10 More General Transaction Management Concepts

Learning Goals:
- understanding the relationship between database systems and transaction processing systems
- understanding the concepts of transaction distribution and different levels of transactions
- understanding the importance of and the problems involved in consistency of distributed transactions as well as strategies to ensure the consistency
- understanding the issues involved in as well as strategies for management of long-term transactions and how they differ from those used for short term transactions
- being able to compare ACID and BASE and being able to evaluate which one is more appropriate in a given situation

The concepts discussed in the previous chapters apply to the traditional type of transactions: with
- simple data types,
- a central database system and
- short duration of transactions (seconds to minutes).

One can also call them "flat" transactions (in the sense that they are not hierarchically nested).

10.1 Nested Transactions

In the case of more complex tasks, a transaction can be subdivided into several subtasks that can run in parallel, possibly in a distributed environment. The transaction then contains subtransactions which in turn can contain other subtransactions => a tree of transactions.
The different subtransactions may run
- within the same DBMS on the same hardware (same CPU, with one or more kernels)
- within the same DBMS on different local hardware (e.g. one server, several clients)
- within the same DBMS on geographically distributed hardware (e.g. with a distributed DBMS)
- within different DBMSs on distributed hardware, coordinated by a system that connects to these DBMSs

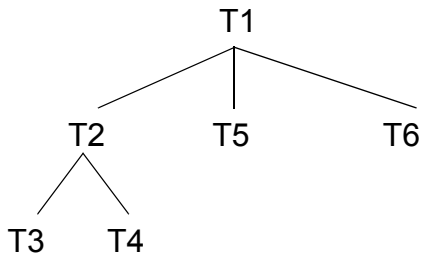
The distribution may be done for the purposes of
- splitting a complex task into more manageable smaller subtasks
- running a problem on parallel hardware to speed up the performance
- managing a transaction that requires access to different DBMSs

Example:
For booking a trip, someone wants to book a flight from Stuttgart to Islamabad, a hotel in Islamabad, and a rented car in Islamabad. One could perform this as a set of independent transactions or as one transaction with child transactions.

Pseudo code:

```
begin_transaction T1                        //Complete reservation

   begin_transaction T2                     //Airline reservation

      begin_transaction T3                        //first flight
         reserveAirlineSeat(Stuttgart, Dubai)
      commit T3;

      begin_transaction T4                     // connecting flight
         reserveAirlineSeat(Dubai, Islamabad)
      commit T4;

   commit T2;

   begin_transaction T5                     // hotel reservation
      bookHotel(Sheraton)
   commit T5;

   begin_transaction T6                     // reserve car
      bookCar(Hertz)
   commit T6;

commit T1;
```

Nesting tree:



Features of nested transactions

- Parent transactions cannot commit while child transactions are still open (i.e. before child transactions are ready to commit or abort).
- Child transactions have to abort when the parent aborts. They are only allowed to commit when the parent tells them to commit.

Abort of a lower-level transaction does not always necessitate abort at a higher level: A parent transaction may decide to use instead additional ways of recovery, for

instance:
- retry the subtransaction
- ignore it (if it is not absolutely necessary for the success of the transaction). For instance, the booking of a rental car may not require an abort of the trip booking if it fails.
- run an alternative subtransaction (for instance try to book the car with a different rental company).

Updates of subtransactions are only visible within the scope of their direct parent transactions (results of T3 may be used by T4 but not higher in the hierarchy and not outside of this transaction tree). => Necessary for the isolation property.
Example: T4 may use the knowledge that T3 could in fact acquire a seat on a given plane from Stuttgart to Dubai and thus also acquire a seat from Dubai to Islamabad. If no seat from Stuttgart to Dubai had been available, T4 might have tried a different flight connection.

=> The top level transaction conforms to the ACID properties. The subtransactions do not necessarily conform to ACID, for instance they may not provide isolation.

The following example uses the Java Transaction API (JTA), which contains interfaces relevant to transaction management, for instance javax.transaction.Transaction. It provides methods like begin(), commit(), rollback().

Parent transaction:

```
void transfer_all_salaries() {
      Transaction tx = new Transaction();
      tx.begin();
      for(int i=0; i<emplCount; i++)
          transfer_salary(employerAccount,employeeAccount[i],
                          amount[i]);
      tx.commit();
}
```

Child transaction:

```
void transfer_salary(from, to, amount) {
      Transaction tx = new Transaction();
      tx.begin();
      from.remove(amount);
      to.add(amount);
      tx.commit();
}
```

Some notes about implementations:

- SQL as such does not support nested transactions.

- Oracle, Microsoft Transaction Server, DB2 support nested transactions.

- CORBA (Common Object Request Broker Architecture) is a specification for object-oriented middleware that is being specified by OMG (Object Management Group). It provides a number of services, one of which is called the Transaction Service (OTS). It specifies the transaction management in a distributed situation.

- The Java Transaction Service (JTS) specifies the implementation of a Java transaction manager. It follows the OMG Transaction Service (OTS) which is part of CORBA.
  An application that implements nested transactions in Java will use the Java Transaction API (JTA) for this purpose. The JTA methods will internally call the lower-level JTS routines.
  JTA follows the XA standard for distributed transactions (developed by the Open Group, XA ="Extended Architecture") which uses the 2 phase commit protocol for synchronization of the transaction outcome.
  For application programming, often an ORM (object-relational mapping) framework is used which has its own functions that then use the JTA methods, so that the programmers don‘t need to use the JTA methods directly. Examples of ORMs are Hibernate, Spring, TopLink.

- MySQL currently implements the resource manager side of the XA protocol for distributed transactions. That means, you cannot have a nested transaction inside one MySQL database. But in a MySQL database, a subtransaction can run that is managed by an outside transaction manager that also implements the XA protocol. In other words: You can use MySQL as a resource manager (meaning here: a local DBMS) but not as a transaction manager (meaning here: the global transaction coordinator).

Quote from the MySQL documentation:

"Applications that use global transactions involve one or more Resource Managers and a Transaction Manager:

A Resource Manager (RM) provides access to transactional resources. A database server is one kind of resource manager. It must be possible to either commit or roll back transactions managed by the RM.

A Transaction Manager (TM) coordinates the transactions that are part of a global transaction. It communicates with the RMs that handle each of these transactions. The individual transactions within a global transaction are “branches” of the global transaction. Global transactions and their branches are identified by a naming scheme described later.

The MySQL implementation of XA MySQL enables a MySQL server to act as a Resource Manager that handles XA transactions within a global transaction. A client program that connects to the MySQL server acts as the Transaction Manager."

Reasons for using the nested transaction model:
- Modularity (a complex task can be considered a single unit, but every subtask can

use (to a certain degree) independent concurrency control and recovery).
- Subtransactions can execute concurrently => This may deliver better performance.
- A transaction may require partial execution in different databases.

10.2 Consistency in Distributed Transactions

Sometimes, a transaction may be distributed over a number of different databases, either managed by a single DBMS (i.e. a) or by different DBMSs. This means that the transaction is composed of several subtransactions (like the nested model discussed above) that are executed at the different databases. The ACID properties should be enforced in this distributed environment as well.

This requires a somewhat different commit and recovery protocol:
The subtransactions cannot commit independently but have to wait for a signal from the parent transaction. The "Two Phase Commit Protocol" (2PC) can for instance be used (see below). It is designed for a distributed situation but can also be employed at a single site (with different processes for the different subtransactions).

Example:
A good example is again the earlier money transfer example. If the two bank accounts are not within the same bank, two different databases are involved.
A coordinating system, for instance a TP monitor, sends messages to both systems, ensuring the ACID properties of the global transaction.

2 Phase Commit Protocol (2PC)

This protocol is executed by a coordinating site and a number of participating sites where the subtransactions are executed. The coordinating site keeps a global log and coordinates the global transaction. The participants keep their local logs and can commit or roll back independently (but in conformance with the global outcome).

The idea is that either all participants commit or none of them do. A failure during phase 1 means that the entire transaction will be rolled back, whereas a failure in phase 2 usually means that a successful transaction will be recovered and committed.

It is assumed that the begin of the transaction has taken place already and the participants are busy executing their parts. The 2PC describes only what must be done for committing.

Point of view of the coordinator:

Phase 1 (voting phase):

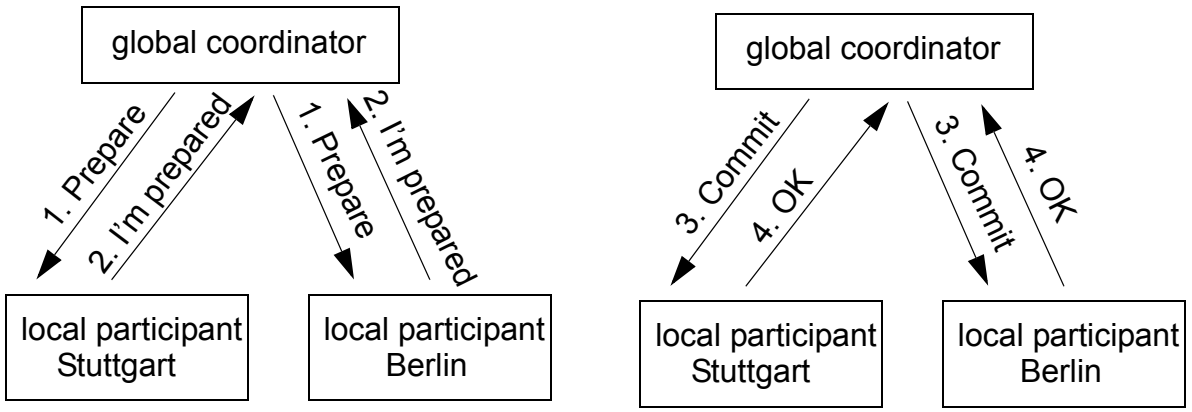
The global coordinator
- writes a BEGIN COMMIT record into the log file and force-writes it to disk,
- sends a PREPARE message to all participants,
- and waits for the participants to reply within a timeout period.

Phase 2 (decision phase):

If the global coordinator receives an ABORT vote from some participant or if one or more participants do not respond within the timeout interval, it
- writes an ABORT record to the log and force-writes it to disk,
- sends a GLOBAL ABORT message to all participants, and
- waits for the participants to acknowledge within a timeout period.

If the global coordinator receives a READY COMMIT message from a participant, it
- updates the list of participants who have responded.
- If all participants have responded within the timeout interval and all voted for commit, it sends a GLOBAL COMMIT message to all participants,
- writes a COMMIT record to the log,
- and waits for the participants to acknowledge within a timeout interval.

After all acknowledgement messages have been received, the global coordinator
- writes an END TRANSACTION record to the log.
- If no acknowledgement is received from some site, the global decision message is resent again and again to this site until the acknowledgement is received.



Point of view of a participant:

The participant has run the transaction and now makes a decision to abort or commit.

Phase 1:

When the participant receives a PREPARE message, it
- writes a READY COMMIT record to the local log, force-writes all log buffers for the transaction to disk, and sends a READY COMMIT message to the coordinator. It waits for the coordinator's next message before being able to proceed with this transaction.
- or: it writes an ABORT record to the local log, force-writes it to disk, sends an ABORT message to the coordinator, and aborts the local transaction.

Phase 2:

- If the participant receives a GLOBAL ABORT message, it writes an ABORT record to its local log and force-writes it to disk. It aborts the transaction and afterwards sends an acknowledgement to the coordinator.
- If the participant receives a GLOBAL COMMIT message from the coordinator, it force-writes a COMMIT record in the log on disk. It commits the local transaction and sends an acknowledgement to the coordinator.

Problem with 2PC:

If a site has signalled ready to commit and then is separated in the network from the global site that issues global commit or abort, it is blocked, i.e. it cannot proceed with the transaction.

A solution is to communicate with another local site that knows what the global decision is. If this is not possible either, the site must wait until the network connection is established again.
To solve the blocking problem, a three phase commit protocol (3PC) has been designed, but even that does not solve all problems.

2PC is more widely used than 3PC.

In recent years, with the explosive growth of widely distributed systems (like for example social networks), these problems have become more urgent.

In 2002, Eric Brewer‘s (from UC Berkeley) CAP Theorem was proven by Gilbert and Lynch (before, it was only an assumption).

CAP stands for:
- ***C***onsistency
  (meaning all network sites see the same data at the same time)
- ***A***vailability
  (meaning every network site receives a response about the success of its requests within a reasonable time)
- ***P***artition Tolerance
  (meaning the global system can continue its work even if some of the network sites are separated from the rest and cannot communicate).

Brewer‘s Theorem states that a distributed computer system cannot guarantee all three of these features at the same time, but only two.

Thus, the following combinations are feasible:
1. The outcome is consistent and available, but not partition tolerant:
   Practical meaning:
   The different sites agree on the same outcome, and this outcome is available right at the end of the transaction to all sites. However, this works only if the network is not partitioned, i.e. has no failures in any links. A network partitioning therefore will

disrupt either availability or the consistency.
-> This is the situation like in 2PC above. It works well, unless network partitioning happens.

2. The outcome is consistent and partition tolerant, but not necessarily available in a reasonable time period.
   Practical meaning:
   This happens for instance with 2PC when a site has signalled ready to commit and then is separated from the network. It is then suspended until the network comes up again.
   -> Hence 2PC can handle the network partitioning and guarantee a globally consistent outcome, but it may take too long for practical applications. 3PC solves the problem for some, but not all situations.
3. The outcome is available and partition tolerant, but not necessarily consistent.
   Practical meaning:
   At each local site, a transaction outcome can be seen, even if there is a network partitioning. However, the different sites do not agree on the outcome. In this case, one should hope that the inconsistent state is only temporary and consistency is achieved at some time in the future.

Replacing ACID with BASE

Case 3 is the place where the NoSQL systems step in with a new approach, replacing the ACID requirement. It is called:

- ***B***asically ***A***vailable
  At all times, data is locally available, but not necessarily all of the data.
- ***S***oft State
  The definition of this part of the strategy seems a little fuzzy. Different sources do not agree on how to define it. Here are some definition examples:
  - Data needs to be refreshed periodically, or it will be automatically deleted or at least marked as expired.
  -> However, NoSQL systems do not typically implement such a strategy.
  - The status of data at the local sites may change over time, even without local user intervention. This is because of the eventual consistency rule, which may cause the system to update some values (even a long time after the original transaction).
  -> This version of the definition seems somewhat redundant with the eventual consistency rule.
- ***E***ventually Consistent
  When accessed at a random time, not all of the different sites involved in a distributed transaction may have a consistent (i.e. the same) outcome, but at some unknown point of time in the future, consistency (i.e. agreement about the data values) will definitely have been achieved.

This approach is known by the acronym built from its first letters: **BASE**.

On the internet, you can find a quote supposedly from Eric Brewer himself that admits a certain redundancy in the acronym definition (http://www.julianbrowne.com/article/viewer/brewers-cap-theorem):
„The term "BASE" was first presented in the 1997 SOSP article that you cite. I came

up with the acronym with my students in their office earlier that year. I agree it is contrived a bit, but so is "ACID" -- much more than people realize, so we figured it was good enough. Jim Gray and I discussed these acronyms and he readily admitted that ACID was a stretch too -- the A and D have high overlap and the C is ill-defined at best. But the pair connotes the idea of a spectrum, which is one of the points of the PODC[1] lecture as you correctly point out."

Two articles discussing aspects of ACID, CAP, and BASE can be found in the Moodle:
- Abadi Thomson: DBMS Musings
- Julian Browne: Brewer‘s CAP Theorem

Browne explains nicely how the CAP theorem becomes important in situations where high scalability is an issue:
„At low transactional volumes, small latencies to allow databases to get consistent has no noticeable affect on either overall performance or the user experience. Any load distribution you do undertake, therefore, is likely to be for systems management reasons. But as activity increases, these pinch-points in throughput will begin limit growth and create errors. It's one thing having to wait for a web page to come back with a response and another experience altogether to enter your credit card details to be met with "HTTP 500 java.lang.schrodinger.purchasingerror"[2] and wonder whether you've just paid for something you won't get, not paid at all, or maybe the error is immaterial to this transaction. Who knows? You are unlikely to continue, more likely to shop elsewhere, and very likely to phone your bank.
Either way this is not good for business. Amazon claim that just an extra one tenth of a second on their response times will cost them 1% in sales. Google said they noticed that just a half a second increase in latency caused traffic to drop by a fifth."

More approaches to relax the requirements of ACID in transactions are described later in this chapter.

10.3 Transaction Processing Systems (TP Systems)

Distributed transaction processing is not only done within a DBMS context, although that is frequently the case. File systems with an understanding of transactions can be used as well.

Definition by Gray and Reuter:
"A transaction processing monitor (TP monitor) is a software module of a transaction processing system (TP system) whose function it is to integrate other system components (like clients, servers, database systems, compilers, runtime systems, operating systems, file systems) and make them work together in a transaction-oriented way. It integrates the different system components to provide a uniform applications and ope-

1. PODC = Symposium on Principles of Distributed Computing
2. I think this is a joke. There is no such schrodinger package; it refers to Erwin Schrödinger and his famous hypothetical cat that was neither dead nor alive.

rations interface with the ACID properties."

A TP monitor can provide transactional access to various commercial DBMSs.
TP monitors are middleware components. They are typically not standalone systems but integrated in architectures like J2EE.

Typical functions are:
- message queuing (asynchronous communication between two applications; messages are stored until they can be delivered or fetched. Messages transferred within a transaction context are for instance start transaction, commit, etc.)
- lock management
- log management
- 2PC synchronization
- rollback
- load balancing

A good discussion of TP systems and TP monitors can be found in
- Gray/Reuter "Transaction Processing - Concepts and Techniques" and in
- Gerhard Weikum, Gottfried Vossen: „Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery," Morgan Kaufmann

Examples of TP monitors:
- IMS (Information Management System) by IBM
- CICS (Customer Information Control System) by IBM
- Oracle Tuxedo (formerly by BEA which was taken over by Oracle)
- Microsoft Transaction Server
- Fujitsu Siemens UTM (Universal Transaction Monitor)
- Guardian 90 by Tandem
- DECdta (distributed transaction processing architecture) by the former Digital Equipment Corporation (DEC) which was bought by Compaq which was bought by HP
- Encina by Transarc Corporation
- ACMS (Application Control and Management System by DEC)

The earlier systems (like IMS, CICS) were single monolithic systems used in proprietary mainframe environments.

More modern systems (e.g. Encina, ACMS, CICS/6000) are more modular and constructed from open transaction middleware.
They contain middleware modules providing the basic functional building blocks for transaction processing: a Transaction Manager, a Lock Manager, a Log Manager and a Resource Manager (e.g. a DBMS). Each module provides an API for its transaction services.

X/Open[1] DTP (Distributed Transaction Processing) is a standard to which most sys-

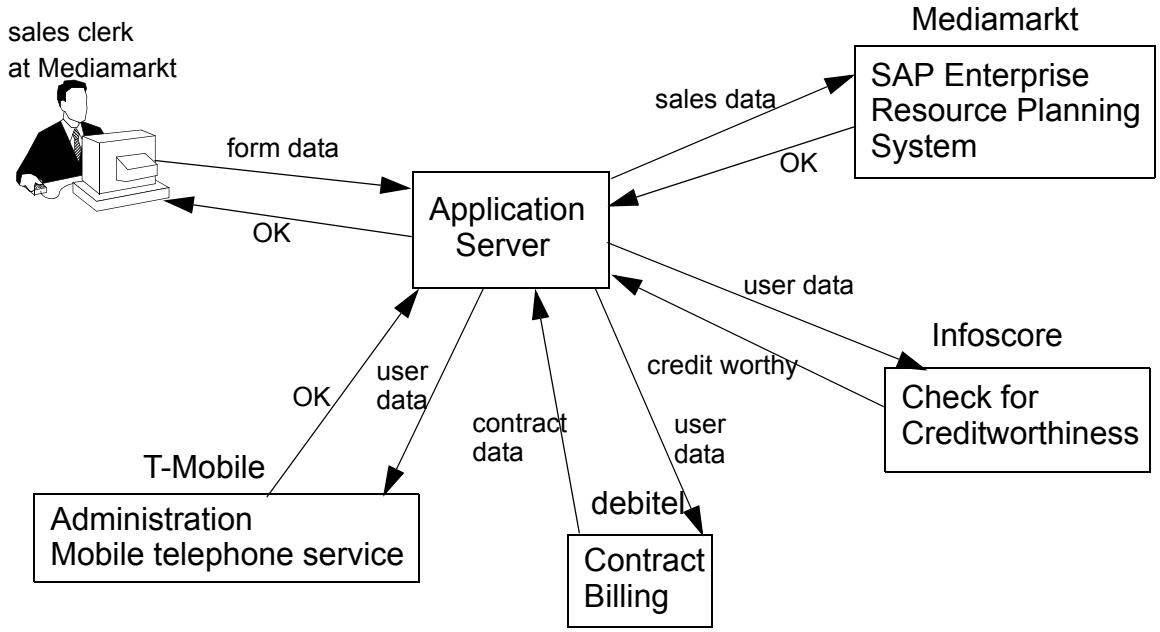
1. The Open Group is a vendor- and technology-neutral consortium, whose vision of Boundaryless Information Flow™ will enable access to integrated information within and between enterprises based on open standards and global interoperability.

tems adhere.
Its model defines three components:
- application programs,
- resource managers,
- and a transaction manager.

This model also specifies functional interfaces between application programs and the transaction manager (known as the ***TX*** interface), and between the transaction manager and the resource managers (the ***XA*** interface). With products complying to these interfaces, one can implement transactions with the two phase commit protocol (2PC).

Example of a distributed transaction situation:

Buying a mobile phone in MediaMarkt (a German retailer for electronic devices) may involve several steps like in the following example:
- Buying the device in the shop (using for instance an enterprise resource planning (ERP) system, like SAP)
- Testing for customer rating (financial credibility) (Arvato Infoscore is a company providing customer ratings, using for instance geocoding)
- Contract with debitel (German Telecom company that has no own telephone carrier but just provides services, like special billing modes)
- Registration with T-Mobile; this is a provider of a carrier (part of German Telecom). This company must enable the usage of the network for the new phone customer.
- Enabling customer access on the debitel portal.

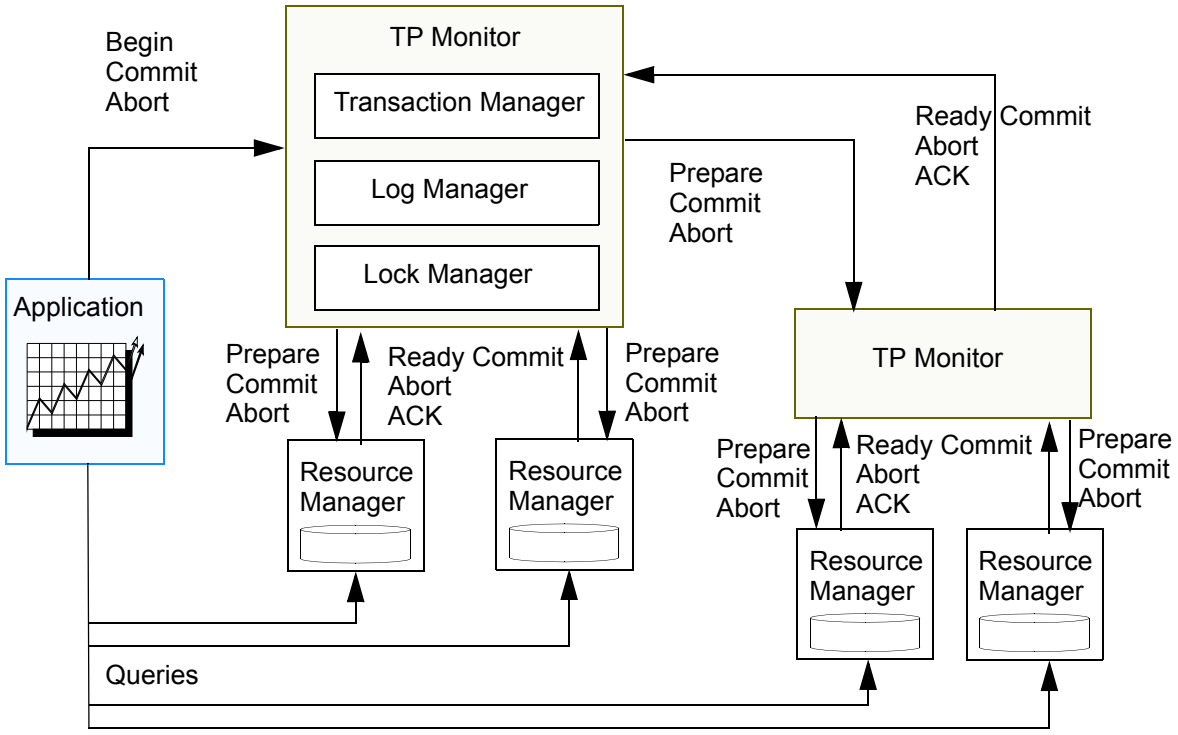


The application server could for instance be BEA Weblogic Server or IBM Websphere Application Server.

One function of the application server is to coordinate the distributed transaction. For this purpose it uses a TP monitor. In the case of Weblogic it would be typically Tuxedo (also a BEA product, now owned by Oracle). Websphere is a set of products that also contains TP functionality (using Websphere MQ (formerly known as MQSeries)

among others).

Typical architecture of a distributed TP system:



Resource manager means typically a DBMS. (It could be a file system as well).

A short excursus about application servers and how they support transactions:

Purpose of application servers: provide a tier for handling the application logic.
Typical products:

- J2EE type application servers: BEA WebLogic, IBM WebSphere, Oracle AS, JBoss, IONA, Sun ONE, Sybase EAServer
- not J2EE: Microsoft .NET

J2EE (Java 2 Enterprise Edition) is a standard for component-based architectures for applications programmed in Java. The components are modules for building the business logic of an application.
Example: In a system for a travel agency, one component could be a ticket reservation function for a hotel room. Another component could make reservations for car rentals.

An application server is an environment for executing application components. There are typically two types of components in J2EE conforming application servers:

- front-end-oriented: servlets or JSPs (Java Server Pages)
- back-end-oriented, business logic: EJBs (Enterprise Java Beans) or normal Java classes

An Object Transaction Monitor is a TP Monitor in a distributed, object-oriented environment.

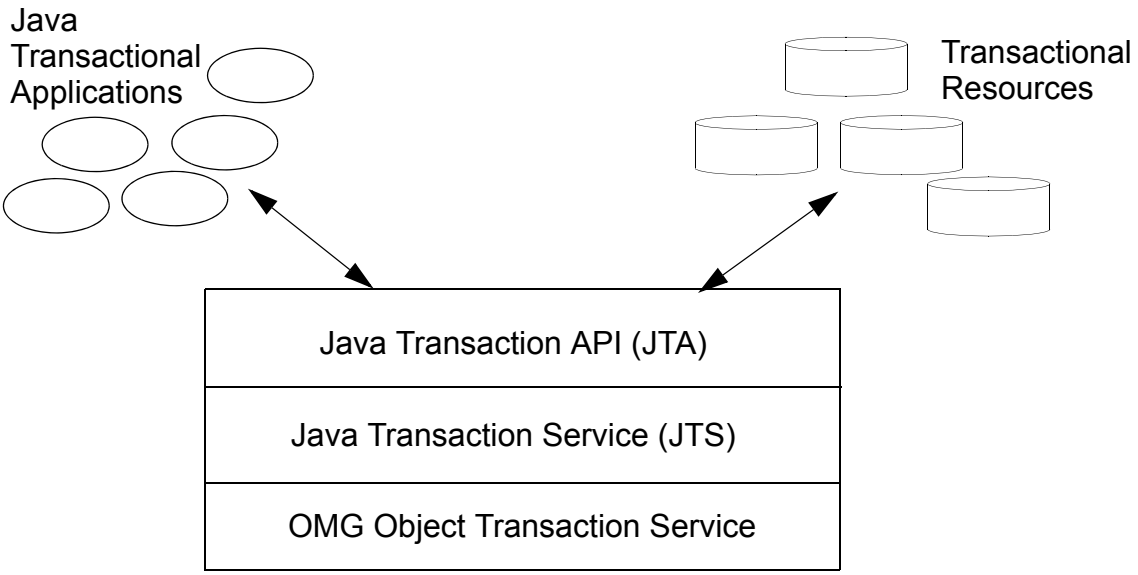
Examples of OTMs:

- CORBA ORB + OTS (ORB = Object Request Broker, OTS = Object Transaction Service), OTS is a standard based on X/Open DTP
- Java Transaction Service (JTS) (an implementation of OTS)
- Microsoft COM+ (DCOM (Distributed Component Object Model) + MTS (Microsoft Transaction Server))

J2EE architectures typically use JTS or directly CORBA.

In J2EE transactions are supported by

- JTA (Java Transaction API, which allows the definition of transaction boundaries). JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications. It includes the packages javax.transaction and javax.transaction.xa. The interfaces Transaction and User-Transaction contain methods such as begin(), commit(), and rollback(). JTA supports only flat (ACID) transactions, no sagas or other nested concepts. OTS, however, supports also nested transactions.
- JTS (Java Transaction Service), which is an implementation of the CORBA OTS standard. JTS specifies the implementation of a Transaction Manager which supports the Java Transaction API specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) specification at the low-level. JTS uses the standard CORBA ORB/OTS interfaces and Internet Inter-ORB Protocol (IIOP) for transaction context propagation between JTS Transaction Managers.
  A JTS Transaction Manager provides transaction services to the parties involved in distributed transactions: the application server, the resource manager, the standalone transactional application, and the Communication Resource Manager (CRM).



A good overview of these concepts can be found at:
http://www.subrahmanyam.com/articles/jts/JTS.html

A printout of this is in the Moodle.

10.4 Problems with Flat ACID Transactions

The typical ACID transactions are also called **"flat transactions"**, as opposed to nested or otherwise structured transactions.

The flat transaction concept as defined so far is fine for the majority of transactions. These are typically of small complexity and of short duration.

In special situations, the ACID features can be problematic, however:

Problems with complex transactions

"All or Nothing" may be too restrictive

- Optional parts
  There may be optional or alternative parts
- Rollback not always sufficient for recovery
  Exception handling (i.e. recovery) may have to be done in part by the application itself because the recovery requires semantic knowledge of the application (example: recover the unavailable rental car by booking a car with a different company).

These problems suggest to *relax the atomicity property* and break the complex action into different, independent parts that will be managed globally.

Problems with long transactions

- Blocking of resources:
  Locking the whole transaction means that other transactions which need the same data cannot access it until the end of this transaction which may take days or even weeks.
- High failure probability:
  During the long execution time a failure is more likely. But aborting the whole transaction and losing a significant amount of work may not be acceptable.
- High deadlock probability:
  The concurrency control protocol of locking is more prone to deadlocks the longer the transaction runs.

These problems suggest trying to *relax the isolation property* and break the transaction up into smaller parts that may publish their results earlier.

10.5 Transaction Savepoints

Savepoints are a way of setting a mark within a transaction and to explicitly rollback only to this point.
=> If something goes wrong later in the transaction that has to be undone, the work up

to the savepoint does not have to be redone; the rollback does recovery only for the later stages.
=> In a longer transaction, rollback may not have to lose quite so much work.

Setting savepoints is part of the SQL standard.

Example:

```
INSERT INTO Student (matNr, sName) values (5000,'Koch');
SAVEPOINT koch;
INSERT INTO Student (matNr, sName) values (6000,'Becker');
SAVEPOINT becker;
INSERT INTO Student (matNr, sName) values (7000,'Wang');
SAVEPOINT wang;
ROLLBACK TO koch
-- This performs a rollback to savepoint koch. => The first INSERT is still maintained.
COMMIT;
-- Now the first INSERT is committed.
```

Note:
Even when savepoints are set, a complete rollback is still possible.

10.6 Long transactions and Sagas

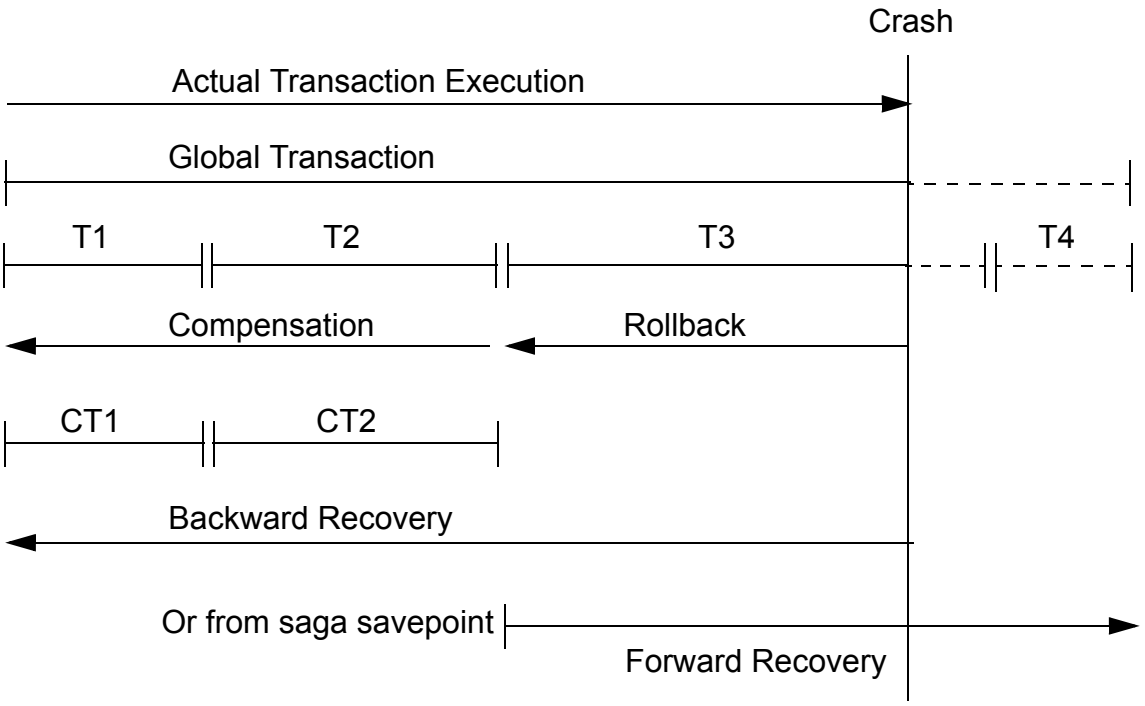
In some application domains, transactions are typically of longer duration, for instance in engineering where developing a new product means that the database is for a long time in an inconsistent state until the product designers release a new consistent version of a product under development.

One approach of dealing with the problem of long transactions is the concept of sagas which was introduced by Hector Garcia-Molina and Kenneth Salem who defined it as "a sequence of transactions that can be interleaved with other transactions".

Interleaving means: some steps of saga #1 may be executed (e.g. committed transactions), then one or more steps from saga #2 are executed, then saga #1 performs some steps again, etc. This can also be done with more than two sagas. The interleaving can mean that the same data is used by the interleaving sagas!

Isolation is provided only within each transaction. The saga as a whole does not provide isolation, since each transaction inside the saga publishes its results after committing.

A saga can be considered a "meta-transaction" in the way that the DBMS guarantees atomicity of the saga. Recovery in the case of failure of a saga is done by providing compensating transactions to undo effects of committed transactions.



By defining a transaction as a saga, a transaction of long duration can be broken up into separate steps using their own concurrency control and recovery and which can be interleaved with other transactions so that a long transaction does not have to mean that other transactions cannot access the locked data.

Sagas are reasonably applicable only if
- the internal transactions are independent from each other
- compensating transactions can be created (example: for withdrawing money from a teller machine, there is no compensating transaction).

Note:
A saga can also be considered a nested transaction, with the difference that the basic concept of a nested transaction enforces isolation (and is therefore called a "closed nested transaction"), whereas sagas do not and are therefore also called "open nested transactions". A saga contains only one level of nesting, however.

10.7 Extended Nested Transaction Models

A nested transaction can have the following properties:

- open or ***closed*** (open means: parts may be published; closed means: full isolation)
- compensatable or ***non-compensatable***
- independent or ***dependent*** of the parent transaction (subtransactions are dependent on the parent transaction: must commit or rollback as the parent does)
- **optional** or *necessary*
- **substitutable** or *non-substitutable* (substitutable: if a subtransaction does not

succeed, it can be replaced by a different subtransaction)

In theory, any combination of the above properties is thinkable. In practice, only some combinations are in use:

- **Classic definition of a nested transaction:** The combination of **boldface** properties in the above list. It is the classic definition of a nested transaction as used in an earlier subsection of these class notes: closed, non-compensatable, dependent, optional, substitutable.
- *Simplest version of a nested transaction:* The combination of *italicized* properties in the above list: closed, noncompensatable, dependent, necessary, non-substitutable.
- Saga: The combination of underlined properties in the above list: open, compensatable, dependent, necessary, non-substitutable.

The general advantages of structuring nested transactions are
- modularity of subtasks (parallel and/or distributed)
- flexibility, semantic exception handling in the application logic