These slides are for use with

Database Systems

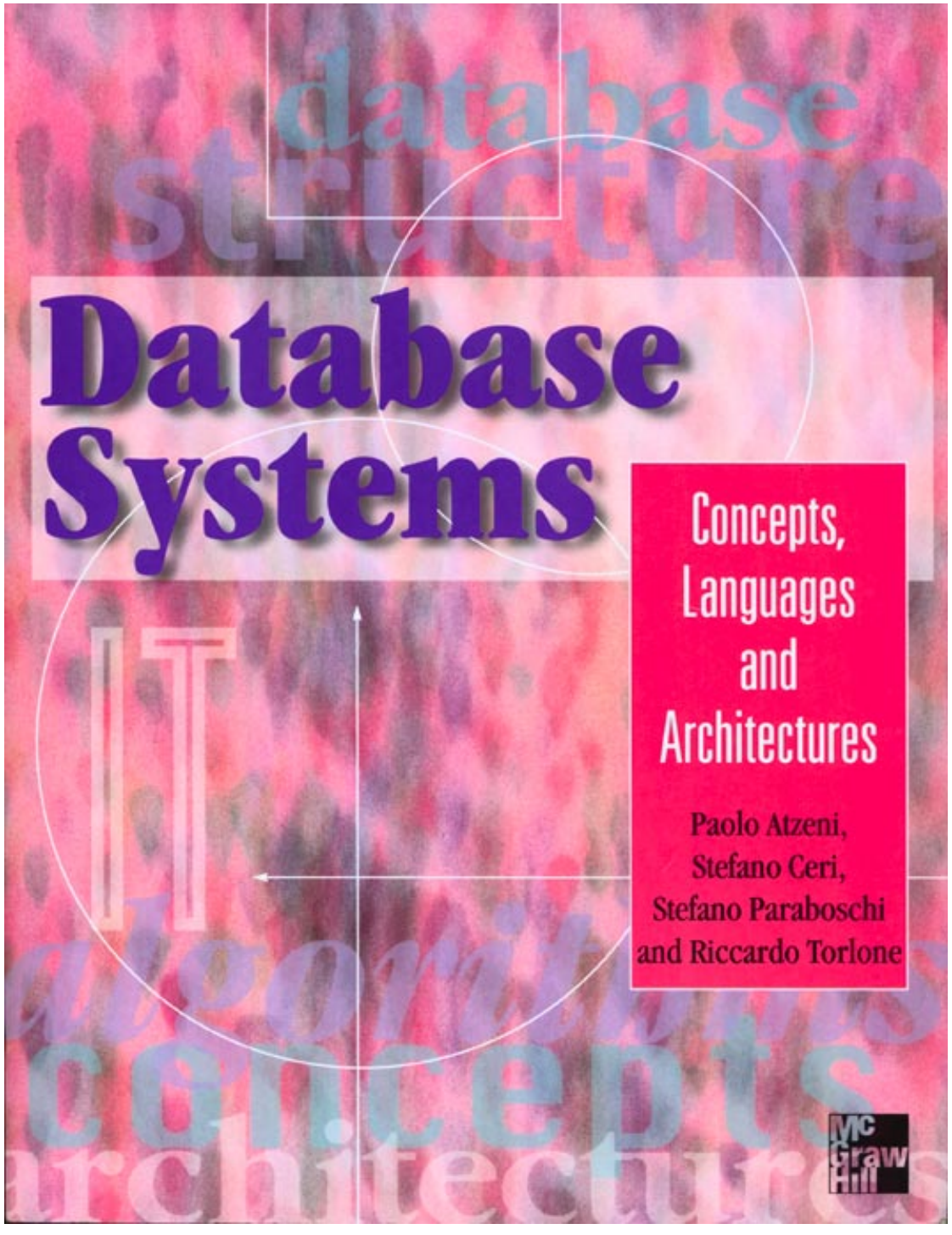
Concepts, Languages and Architectures

Paolo Atzeni • Stefano Ceri • Stefano Paraboschi • Riccardo Torlone

To view these slides on-screen or with a projector use the arrow keys to move to the next or previous slide. The return or enter key will also take you to the next slide. Note you can press the 'escape' key to reveal the menu bar and then use the standard Acrobat controls — including the magnifying glass to zoom in on details.

To print these slides on acetates for projection use the escape key to reveal the menu and choose 'print' from the 'file' menu. If the slides are too large for your printer then select 'shrink to fit' in the print dialogue box.

Press the 'return' or 'enter' key to continue . . .



Chapter 10

Distributed architectures

Click here for help

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.

Separation of functionalities

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

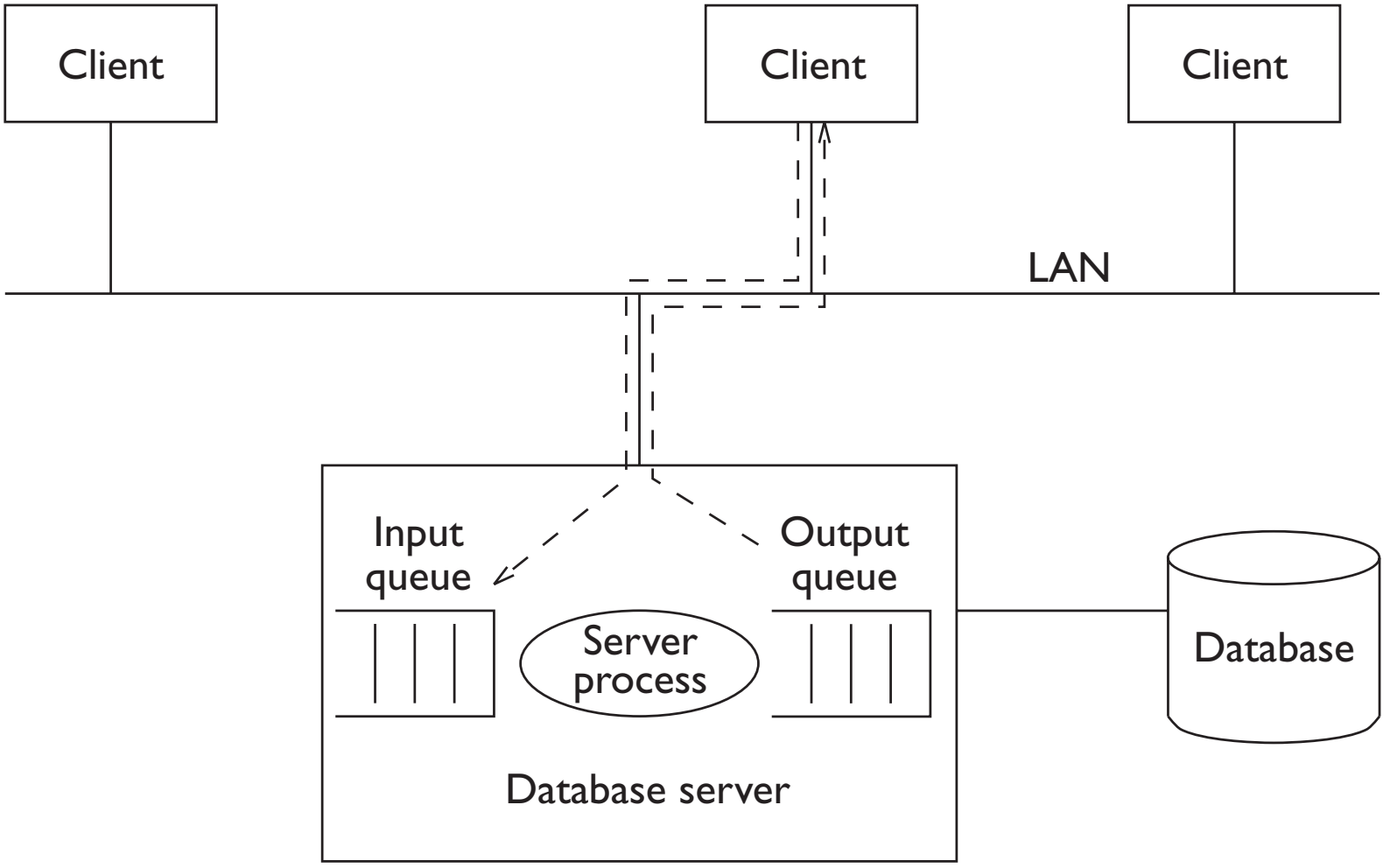
Properties of highly interactive systems

- *Portability* denotes the possibility of transporting programs from one environment to another (and it is thus a typical property of *compilation time*)
  - Facilitated by language standards (e.g.: SQL-2, SQL-3)
- *Interoperability* denotes the ability of interacting between heterogeneous systems (and it is thus a typical property of *execution time*)
  - Facilitated by standard data access protocols, including *Database Connectivity* (*ODBC*) and *X-Open Distributed Transaction Processing* (*DTP*)

Client-server architecture

- **Client-server**: a general model of interaction between software processes, where interacting processes are sub-divided among *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.

Client-server architecture



Client server architecture and data management

- In data management, allocation of client and server processes to distinct computers is now widespread, because:
  - The functions of client and server are well identified
  - They give rise to a convenient separation of design and management activities
- SQL offers an ideal programming paradigm for the identification of the ‘service interface’
  - SQL queries are formulated by the client and sent to the server
  - The query results are calculated by the server and returned to the client
  - The standardization, portability and interoperability of SQL allows the construction of client applications that involve different server systems

Allocation of servers and clients 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 architecture

- Often, the server that manages such requests is *multi-threade*d:
  - Behaves like a single process that works dynamically on behalf of different 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 architecture

- *Two-tier* architecture: 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* architecture: a second server is present, known as the *application serve*r, responsible for the management of the application logic common to many clients
  - The client is named *thin-clien*t; it is responsible only for the interface with the final user. Can be deployed using the browser technology

Distributed databases

- A *distributed database* is a system in which at least one client interacts 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:
  - Enterprises are structurally distributed, distributed data management allows the distribution of data processing and control to the environment where it 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, with great flexibility and modularity
  - Although they are more vulnerable to failures due to their structural complexity, they support ‘graceful degradation’ (respond to failures with a reduction in performance but without a total failure)

Classification of applications

- Based on the type of DBMS involved:
  - Homogeneous DDB: When all the servers have the same DBMS
  - Heterogeneous DDB: 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 co-operation

- 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 normally disjoint
  - In *vertical fragmentation*, each fragment R_i has a subset of the schema of *R* (as after a projection applied on *R*). Vertical fragments include 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
- A technique for data organization that allows efficient data distribution and processing

Example

- Consider the relation:
  - EMPLOYEE (Empnum, Name, Deptnum, Salary, Taxes)
- Horizontal fragmentation
  - EMPLOYEE1 = $\sigma_{Empnum<=3}$ EMPLOYEE
  - EMPLOYEE2 = $\sigma_{Empnum>3}$ EMPLOYEE
- Reconstruction requires a union:
  - EMPLOYEE = EMPLOYEE1 U EMPLOYEE2
- Vertical fragmentation:
  - EMPLOYEE1 = $\Pi_{EmpNum,Name}$EMPLOYEE
  - EMPLOYEE2 = $\Pi_{EmpNum,DeptName,Salary,Tax}$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

Example of 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 schemas

- 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 the 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-redundan*t, when each fragment or relation is allocated to a single server
  - *redundan*t, when at least one fragment or relation is allocated to more than one server

Transparency levels

- There are three significant levels of transparency: transparency of fragmentation, of allocation and of language
- In *absence of transparency*, each DBMS accepts its own SQL 'dialect': the system is heterogeneous and the DBMSs do not support a common interoperability standard
- Assume:
  - SUPPLIER(SNum,Name,City)
- with two 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

- On 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

- On this level, 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

- On this level the programmer must indicate in the query both the structure of the fragments and their allocation
- Queries expressed at a higerh level of transparency are transformed to this level by the distributed query optimizer, aware of data fragmentation and allocation
- 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 the *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
  - The 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
  - Requires the two-phase commit protocol
- *Distributed requests* are arbitrary transactions, in which each SQL command can refer to any DBMS
  - Requires a distributed optimizer

Typical transaction: fund transfer

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

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

- Comment: It is 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.

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

  $t_2 : r_{22}(y)\ w_{22}(y)\ r_{21}(x)\ w_{21}(x)$

- The local serializability within the schedulers is not a sufficient guarantee of serializability. Consider the two schedules at nodes 1 and 2:

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

  $S_2 : r_{22}(y)\ w_{22}(y)\ r_{12}(y)\ w_{12}(y)$

- These are locally serializable, but their 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

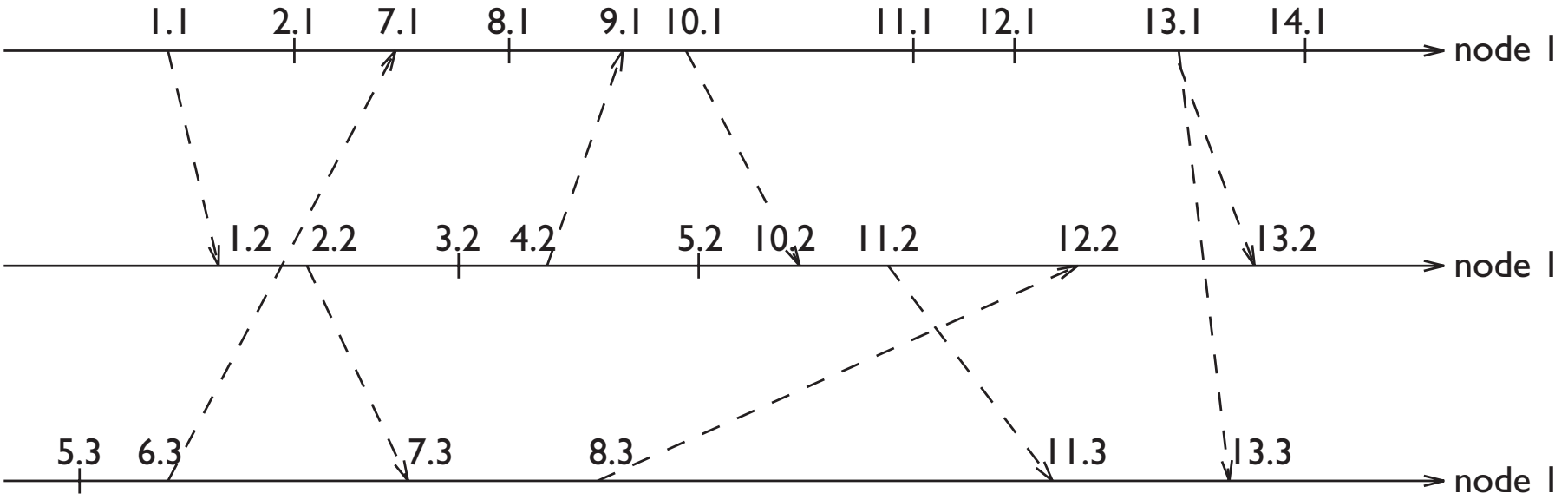
Global serializability

- 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 produced at each node
- The following properties are valid.
  - If each scheduler of a distributed database uses the two-phase locking method on each node and carries out the commit action when all the sub-transactions have acquired all the resources, then the resulting schedules are globally conflict-serializable
    - This is imposed by 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 method for assigning timestamps

- The *Lamport method* for assigning timestamps reflects the precedence among events in a distributed system
- A timestamp is a number characterized by two groups of digits
  - The least significant digits identify the node at which the event occurs
  - The most significant digits identify the events that happen at that node. They can be obtained from a local counter, which is incremented at each event
- Each time 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 the increasing of the local counter on the receiving node

Example of assignment of timestamps using the Lamport method



Distributed deadlocks

- Distributed deadlocks can be due to 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

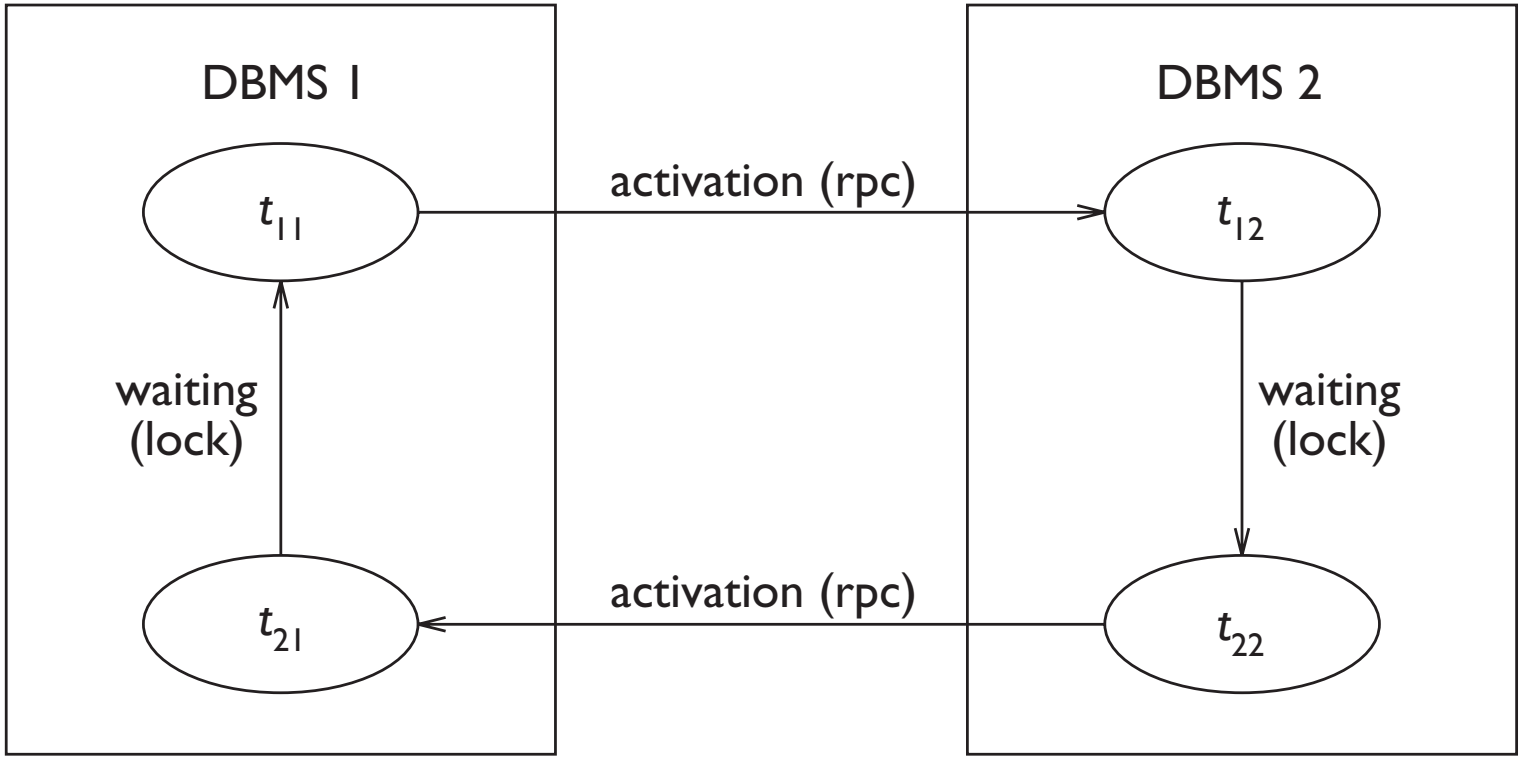
- Assume that sub-transactions are activated by using a *remote procedure call*, that is, a synchronous call to a procedure that is remotely executed. This model allows for two distinct types of waiting
  - Two sub-transactions of the same transaction can be in waiting in distinct DBMSs as one waits 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 a distributed deadlock



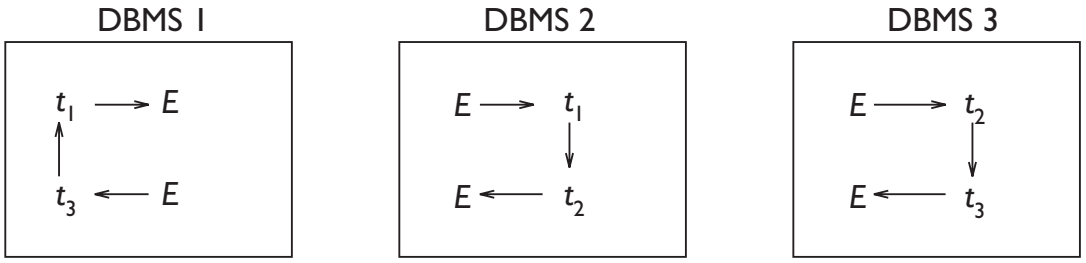
Representation of waiting conditions

- 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: $EXT < t_{12} < t_{22} < EXT$
- The general format of waiting condition is summarized using a *wait sequence*: $EXT < t_i < t_j < EXT$

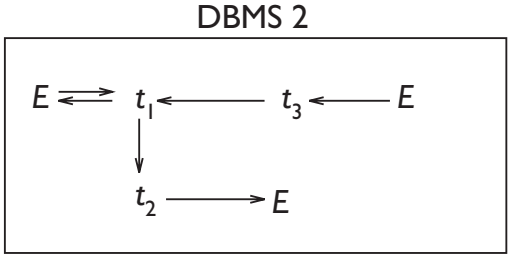
Algorithm

- The algorithm for distributed deadlock detection is periodically activated on the 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 the wait conditions on its DBMS and detects local deadlocks
  - communicates the wait sequences to other instances of the same algorithm
- To avoid the situation in which 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

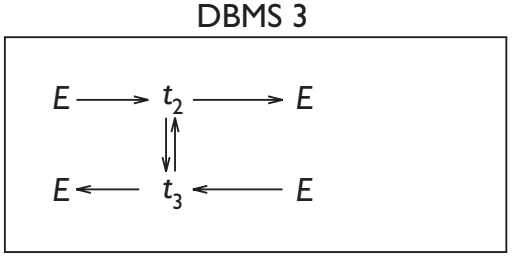
Example of a distributed deadlock detection



a. initial situation



b. first pass of the algorithm



c. second pass of the algorithm

Failures in distributed systems

- A distributed system is subject to failures, message losses, or network partitioning
- *Node failures* may occur on any node of the system and be soft or hard, as discussed before
- *Message losses* leave the execution of a protocol in an uncertain situation
  - 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.* This is a failure of the communication links of the computer network which divides it into two sub-networks that have no communication between each other
  - A transaction can be simultaneously active in more than one sub-network

Two-phase commit protocol

- *Commit protocols* allow a transaction to reach the correct commit or abort decision at all the nodes that participate in a transaction
- The *two-phase commit protocol* is similar in essence to a marriage, in that the 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 celebrant (or coordinator) is allocated to a process, called the *transaction manager* (TM)
- It 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 reaches the final decision
  - The `complete` record is written at the end of the two-phase commit protocol

New log records

- Records of RM
  - The `ready` record indicates the irrevocable availability to participate in the two-phase commit 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, by undoing the effects, without participating to the two-phase commit protocol

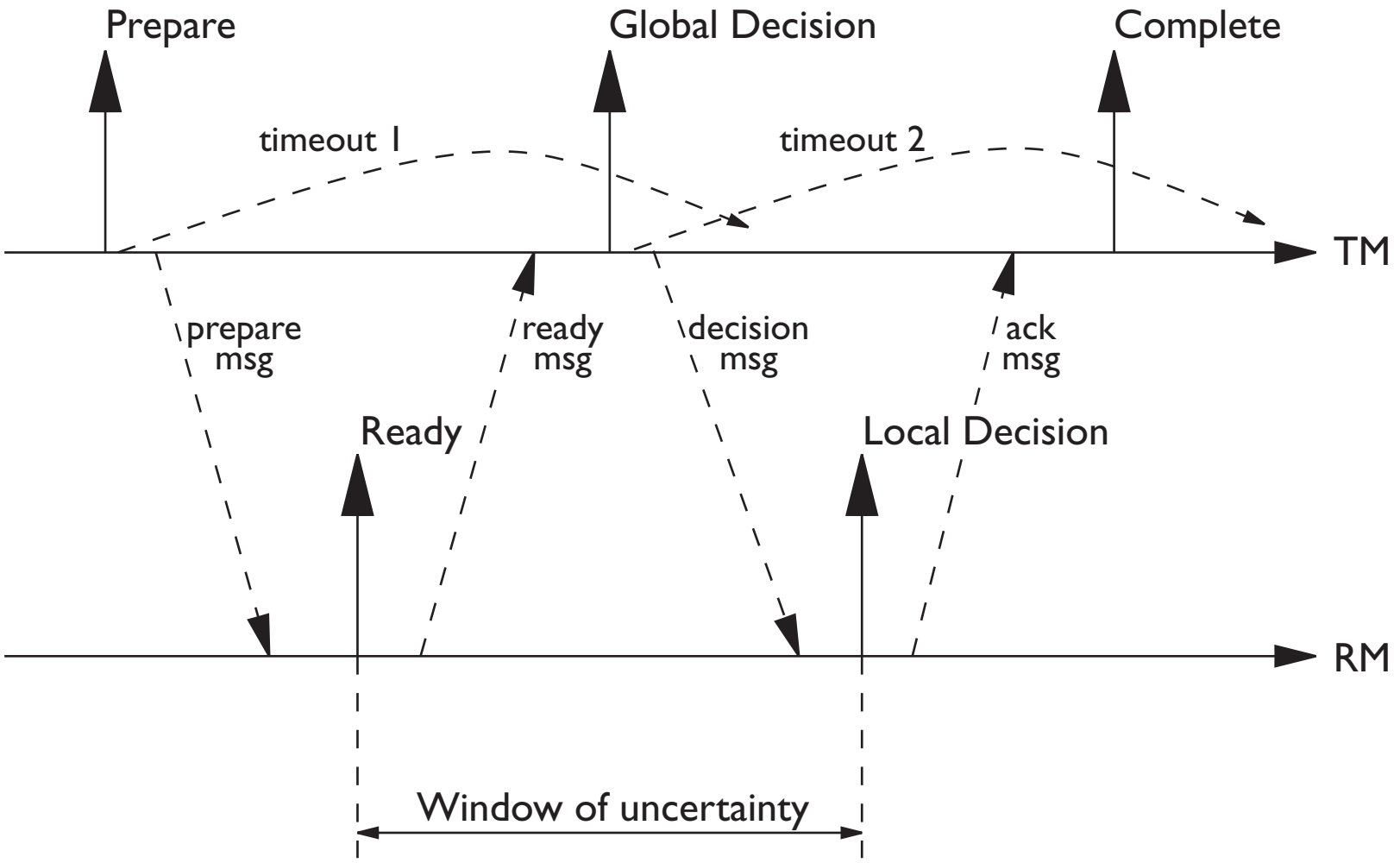
First phase of the basic protocol

- 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 the first 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

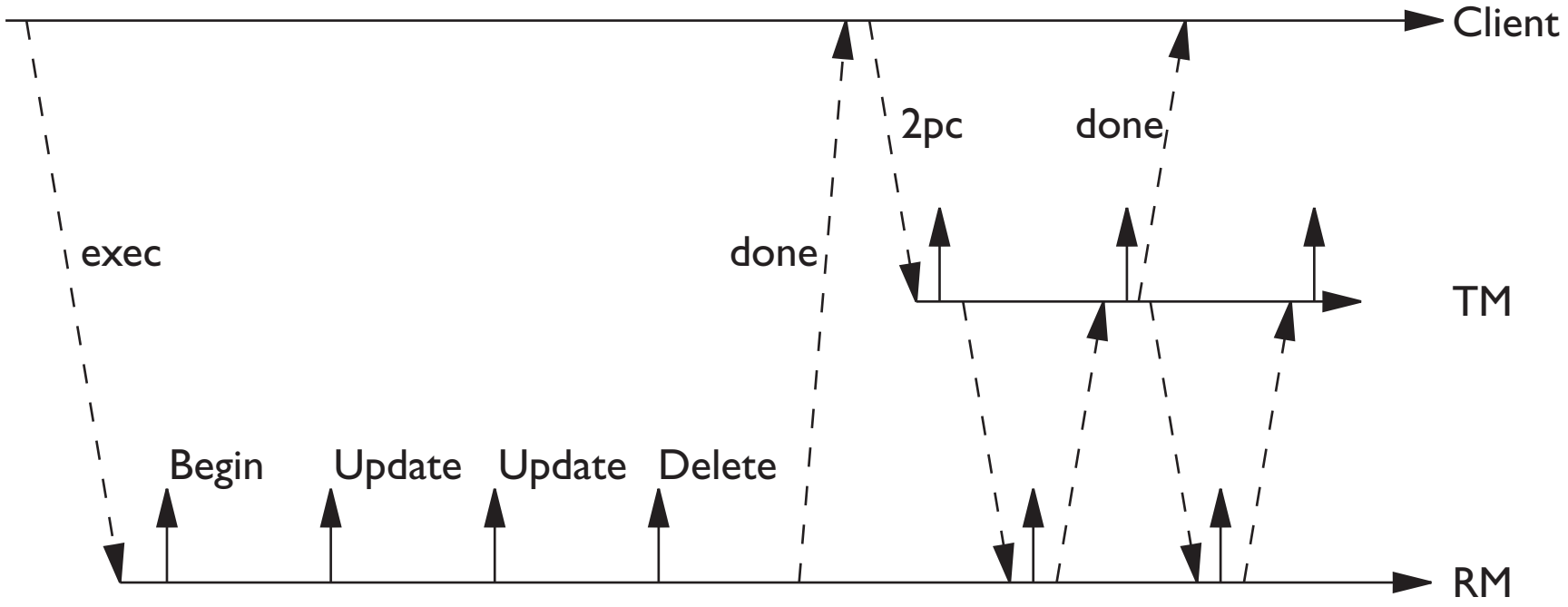
Second phase of the basic protocol

- 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 before
- 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



Two-phase commit protocol in the context of a transaction



Blocking, uncertainty, recovery protocols

- 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 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. Depends on the last record written in the log:
  - when it is an action or `abort` record, the actions are *undone*; when it is a `commit`, the actions are *redone*; in both cases, the failure has occurred before starting the commit protocol
  - when the last record written in the log is a `ready`, the failure has occurred during the two-phase commit. The participant is *in doubt* about the result of the transaction
- During the warm restart protocol, the identifier of the transactions in doubt are collected in the *ready* set. For each of them the final transaction outcome must be requested to 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` messages are 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

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 it does not 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 needs not be forced; in some systems, it is omitted. In conclusion the records to be forced are `ready`, `global commit` and `commit`

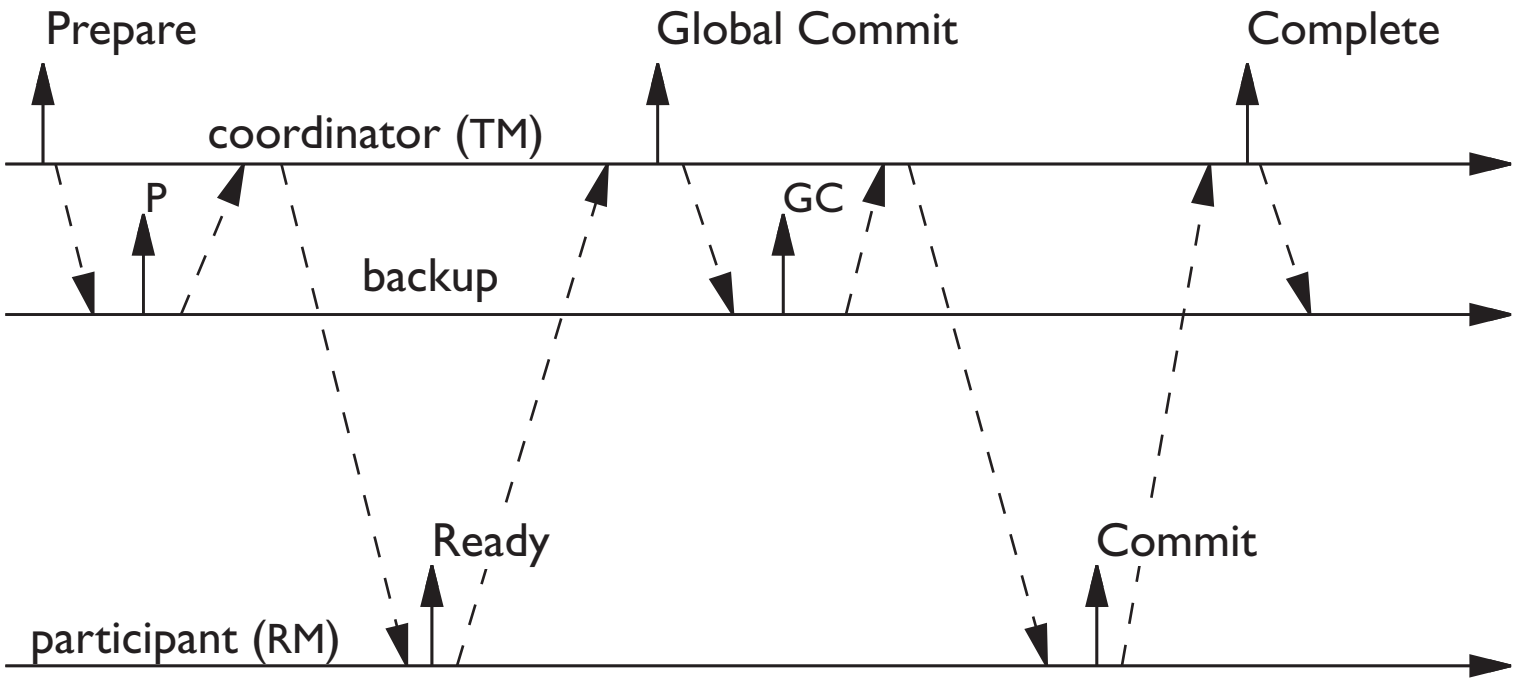
Read-only optimization

- 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

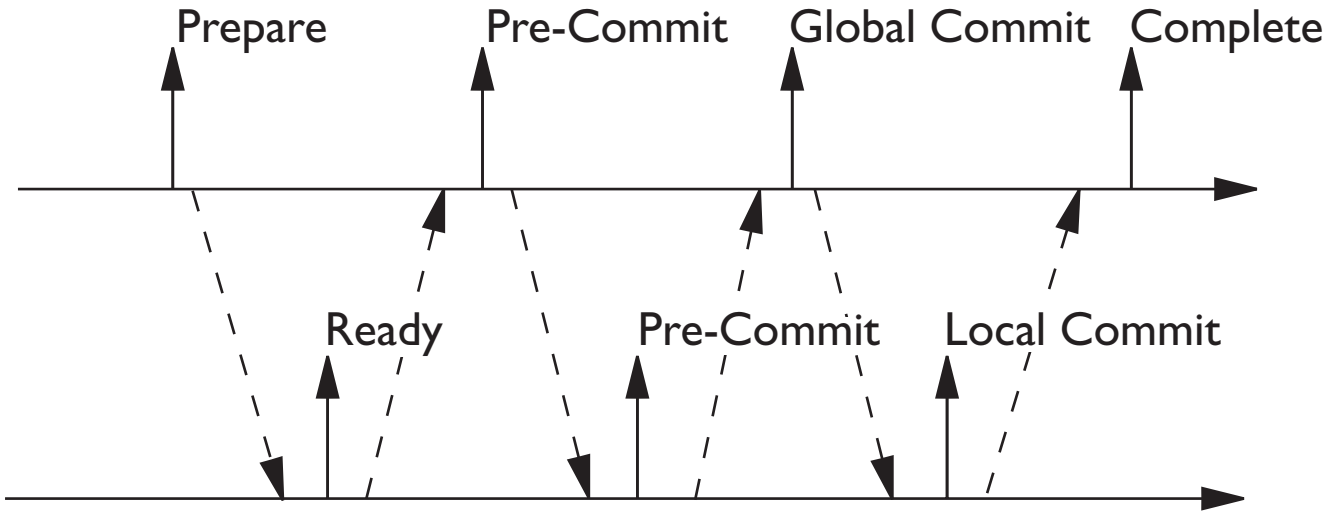
Four-phase commit protocol



Three-phase commit protocol

- The basic idea is to introduce a third pre-commit phase in 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 can have gone beyond the pre-commit condition, and thus can make the decision 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 inconveniences and has not been successfully implemented:
  - It lengthens the window of uncertainty
  - It is not resilient to network partitioning, unless with additional quorum mechanisms

Three-phase commit protocol



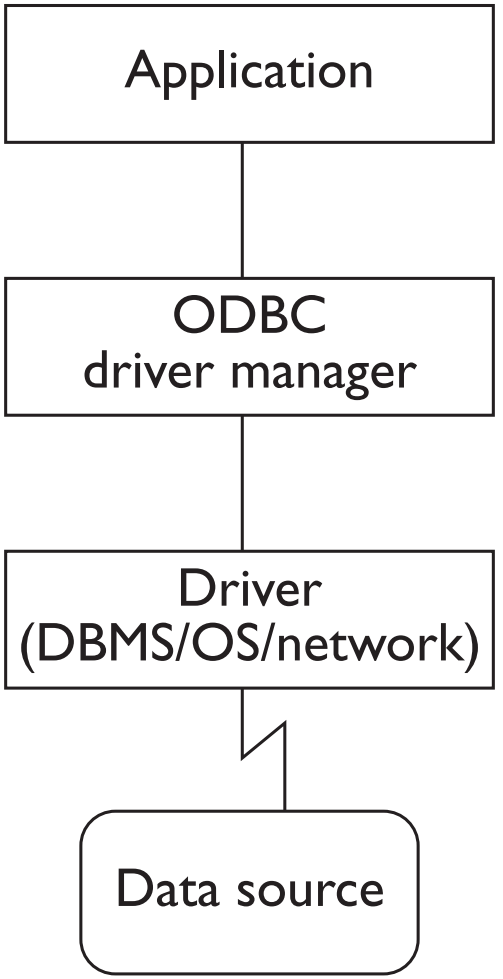
Interoperability

- Interoperability is the main problem in the development of heterogeneous applications for distributed databases
- It requires the availability of functions of adaptability and conversion, which make it possible to exchange information between systems, networks and applications, even when heterogeneous
- 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 specific driver
- ODBC does not support the two-phase commit protocol

Architecture of ODBC



ODBC Components

- The *application* issues SQL queries
- The *driver manager* loads the drivers at the request of the application and provides naming conversion functions. This software is supplied by Microsoft
- The *drivers* perform ODBC functions. They execute SQL queries, possibly translating them to adapt to the syntax and semantics of specific products
- The *data source* is the remote DBMS system, which carries out the functions transmitted by the client

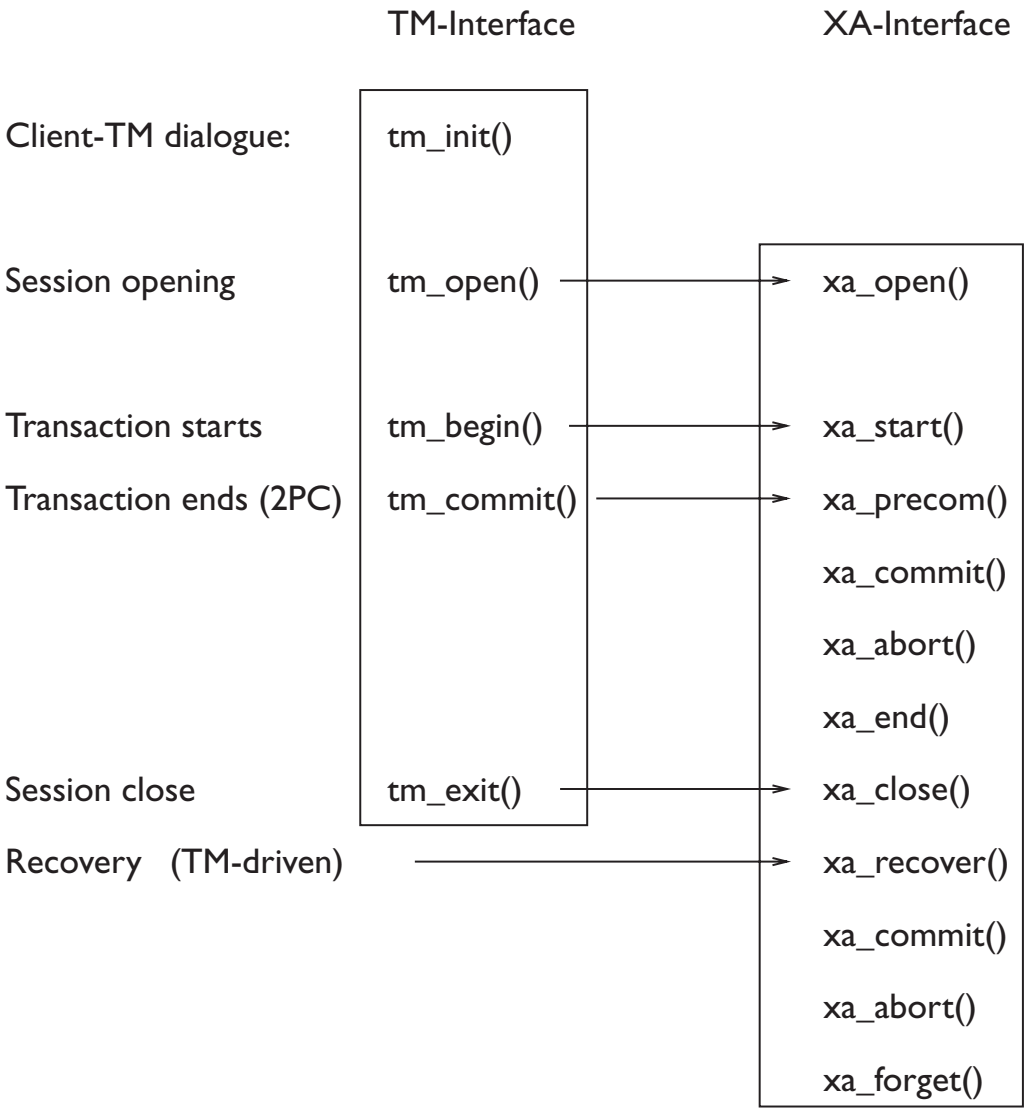
X-Open distributed transaction processing (DTP)

- A protocol that guarantees the interoperability of transactional computations 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

- RM are passive; they respond to remote procedure calls issued by the TM
- The protocol uses the two-phase commit protocol with the 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 the abort), thus allowing the release of the 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

Interactions among client, TM and server with the X-OPEN DTP protocol



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

XA Interface

- `xa_open` and `xa_close` open and close a session between TM and a given RM
- `xa_start` and `xa_end` activate and 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
  - Transactions decided by a *heuristic commi*t
  - Transactions decided by a *heuristic abor*t
- `xa_forget` allows an RM to forget transactions decided in a heuristic manner

Co-operation among pre-existing systems

- *Co-operatio*n is the capacity of the applications of a system to make use of application services made available by other systems, possibly managed by different organizations
- Needs for co-operation 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

Data-centered co-operation

- Two kinds of co-operation:
  - *process-centered co-operation*: the systems offer one another services, by exchanging messages, information or documents, or by triggering activities, without making remote data explicitly visible
  - *data-centered co-operatio*n, in which the data is naturally distributed, heterogeneous and autonomous, and accessible from remote locations according to some co-operation agreement
- We will concentrate on data-centered co-operation, characterized by data autonomy, heterogeneity and distribution

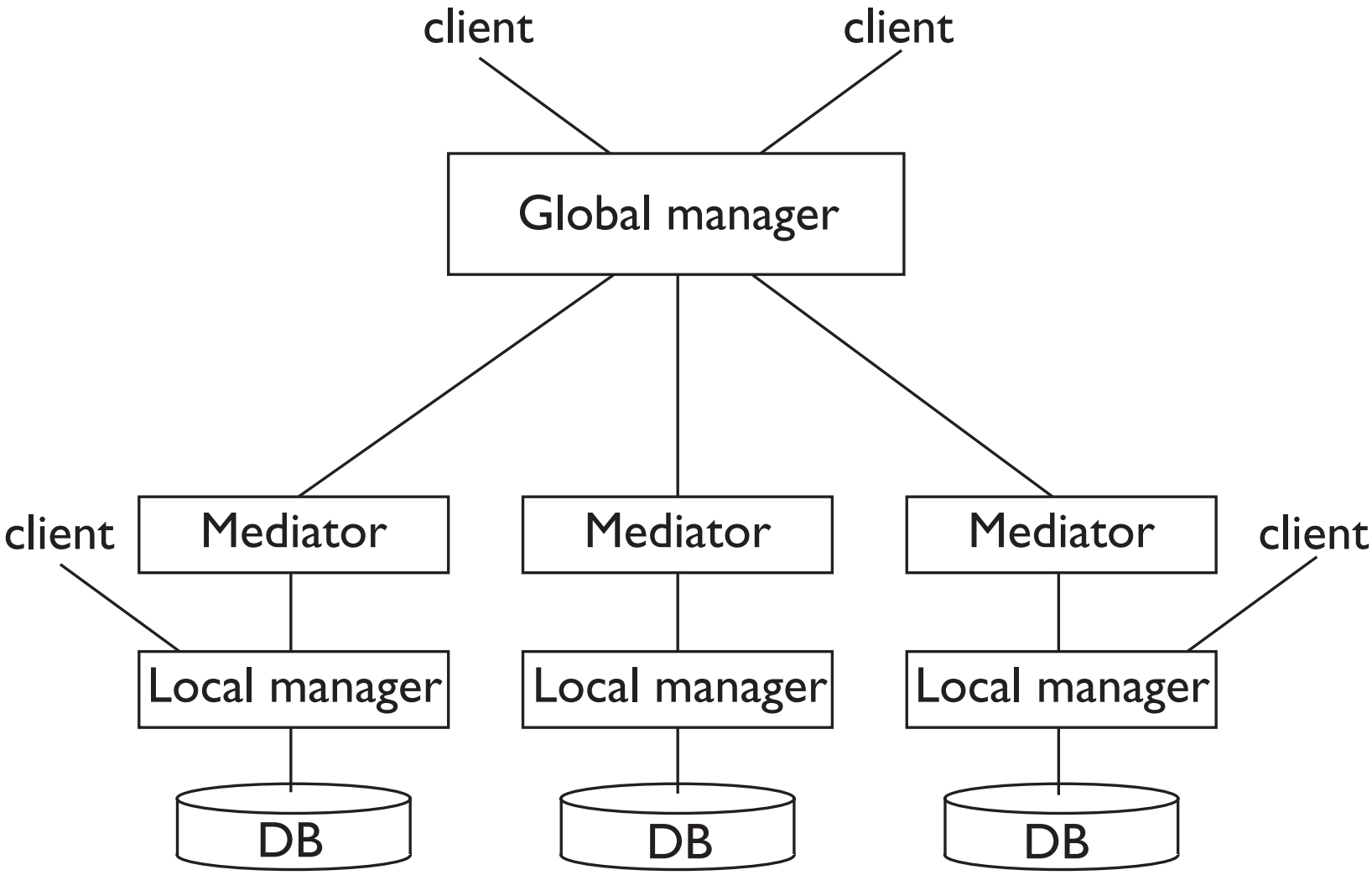
Features of data-centered co-operation

- The *transparency level* measures how the distribution and heterogeneity of the data are masked
- The *complexity of distributed operations* measures the degree of coordination necessary to carry out operations on the co-operating databases
- The *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 co-operation

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 *mediator*s, which show only the portion of database that must be exported. They make it available to a *global manage*r, which carries out the integration
- In general, data cannot be modified by means of mediators, because each source system is autonomous
- Features:
  - presents an integrated view to the users, as if the database were integrated
  - provides a high level of transparency
  - currency is also high, because data is accessed directly

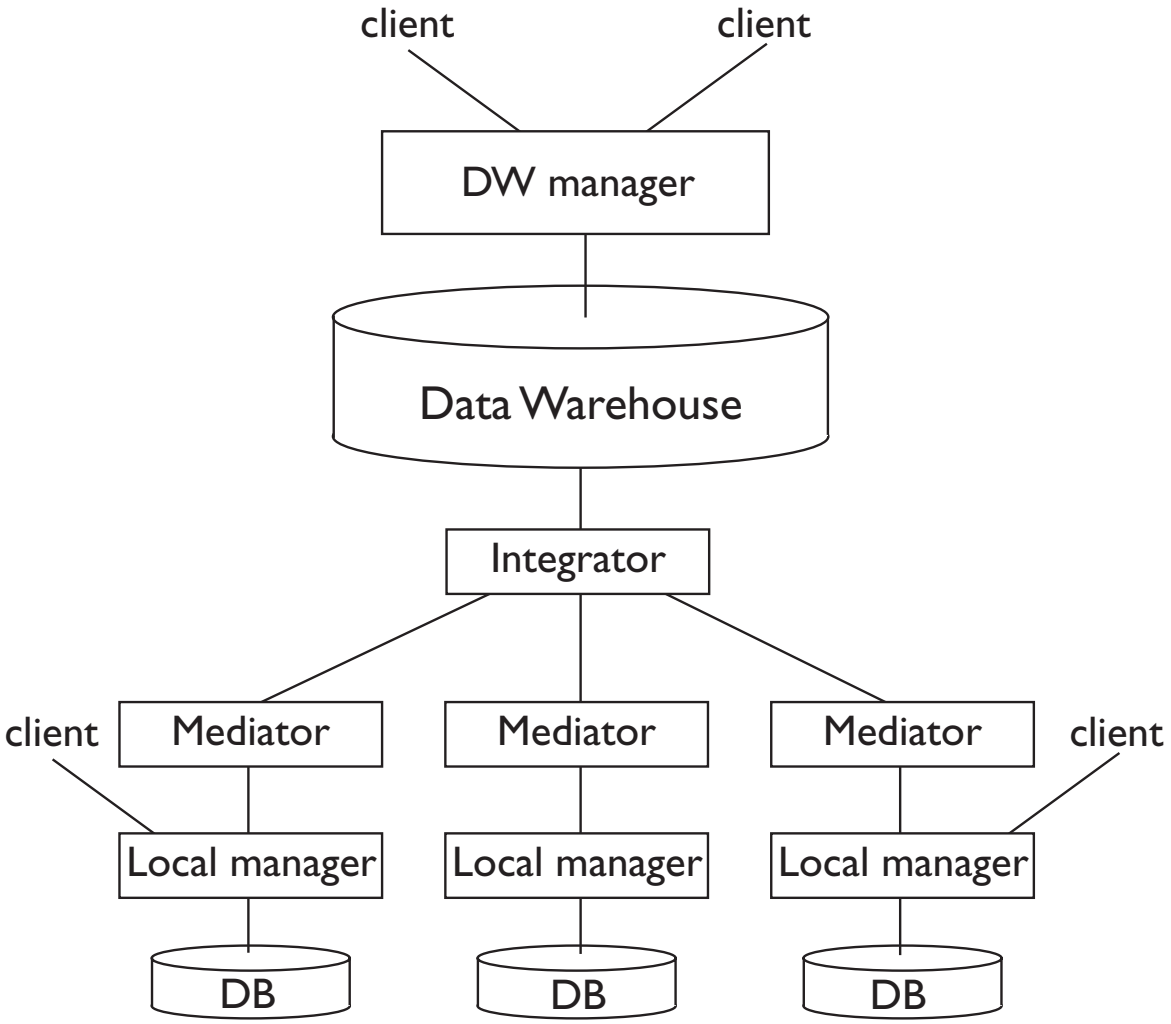
Architecture of a multi-database system



Systems based on replicated data

- They guarantee read only access to secondary copies of the information provided externally
- These may be stored in the *data warehouse*, which contains data extracted from various heterogeneous distributed systems and offers a global view of data
- Features:
  - present a high level of integration and transparency, but have a reduced degree of currency

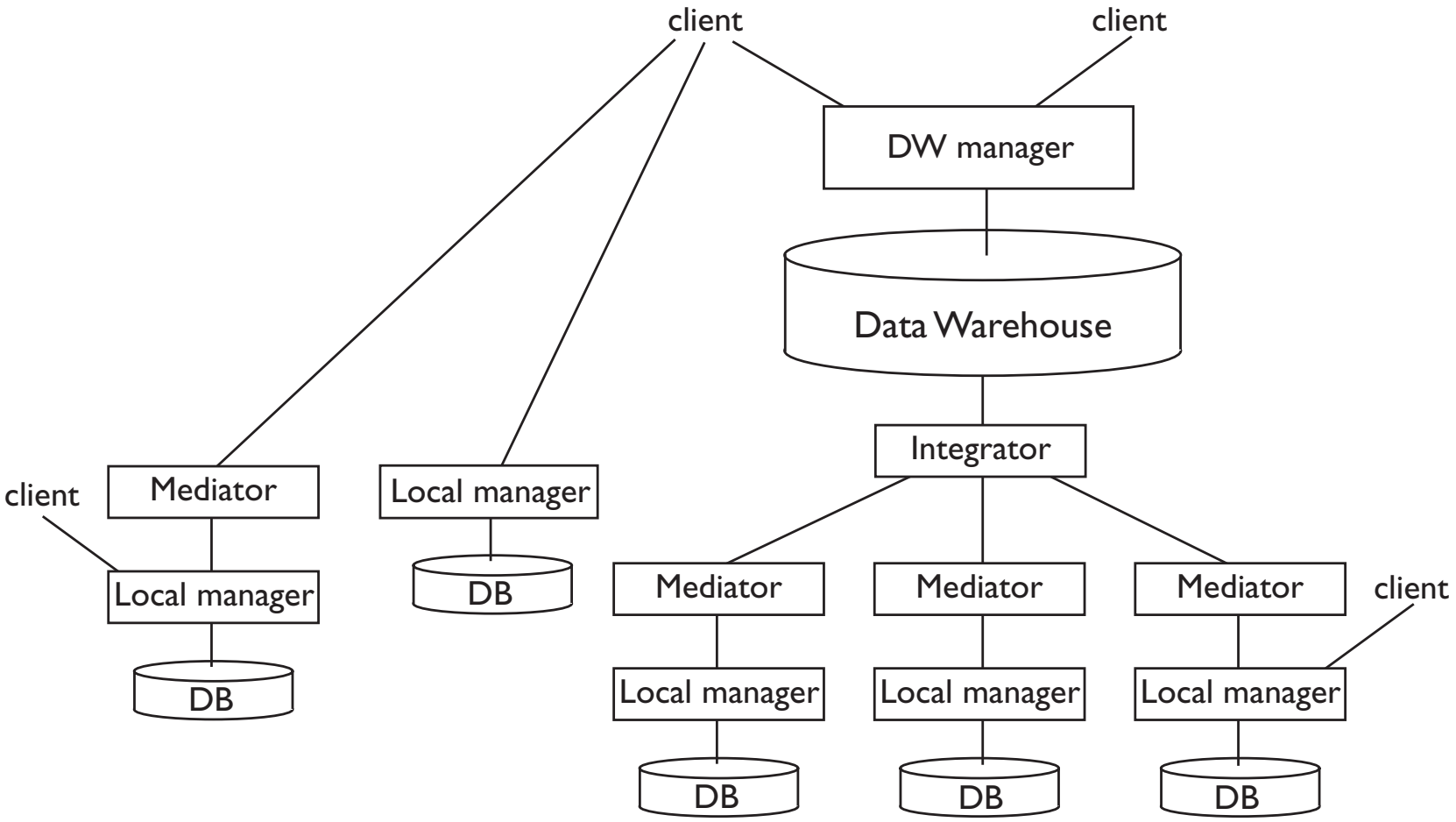
Architecture for data warehouse systems



Systems based on external data access

- 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

Architecture with external data access



Parallelism

- Was developed during the nineties along with the spread of standard multiprocessor architectures, after the failure of special architectures for databases (the so-called *database machines*) during the eighties
- Parallelism is possible with multiprocessor architectures both with and without shared memory, though with different technical solutions
- The reason for the success of parallelism in databases is that data management operations are quite repetitive in nature, and can be carried out in parallel with great efficiency
  - A complete scan of a large database can be executed using *n* scans, each on a portion of the database. If the database is stored on *n* different disks managed by *n* different processors, the response time will be approximately 1/*n* of the time required for a serial search

Inter-query parallelism

- Parallelism is called *inter-query* when it carries out different 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 carries out part 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 is normally associated with 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,TransactionType,Amount)

  Fragmented 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

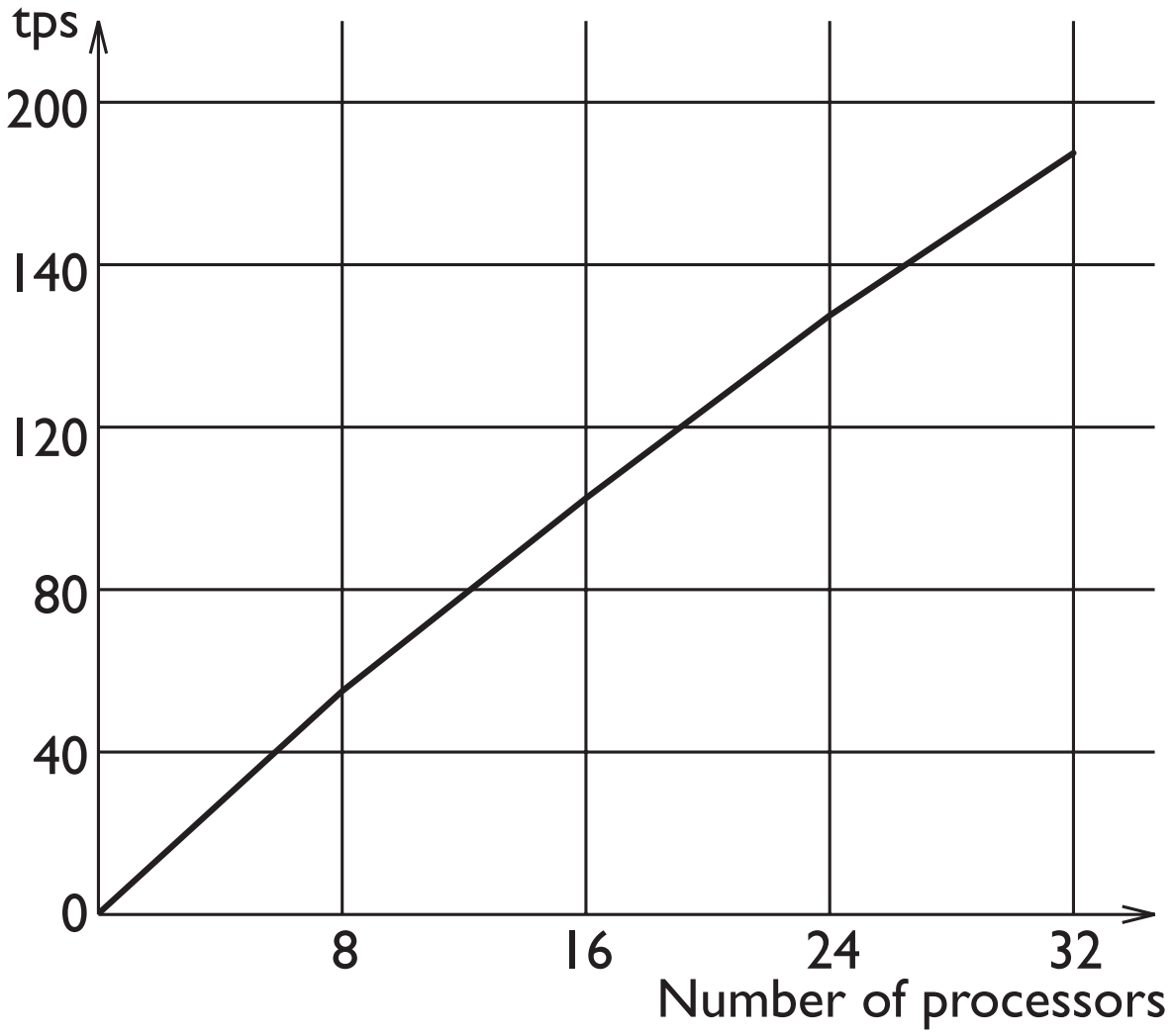
Distributed Joins

- 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

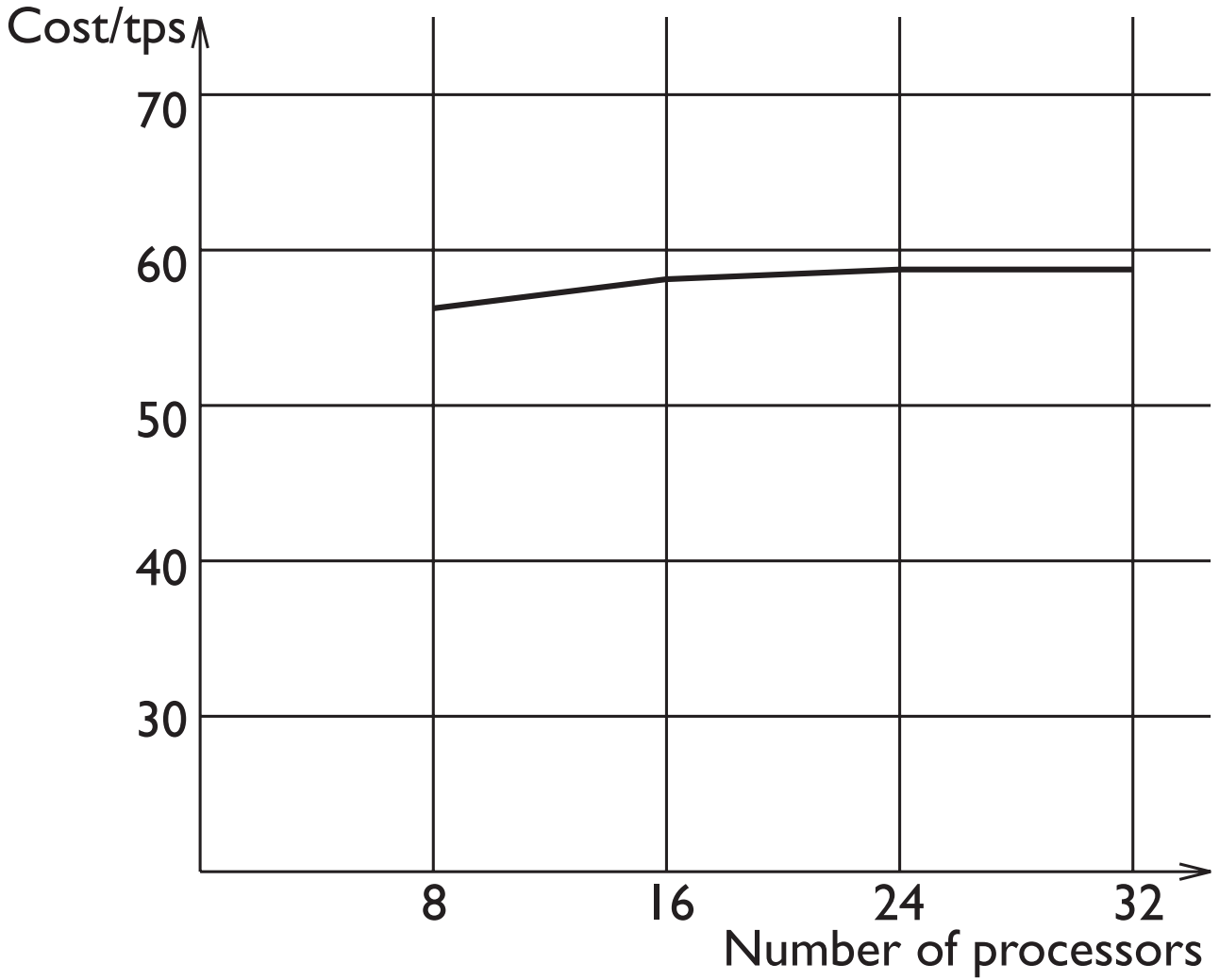
Speed-up and scale-up

- 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. We say that the system ‘scales’ well

Speed-up in a parallel system



Scale-up in a parallel system



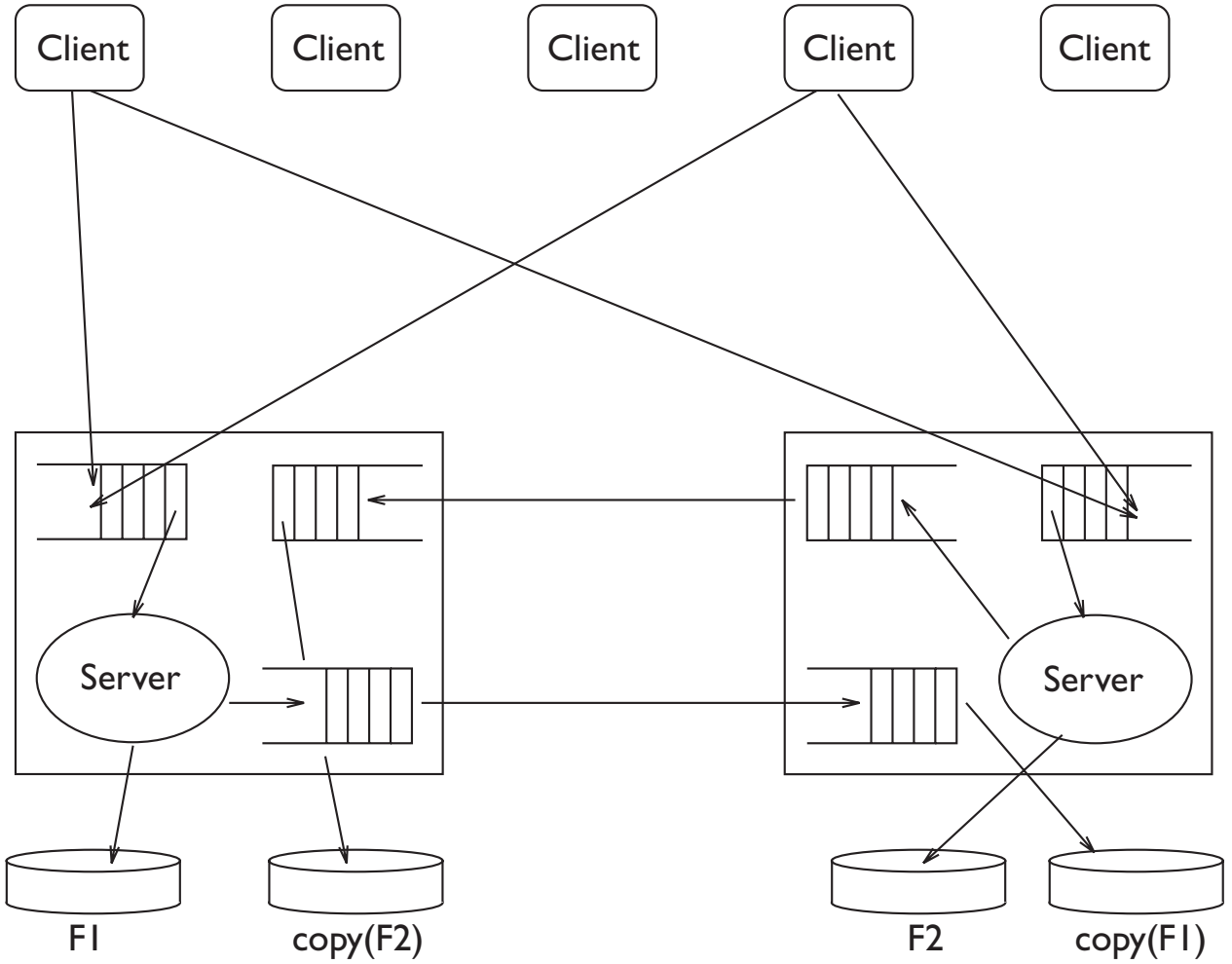
Transaction benchmarks

- Specific tests for measuring the efficiency of (possibly parallel) DBMS; used for producing speed-up and scale-up curves under standard 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:
  - The transaction code
  - The size of the database
  - The method used for generating data
  - The distribution of the arrivals of transactions
  - 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 main function is to maintain the 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

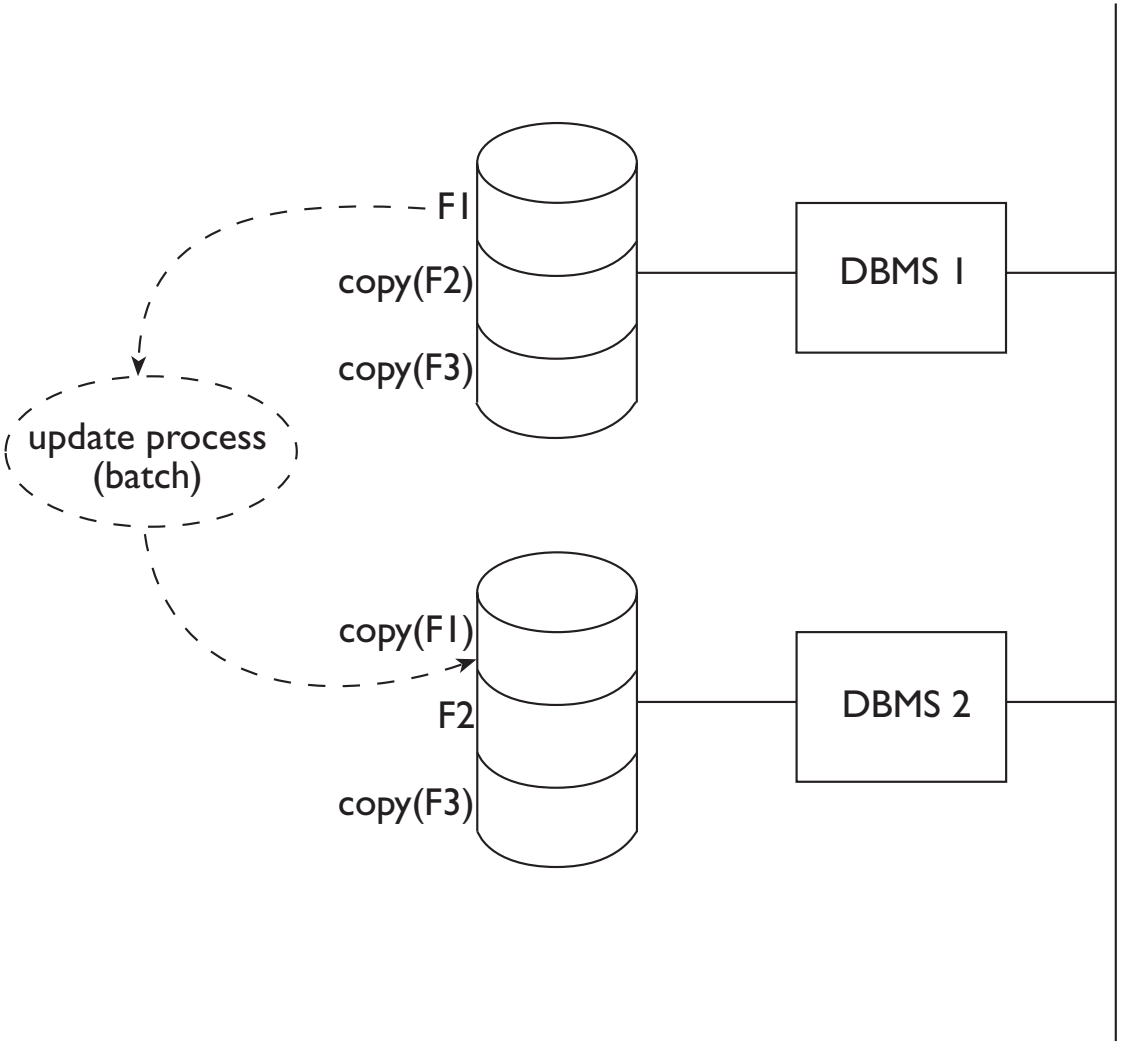
Example of architecture with replicated data



Description of the architecture

- The architecture has two identical sites. Each site manages 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 redirected to the secondary copy
- Each ‘access point’ to the system is connected to both sites
- In the case of a failure that involves only one site, the system is capable of commuting almost instantly all the transactions onto 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

Tandem information system



Description of the architecture

- An application created by Tandem towards the mid-eighties. Tandem had about 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 components, 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 components described in that fragment
  - secondary copies were replicated to 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 functions of replication managers

- *Symmetrical replicatio*n: modifications can be carried out 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*: occurs with *mobile* systems, in which the connection with the database can be broken
  - For example, a salespersons can connect to the database in order to download the availability of merchandise and upload the 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