XIII. Distributed Transactions

Distributed Transactions
Classification of Transactions
2PC, 4PC, and 3PC Protocols
Interoperability

Classification of Transactions

- ***Remote requests:*** read-only transactions made up of an arbitrary number of SQL queries, addressed to a single remote DBMS (remote DBMS can only be queried)
- ***Remote transactions*** made up of any number of SQL commands (select, insert, delete, update) directed to a single remote DBMS (each transaction writes on one DBMS.)
- ***Distributed transactions*** made up of any number of SQL commands (select, insert, delete, update) directed to an arbitrary number of remote DBMSs, but each SQL command refers to a single DBMS (Transactions may update more than one DBMS, require the two-phase commit protocol)
- ***Distributed requests*** are arbitrary transactions, in which each SQL command can refer to any DBMS; assumes a distributed optimizer

Typical Transaction: Fund Transfer

- Assume: ACCOUNT (AccNum,Name,Tl) with accounts lower than 10000 allocated on fragment ACCOUNT1 and accounts above 10000 allocated on fragment ACCOUNT2

```
begin transaction
update Account1
  set Tl = Tl – 100000 where AccNum = 3154;
update Account2
  set Tl = Tl + 100000 where AccNum = 14878;
commit work;
end transaction
```

- Note: It is generally an unacceptable violation of atomicity that one of the modifications is executed while the other is not.

Technology of Distributed Databases

- Data distribution does not influence consistency and durability
  - ✓ ***Consistency*** of transactions does not depend on data distribution, because integrity constraints describe only local properties (a limit of the actual DBMS technology)
  - ✓ ***Durability*** is not a problem that depends on the data distribution, because each system guarantees local durability by using local recovery mechanisms (logs, checkpoints, and dumps)
- Other subsystems require major enhancements:
  - ✓ Query optimization
  - ✓ Concurrency control
  - ✓ Reliability control

Distributed Query Optimization

- Required when a DBMS receives a distributed request; the DBMS that is queried is responsible for the 'global optimization'
  - ✓ It decides on the breakdown of the query into many sub-queries, each addressed to a specific DBMS
  - ✓ It builds a strategy (plan) of distributed execution: consisting of the coordinated execution of various programs on various DBMSs and in the exchange of data among them
- The cost factors of a distributed query include the quantity of data transmitted on the network

  $C_{total} = C_{I/O} \text{ x } n_{I/O} + C_{cpu} \text{ x } n_{cpu} + C_{tr} \text{ x } n_{tr}$

  n_{tr}: the quantity of data transmitted on the network

  C_{tr}: unit cost of transmission

Concurrency Control

- In a distributed system, a transaction t_i can carry out various sub-transactions t_{ij} , where the second subscript denotes the node of the system on which the sub-transaction works.

  $t_1 : r_{11}(x)\ w_{11}(x)\ r_{12}(y)\ w_{12}(y)$

  $t_2 : r_{22}(y)\ w_{22}(y)\ r_{21}(x)\ w_{21}(x)$
- Local serializability is not a sufficient guarantee of serializability. Consider schedules at nodes 1 and 2:

  $S_1 : r_{11}(x)\ w_{11}(x)\ r_{21}(x)\ w_{21}(x)$

  $S_2 : r_{22}(y)\ w_{22}(y)\ r_{12}(y)\ w_{12}(y)$
- They are locally serializable, but global conflict graph has a cycle:
  - ✓ on node 1, t_1 precedes t_2 and is in conflict with t_2
  - ✓ on node 2, t_2 precedes t_1 and is in conflict with t_1

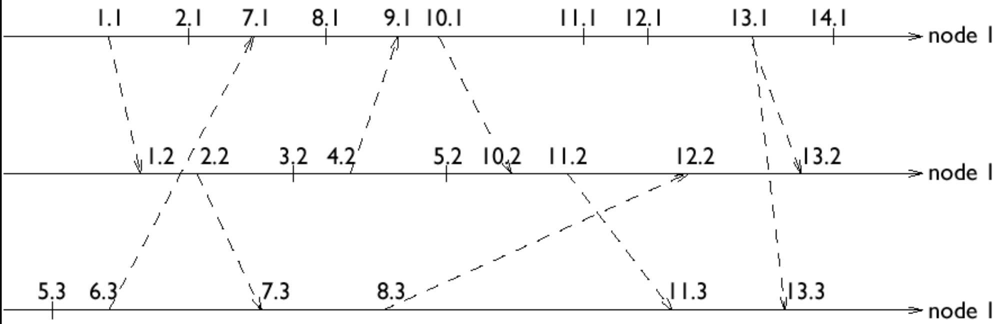
Global Serializability

- Global serializability of distributed transactions over the nodes of a distributed database requires the existence of a unique ***serial schedule*** *S* equivalent to all the local schedules S_i
- The following properties are valid.
  - ✓ If each scheduler uses two-phase locking on each node and commits when all the sub-transactions have acquired all the resources, then the resulting schedules are globally conflict-serializable; thanks to the 2-phase commit protocol
  - ✓ If each distributed transaction acquires a single timestamp and uses it in all requests to all the schedulers that use concurrency control based on timestamp, the resulting schedules are globally serial, based on the order imposed by the timestamps

Lamport Timestamp Method

- The ***Lamport method*** for assigning timestamps reflects the precedence among events in a distributed system.
- A timestamp is a number consisting of two groups of digits:
  - ✓ Least significant digits identify the node ofr the event;
  - ✓ The most significant digits identify the events that happen at that node; they can be obtained from a local counter, which is incremented with each event.
- Whenever two nodes exchange a message, the timestamps become synchronized:
  - ✓ The receiving event must have a timestamp greater than the timestamp of the sending event;
  - ✓ This may require increasing of the local counter on the receiving node.

Example of Lamport's Method



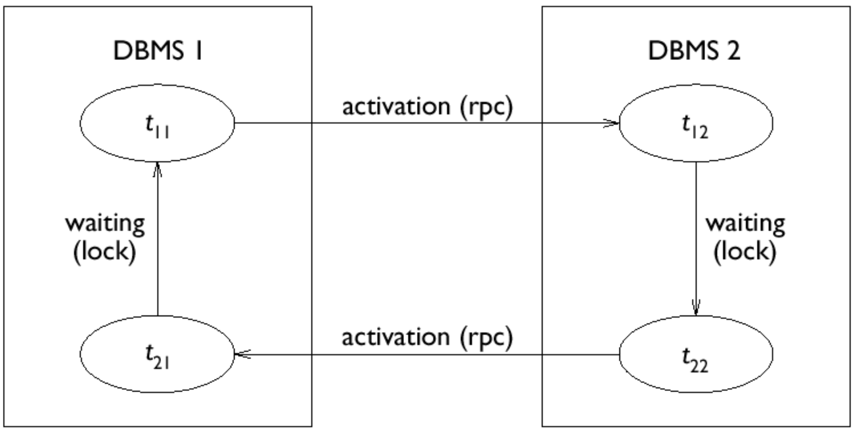
Distributed Deadlocks

- Distributed deadlocks can be caused by circular waiting situations between two or more nodes
- The ***time-out*** method is valid and most used by distributed DBMSs
- Deadlock resolution can be done with an asynchronous and distributed protocol (implemented in a distributed version of DB2 by IBM)

Distributed Deadlock Resolution

- Assume that sub-transactions are activated by using a ***remote procedure call***; that is a synchronous call to a procedure that is remotely executed; this model allows for two distinct types of waiting
  - ✓ Two sub-transactions of the same transaction can be in waiting in distinct DBMSs as one waits for the termination of the other
    - If t_{11} activates t_{12}, it waits for the termination of t_{12}
  - ✓ Two different sub-transactions on the same DBMS can wait as one blocks a data item to which the other one requires access
    - If t_{11} locks an objects requested by t_{21}, t_{21} waits for the termination of t_{11}

Example of Distributed Deadlock



Representation of Waiting Conditions

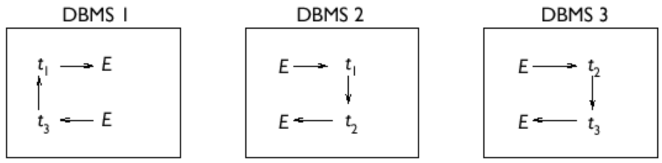
- The waiting conditions at each DBMS can be characterized using precedence conditions, where EXT represents executions at a remote DBMS:
  - ✓ On DBMS1 we have: $EXT \dashrightarrow t_{21} \dashrightarrow t_{11} \dashrightarrow EXT$
  - ✓ On DBMS2 we have: $EXT \dashrightarrow t_{12} \dashrightarrow t_{22} \dashrightarrow EXT$

  , where EXT represents a remote procedure call
- The general format of a waiting condition is summarized using a ***wait sequence***: $EXT \dashrightarrow t_i \dashrightarrow t_j \dashrightarrow EXT$

Algorithm

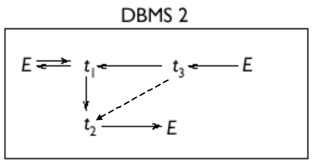
- The algorithm is periodically activated on various DBMSs of the system. When it is activated, it:
  - ✓ integrates new wait sequences with the local wait conditions as described by the lock manager;
  - ✓ analyzes wait conditions on its DBMS and detects deadlocks
  - ✓ communicates the wait sequences to other instances of the same algorithm
- To avoid situations where the same deadlock is discovered more than once, the algorithm sends wait sequences:
  - ✓ 'ahead', towards the DBMS which has received the remote procedure call
  - ✓ only if, for example, $i > j$ where i and j are the identifiers of the transactions

Application of the Algorithm



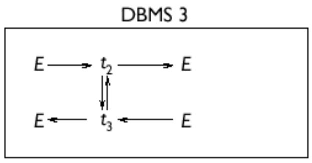
a. initial situation

Send wait sequence from DBMS 1 to 2



b. first pass of the algorithm

Send sequence (t3 waits for t2) from DBMS 2 to 3



c. second pass of the algorithm

Failures in Distributed Systems

A distributed system is subject to a variety of failures:

- ***Node failure*** may occur on any node of the system and be soft or hard, as discussed before
- ***Message losses*** leave a protocol in an uncertain state:
  - ✓ Each protocol message (*msg*) is followed by an acknowledgement message (*ack*)
  - ✓ The loss of either one leaves the sender uncertain about whether the message has been received.
- ***Network partitioning*** is a failure of the communication links of the computer network resulting in two sub-networks that have no communication between each other.
  - ✓ A transaction can be simultaneously active in more than one sub-network.

The Two-Phase Commit Protocol

- ***Commit protocols*** allow a transaction to reach the correct commit or abort decision at all participating nodes.
- The ***two-phase commit protocol*** (2PC) is like a marriage, in that a decision of two parties is received and registered by a third party, who ratifies the marriage.
  - ✓ The servers – who represent the participants to the marriage – are called ***resource managers*** (RM)
  - ✓ The ceremonial agent (or coordinator) is allocated to a process, called the ***transaction manager*** (TM)

More on Two-Phase Commit (2PC)

- The 2PC protocol takes place by means of a rapid exchange of messages between TM and RM and writing of records into their logs. The TM can use:
  - ✓ broadcast mechanisms (transmission of the same message to many nodes, collecting responses arriving from various nodes);
  - ✓ serial communication with each of the RMs in turn.

New Log Records

- Records of TM
  - ✓ The `prepare` record contains the identity of all the RM processes (that is, their identifiers of nodes and processes.)
  - ✓ The `global commit` or `global abort` record describes the global decision. When the TM writes in its log the `global commit` or `global abort` record, it has reached a final decision.
  - ✓ The `complete` record is written at the end of the protocol.

New Log Records

- Records of RM
  - ✓ The `ready` record indicates the irrevocable availability to participate in the 2PC protocol, thereby contributing to a decision to commit. Can be written only when the RM is **recoverable**, i.e., possesses locks on all resources that need to be written. The identifier (process identifier and node identifier) of the TM is also written on this record.
  - ✓ In addition, `begin`, `insert`, `delete`, and `update` records are written as in centralized servers.
- At any time an RM can autonomously abort a sub-transaction, thereby ending its participation to the 2PC protocol. This causes a global abort.

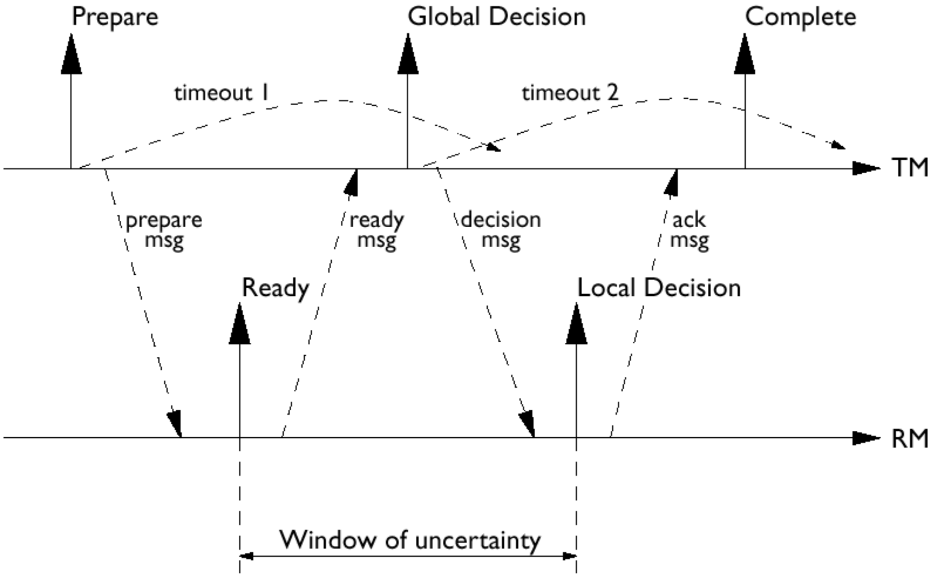
First Phase of the Basic Protocol

- The TM writes the `prepare` record in its log and sends a `prepare` message to all the RMs. Sets a timeout indicating the maximum time allocated to the completion of this phase.
- The recoverable RMs write on their own logs the `ready` record and transmit to the TM a `ready` message, which indicates the positive choice of commit participation.
- The non-recoverable RMs send a `not-ready` message and end the protocol.
- The TM collects the reply messages from the RMs:
  - ✓ If it receives a positive message from all the RMs, it writes a `global commit` record on its log;
  - ✓ If one or more negative messages are received or the time-out expires without the TM receiving all the messages, it writes a `global abort` record on its log.

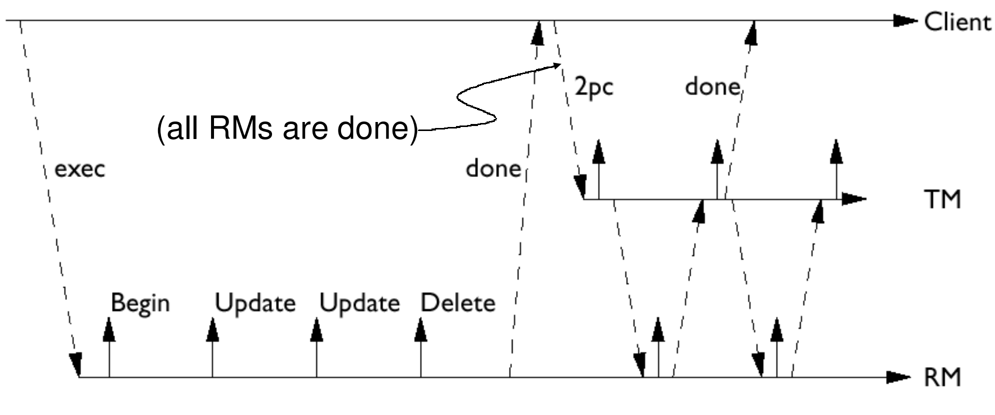
Second Phase of the Basic Protocol

- The TM transmits its global decision to the RMs. It then sets a second time-out
- The RMs that are ready receive the decision message, write the `commit` or `abort` record on their own logs, and send an acknowledgement to the TM. Then they implement the commit or abort by writing the pages to the database (as discussed earlier.)
- The TM collects all the *ack* messages from the RMs involved in the second phase. If the time-out expires it sets another time-out and repeats the transmission to all the RMs from which it has not received an *ack*.
- When all the *ack*s have arrived, the TM writes the `complete` record on its log.

Two-Phase Commit Protocol



Actions of Client, TM, and RM for 2PC



Blocking, Uncertainty, Recovery

- An RM in a `ready` state loses its autonomy and awaits the decision of the TM. A failure of the TM leaves the RM in an uncertain state. The resources acquired by using locks are blocked.
- The interval between the writing on the RM's log of the `ready` record and the writing of the `commit` or `abort` record is called the ***window of uncertainty***. The 2PC protocol is designed to keep this interval to a minimum.
- Recovery protocols are performed by the TM or RM after failures; they recover a final state which depends on the global decision of the TM.

Recovery of Participants

- Performed by the warm restart protocol, based on the last record written in the log:
  - ✓ when it's an action or `abort`, actions are ***undone***; when it's a `commit`, actions are ***redone***; either way, the failure has occurred before starting the commit protocol;
  - ✓ when it's `ready`, failure has occurred during 2PC; participant is ***in doubt*** about the result of the transaction.
- During a warm restart, the identifiers of the transactions in doubt are collected in the ***ready set***. For each of them the final transaction outcome must be requested from the TM.
- This can happen as a result of a direct (***remote recovery***) request from the RM or as a repetition of the second phase of the protocol.

Recovery of the Coordinator

- When the last record in the log is a `prepare`, the failure of the TM might have placed some RMs in a blocked situation. Two recovery options:
  - ✓ Write `global abort` on the log, and then carry out the second phase of the protocol.
  - ✓ Repeat the first phase, trying to arrive to a global commit.
- When the last record in the log is a global decision, some RMs may have been correctly informed of the decision and others may have been left in a blocked state. The TM must repeat the second phase.

Message Loss and Network Partitioning

- The loss of a `prepare` or `ready` message is not distinguishable by the TM. In both cases, the time-out of the first phase expires and a global abort decision is made.
- The loss of a decision or *ack* message are also indistinguishable. In both cases, the time-out of the second phase expires and the second phase is repeated.
- A ***network partitioning*** does not cause further problems, in that the transaction will be successful only if the TM and all the RMs belong to the same partition.

The Presumed Abort Protocol

- The ***presumed abort*** protocol is used by most DBMSs.
- Based on the following rule: *when a TM receives a remote recovery request from an in-doubt RM and doesn't know the outcome of that transaction, the TM returns a global abort decision as default.*
- As a consequence, the ***force*** of `prepare` and `global abort` records can be avoided, because in the case of loss of these records the default behavior gives an identical recovery.
- Furthermore, the `complete` record is not critical for the algorithm, so it need not be forced; in some systems, it is omitted. In conclusion the records to be forced are `ready`, `global commit` and `commit`.

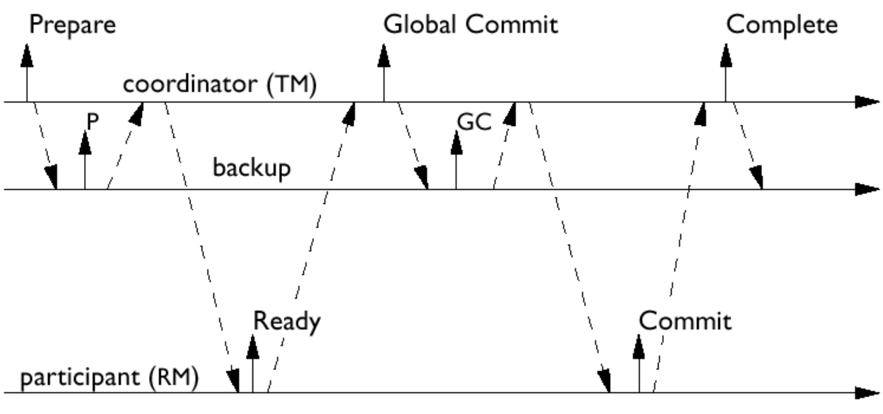
Read-only Optimization

- When a participant is found to have carried out only read operations (no write operations.)
- It responds `read-only` to the `prepare` message and suspends the execution of the protocol.
- The coordinator ignores read-only participants in the second phase of the protocol.

Four-Phase Commit Protocol

- Created by *Tandem*, a provider of hardware-software solutions for data management based on the use of replicated resources to obtain reliability.
- The TM process is replicated by a backup process, located on a different node. At each phase of the protocol, the TM first informs the backup of its decisions and then communicates with the RMs.
- The backup can replace the TM in case of failure:
- When a backup becomes TM, it first activates another backup, to which it communicates the information about its state, and then continues the execution of the transaction.

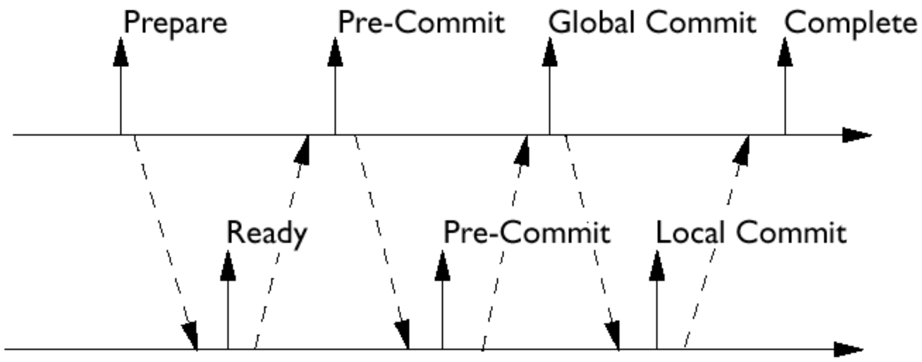
Four-Phase Commit Protocol



Three-Phase Commit Protocol

- Basic idea is to introduce a third pre-commit phase to the standard protocol; if the TM fails, a participant can be elected as new TM and decide the result of the transaction by looking at its log:
  - ✓ If the new TM finds `ready` as last record, no other participants in the protocol has gone beyond the pre-commit condition, and the decision is to abort;
  - ✓ If the new TM finds `pre-commit` as last record, it knows that the other participants are at least in the ready state, and thus can make the decision to commit.
- The three-phase commit protocol has serious drawbacks and has not been successfully implemented: it lengthens the window of uncertainty, and is not resilient to network partitioning, without additional quorum mechanisms.

The Three-Phase Commit Protocol



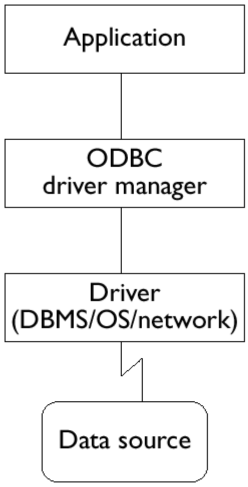
Interoperability

- Interoperability is an important problem in the development of heterogeneous applications for distributed databases
- Interoperability means that there are conversion and translation functions available which make it possible to exchange information between systems, networks and applications, despite their heterogeneity.
- Interoperability is made possible by means of standard protocols such as FTP, SMTP/MIME, and so on
- With reference to databases, interoperability is guaranteed by the adoption of suitable standards

Open Database Connectivity (ODBC)

- It is an application interface proposed by Microsoft in 1991 for the construction of heterogeneous database applications, supported by most relational products.
- The language supported by ODBC is a restricted SQL, characterized by a minimal set of instructions.
- Applications interact with DBMS servers by means of a *driver*, a library that is dynamically connected to the applications. The driver masks the differences of interaction due to the DBMS, the operating system and the network protocol; for example, the trio (*Sybase*, *Windows/NT*, *Novell*) identifies a single driver.
- ODBC does not support the two-phase commit protocol.

The ODBC Interface



X-Open Distributed Transaction Processing (DTP)

- Guarantees the interoperability of transactions on DBMSs of different suppliers ; assumes the presence of one client, several RMs and one TM.
- The protocol consists of two interfaces:
  - ✓ Between client and TM, called *TM-interfac*e;
  - ✓ Between TM and each RM, called *XA-interfac*e.
- Relational DBMSs must provide the XA-interface.
- Various products specializing in transaction management, such as *Encina* (a product of the Transarc company) and *Tuxedo* (from Unix Systems, originally AT&T) provide the TM component.

Features of X-Open DTP

- RMs are passive; they respond to RPCs issued by the TM.
- Uses 2PC with presumed abort and read-only optimizations.
- The protocol supports ***heuristic decisions***, which in the presence of failures allow the evolution of a transaction under the control of the operator:
  - ✓ When an RM is blocked because of the failure of the TM, an operator can impose a heuristic decision (generally, an `abort`), thus allowing the release of locked resources;
  - ✓ When heuristic decisions cause a loss of atomicity, the protocol guarantees that the client processes are notified;
  - ✓ The resolution of inconsistencies due to erroneous heuristic decisions is application-specific.

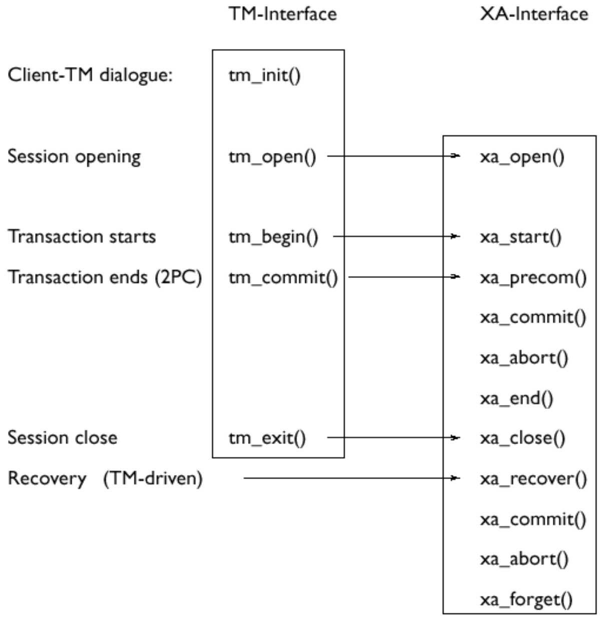
The TM Interface

- `tm_init` and `tm_exit` initiate and terminate the client-TM dialogue.
- `tm_open` and `tm_term` open and close a session with the TM.
- `tm_begin` begins a transaction.
- `tm_commit` requests a global commit.

The XA Interface

- xa_open and xa_close open and close a session between TM and a given RM.
- xa_start, xa_end activate, complete a new transaction
- xa_precom requests that the RM carry out the first phase of the commit protocol; the RM process can respond positively to the call only if it is in a recoverable state.
- xa_commit and xa_abort communicate the global decision about the transaction.
- xa_recover initiates a recovery procedure after the failure of a process (TM or RM); the RM consults its log and builds three sets of transactions: transactions *in doubt*, ones decided by a *heuristic commit*, and ones decided by a *heuristic abort*.
- xa_forget allows an RM to forget transactions decided in a heuristic manner.

The XA Interface



Cooperation Among Pre-existing Systems

- ***Cooperation*** is the ability of a system to make use of application services made available by other systems, possibly managed by different organizations.
- Needs for cooperation rise for different reasons, which range from the simple demand for integration of components developed separately within the same organization, to the co-operation or fusion of different companies and organizations.
- The integration of databases is quite difficult. Over-ambitious integration and standardization objectives are destined to fail. The 'ideal' model of a highly integrated database, which can be queried transparently and efficiently, is impossible to develop and manage in most cases.

System Cooperation Types

- Two types of cooperation:
  - ✓ ***process-centered cooperation***: systems cooperate by exchanging messages, information or documents, or by triggering activities, without making remote data explicitly visible;
  - ✓ ***data-centered cooperation***, where the data is distributed, heterogeneous and autonomous, and accessible by cooperating systems, according to a cooperation agreement.
- We concentrate on data-centered cooperation, characterized by data autonomy, heterogeneity and distribution

Features of Data-Centered Cooperation

- ***Transparency level*** measures how the distribution and heterogeneity of the data are masked.
- ***Complexity of distributed operations*** measures the degree of coordination necessary to carry out operations among the cooperating databases.
- ***Currency level*** indicates whether the data being accessed is up-to-date or not.
- Based on the above criteria, we can identify three architectures for guaranteeing data-based cooperation.