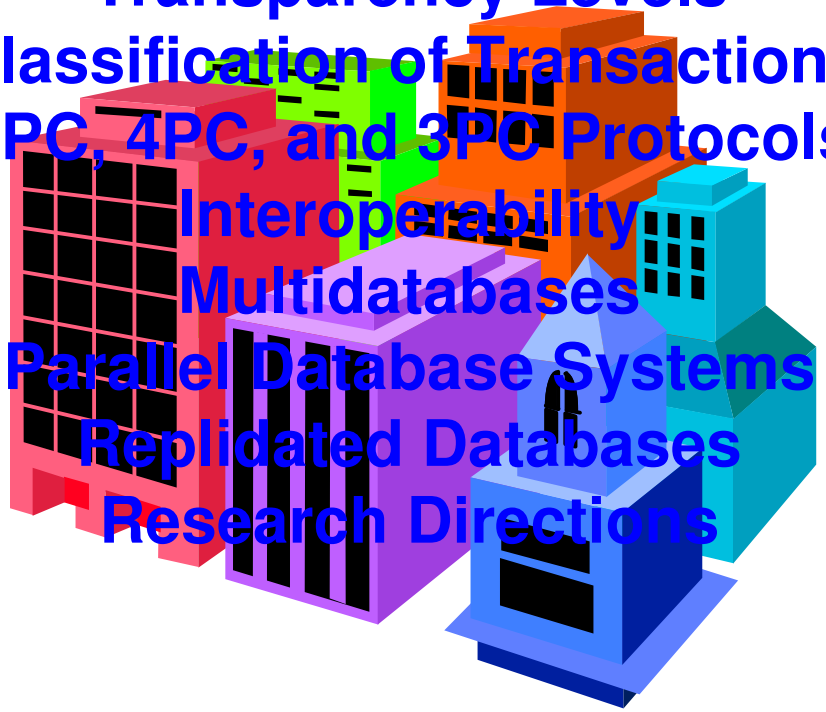
X. Distributed Architectures

Types of Distributed Architectures
Client-Server Architectures
Distributed Databases
Data Fragmentation and Allocation
Transparency Levels
Classification of Transactions
2PC, 4PC, and 3PC Protocols
Interoperability
Multidatabases
Parallel Database Systems
Replidated Databases
Research Directions



Paradigms for Data Distribution

- **Client-server architecture**: separation of the database server from the client
- **Distributed databases**: several database servers used by the same application
- **Parallel databases**: several data storage devices and processors operate in parallel for increasing performances
- **Replicated databases**: data logically representing the same information and physically stored on different servers
- **Data warehouses**: servers specialized for the management of data dedicated to decision support.

Technologies for Distributed Systems

- Networks -- local area and wide area networks (LAN and WAN respectively); you need to install network cables.
- Internet -- use telephone lines to transfer data and programs, thereby facilitating distribution.
- Wireless communication technologies -- the way of the future
  - ✓ Packet radio technology; generally works for short distances, low transfer rates;
  - ✓ Cellular networks, support cell phones;
  - ✓ Commercial satellite systems, such as iridium
- Telephone companies world-wide are scrambling to get out of wired telephony and into wireless services.

How Much Distributed Data is there?

- It is estimated that we produce about 1 hexabyte of unique web data per year; that's 10^{18}Bytes, or ~250MB per person on earth; no disk can hold so much data.
- We produce 500 times more information through email messages than through the web.
- The great challenge (some say the greatest ever!) of technology is to search, retrieve, organize and manage all these data, and make them useful to people.
- Someone (Brewster Kahle, an archivist) is trying to archive the web…

…a lot!!!

Separation of Functionalities

- OLTP (*On-Line Transaction Processing*) systems, aims at optimized management and reliable transactions on *database servers*, specialized for supporting hundreds or even thousands of transactions per second
- OLAP (*On-Line Analytic Processing*) systems, aimed at data analysis, which operate on *data warehouse* servers, specialized for data management for decision support

Desirable Properties for Highly Interactive Systems

- **Portability** refers to the ease of transporting programs from one environment to another (and it is thus a typical *compile-time* property)

Facilitated by language standards (e.g., SQL-2, SQL-3)

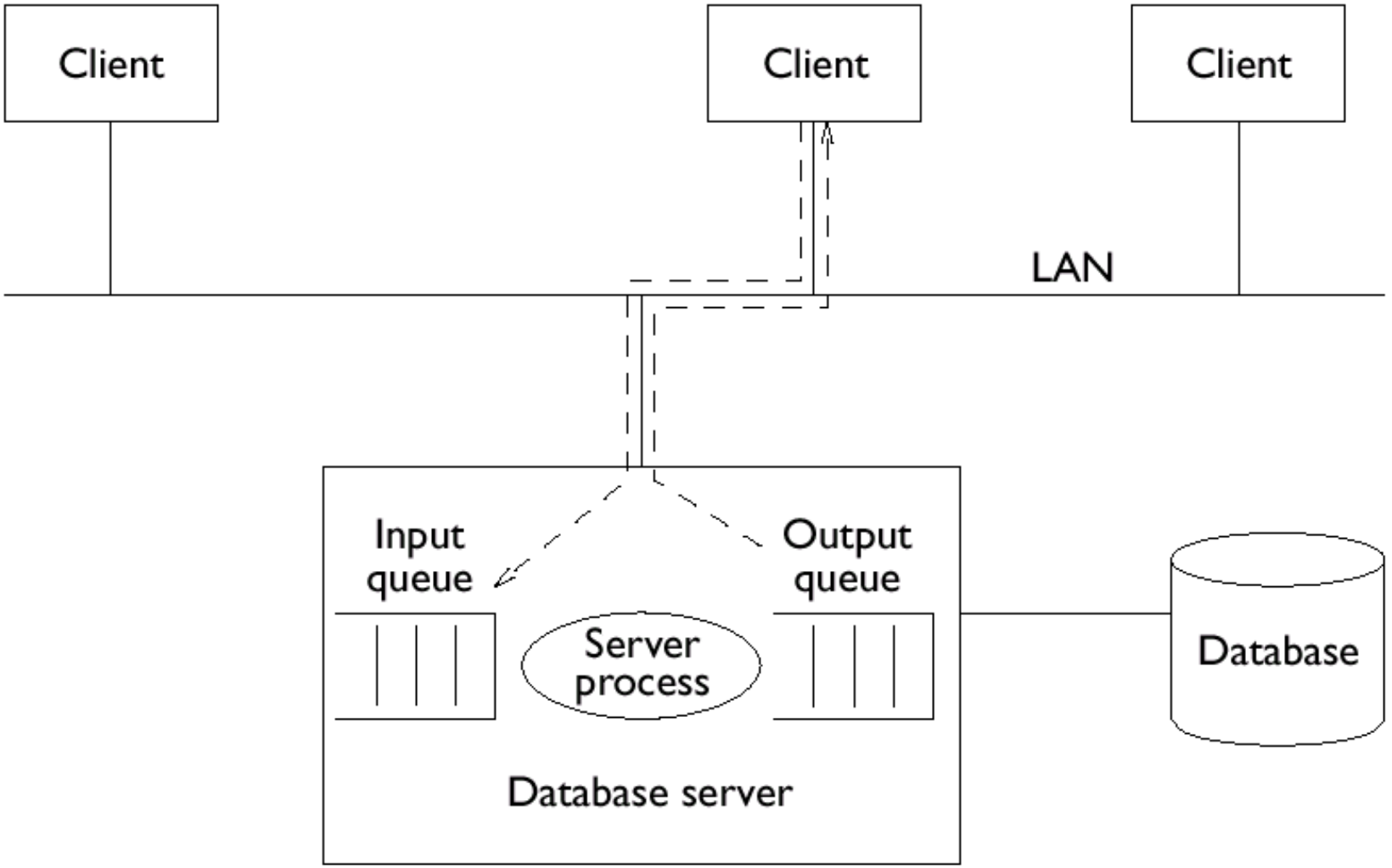
- **Interoperability** refers to a system's ability to interact with other, possibly heterogeneous, systems (and it is thus a typical *execution-time* property.)

Facilitated by standard data access protocols, including Database Connectivity (ODBC) and X-Open Distributed Transaction Processing (DTP)

Client-Server Architectures

- **Client-server** is a general model of interaction between software processes, where interacting processes are classified into ***clients*** (which require services) and ***servers*** (which offer services)
- Requires a precise definition of a ***service interface***, which lists the services offered by the server
- The client process performs an active role, the server process is reactive
- Normally, a client process requests few services in sequence from one or more server processes, while a process server responds to multiple requests from many process clients.

Typical Client-Server Architecture



Client Server Architecture for Data Management

- It is a good idea to assign clients and servers to distinct computers because:
  - ✓ The functions of clients and servers are clearly identified
  - ✓ They give rise to a convenient separation of application and management activities
- SQL offers an ideal programming paradigm for the identification of the 'service interface'
  - ✓ SQL queries are sent from the client to the server
  - ✓ The query results are returned to the client
  - ✓ The standardization, portability and interoperability of SQL allows the construction of client applications that involve different server systems

Allocation of Clients and Servers to Different Computers

- The computer dedicated to the client must be suitable for interaction with the user and support productivity tools (electronic mail, word processing, spreadsheet, Internet access, and workflow management)
- The server computer must have a large main memory (to support buffer management) and a high capacity disk (for storing the entire database)

Multi-Threaded Architectures

- Often, database (and other) servers are ***multi-threaded***:
  - ✓ Behave like a single process that works dynamically on behalf of several transactions;
  - ✓ Each unit of execution of the server process for a given transaction is called a ***thread;***
- Servers are permanently active processes that control an ***input queue*** for client requests and an ***output queue*** for the query results
- Often, a ***dispatcher*** process distributes requests to the servers and returns the responses to the clients
- When the dispatchers can dynamically define the number of active server processes as a function of the number of requests received: we say that a ***server class*** is available

Two-Tier vs Three-Tier Architectures

- **Two-tier architectures**: the client is both the user interface and the application manager
  - ✓ The client is called ***thick-client***, as it supports the application logic
- **Three-tier architectures**: a second server is present, known as the ***application server***, responsible for the management of the application logic common to many clients
  - ✓ The client is named ***thin-client***; it is responsible only for the interface with the final user; such clients can be deployed using browser technology

Distributed Databases

- A ***distributed database*** is a system in which some clients interact with multiple servers for the execution of an application
- We discuss separately:
  - ✓ How a user can specify distributed queries
  - ✓ How the server technology is extended in a distributed database

Advantages of Distributed Databases

- Distributed databases respond to application needs. After all, enterprises are geographically distributed; distributed databases allow the distribution of data processing and control to the environment where data is generated and largely used
- Distributed databases offer greater flexibility, modularity and resistance to failures
  - ✓ Distributed systems can be configured by the progressive addition and modification of components;
  - ✓ Although they are more vulnerable to failures, they support 'graceful degradation' (respond to failures with a reduction in performance but without total failure)

Classification of Applications

- Based on the type of DBMS involved:
  - ✓ Homogeneous Distributed Databases: When all the servers have the same DBMS
  - ✓ Heterogeneous Distributed Databases: When the servers support different DBMSs
- Based on the network:
  - ✓ Can use a ***local area network*** (LAN)
  - ✓ Can use a ***wide area network*** (WAN)

Classification of Applications

Type of DBMS	Network type: LAN	Network type: WAN
Homogeneous	Data management and financial applications	Travel management and financial applications
Heterogeneous	Inter-divisional information systems	Integrated banking and inter-banking systems

Local Independence and Cooperation

- In a distributed database, each server has its own capacity to manage applications independently:
  - ✓ A distributed database should not maximize the interaction and the necessity of transmitting data via networks
  - ✓ On the contrary, the planning of data distribution and allocation should be done in such a way that applications should operate independently on a single server

Data Fragmentation and Allocation

- Given a relation R, its fragmentation consists of determining fragments R_i by applying algebraic operations to R.
  - ✓ In ***horizontal fragmentation***, fragments R_i are groups of tuples having the same schema as R (as after selection on R); Horizontal fragments are usually disjoint;
  - ✓ In ***vertical fragmentation***, each fragment R_i has a subset of the schema of R (as though projection was applied on R). Each vertical fragment includes the primary key of R
- The fragmentation is correct if it is:
  - ✓ ***Complete***: each data item of R must be present in one of its fragments R_i
  - ✓ ***Restorable***: the content of R must be restorable from its fragments

Example

- Consider EMPLOYEE (Emp#, Name, Dept#, Salary, Taxes)
- Horizontal fragmentation
  - ✓ EMPLOYEE1 = $\sigma_{Emp\#<=3}$ EMPLOYEE
  - ✓ EMPLOYEE2 = $\sigma_{Emp\#>3}$ EMPLOYEE
- Reconstruction requires a union:
  - ✓ EMPLOYEE = EMPLOYEE1 U EMPLOYEE2
- Vertical fragmentation:
  - ✓ EMPLOYEE1 = $\Pi_{Emp\#,Name}$EMPLOYEE
  - ✓ EMPLOYEE2 = $\Pi_{Emp\#,DeptName,Salary,Tax}$EMPLOYEE
- Reconstruction requires an equi-join on key values (natural join).
  - ✓ EMPLOYEE = EMPLOYEE1 $\triangleright\triangleleft$ EMPLOYEE2

Initial Table

EMPLOYEE

EmpNum	Name	DeptName	Salary	Tax
1	Robert	Production	3.7	1.2
2	Greg	Administration	3.5	1.1
3	Anne	Production	5.3	2.1
4	Charles	Marketing	3.5	1.1
5	Alfred	Administration	3.7	1.2
6	Paolo	Planning	8.3	3.5
7	George	Marketing	4.2	1.4

Example of Horizontal Fragmentation

EMPLOYEE1

EmpNum	Name	DeptName	Salary	Tax
1	Robert	Production	3.7	1.2
2	Greg	Administration	3.5	1.1
3	Anne	Production	5.3	2.1

EMPLOYEE2

EmpNum	Name	DeptName	Salary	Tax
4	Charles	Marketing	3.5	1.1
5	Alfred	Administration	3.7	1.2
6	Paolo	Planning	8.3	3.5
7	George	Marketing	4.2	1.4

Vertical Fragmentation

EMPLOYEE1

EmpNum	Name
1	Robert
2	Greg
3	Anne
4	Charles
5	Alfred
6	Paolo
7	George

EMPLOYEE2

EmpNum	DeptName	Salary	Tax
1	Production	3.7	1.2
2	Administration	3.5	1.1
3	Production	5.3	2.1
4	Marketing	3.5	1.1
5	Administration	3.7	1.2
6	Planning	8.3	3.5
7	Marketing	4.2	1.4

Fragmentation and Allocation Schemata

- Each fragment R_i corresponds to a different physical file and is allocated to a different server.
- Thus, the relation is present in a virtual mode (like a view), while fragments are actually stored.
- The *allocation schema* describes the mapping of relations or fragments to the servers that store them. This mapping can be:
  - ✓ ***non-redundant***, when each fragment or relation is allocated to a single server
  - ✓ ***redundant***, when at least one fragment or relation is allocated to more than one server

Transparency Levels

- There are three significant levels: transparency of fragmentation, allocation and language
- In the ***absence of transparency***, each DBMS accepts its own SQL 'dialect': the system is heterogeneous and the DBMSs do not support a common interoperability standard
- Given SUPPLIER(S#,Name,City), with horizontal fragments
  - ✓ SUPPLIER1 = $\sigma_{City='London'}$ SUPPLIER
  - ✓ SUPPLIER2 = $\sigma_{City='Manchester'}$ SUPPLIER
- …and the allocation schema:
  - ✓ SUPPLIER1@company.London.uk
  - ✓ SUPPLIER2@company.Manchester1.uk
  - ✓ SUPPLIER2@company.Manchester2.uk

Fragmentation Transparency

- At this level, the programmer should not worry about whether or not the database is distributed or fragmented
- Query:

```
procedure Query1(:snum,:name);
select Name into :name
   from Supplier
   where SNum = :snum;
end procedure
```

Allocation Transparency

- The programmer should know the structure of the fragments, but does not have to indicate their allocation
- With replication, the programmer does not have to indicate which copy is chosen for access (*replication transparency*)
- Query:

```
procedure Query2(:snum,:name);
select Name into :name from Supplier1
  where SNum = :snum;
if :empty then
  select Name into :name
    from Supplier2 where SNum = :snum;
end procedure;
```

Language Transparency

- The programmer indicates both the structure of the fragments and their allocation
- Queries expressed at a higher level of transparency are transformed to this level by the distributed query optimizer.
- Query:

```
procedure Query3(:snum,:name);
select Name into :name
  from Supplier1@company.London.uk
  where SNum = :snum;
if :empty then select Name into :name
    from Supplier2@company.Manchester1.uk
    where SNum = :snum;
end procedure;
```

Optimizations

This application can be made more efficient in two ways:

- By using *parallelism*: instead of submitting the two requests in sequence, they can be processed in parallel, thus saving on the global response time
- By using *knowledge on the logical properties of fragments* (but then the programs are not flexible)

```
procedure Query4(:snum,:name,:city);
case :city of
"London": select Name into :name
    from Supplier1 where SNum = :snum;
"Manchester": select Name into :name
    from Supplier2 where SNum = :snum;
end procedure;
```

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

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

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

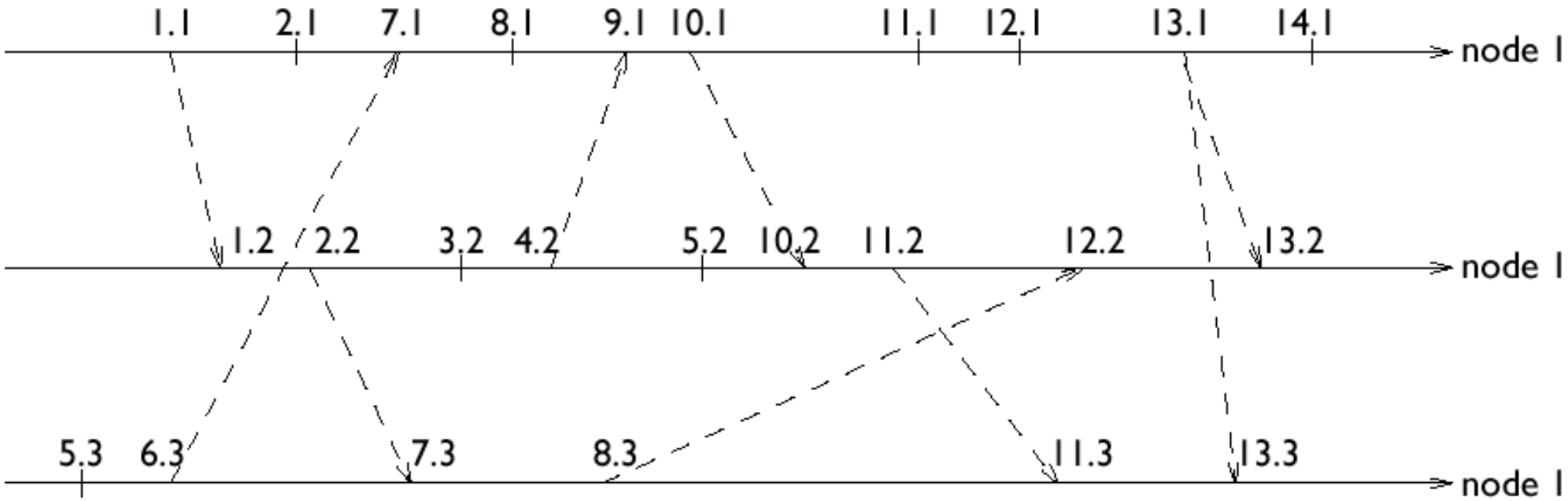
Global Serializability

- 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)

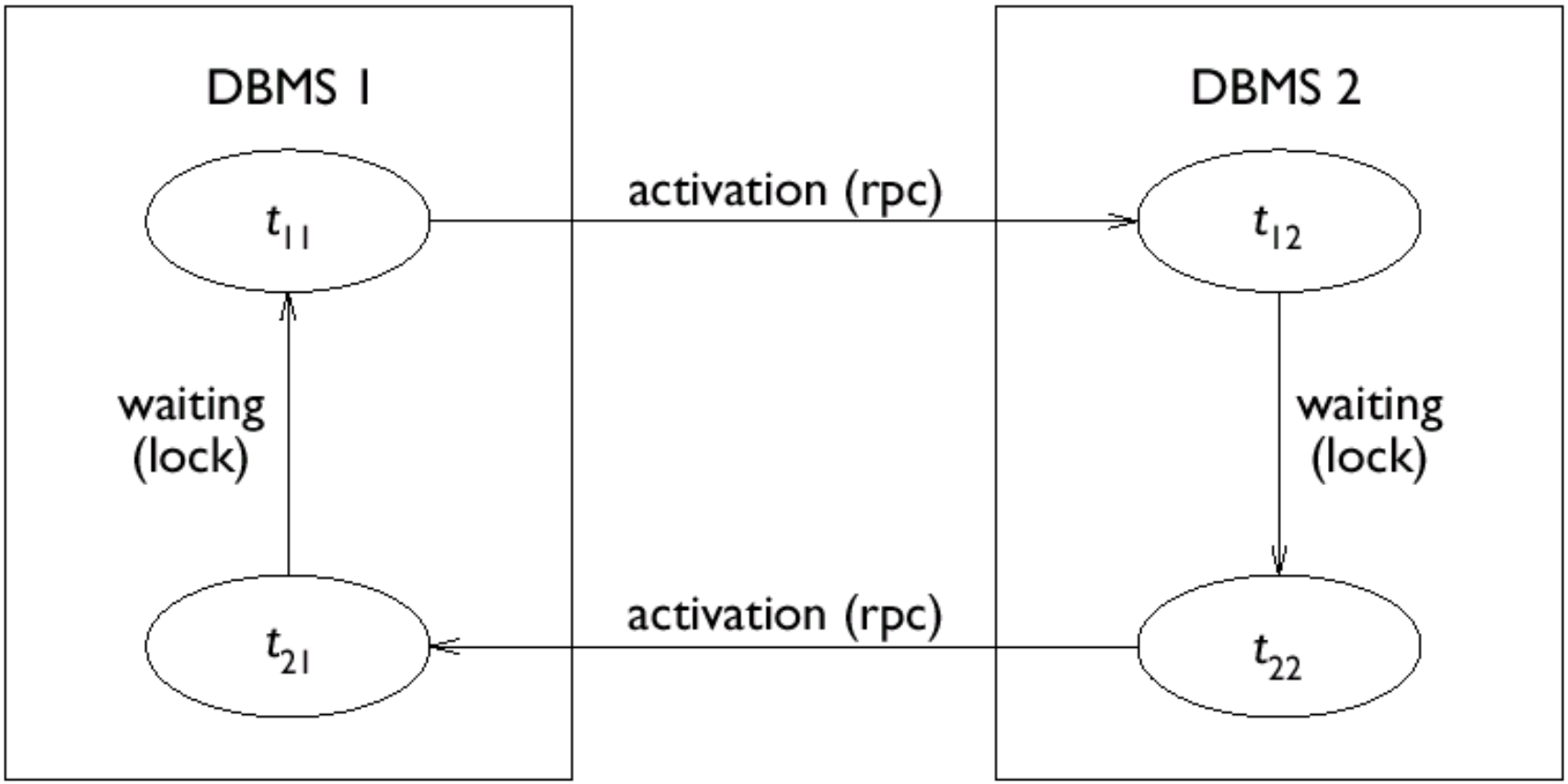
Distributed Deadlock Resolution

- 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



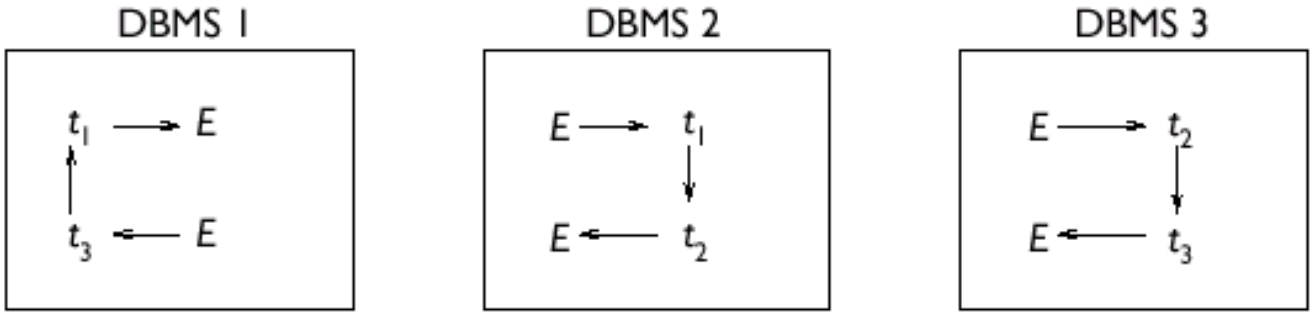
Representation of Waiting Conditions

- 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* --> t_{21} --> t_{11} --> *EXT*
  - ✓ On DBMS2 we have*: EXT* --> t_{12} --> t_{22} --> *EXT*

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

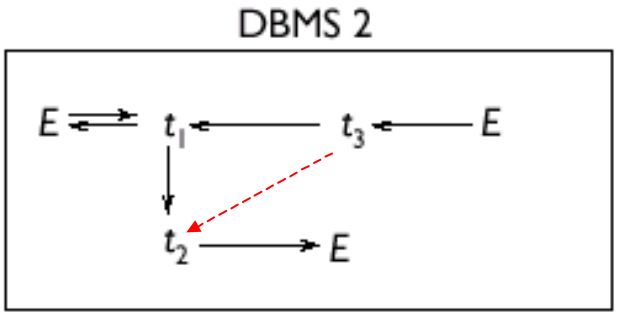
Algorithm

- 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



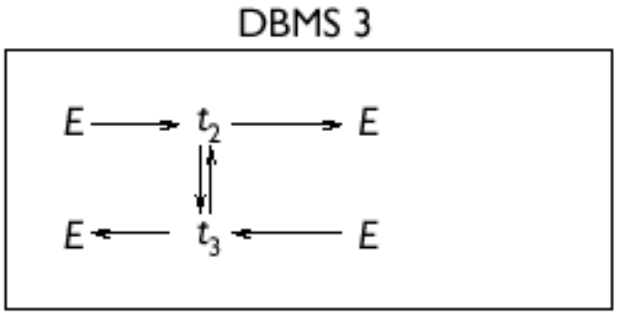
a. initial situation

Send wait sequence from DBMS 1 to 2



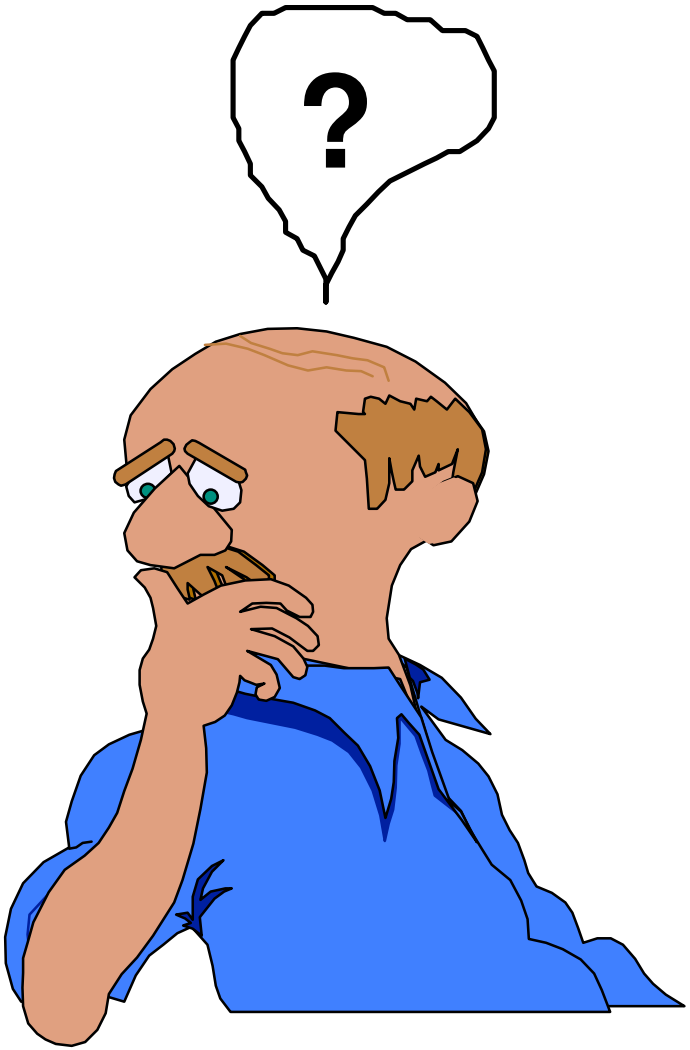
b. first pass of the algorithm

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



c. second pass of the algorithm

Application of the Algorithm



Questions?

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.

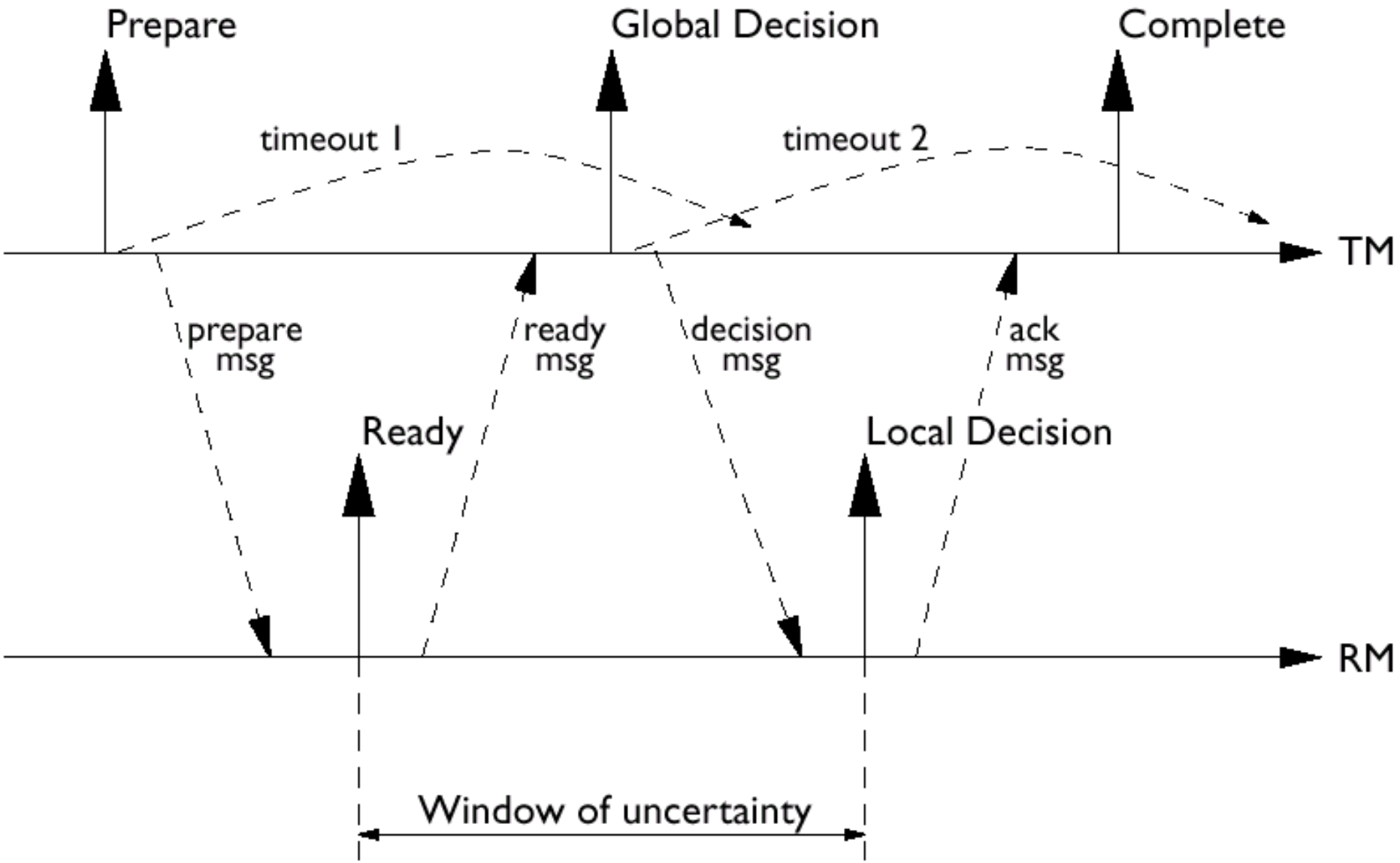
First Phase of the Basic Protocol

- 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.

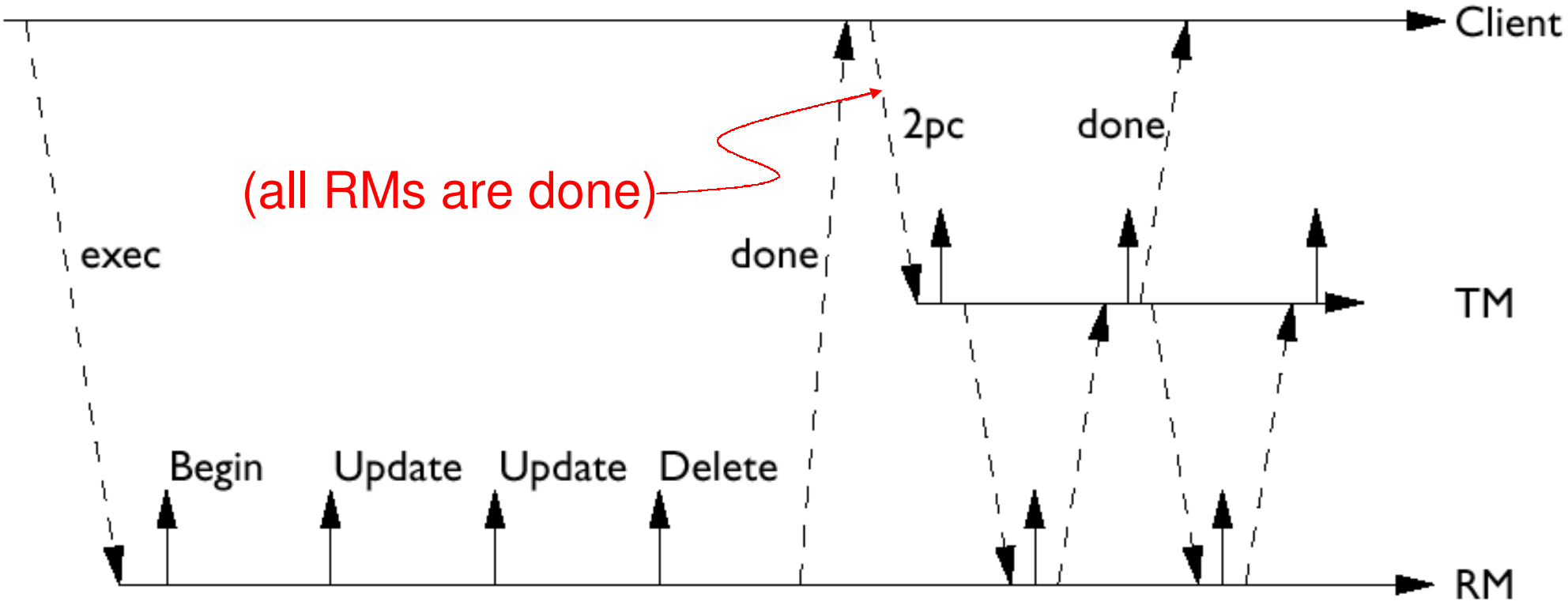
Second Phase of the Basic Protocol

- 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`.

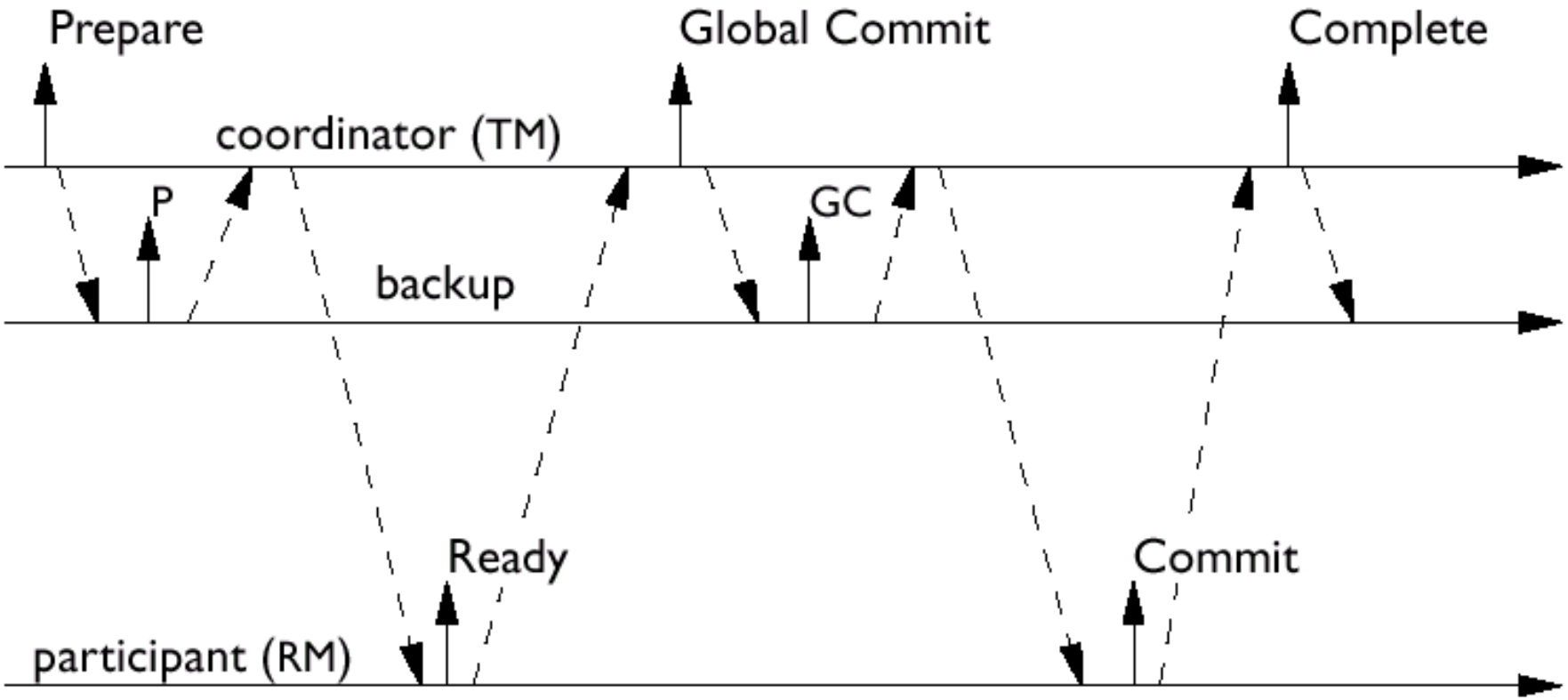
Read-only Optimization

- 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.

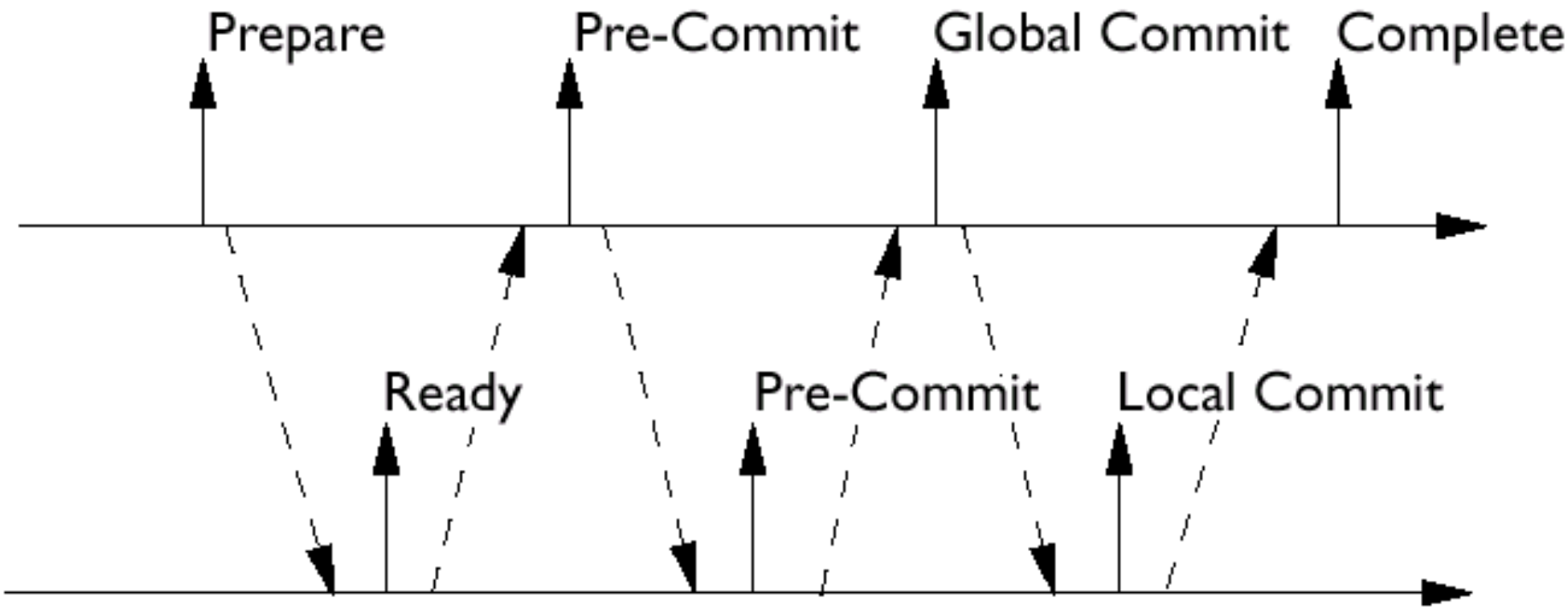
Four-Phase Commit Protocol



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



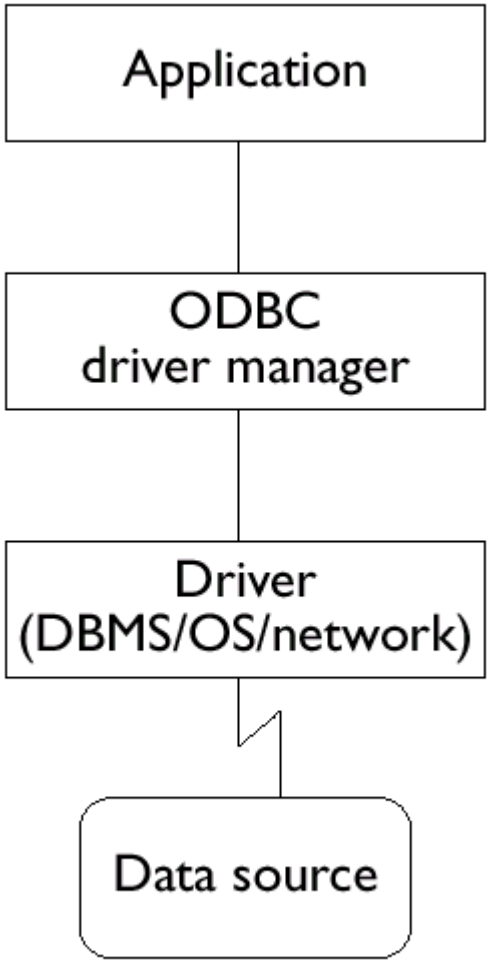
Interoperability

- 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-interface*;
  - ✓ Between TM and each RM, called *XA-interface*.
- 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.

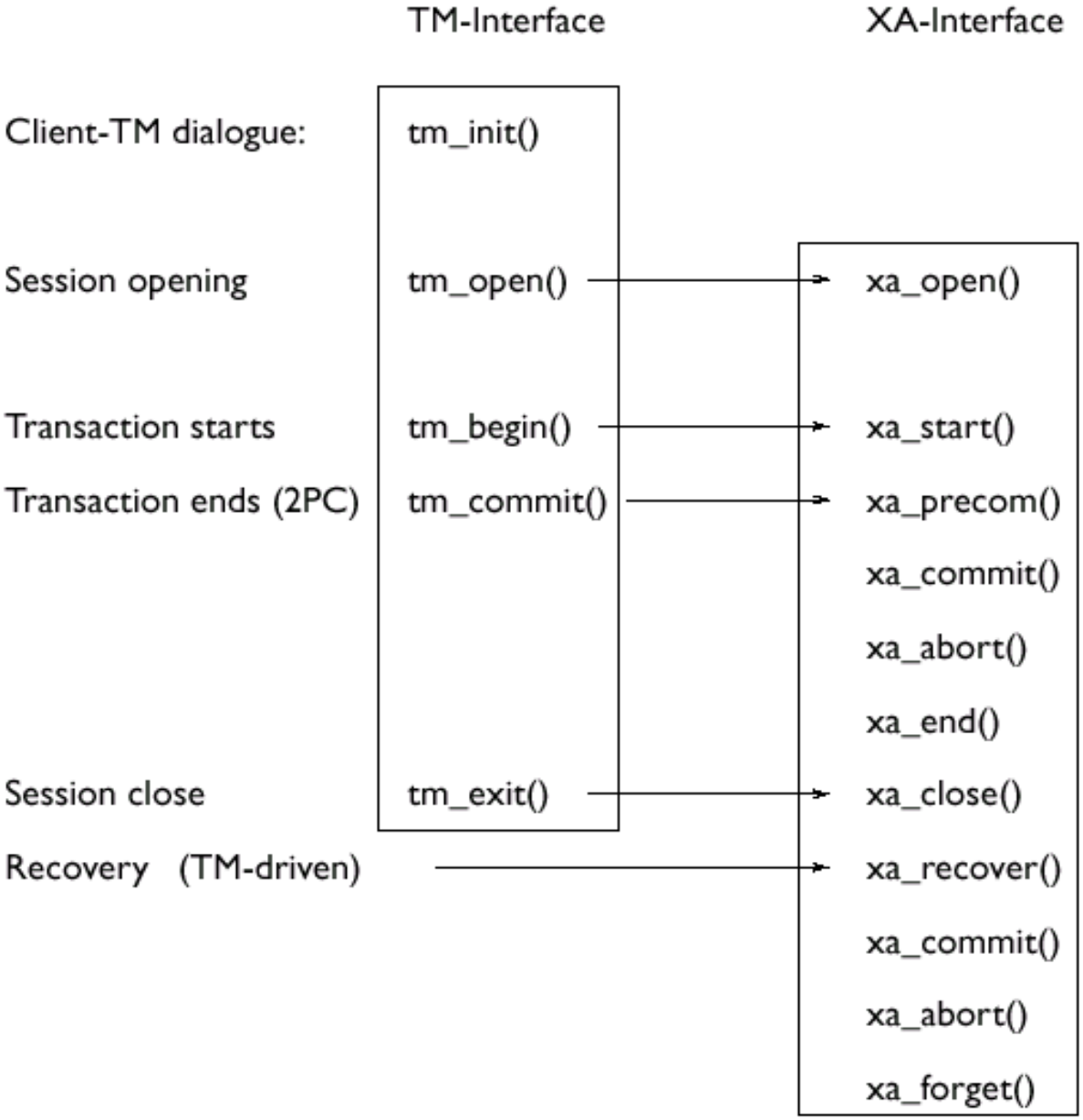
The TM Interface

- `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 doub*t, ones decided by a *heuristic commit*, and ones decided by a *heuristic abor*t.
- `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

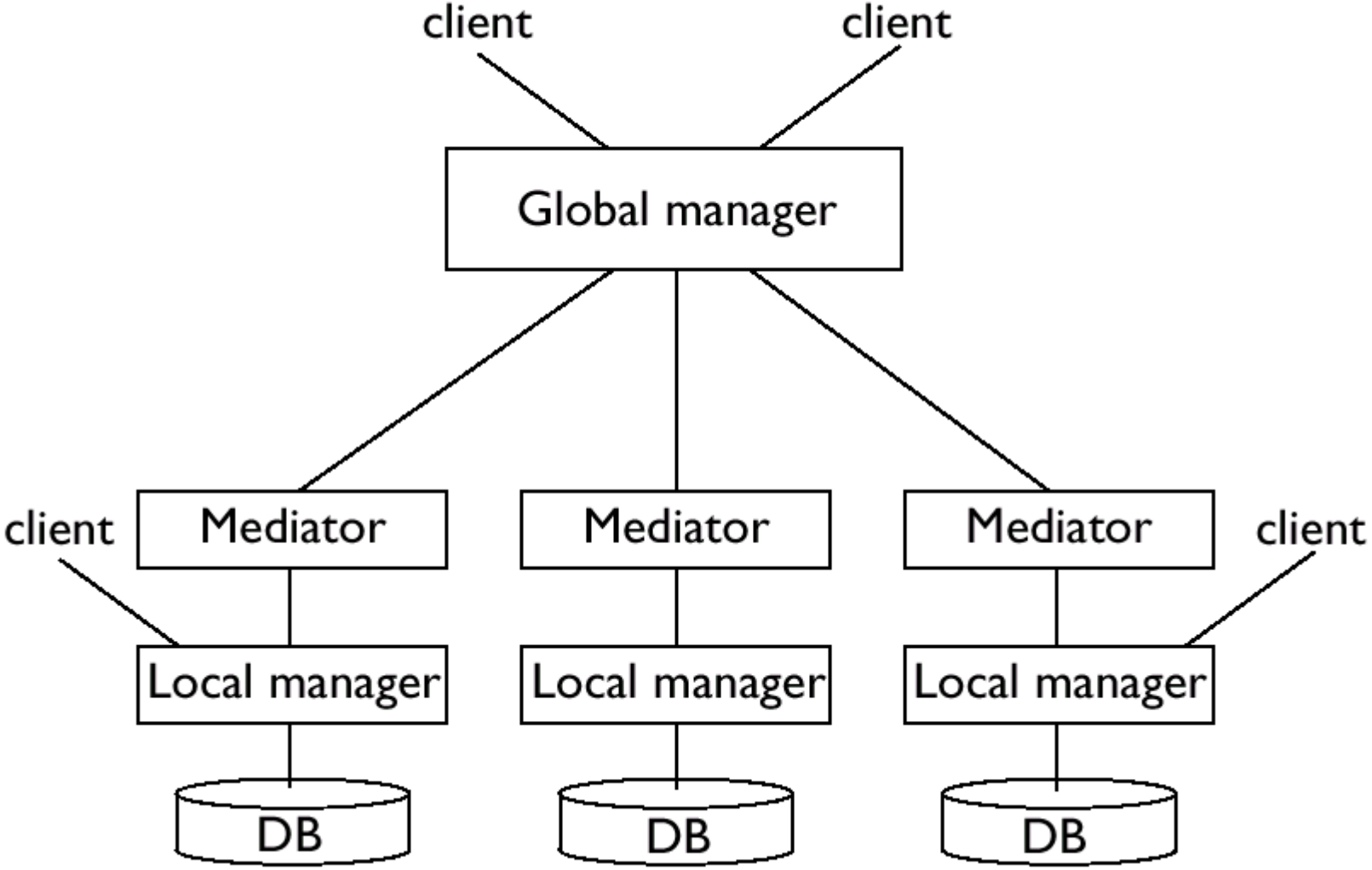
Features of Data-Centered Cooperation

- ***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.

Multidatabases

- Each of the participating databases continues to be used by its respective users (programs or end users.)
- Systems are also accessed by modules, called ***mediators***, which show only the portion of a database that is to be exported; mediators rely on a ***global manager***, to carry out the integration.
- In general, data cannot be modified by the mediators, because each source system is autonomous.
- Features:
  - ✓ presents an integrated view to the users, as if the databases were integrated;
  - ✓ provides a high level of transparency;
  - ✓ currency is also high, because data is accessed at source;

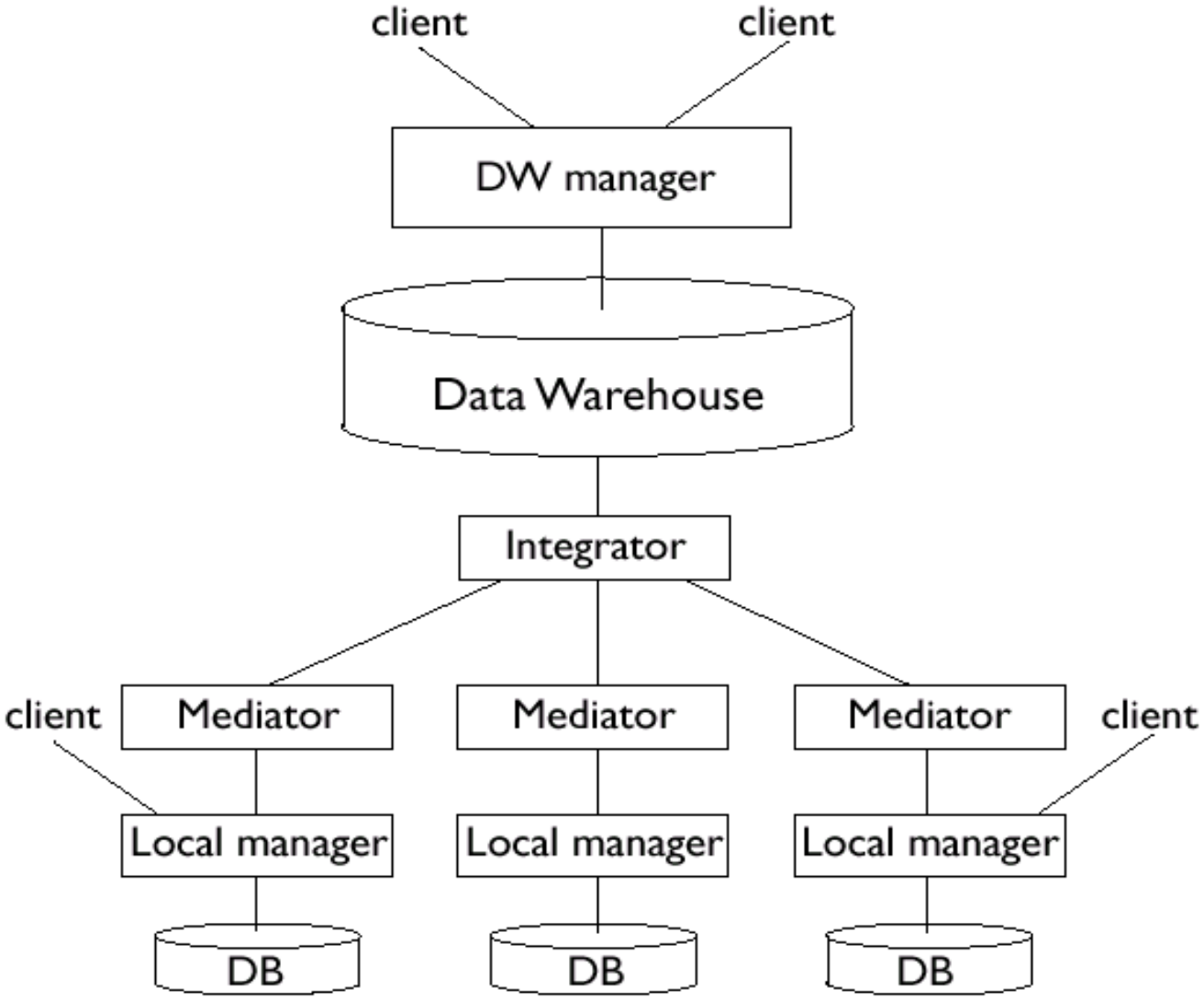
A Multidatabase System



Replicated Data Architectures

- They guarantee read-only access to secondary copies of a database.
- An example of replication is a ***data warehouse***, which downloads periodically data extracted from various heterogeneous distributed systems for purposes of analysis and decision support.
- Key feature: offers a high level of integration and transparency, but a reduced degree of currency.

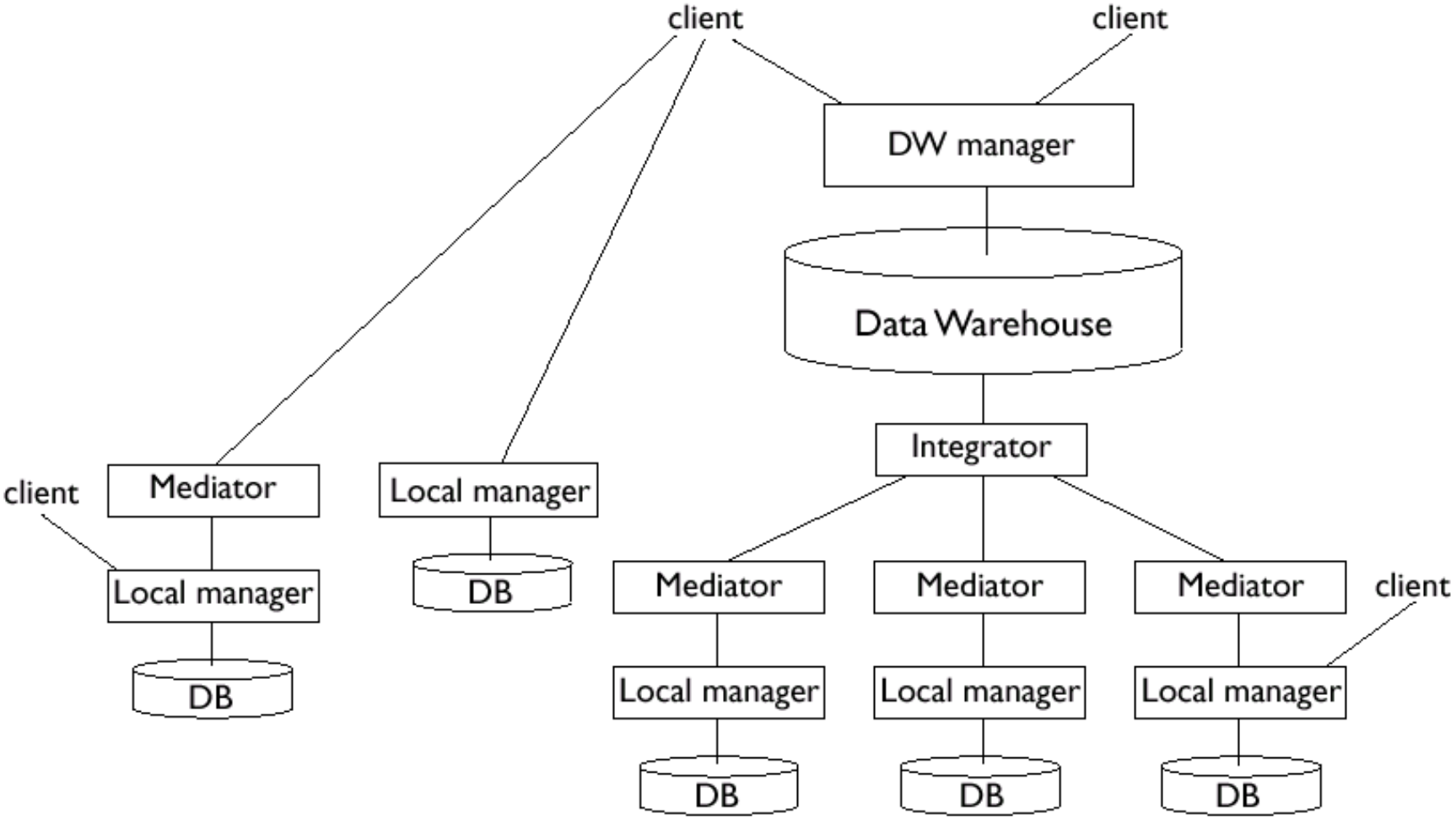
Systems Based on Replicated Data



Application-Level Data Integration

- Data integration is carried out explicitly by the application
- In the next example, three sources are integrated: an external database, a local database and a data warehouse, which in turn uses three sources of information
- Features:
  - ✓ Low degree of transparency and integration, with a degree of currency that depends on specific demands

Example of Application-Level Integration



Parallel Databases

- Were developed in the nineties along with the spread of standard multiprocessor architectures, after the failure of specialized database architectures (the so-called ***database machines***) during the eighties.
- Parallelism is possible with multiprocessor architectures both with and without shared memory.
- Parallelism in databases is useful because many data management operations are repetitive in nature, and can be carried out in parallel with great efficiency.
- For example, a complete scan of a large database can be done using *n* scans, each on a portion of the database. If the database is stored on *n* different disks managed by *n* different processors, resulting in a speedup of approximately 1/*n* of the time required for a serial scan,

Inter-Query Parallelism

- Parallelism is called ***inter-query*** when it carries out several queries in parallel.
  - ✓ The load imposed on the DBMS is typically characterized by simple and frequent transactions (up to thousands of transactions per second.)
  - ✓ Parallelism is introduced by multiplying the number of servers and allocating an optimal number of requests to each server.
  - ✓ In many cases, the queries are redirected to servers by a ***dispatcher*** process.
  - ✓ Useful for OLTP systems.

Intra-Query Parallelism

- Parallelism is known as ***intra-query*** when it executes parts of the same query in parallel:
  - ✓ The load on the DBMS is characterized by a few extremely complex queries, which are decomposed into various partial sub-queries, to be executed in parallel.
  - ✓ In general, queries are carried out one after another, using the entire multi-processor system for each query.
  - ✓ Useful for OLAP systems.

Parallelism and Data Fragmentation

- Parallelism for databases requires some form of data fragmentation: the fragments are distributed among many processors and allocated to distinct secondary memory devices.
- Consider:

```
ACCOUNT(AccNum, Name, Balance)
TRANSACTION(AccNum,Date,SerialNumber,
  TransType,Amount)
```

  Fragmentation based on predefined intervals of account number

Example of a Typical OLTP Query

- A typical OLTP query with inter-query parallelism:

```
procedure Query5(:acc_num, :total);
select Balance into :total
  from Account
  where AccNum = :acc_num;
end procedure;
```

- Directed towards specific fragments depending on their selection predicates

Example of a Typical OLAP Query

- A typical OLAP query with intra-query parallelism:

```
procedure Query6();
select AccNum, sum(Amount)
  from Account join Transaction
  on Account.AccNum = Transaction.AccNum
   where Date >= 1.1.1998
      and Date < 1.1.1999 group by AccNum
   having sum(Amount) > 100000;
end procedure;
```

- Carried out on all of the fragments in parallel

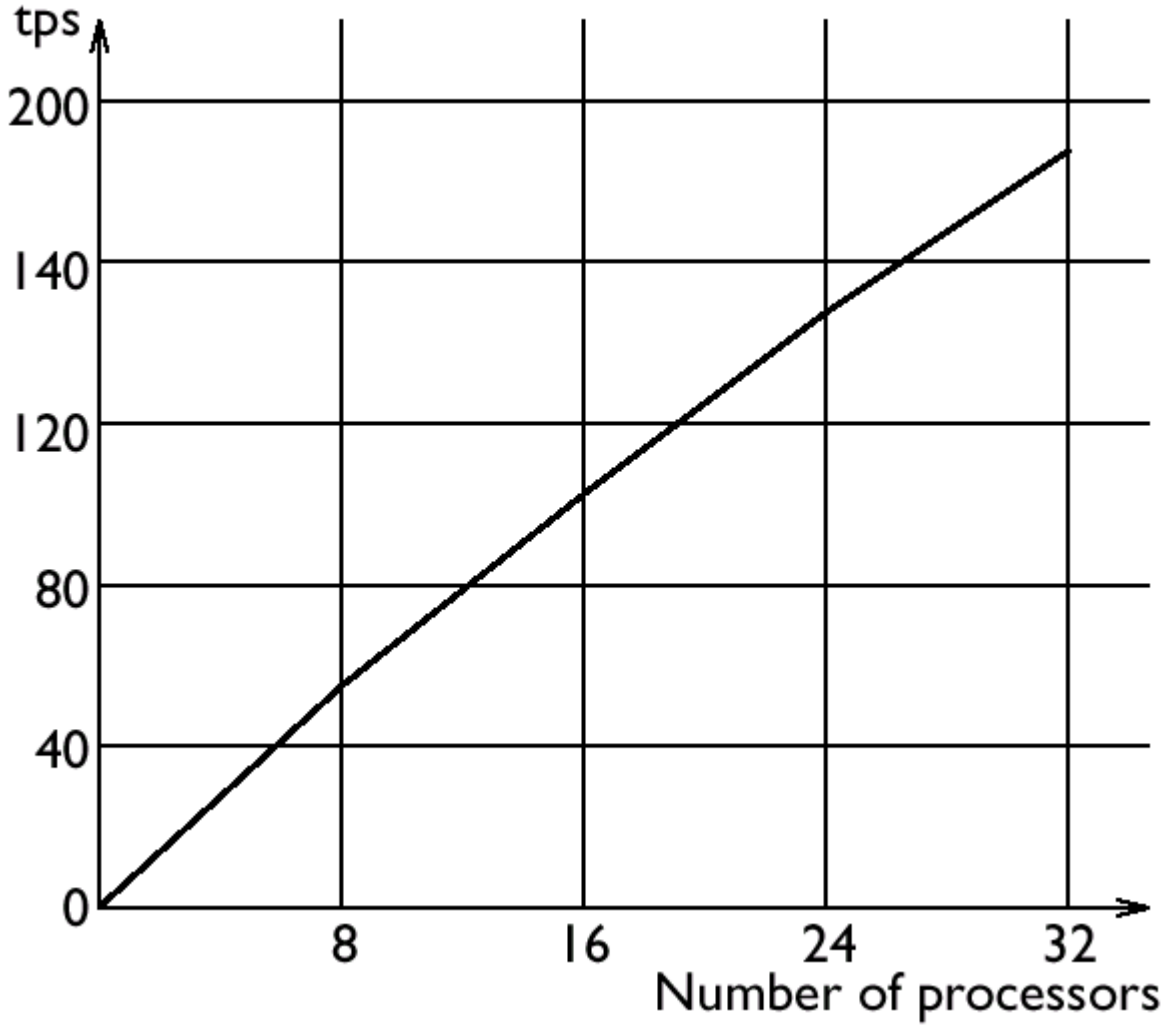
Distributed Joins

- The join of pairs of fragments corresponding to the same account number interval; the joins between the matching fragments can be carried out in parallel.
- Essential for intra-query parallelism: The parallel execution of n joins on fragments of dimension $(1/n)$ is obviously preferable to the execution of a single join that involves the entire table
- In general, when the initial fragmentation does not allow the distributed execution of the joins present in the query, data is dynamically redistributed to support distributed joins.

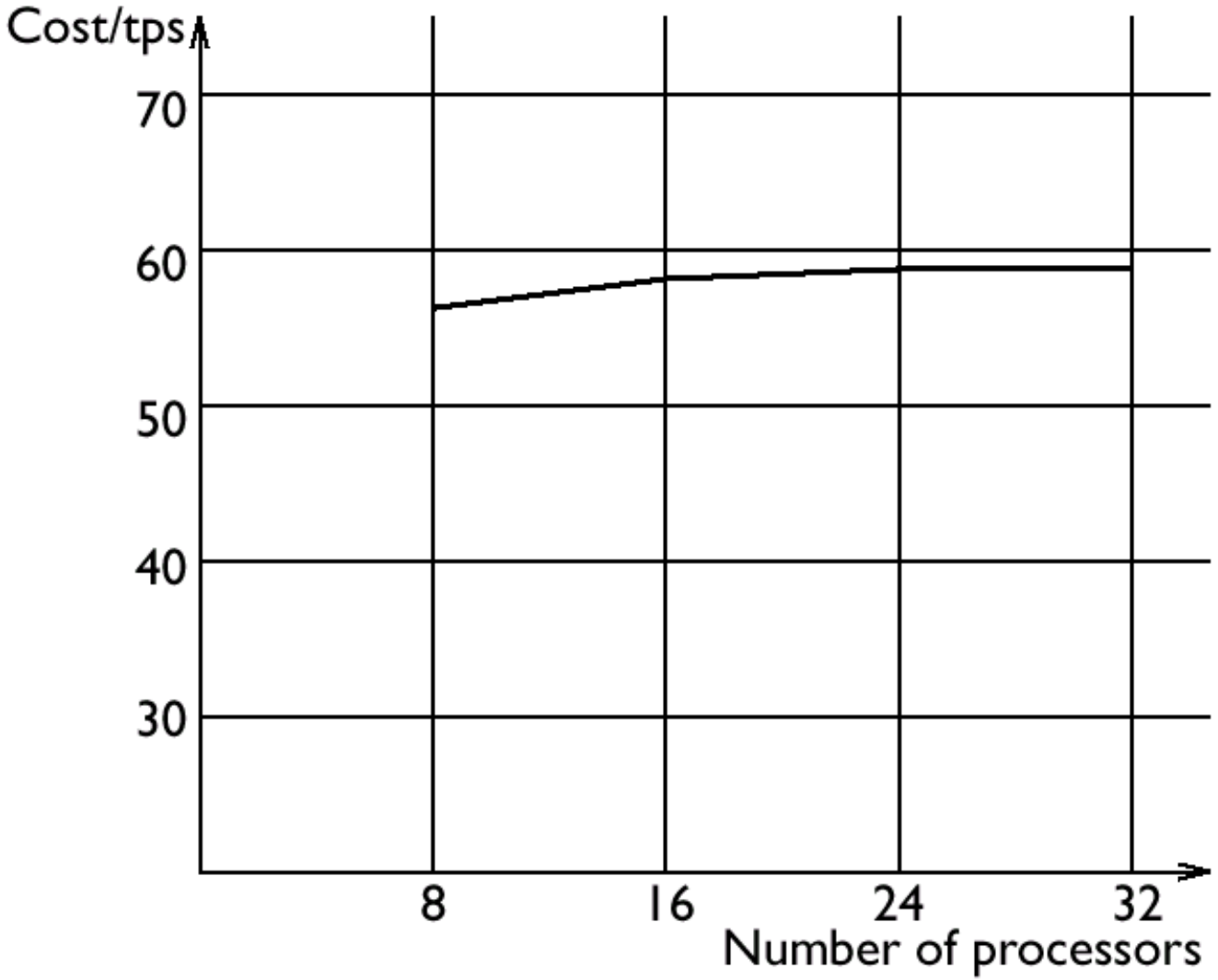
Speed-Up and Scale-Up

- The ***speed-up*** curve characterizes only inter-query parallelism and measures the increase of services, measured in ***tps*** (transactions per second), against the increase in the number of processors; In the ideal situation, services increase almost linearly against the increase in processors.
- The ***scale-up*** curve characterizes both inter-query parallelism and intra-query parallelism, and measures the average cost of a single transaction against the increase of the number of processors; In the ideal situation, the average costs remain almost constant with an increase in processors. In such cases, we say that the system 'scales well'.

A Typical Speed-Up Curve



A Typical Scale-Up Curve



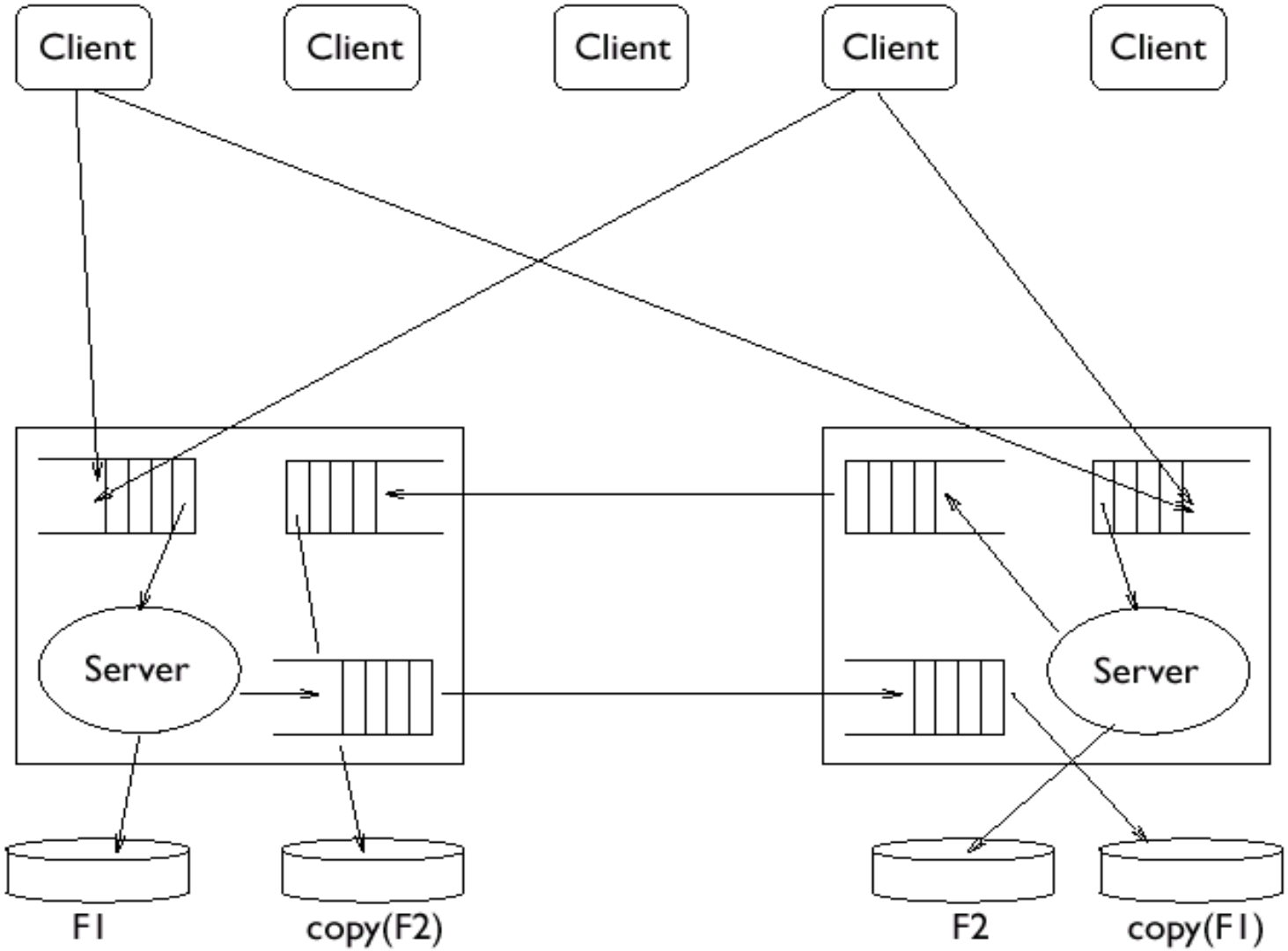
Transaction Benchmarks

- Specific tests for measuring the efficiency of (possibly parallel) DBMS; used to produce speed-up and scale-up curves under standard workload conditions.
- Standardized by TPC (Transaction Processing Performance Council), a committee of about thirty suppliers of DBMSs and transaction systems
- Three main benchmarks (TPC-A, TPC-B and TPC-C) respectively for OLTP, mixed, and OLAP applications. Can refer to a mainframe-based, client-server, or parallel architecture
- Parameters of benchmarks: transaction code, size of the database, the method used for generating data, the distribution of the arrivals of transactions, also the techniques for measuring and auditing the benchmark

Replicated Databases

- Data replication is an essential service for the creation of many distributed applications
- Provided by products called ***data replicators***, whose function is to maintain consistency among copies. They operate transparently to applications running on the DBMS server
- In general, there is one ***main copy*** and various ***secondary copies***, and updates are propagated asynchronously (without the two-phase commit protocol)
- Propagation is *incrementa*l when it is based on data variations, sent from the main copy to the secondary copy
- The use of replication makes a system less sensitive to failure, because if the main copy is unavailable it is possible to use one of its copies

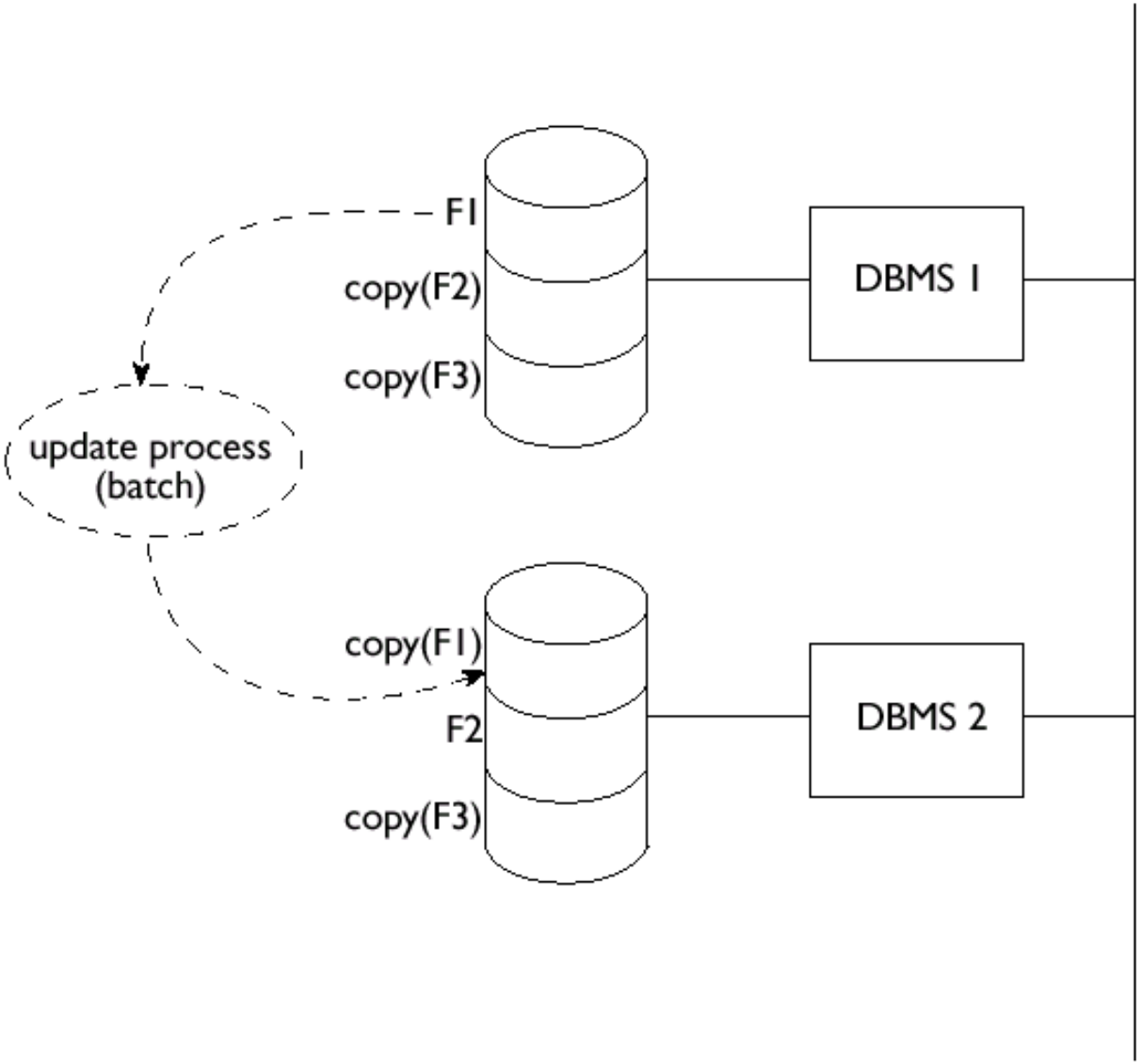
Typical Architecture for Data Replication



Discussion of the Architecture

- The architecture has two identical sites, each managing the entire database; half is the main copy and the other half is the secondary copy.
- All transactions are sent to the main copy and then possibly redirected to the secondary copy.
- Each ‘access point’ to the system is connected to both sites.
- In the case of a failure for one site, the system commutes almost instantly all the transactions to the other site, which is powerful enough to sustain the entire load.
- When the problem is resolved, the replication manager restores the data transparently and then resets the two sites to normal operations.
- Specialized for high availability.

Example Architecture for Data Replication



The Tandem Information System

- An application created by Tandem in the mid-eighties; had ten factories in various parts of the world, each responsible for the production of a specific part of the architecture of a computer.
- Tables representing the available parts in the company were fragmented to reflect the physical distribution of the parts, and then allocated to the nodes, co-located with a factory, in a redundant way: the main copy of each fragment was on the node responsible for the production process of the parts described in that fragment, while secondary copies were replicated in all other nodes
- The replication manager acted periodically, by collecting a batch of modifications on a given fragment and applying them asynchronously to all the other fragments

Advanced Replication Mechanisms

- ***Symmetrical replication*** allows modifications on any copy, with a 'peer-to-peer' situation among the copies:
  - ✓ It is possible to introduce conflicts, in that two copies of the same information are managed in a concurrent way *without* concurrency control;
  - ✓ Techniques are capable of revealing inconsistencies after their occurrence; repair is application-specific.
- ***Disconnected replication***: used with *mobile* systems, in which the connection with the database can be broken; for example, a salesperson can connect to the database in order to download data on merchandise and upload orders received. The salesperson is normally disconnected from the database and accepts transactions on the copy; the copy is 'reconciled' with the main copy at the end of the sale activity.

Research Directions for Multidatabases

- Consider a company database:

  `Cust(name,addr,phone)`

  `Sales(custName,prod#,price,amount,date)`

  `Prod(prod#,name,price,inStock)`

- Each salesperson leaving for a trip downloads parts of the `Prod`, `Sales` and `Cust` relations. On their trip, they update customer, and sales information.
- Each of these databases evolves autonomously from the original, and there is no global manager. However, we'd like to enforce **coordination rules**, such as:

  "Updates to a customer address must be propagated to other databases"

Coordination Rules

- These are inter-database soft constraints. They are checked every time there is an updated to one of the relevant databases, and are enforced through some protocol.
- Examples:
  - ✓ `Master:Cust(n,x) and SalesN:Cust(n,y)`

    `—> x=y)  propagate last`

    /* the latest addr is propagated to the other database */
  - ✓ `Master:Prod(p#,p) and SalesN:Prod(p#,p')`

    `—> p=p')  propagate (Master->SalesN)`

    /* the Master copy prices are always propagated to the other databases, not the other way around */

More Coordination Rules

- ✓ `Maria:TravelB=x and Paolo:TravelB=y and Fausto:TravelB=z —> x+y+z≤15MLit`

  `equi-distribute`

/* if their total budget is x > 15MLit, reduce each budget by (x - 15ML)/3 */

- ✓ `Master:Prod(p#,n) and SalesN:Prod(p#,n')`

  `undo`

/* no updates allowed to product names */

- ✓ `Master:Prod(p#,.) == SalesN:Prod(p#,.)`

  `propagate (Master->SalesN)`

/* propagate added or deleted Product tuples in the Master database */

Compatibility Rules

- They define correspondences between different databases at two levels:
  - ✓ Constant to constant, e.g., ‘one’ — ‘uno’;
  - ✓ Relational signature to relational signature, e.g.,

    `Cust(name,addr)` — `Customer(nm,ad)`
- Of course, such correspondences can be one-to-one, one-to-many or many-to-many.

A Data Model for Multidatabases

- A Multidatabase system consists of one or more databases and a set of coordination rules.
- Operations that can be performed on the system include:
  - ✓ Add or delete a database;
  - ✓ Update or query a database;
  - ✓ Add, delete or update a coordination rule.
- Each database 'knows' ('is acquainted with') only with databases it shares coordination and compatibility rules.
- Queries are local; global queries can be expressed by using compatibility rules:

  "Give me all customers in all acquainted databases"

  fetches customers from all databases in the transitive closure of the aquaintaince relation wrt this database.

Research Problems

- A formal semantics to the Multidatabase Model (note: hard problem, standard Logic techniques assume a global model which serves as interpretation of whatever you are trying to formalize.)
- Efficient global query processing techniques (possibly exploiting parallelism.)
- A formal transaction model for coordination rules, supporting ‘soft’ enforcement mechanisms.
- Efficient implementation techniques for coordination rule enforcement.
- …more...