more difficult it is to maintain the system:
  - even if it is modular, the components cannot do anything without the rest of the system
  - upgrades, corrections, general maintenance becomes very difficult because they might require to shut the system down

Two solutions

Enhanced Support

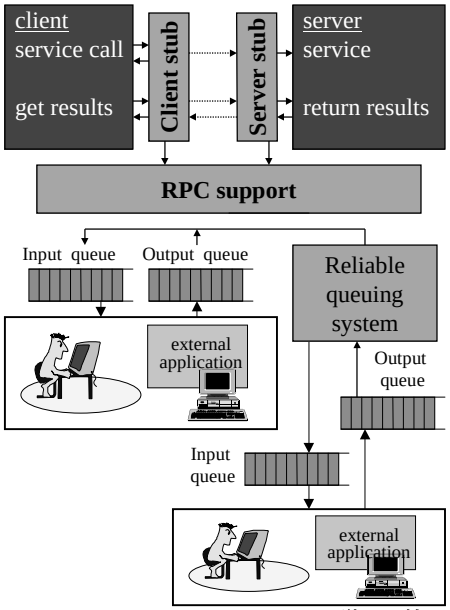
- Client/Server middleware provides a number of mechanisms to deal with the problems created by synchronous interaction:
  - Transactional RPC: to enforce exactly once execution semantics and enable more complex interactions with some execution guarantees
  - Service replication and load balancing: to prevent the system from having to shut down if a given service is not available; this also gives a chance to maintain and upgrade the system while keeping it online

ASYNCHRONOUS INTERACTION

- Using asynchronous interaction, the caller sends a message that gets stored somewhere until the receiver reads it and sends a response. The response is sent in a similar manner
- Asynchronous interaction can take place in two forms:
  - non-blocking invocation (RPC but the call returns immediately without waiting for a response, similar to batch jobs)
  - persistent queues (the call and the response are actually persistently stored until they are accessed by the client and the server)

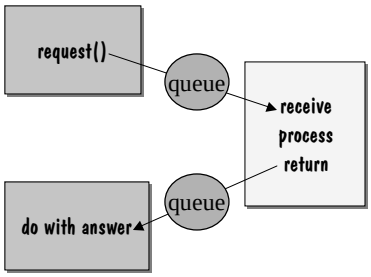
TP-Monitors

- The problems of synchronous interaction are not new. The first systems to provide alternatives were TP-Monitors which offered two choices:
  - asynchronous RPC: client makes a call that returns immediately; the client is responsible for making a second call to get the results
  - Reliable queuing systems (e.g., Encina, Tuxedo) where instead of through procedure calls, client and server interact by exchanging messages. Making the messages persistent by storing them in queues added considerable flexibility to the system

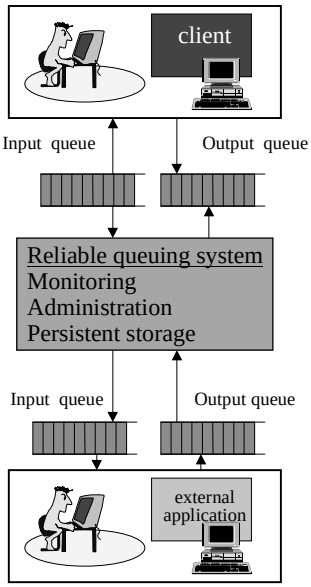


Reliable queuing

- Reliable queuing turned out to be a very good idea and an excellent complement to synchronous interactions:
  - Suitable to modular design: the code for making a request can be in a different module (even a different machine!) than the code for dealing with the response
  - It is easier to design sophisticated distribution modes (multicast, transfers, replication, coalescing messages) an it also helps to handle communication sessions in a more abstract way
  - More natural way to implement complex interactions (see next)



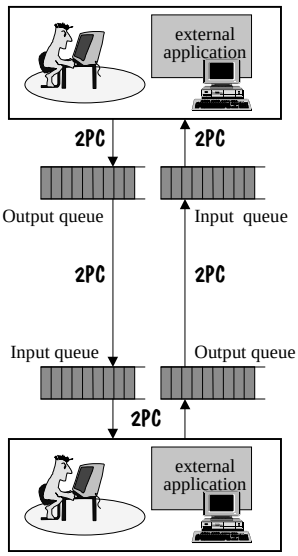
Queuing systems



- Queuing systems implement asynchronous interactions.
- Each element in the system communicates with the rest via persistent queues. These queues store messages transactionally, guaranteeing that messages are there even after failures occur.
- Queuing systems offer significant advantages over traditional solutions in terms of fault tolerance and overall system flexibility: applications do not need to be there at the time a request is made!
- Queues provide a way to communicate across heterogeneous networks and systems while still being able to make some assumptions about the behavior of the messages.
- They can be used embedded (workflow, TP-Monitors) or by themselves (MQSeries, Tuxedo/Q).

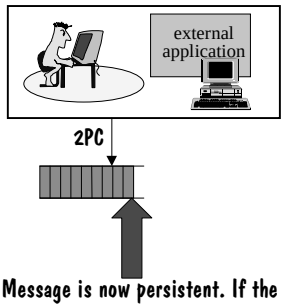
Transactional queues

- Persistent queues are closely tied to transactional interaction:
  - to send a message, it is written in the queue using 2PC
  - messages between queues are exchanged using 2PC
  - reading a message from a queue, processing it and writing the reply to another queue is all done under 2PC
- This introduces a significant overhead but it also provides considerable advantages. The overhead is not that important with local transactions (writing or reading to a local queue).
- Using transactional queues, the processing of messages and sending and receiving can be tied together into one single transactions so that atomicity is guaranteed. This solves a lot of problems!



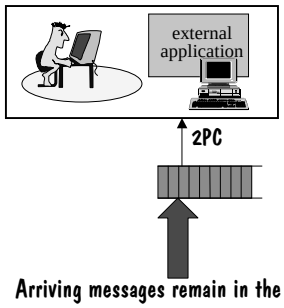
Problems solved (I)

SENDING



Message is now persistent. If the node crashes, the message remains in the queue. Upon recovery, the application can look in the queue and see which messages are there and which are not. Multiple applications can write to the same queue, thereby "multiplexing" the channel.

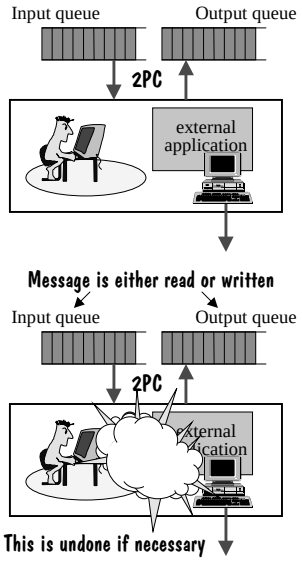
RECEIVING



Arriving messages remain in the queue. If the node crashes, messages are not lost. The application can now take its time to process messages. It is also possible for several applications to read from the same queue. This allows to implement replicated services, do load balancing, and increase fault tolerance.

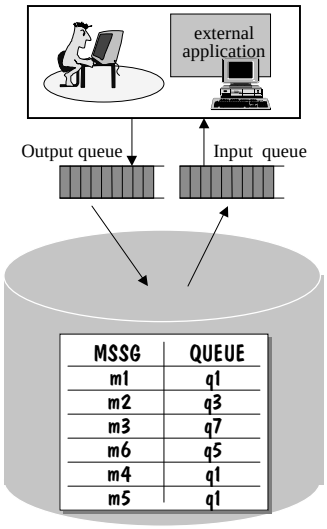
Problems solved (II)

- An application can bundle within a single transaction reading a message from a queue, interacting with other systems, and writing the response to a queue.
- If a failure occur, in all scenarios consistency is ensured:
  - Þ if the transaction was not completed, any interaction with other applications is undone and the reading operation from the input queue is not committed: the message remains in the input queue. Upon recovery, the message can be processed again, thereby achieving exactly once semantics.
  - Þ If the transaction was completed, the write to the output queue is committed, i.e., the response remains in the queue and can be sent upon recovery.
  - Þ If replicated services are used, if one fails and the message remains in the input queue, it is safe for other services to take over this message.



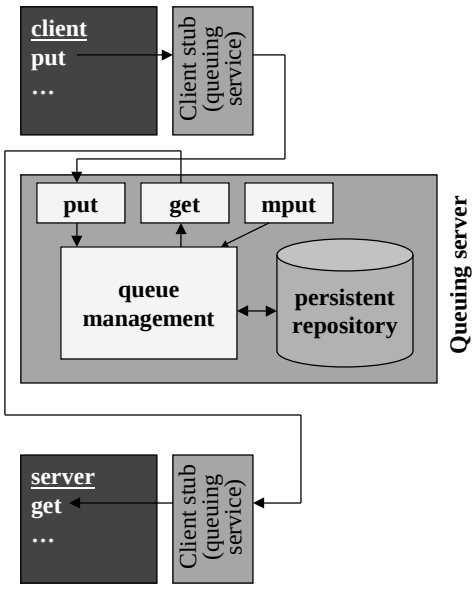
Simple implementation

- Persistent queues can be implemented as part of a database since the functionality needed is exactly that of a database:
  - a transactional interface
  - persistence of committed transactions
  - advanced indexing and search capabilities
- Thus, messages in a queue become simple entries in a table. These entries can be manipulated like any other data in a database so that applications using the queue can assign priorities, look for messages with given characteristics, trigger certain actions when messages of a particular kind arrive ...



Queues in practice

- To access a queue, a client or a server uses the queuing services, e.g., :
  - put (enqueue) to place a message in a given queue
  - get (dequeue) to read a message from a queue
  - mput to put a message in multiple queues
  - transfer a message from a queue to another
- In TP-Monitors, these services are implemented as RPC calls to an internal resource manager (the reliable queuing service)
- These calls can be made part of transaction using the same mechanisms of TRPC (the queuing system uses an XA interface and works like any other resource manager)



Advanced functionality

- Queues allow to implement complex interaction patterns between modules:
  - 1-to-1 interaction with failure resilience
  - 1-to-many (multicast: put in a queue and then send from this queue to many other queues) this is very helpful for "subscriptions". The fact that the queues are implemented in the database even helps with performance since the logic for distribution can be embedded in the database itself
  - many-to-1 many modules send their request to a single module that can then assign priorities, reorder, compare, etc.
  - many-to-many as in replicated services for large amount of clients
- In some cases queues are being used for interactions that are also on-line. If the queues are fast enough (like in a cluster) one can take advantage of the properties of queues at the expense of performance. Building computer farms becomes easier since messages are one more element that can be moved, copied and stored.
- Incorporating queues into databases provides databases with a very powerful tool for designing distributed applications (TP-light).

Types and messages

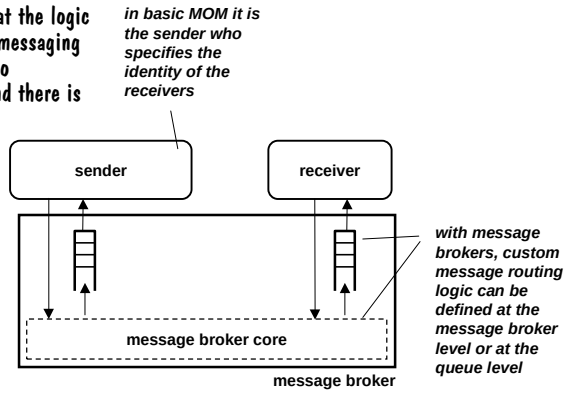
- Queues are very useful but they also have their disadvantages from the programming point of view:
  - In RPC, the type of the parameters exchanged between client and server is determined by the IDL definition and available in the stubs. The RPC infrastructure takes care of marshalling, unmarshalling, serializing, etc.
  - When queues are used, there is no IDL determining the interface. The type and format of the data in a queue must be agreed upon before hand but the system does not have much control over it
  - The role of IDL is now taken over by the message format (it is not in the stubs)
- The way to develop a system is as follows:
  - define message formats by creating complex types (records, objects)
  - create the queues and the access policies for those queues
  - program the server and clients according to the type definitions of the messages
- The system uses the types defined for the messages to set up the RPC calls needed to do marshalling, unmarshalling, serialization, etc.

Beyond client/server

- Persistent queues are most useful when the interactions are not simple client/server calls
  - Þ workflow processes can be easily implemented as a sequence of services that pass messages to each other along a well defined set of queues
  - Þ information dissemination and event notification can be directly and efficiently implemented on top of queues
  - Þ publish/subscribe systems are, in essence, event systems implemented on top of modified queuing systems
- Because these interactions are also very common and have increased in importance, queuing systems are no longer just one more module in TP-Monitors but have become products in their own right (e.g., MQSeries of IBM)
- Once they became products, queuing systems started to be subjected to the same evolutionary forces as other forms of middleware:
  - Þ integration in larger, more comprehensive tools
  - Þ enhancements to the basic functionality by making the queues active processing entities = Information Brokers

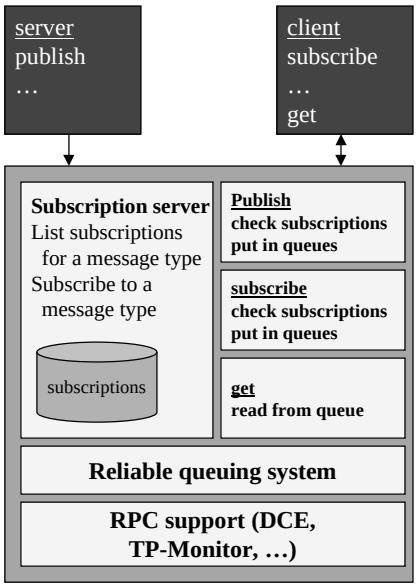
Message brokers

- Message brokers add logic to the queues and at the level of the messaging infrastructure.
- Messaging processing is no longer just moving messages between locations but designers can associate rules and processing steps to be executed when given messages are moved around
- The downside of this approach is that the logic associated with the queues and the messaging middleware might be very difficult to understand since it is distributed and there is no coherent view of the whole

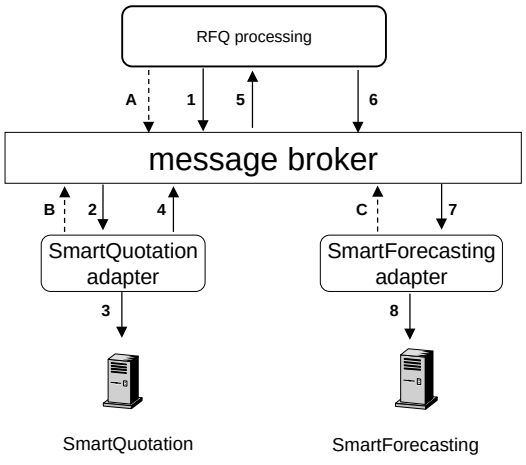


Publish/Subscribe

- Standard client/server architectures and queuing systems assume the client and the server know each other (through an interface or a queue)
- In many situations, it is more useful to implement systems where the interaction is based on announcing given events:
  - a service publishes messages/events of given type
  - clients subscribe to different types of messages/events
  - when a service publishes an event, the system looks at a table of subscriptions and forwards the event to the interested clients; this is usually done in the form of a message put into a queue for that client
- publish, subscribe, get, .. are also RPC calls to a resource manager



Subscription in message brokers



at systems startup time (can occur in any order, but all must occur before RFQs are executed)

A: subscription to message *quote*

B: subscription to message *quoteRequest*

C: subscription to message *newQuote*

at run time: processing of a request for quote.

1: publication of a *quoteRequest* message

2: delivery of message *quoteRequest*

3: synchronous invocation of the *getQuote* function

4: publication of a *quote* message

5: delivery of message *quote*

***6:* publication of a new*Quote* message**

7: delivery of message *newQuote*

***8:* invocation of the *createForecastEntry* procedure**