Transactions

- From the perspective of the user a transaction is a *consistent* and *failure-resilient* execution of application(s).
- Example: withdraw $100 from my checking account using an ATM.
- A transaction can be viewed as a computation that transform the system state.
- State before the execution of the transaction: my account balance $500, no money dispensed by the ATM.
- State after the execution of the transaction: my account balance $400, and $100 dispensed by the ATM.
- Consistent transformation: the transaction execution causes the ATM to dispense exactly the same amount as is debited from the bank account.
- failure-resilience: if the money is dispensed by the ATM it is debited from my account inspite of all failures.

ACID Properties of Transactions

- **Atomicity:** if the transaction terminates *successfully*, the state of the real-world reflects all the effects of the transaction. Else, if something goes wrong, none of the effects of the transaction apear.
  - My account is debited if and only if I get the money from the ATM.
- **Consistency:** The transformation caused by the transaction is "correct".
  - Balance in my account after debit is positive.
- **Isolation:** Effects of a transaction are hidden from other concurrently executing transactions.
- **Durability:** Once the transaction executes successfully, its changes to the state survive all subsequent malfunctions.
  - After the transaction terminates, and the ATM dispenses the money, the state of my account must always reflect that $100 were withdrawn– inspite of all failures.
- In later classes, we will give precise meaning to each of the above terms and study algorithms that are used to implements these properties.

What Transaction Semantics provides to a User

- *Simple failure semantics*–
  - either the ATM machine gives me the money, or else if something goes wrong my account is not debited.

- *An isolated view of the world*–
  - noone else is simultaneously accessing my account.

- Thus, it provides an environment for developing complex (possibly distributed) applications in which the the application designer does not need to worry about synchronization and fault-tolerance– the system does it automatically!

Examples

- Communication Industry: Setting up of an international call by Sprint long distance carrier, which the user charges on her discover card.
  - Authentication of the identity by the discover card using their database.
  - allocation of resources (telephone lines) by Sprint for the call.
  - Sprint may need to hire the telecommunication services of AT&T (and multiple other companies) to establish the call since there may not be a direct line owned by Sprint that can establish the call.
  - Call forwarding – the callee has forwarded his calls to his car phone.
  - Billing for the usage of the resources by everyone involved.
- Notice the importance of ACID properties for the above application–
  - A user should not be charged if the call does not go through.
  - Concurrent execution of multiple transactions must not result in allocation of the same line to two different calls at the same time.

Other Examples

- *Finance:* Banking, stock trading, point of sale.
- *Travel:* Reservations and billing of airline, cars, trains, hotel.
- *Process Control:* control of factories, warehouses, steel, paper, chemical plants.
- *Manufacturing:* order entry, job and inventory planning and scheduling, accounting.

Limitations of Basic Transactions

- Basic transactions simplify application development, but they may be too limiting at times…
- Example: Modeling Trip Planning as a transaction:

  BEGIN WORK
  - S1: book flight from San Francisco to Frankfurt
  - S2: book flight from Frankfurt to Milan, same day
  - S3: book flight from Milan to Pisa, same day

  – **Problem: There is no way to get to Ripa from Pisa the same day, except in a rental car. But you do not want to drive at night.**

  – **To roll back S3, we have to rollback the whole transaction**
  – **Loss of work, the deals available as part of S1 and S2 might no longer be available, …**

- **Many extensions to basic transaction model to overcome limitations**

Transactions with Savepoints

- **Using savepoints inside a transaction:** Savepoints are explicitly established by the application program and can be reestablished by invoking a modified ROLLBACK function, which is aimed at an internal savepoint rather than at the beginning of the transaction. For reference, savepoints are numbered. The BEGIN WORK of a transaction implicitly establishes savepoint number 1.

- Example: to allow for selective rollback, trip planning application can add savepoint after each step

- What if rollback is a result of system failure?
  - Can save points be persistent?

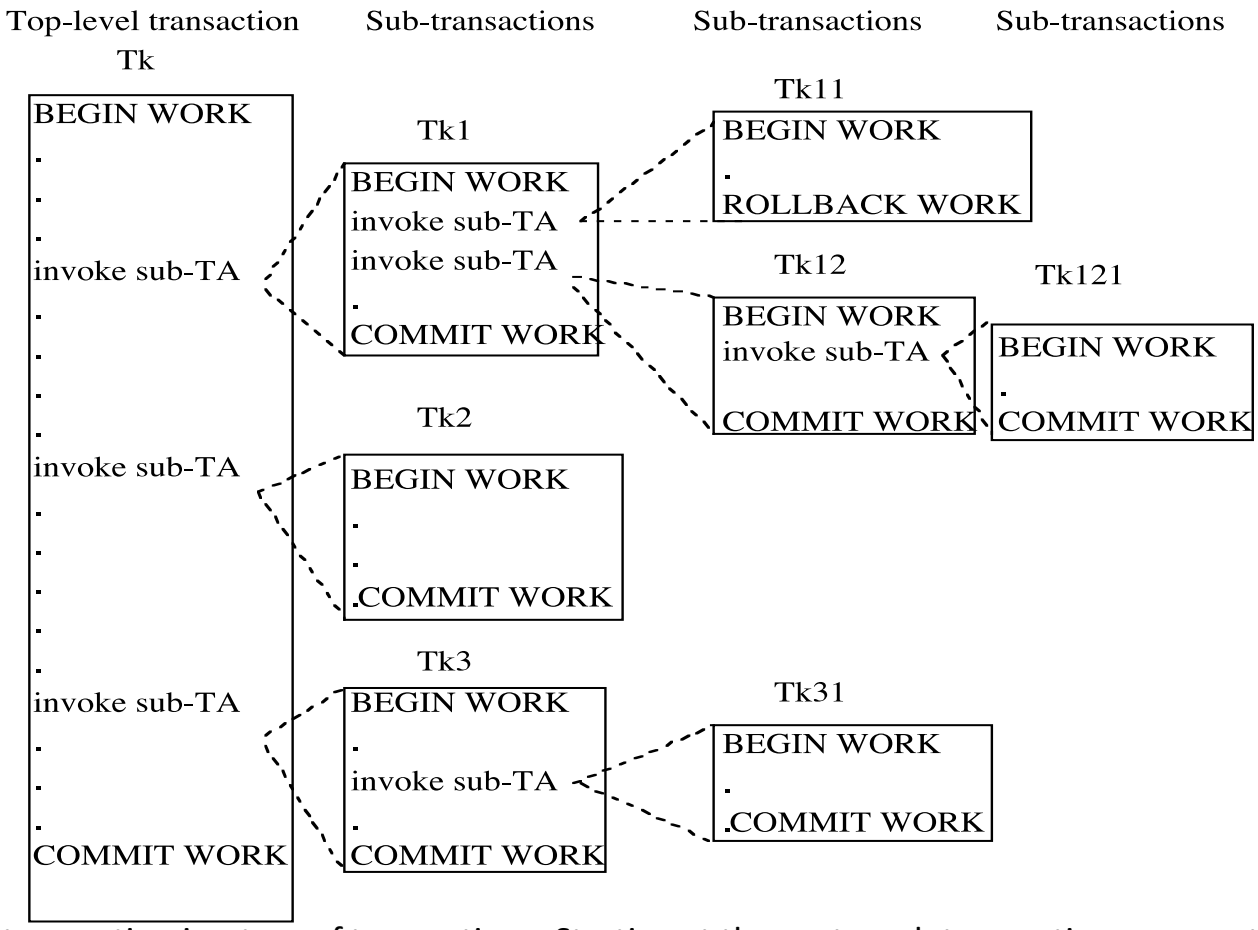
Nested Transactions

- A nested transaction is a tree of transactions, the sub-trees of which are either nested or flat transactions.
- Transactions at the leaf level are basic transactions. The distance from the root to the leaves can be different for different parts of the tree.
- The transaction at the root of the tree is called **top-level transaction**, the others are called **sub-transactions**. A transaction's predecessor in the tree is called **parent**; a sub-transaction at the next lower level is also called a **child**.
- A sub-transaction can either commit or rollback; its commit will not take effect, though, unless the parent transaction commits. So, by induction, any sub-transaction can finally commit only if the root transaction commits.
- The rollback of a transaction anywhere in the tree causes all its sub-transaction to roll back. This combined with the previous point is the reason why sub-transactions have only ACI, but not D

Dynamic Behavior of Nested Transactions

- **Commit rule:** The commit of a sub-transaction makes its results accessible to the parent transaction only. The sub-transaction will **finally** commit (i.e. release its results to the outside world) if and only if it has committed itself locally and all its ancestors up to the root have finally committed.
- **Rollback rule:** If a (sub-) transaction at any level of nesting is rolled back, all its sub-transactions are also rolled back, independent of their local commit status. This is applied recursively down the nesting hierarchy.
- **Visibility rule:** All changes done by a sub-transaction become visible to the parent transaction upon the sub-transaction's commit. All objects held by a parent transaction can be made accessible to its sub-transactions. Changes made by a sub-transaction are not visible to its siblings

Nested Transaction

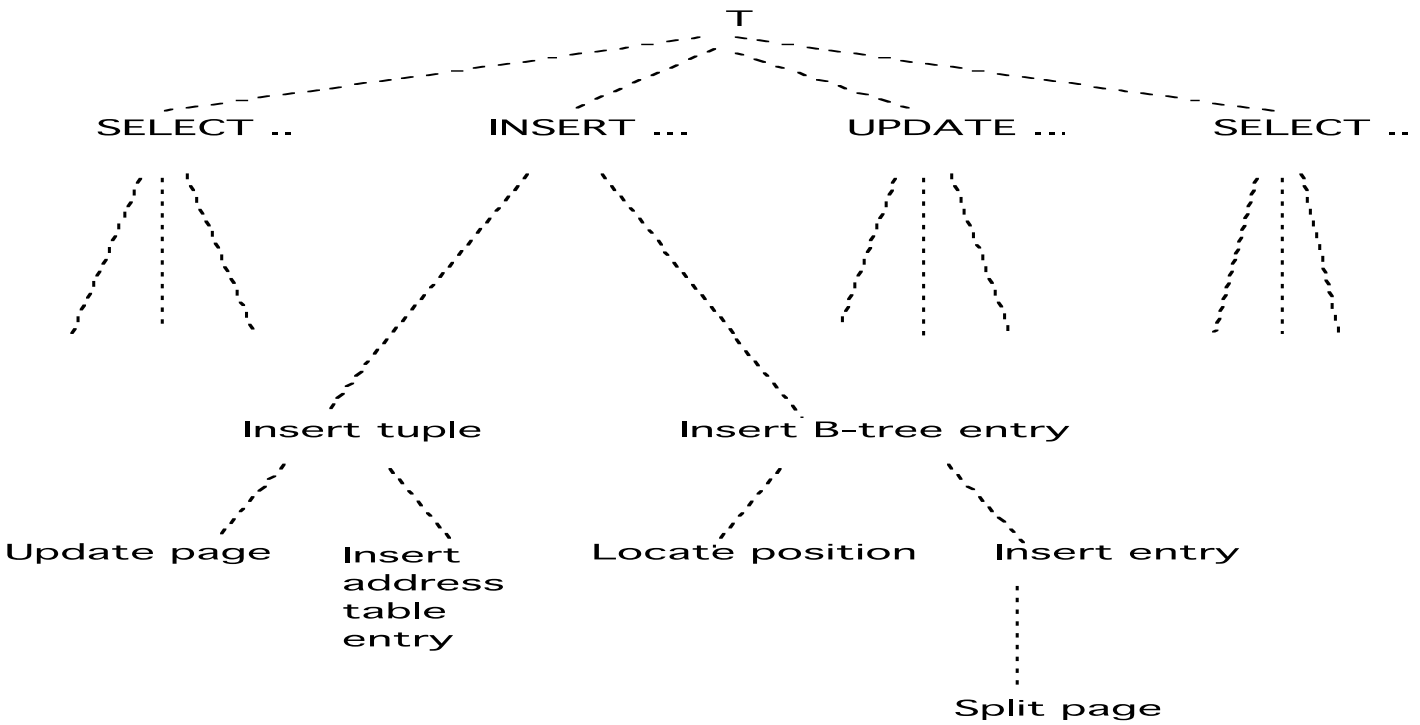


A nested transaction is a tree of transactions. Starting at the root, each transaction can create lower-level transactions (subtransactions), which are embedded in the sphere of control of the parent. Transactions at the leaf level are flat transactions, except that they lack the durability of non-nested flat transactions.

Multilevel Transactions

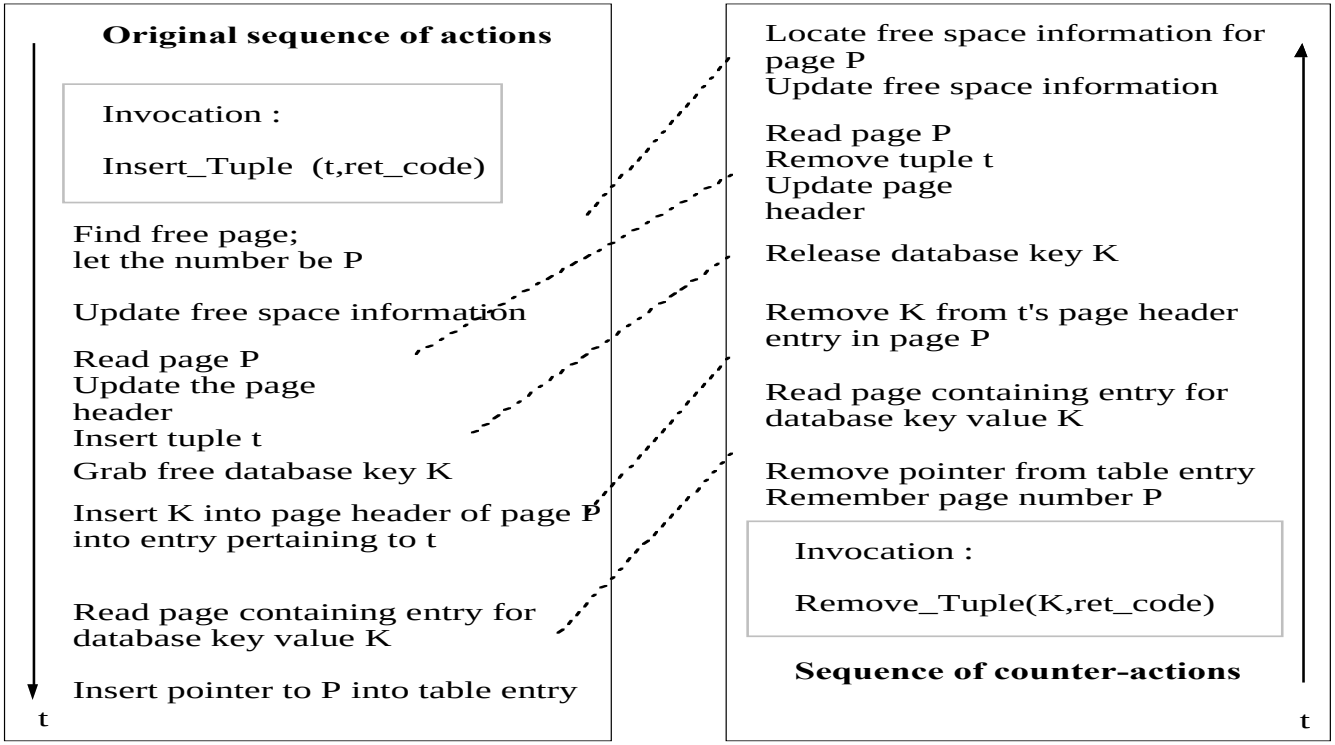
- A generalized notion of nested transactions
- Subtransactions can commit (and expose data to outside world) even before the transaction commits.
- For each subtraction, there MUST exist a compensating transaction which can “semantically” reverse the original subtraction in case the parent transaction decides to (or is forced to) roll back.
- Compensating transaction may itself be a nested or a multilevel transaction in itself.
- A compensating transaction MUST be such that it is guaranteed to commit (it cannot abort since by definition the abort of the original transaction cannot fail).

Use of Multilevel Transactions



Multi-level transactions can be used to structure operations on abstract objects, which are implemented by more primitive operations on lower-level objects, and so forth. For each operation at each level, there must be an inverse operation (compensation). As an example, consider SQL operations on tuple objects. The statements are nested within the top-level application transaction. The effects of each statement can be committed at their level of implementation before the surrounding top-level transaction commits, provided the objects they implement are commitment-controlled at a higher level, and provided there is a counter-action that can compensate the original action.

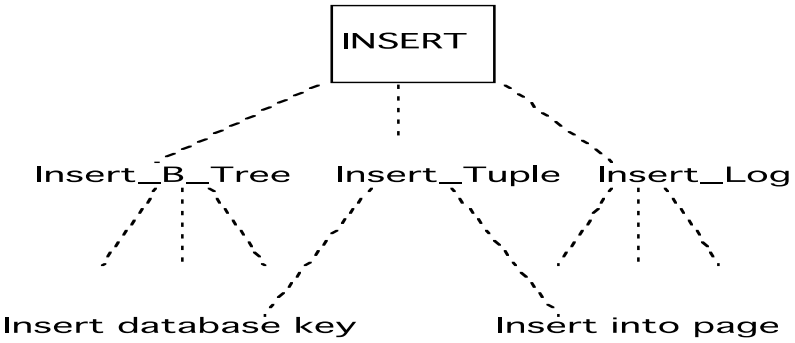
Using Multilevel Transactions



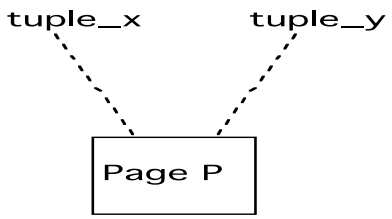
- **Maintaining tuples in a page by nested subtransactions:** If inserting a tuple into a page is implemented as a nested transaction that commits before the root transaction does, there must be a corresponding compensating transaction that guarantees to remove the tuple from the page if this is necessary—even if the page has subsequently been modified by other (unrelated) transactions.

Using MultiLevel Transactions

- Multi-level transactions can be used if the following structural requirements are fulfilled:
  - **Abstraction hierarchy:** The entire system consists of a strict **hierarchy** of objects with their associated operations.
  - **Layered abstraction:** The objects of layer **n** are completely implemented by using operations of layer **n-1**.
  - **Discipline:** There are no shortcuts that allow layer **n** to access objects on a layer other than **n-1**.
- Note the difference between implementation layers and object mapping layers:



mapping of operations



mapping of objects

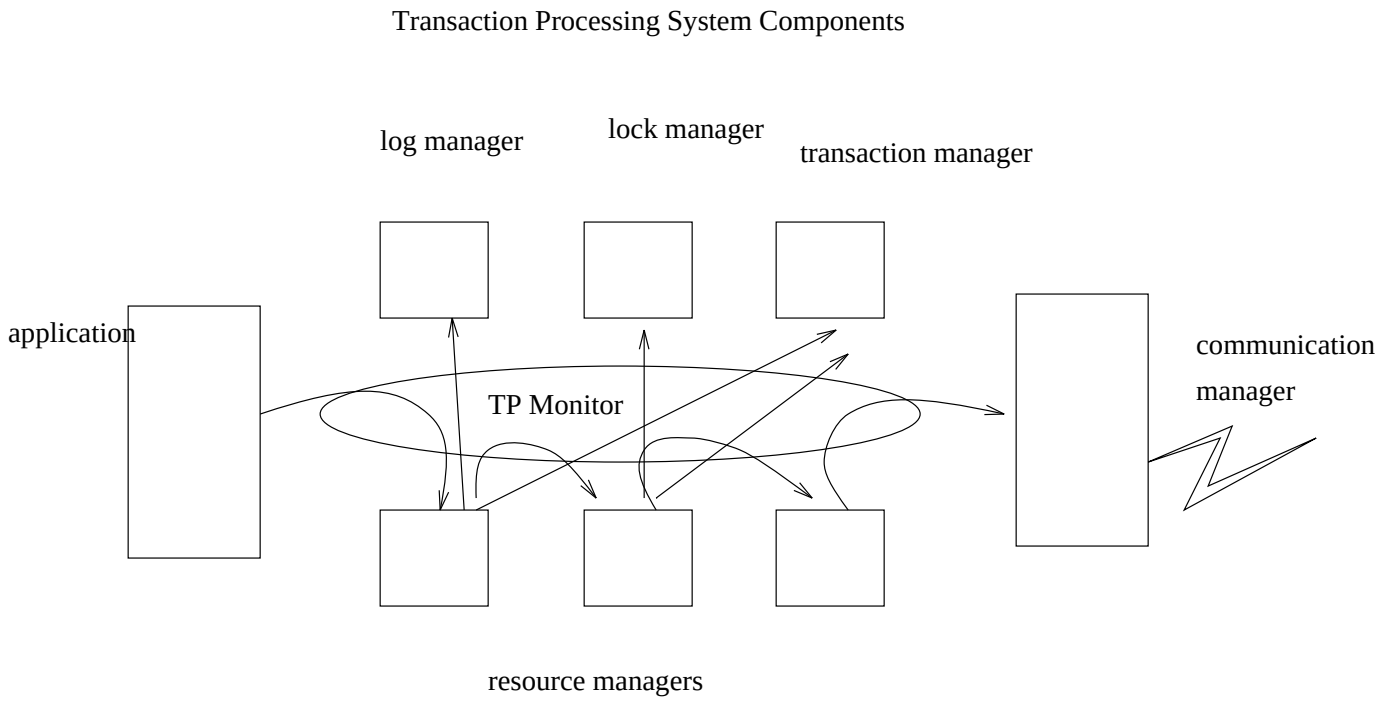
Other Extensions …

- Chained Transactions
- Sagas
- CAD transactions
- Long-lived transactions
- Transactions with context …
- ACTA
- ..

Transaction Processing Systems

- Transaction processing (TP) systems are systems that supports transactions.
- Large system– may consist of thousands of terminals, hundreds of computers, 100's of gigabytes of data.
- 1000's of transaction per second.
- Zero downtime– may be required to be operational even in presence of multiple hardware/ software failures as well as natural disasters (e.g., fire, earthquakes, etc).

Components of a TP System



- A TP monitor that controls the flow of transactions through the system.
- A lock manager, log manager, transaction manager.
- Resource Managers — databases, queue managers (spoolers), etc.
- Applications.
- Terminal Presentation Services.
- Communication Manager.

TP Monitors

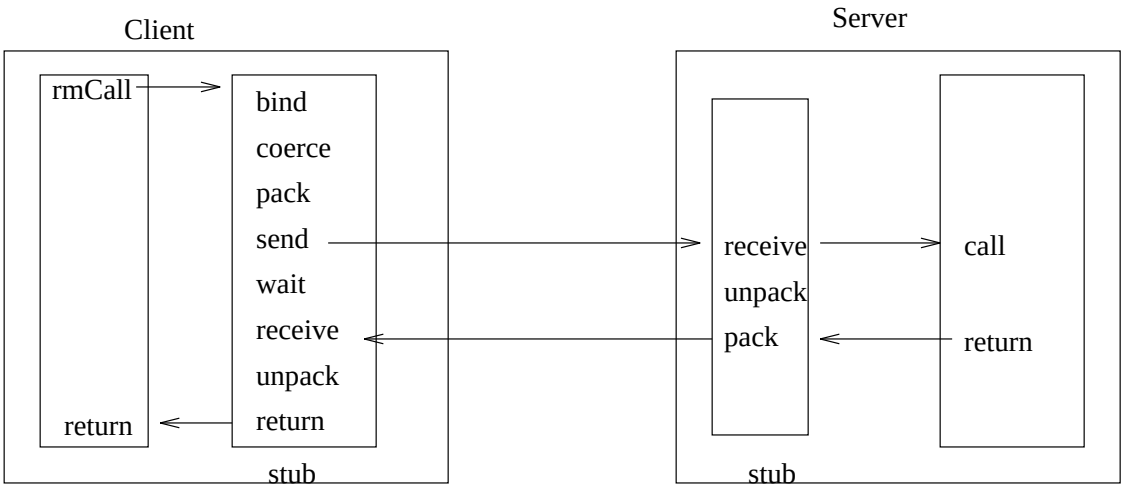
- A TP monitor is to transactions what an operating systems are to processes.

- It supports communication between different resource managers– *transactional remote procedure calls* (TRPC).
  - TRPC makes it possible for different resource managers to cooperate in the execution of transactions.

- TP monitor is responsible for:
  - system load balancing,
  - user authentication— checking user identity.
  - authorization— checking if the authenicated user has permissions to do what they want to do.

- Bringing up each resource manager after recovery from a system failures.

Transaction Remote Procedure Call



- A process makes a TRPC call similar to a local procedure call.
- The stub packs the parameters for the call.
- The RPCRuntime (shared by all processes) transmits the call to the server and receives the reply.
- The call is received by the RPCRuntime on the server.
- The server stub unpacks the parameters and the control goes to the server code.
- The server stub packs the result.

Application Writer's View of the TP System

- Application writers specify what should execute as a transaction using the BEGIN_WORK, COMMIT_WORK transaction verbs.

- A trip planning transaction.

```
BEGIN_WORK
OK = make-airline-reservation(...);
if (OK)
        OK =
make-rental-car-reservation(...);
else ABORT_WORK;
if (OK)
        OK = make-hotel-reservation(...);
else ABORT_WORK;
if (OK)
        COMMIT_WORK;
else ABORT_WORK;
```

- System ensures that all the effects of a committed transaction are persistent and the effects of an aborted transaction are undone completely.

Interaction between Components of the TP System

- Applications make service requests to resource managers using TRPCs.
- The identity of the transaction is (trid) is encoded within the TRPC.
- The resource manager, in turn, may make calls to other resource managers (e.g., lock manager, log manager) for the execution of the service request, again using TRPC.
- Before we study the structure of the TP system and how the TP monitor controls the flow of transaction we need a basic understanding of *transaction management* – that is, the role of logging, locking, and the transaction manager in implementing transactions.

Basic Transaction Management

- For simplicity, let us assume that the system consists of only a single resource manager – say an SQL database called SQLDB.
- The application programs send requests for service (possibly multiple times) to the SQLDB (using a TRPC).
- Our task is to understand what the system needs to do to ensure ACID properties of such transactions.

Concurrency Control

- Let R be a relation in SQLDB containing information about bank accounts.
- Let R be stored in a file f on disk.
- Let x and y be tuples containing information about two bank accounts in R, where x is located on page p_1 and y on page p_2 of file f.
- Consider an application AP1 that transfers \$100 from x to y.
- To execute AP1, the SQLDB fetches pages p_1 and p_2 into $database\ file\ buffers$ in main memory (if they are not already present).
- Consider another application AP2 that sums the total of balance in x and y.
- Similar to AP1, to execute AP2, the SQLDB fetches pages p_1 and p_2 into $database\ file\ buffers$ in main memory (if they are not already present).

Concurrency Control

- Let original balance in x be \$1000 and in y be \$2000.
- It is possible that the concurrent execution of AP1 and AP2 results in AP2 seeing an incorrect balance of accounts!

Locking

- To prevent incorrect interleaving of transactions (that is, to ensure isolation), resource manager acquires *locks* on behalf for transactions.
- A lock is similar to a semaphore– no two transactions can hold a lock on an object concurrently.
- After acquiring the locks on the objects, the resource manager can access/ modify the objects.
- What should we lock— *granule of locking*?
  - field in tuple, tuple, page containing tuples, a subset of tuples in a relation (possibly defined by a predicate), a relation, a set of relations, ..., database.
- How are locks implemented?
- How do locks ensure isolation? When are locks released?
- What exactly is isolation?

Logging

- Consider an application AP1 that transfers \$100 from account x to y.
- Let the initial values of x and y be x_{old} and y_{old} where:
  - $x_{old} =$ \$1000.
  - $y_{old} =$ \$2000.
- The value of x and y after execution of AP1 will be:
  - $x_{new} =$ \$900.
  - $y_{new} =$ \$2100.
- Since AP1 may decide to execute an ABORT_WORK, the system must save enough information to reconstruct the state of x and y to the state in which they were before modification—that is, the value x_{old} and y_{old}.

Logging

- One way to do so is to prevent pages p_1 and p_2 to be flushed to disk before the transaction commits ($no\text{-}steal\ buffer\ replacement\ policy$).
- Alternatively, this can be achieved by logging the $before\text{-}value$ of the objects that are modified– that is, x_{old} and y_{old}.
- These log records are called UNDO logs.
- UNDO log records satisfy the following:
  - UNDO log of $x + x_{new} = x_{old}$
  - UNDO log of $y + y_{new} = y_{old}$
- What to log— physical logs, differential logs, logical (operation) logs, physiological logs.
- How are logs implemented and stored?

What we have said so far

- As the transaction progresses, the resource manager acquires locks on objects, and writes out the before-values (UNDO logs) of the objects being modified on behalf of the transaction.

- If the transaction requests an ABORT_WORK, its before-values are installed in the database using the transactions UNDO logs.

Commit of a Transaction

- If the transaction requests a COMMIT_WORK, the resource manager must make effects (modifications) of the transaction on the database persistent (durability).

- When would effects of a transaction not be persistent?

- Assume there is a system failure and the contents of the electronic memory (main memory) are lost.

- If the effects of the transaction were not reflected on the database on the disk, a system failure will result in a loss of durability!

- $\Rightarrow$

  force all the updated objects on the disk before committing a transaction ($force$ $policy$).

- A transaction is committed only if all the updates could be forced on the disk.

- If a failure occurs before the updated objects could be forced onto the disk, use the UNDO logs to undo the effects of the transaction and consider it aborted.

Alternative to ensure durability: Redo Logging

- When an object is modified, not only log its before values (UNDO logging), but also log its *after-values* (REDO logging).
- Before the transaction commits, the REDO logs and not the actual objects are flushed to the disk.
- If a system failure occurs after the transaction commits, the effects of the transaction be reinstalled by using the REDO logs.
- Is there any advantage to forcing REDO logs instead of forcing the modified objects on the disk?

- Yes since forcing REDO logs means sequential access whereas forcing modified objects generally means random access.

Force Vrs REDO Logging?

- Disk Access Time = Seek Time + Rotational Latency + transfer size / transfer rate.
- Average seek time is approx. 10ms, 1 complete disk rotation is 10ms, Transfer rate is approx. 10MBps.
- Thus, the time to access one random disk block (1KB) is 10 + 5 + 0.1 = 15.1ms.
- Access time for 1000 random disk blocks is 15.1 seconds.
- On the other hand access time for 1000 disk blocks placed contigously on disk is 10 + 5 + 100 = 115ms.
- Sequential access on a magnetic disk far outperforms random access!
- REDO logging results in sequential disk access, whereas forcing objects on the disk may result in random access.

Force vrs REDO Logging?

- REDO logging nevertheless complicates recovery. To see why let us look at the system from the perspective of fault-tolerance.

- system availability $= \frac{MTTF}{MTTF+MTTR}$.
  - MTTF = mean time to failure.
  - MTTR = mean time to repair.

- Not forcing objects onto the disk at transaction commit time may result in the copy of the database on the disk rapidly diverging from the current state of the database (recall that TP systems may be running at 1000 tps).
- Thus, if we do not force objects onto the disk at transaction commit, as the state of the database on the disk diverges from the current database state, MTTR will keep increasing thereby reducing system availability in the case of faults.

Checkpointing

- To reduce MTTR (thereby increasing system availability) we need to explicitly *checkpoint* the database periodically. A checkpoint brings the database on disk in sync with the current database state.
- There are two strategies for checkpointing–
  - *synchronous:* stop processing transactions. Make the state of the database on the disk current. Restart transaction processing (simple).
  - *Asynchronous:* the checkpointing goes on concurrently with the transaction processing (complicated).
- Since TP systems provide online transactions, asyncronous checkpointing is preferable to synchronous checkpointing.

Write Ahead Logging

- REDO log records are made stable before the transaction commits. What about UNDO log records?

- Note that UNDO log records are used to reinstall the state before the modification of an object in case of system failure before the transaction successfully executes a COMMIT_WORK.

- $\Rightarrow$

  the UNDO log record for an object must be made stable before the modified object is written to the stable database.

- The above rule is referred to as *write ahead logging* (WAL).

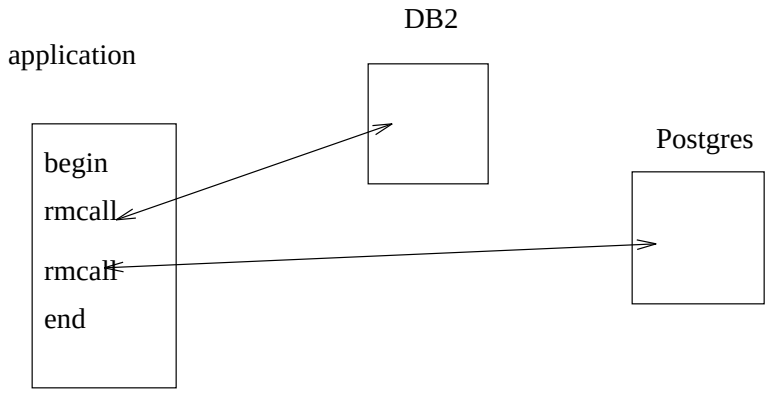
Overview of Transaction Management

- Assume the system contains only a single resource manager – say an SQL database called SQLDB.
- The application programs send requests for service (possibly multiple times) to the SQLDB (using a TRPC).
- Concurrent execution of the applications may result in loss of isolation.
- To ensure isolation, transactions acquire locks.
- Locks are held until the transaction commits or aborts.
- Issues we will study in later classes:
  - formal model of concurrency and isolation.
  - lock implementation.
  - granularity of lock– tuple, page, relation,etc.

Overview of Transaction Management with a Single Resource Manager

- So far, we have studied basic transaction management issues in case applications access a single resource manager.
  - Resource manager acquires (and release) locks for transactions.
  - Resource manager writes log records (possibly both UNDO and REDO logs) for transactions to ensure atomicity and durability in the presence of failures.
  - If an application requests a transaction abort, then UNDO logs are used to reinstate the old state of the object before the execution of the transaction.
  - If a system failure occurs,
    * REDO logs used reinstall the effects of the committed transactions.
    * UNDO logs used to remove effects of transactions that have not committed.

TM with Multiple Resource Managers



- A transaction may access multiple resource managers.
- Resource managers may be located at the same node or distributed over the network.
- Each resource manager acquires locks and writes log records for the transaction.
- Resource managers can execute ABORT_WORK asynchronously.
- Can COMMIT_WORK be executed asynchronously by the resource managers?

Executing the Commit Operation

- Atomicity of the transaction dictates that either all the resource managers commit the transaction or all abort the transaction.
- Consider a transaction T that executes at two resource managers R_1 and R_2.
- Say the system sends COMMIT_WORK to the R_1 and R_2 asynchronously.
- It is possible that R_1 aborts T
  - a failure occurs before the REDO logs of T written by R_1 have been forced to stable storage.
  - Committing T may violate the consistency of R_1.
- Transaction T may abort at R_1 and commit at R_2.
- Thus, if resource managers asynchronously execute COMMIT_WORK, loss of atomicity may result.

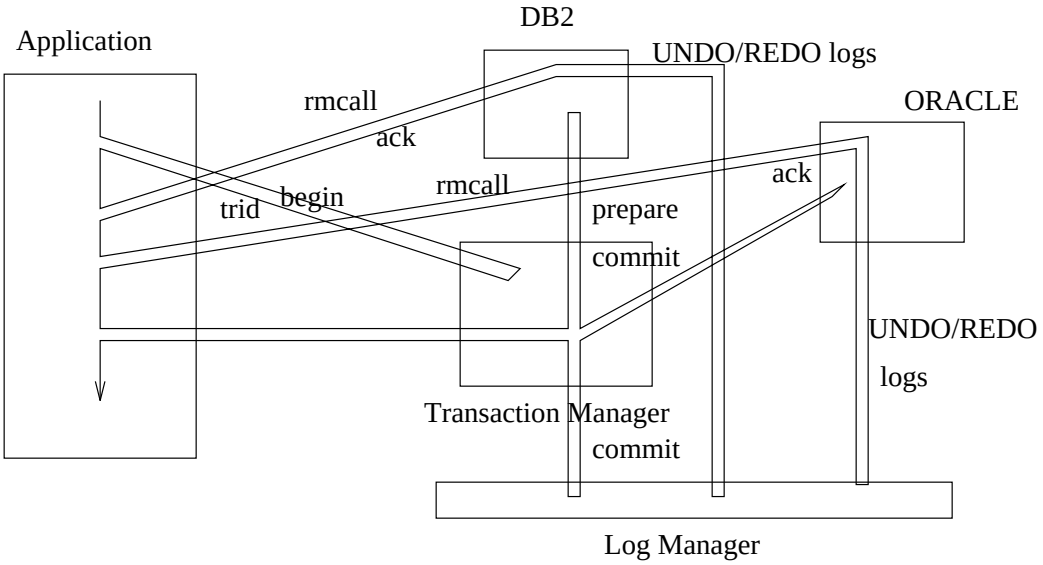
Two-Phase Commit Protocol

- To process COMMIT_WORK a two-phase commit protocol (2PC) is used.
- The transaction manager sends a $PREPARE$ request to each of the resource manager at which the transaction executes.
- Each resource manager $votes$: YES (commit) or NO (abort), and responds to the transaction manager with its vote.
- A resource manager votes YES only if its REDO logs have successfully been forced onto stable storage and all the integrity checks have been successful.
- A vote YES is a $guarantee$ by the resource manager that if asked to do so, it will surely commit the transaction.
- If each resource manager votes YES, the TM decides to commit the transaction. Else, if some resource manager votes NO, then the TM decides to abort the transaction.
- The TM sends its vote to the resource managers.
- Issues we will study later: failures during 2PC, 2PC overhead, 2PC in distributed environment.

What have we learnt so far

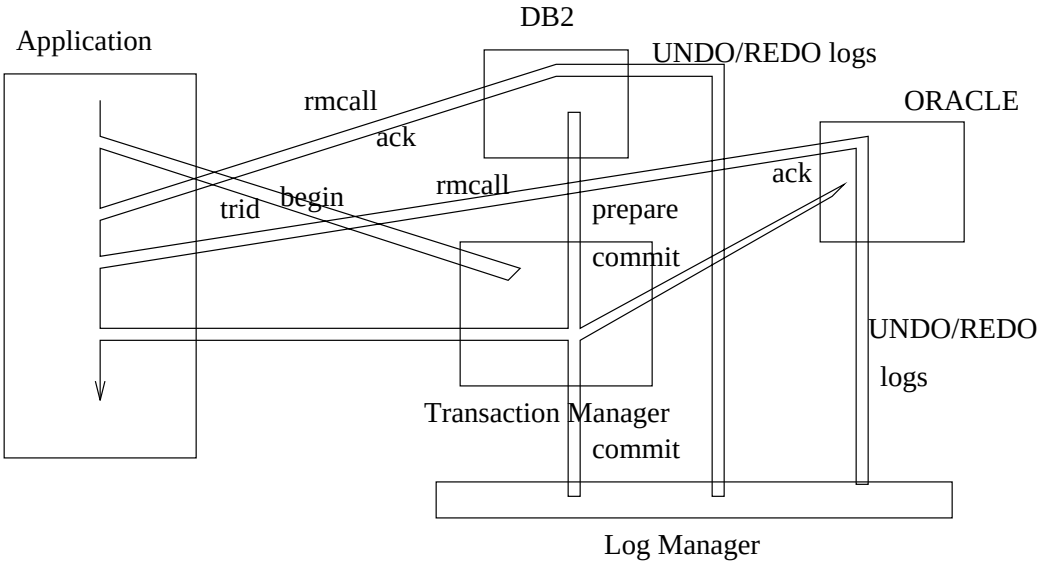
- Transactions may access data managed by multiple resource managers.
- Resource managers acquire locks and write log records (possibly both UNDO and REDO)
- To commit a transaction, a 2PC protocol is used.
- Let us now return to our original goal of understanding how a transaction flows in a TP system and how the components of the TP system interact to ensure ACID properties.

Flow of a Transaction Through the System



- Recall that an application begins a transaction by executing a BEGIN_WORK.
- A BEGIN_WORK is a TRPC to the transaction manager (TM)
- The TM responds to the BEGIN_WORK request by assigning a *transaction identifier* (TRID) which uniquely identifies the transaction.
- All future TRPCs of the transaction carry with them the transaction's TRID.

Flow of Transaction Through the System



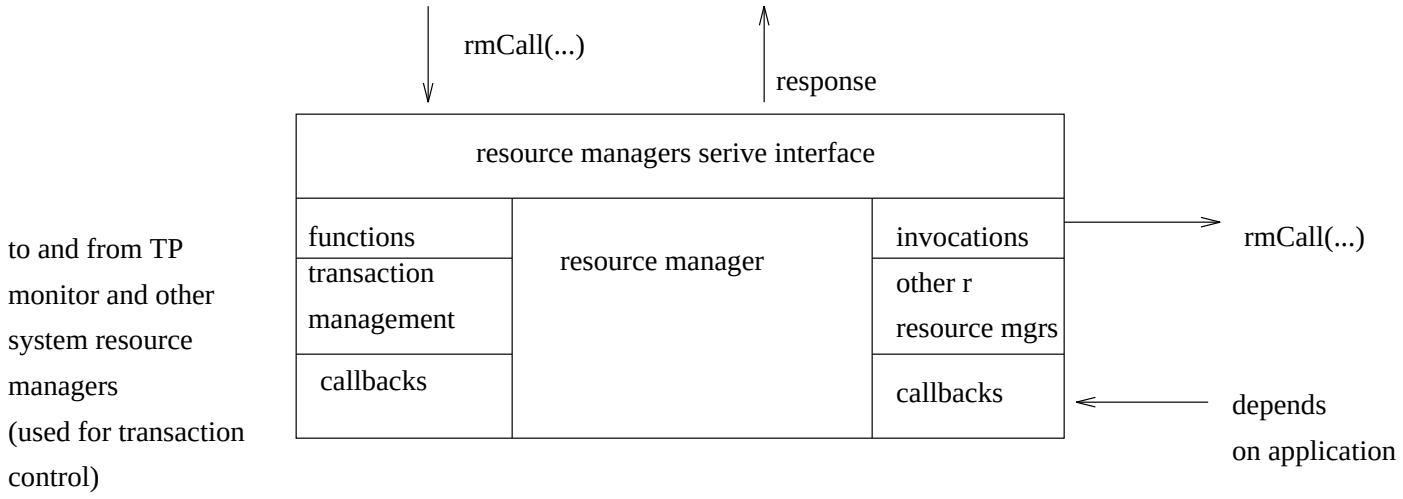
- TM keeps trace of the resource manager that execute on behalf of a transaction.
- To facilitate this, the TP monitor carrying the TRPC checks if the resource manager has already checked in with the TM for the current transaction. If not an appropriate call is made to the TM.
- A COMMIT_WORK and ABORT_WORK are TRPCs to the TM.
- On the receipt of the ABORT_WORK, the TM sends TRPC to resource manager's to execute ABORT_WORK for the transaction.
- On the receipt of the COMMIT_WORK, TM coordinates the execution of the 2PC with the resource managers.

Resource Managers

Resource Managers in a TP system can be classified into two types:

- Transactional Resource Managers: those that offer an interface for the execution of transactions.
- Simple Resource Managers: those that do not offer such an interface.

Tranasctional Resource Manager Interface



- A resource manager interface supports multiple entry points for the execution of the transactions.
  - RM_CALL(...): for the execution of regular service requests.
  - RM_UNDO, RM_REDO, RM_ABORT, RM_COMMIT, RM_PREPARE, RM_CHECKPOINT: to support transactions or implementation of ACID properties.
  - RM_INSTALL, RM_REMOVE, RM_STARTUP, RM_SHUTDOWN: for the TP monitor to manage the resource manager.

Simple Resource Managers

- Not every resource manager in the system is *transactional*– that is support an interface to participate in a 2PC, abort, checkpoint, etc.

- Simple resource manager's can participate in a transaction but should not modify any durable data– since they do not do any logging and do not participate in the 2PC.

- Thus, the ACID properties of the computations in these simple resource managers is controlled by these resource manager's themselves.

- Simple resource managers can still make calls (TRPC) to the other resource managers that control durable data.

- The TP monitor carrying the TRPC ensures that the resource managers join the transaction (by registering it to the TM) – notice the importance of associating TRID's with the TRPCs.

So far We have Seen ...

- A TP system consists of:
  - Resource managers.
  - Application managers.
  - Transaction manager, log manager, lock manager.
  - TP monitor.

- TP monitor supports TRPCs that enables different resource managers to request work from each other and to utilize the services of the log, lock and transaction managers for ensuring ACID properties of transactions.

- Each resource manager (if it is to manage durable, shared data) must support an appropriate interface.

Other Responsibilities of the TP Monitor

- Client authentication and authorization.
- Managing server classes and load balancing.
- Context Management.
- Restart, Recovery, Resource manager installation, closeup.
- Request and Response Queue management.

TP System Process Structure

- One OS Process per terminal:

  too many processes (as high as 10^5), too many file control blocks, too many process switches.

- Only one OS process for all terminals: E.g., CICS, ComPlete.

  huge process, cannot exploit multiprocessor architecture, single process may become a bottleneck, page fault will block entire system, no fault containment.

- Many servers, one scheduler: E.g., IMS/DC.

  single monitor process may become a bottleneck.

- Many servers, many schedulers: E.g., DEC ACMS, Tandem Pathway.

Server Class Management

- The TP monitor routes a service request to the resource manager.
- A resource manager is typically implemented as a collection of processes (possibly light-weight threads that share most of their address space).
- The TP monitor at the startup of the resource manager creates server processes — *server class*.
- At run time, the TP monitor assigns a server process to the service request.
- It may assign an available process, dynamically create a new process, or make the request wait in a server queue.
- The TP monitor is responsible for load balancing of the server class.

Context Management

A resource manager maintains two types of context:

- *Session Context:* the state of the resource manager within a session –e.g., authetication information, SQL cursor, value of some local variables.
- *Transaction Context:* the state of the resource manager with respect to the transaction– e.g., the locks maintained for a transaction, log records written for a transaction, value of some local variables.

Sessions

- A given client may make multiple calls to the same resource manager– either on behalf of the same transaction, or on behalf of multiple different transactions.

- Different invocations of the resource manager by a given client are independent of each other (context-free, stateless), or the work that resource manager performs depends upon the previous invocations (context-sensitive,stateful).

- If the invocations are stateful, the client and the server are said to be *in session*.

- Example of a session within a transaction: client acting within a transaction repeatedly requests the SQL database for the next record. Context in this case is the cursor.

- Example of a session that spans multiple transactions: Consider using a bank card on an ATM machine. Entry of the password corresponds to starting a session with the account manager. Multiple independent transactions (e.g., withdraw, balance inquiry) may be executed within the

session. Removing the card from the ATM corresponds to closing the session. The context in such a session includes customer's password.

Transaction Context

- A resource manager besides maintaining context related to sessions it participates in, needs also to maintain transaction context. Example of transaction context is locks, logs, other program variables.

- Recall that a resource manager may be called on behalf of the $same$ transaction multiple times by $different$ clients.

- The resource manager can maintain the transaction context since it can identify that the calls from the different clients are for the same transaction using the TRID.

- Note that session context and transaction context are different concepts.

Techniques for Context Maintainance

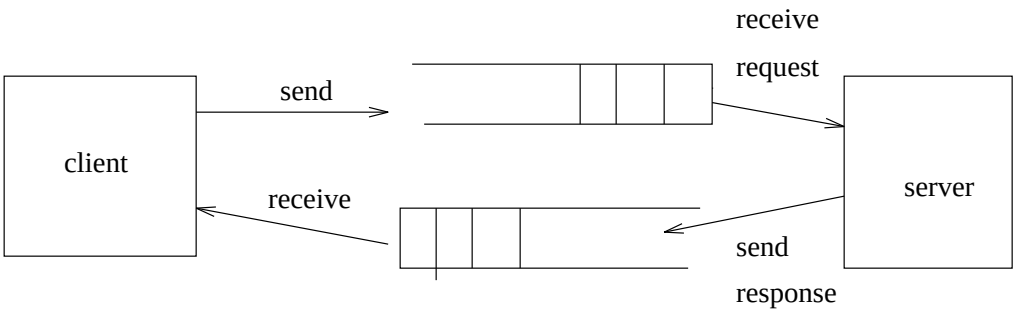
- The TP monitor maps a session (transaction) to a unique process in the server class for the entire duration of the session (transaction). This process executes exclusively on behalf of the session (transaction).

- The client and resource manager pass the relevant context back and forth explicitly with each request and reply. This scheme is applicable for managing only session context.

- Server process writes the relevant context to a *context database*. This database may be private to the resource manager or provided by the TP monitor. Before processing a request, server process reads the relevant context from the context database.

- All servers of the same server class (i.e., executing for the same resource manager) share a segment of read/write memory in which invocation context is kept. Synchronization of this shared memory is done by the servers.

Durable Queues



- Direct Transactions: in which the user interacts directly with the server programs handling the request (syncronous transaction processing).

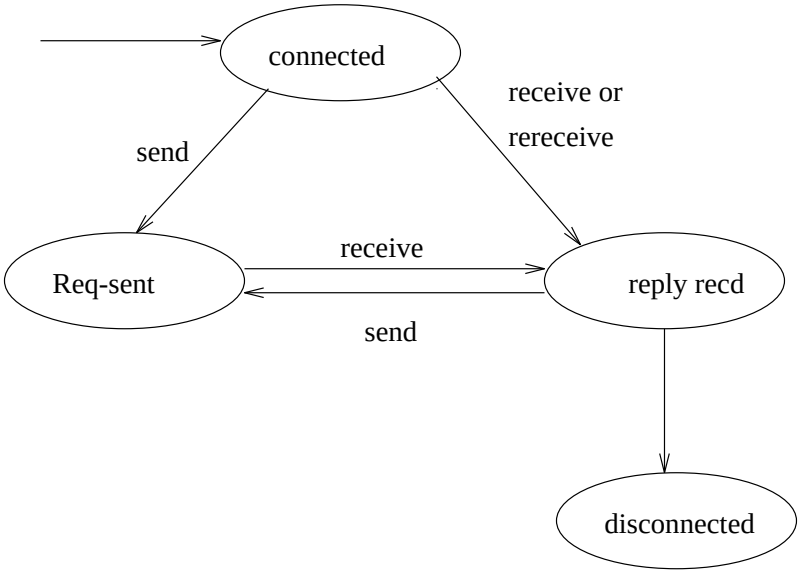
- Queued Transactions: a system provides request and response queues. The requests are put in a request queue, the system asyncronously processes the requests and puts the answer in the answer queue.

- TP system implements durable queueing service for asyncronous transaction processing.

Durable Queueing Service

- Request-Reply matching: system guarantees that for each request there is a reply.
- Exactly once Request Handling: system executes request exactly once.
- At least once Response Processing: the client is guaranteed to receive the reply at least once.

Client Model Supported by Durable Queues



Summary of Transaction Processing Systems

- Transaction: ACID computation units.
- TP systems: large systems, 1000's of terminals, gigabytes of data, 100's of processors.
- Performance requirements: 0 downtime and 1000's of transactions per second.
- TP system components: presentation and terminal services, log and lock manager, transaciton manager, TP monitor.
- TP monitor is similar to an OS:
  - monitoring and and maintainance of the TP system components.
  - Providing an environment in which components can interact for implementing transactions.
  - TRPC, context management and binding, server class management, startup and shutdown, queuing services.

TP Monitor Responsibilities

- Supports TRPC for communication between resource managers.
- Permits (possibly multivendor, heterogeneous) resource managers to participate in a transaction.
- Context Management.
- Client authentication and authorization.
- Managing server classes and load balancing.
- Restart, Recovery, Resource manager installation, closeup.
- Request and Response Queue management.

Next Few Classes

- Formal model of concurrency control in database systems.
- Concurrency control algorithms.