

---

# Conception of Information Systems

## Lecture 6: Transaction Monitors

19 April 2005

<http://lsirwww.epfl.ch/courses/cis/2005ss/>

## Outline

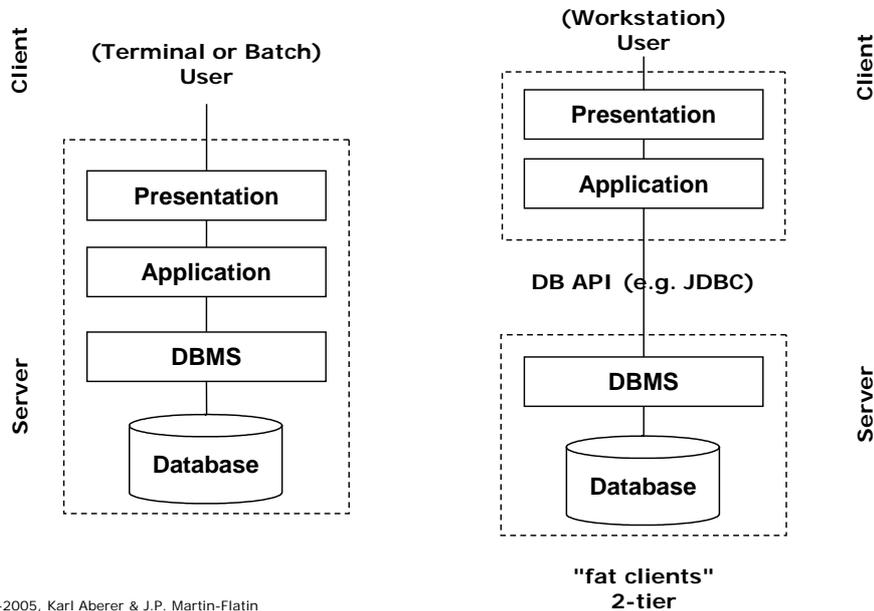
---

1. Client-Server Information Systems
2. Overview of Transaction Monitors
3. Transactional RPCs and Persistent Message Queues
4. Distributed Transaction Processing
5. An Architecture for Transaction Monitors: X/Open DTP

Transaction monitors (a.k.a. transaction processing monitors, a.k.a. TP monitors) were the first kind of middleware to support distributed transaction processing, and thus also the first kind of application servers. They were (originally) designed for environments with very high processing demands, such as banks or airlines, that could not be properly supported by 2-tier architectures and by DBMS alone.

Architectures using transaction monitors as middleware for transaction control are also called TP-heavy transaction processing environments, because this solution is much heavier to implement.

## 5.1 Client-Server Information Systems



The role of application programs in information systems underwent a continuous evolution together with the general development of computing infrastructure. The development of information systems was for a long period based on completely centralized mainframe systems, which were accessed by users by means of "dumb" terminals, i.e. pieces of equipment that were only able to transfer ASCII input and output.

In the early 1980s, a major technological revolution in computer hardware took place, which had a profound impact on the architecture of information systems: the concept of Personal Computer (PC). Suddenly, instead of "dumb" terminals, users had real computing power on their desk. Client-server computing was born.

This led in the first place to the "outsourcing" of the presentation functionality to the client computer. But, as a consequence, it was no longer clear where the application programs should run. As the clients had significant computational power, the application logic was often executed on the client. The servers kept essentially only the shared resources, namely the databases, and thus functioned as database servers.

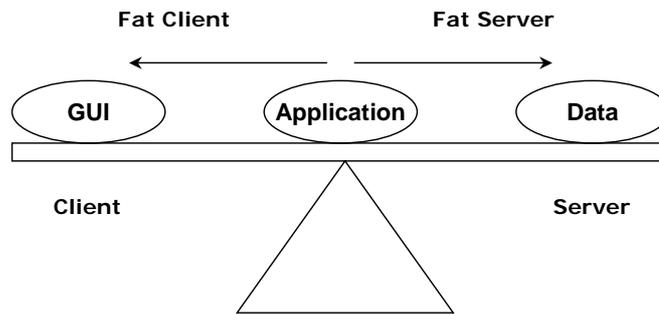
This approach to client-server computing is called "fat client" 2-tier architecture (the word *tier* is synonym to *level* or *layer*).

One concrete example of implementing this architecture is to access a database from an application implemented as a Java program, which is running on the user's workstation and accessing databases via the JDBC API over the network.

## Fat Clients

---

- Running applications on clients is not necessarily a good idea
  - Performance: communication overhead and data transfers
  - Maintenance: cost and time of updates



©2004-2005, Karl Aberer & J.P. Martin-Flatin

4

However, running applications on clients is not always a good idea. From a performance standpoint, it leads to a large number of requests and data transfers over the network. If we consider the previous example, a number of DB accesses are performed just to execute a single transaction.

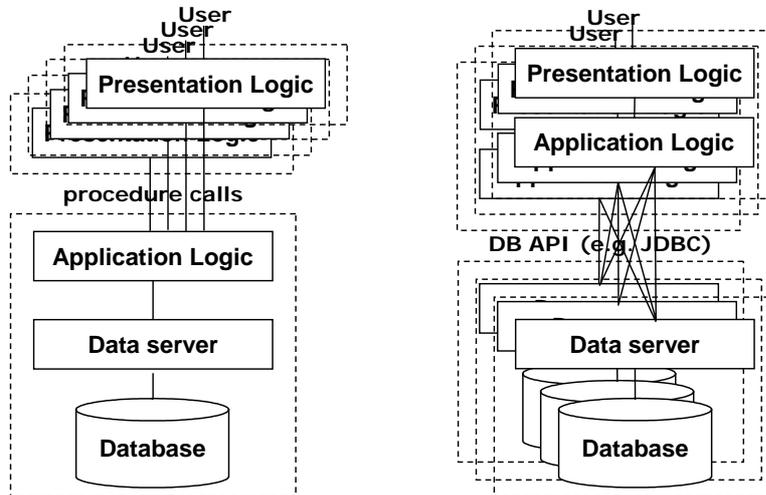
Another problem is related to development and maintenance. Because an instance of the application program must be installed at every client, all the clients need to be updated whenever the application software is updated. In large organizations, this can cause substantial problems to maintain consistency of program versions, and thus lead to unacceptably large costs.



## Limitations of Two-Tier Architectures (1/2)



- Two-tier architectures do not scale well to large no. of clients and/or many servers



© 2004-2005, Karl Aberer & J.P. Martin-Flatin

6

When looking at the problem of scaling systems up to a very large number of clients (e.g., 1000s) and a moderately large number of servers (e.g., 10s), both two-tier approaches (i.e., fat and thin clients) do not perform very well.

If the applications run on the client (fat clients), the following problems occur:

- Poor reusability and difficult maintenance:** As client programs have to cover many different application needs, they tend to grow into huge monolithic applications with little reusability (unless outstanding software engineering principles are enforced). As applications grow in complexity, they become also more difficult to maintain, which is further aggravated by the fact that maintenance has to be done in a distributed environment.
- Performance:** As the client applications access databases with a very fine granularity (e.g., single statements in JDBC) and as multiple databases often need to be accessed by these client applications, many communication channels are required between clients and servers and a large number of messages need to be exchanged over these channels.
- Heterogeneity:** As the number of servers increases, applications face more and more difficult heterogeneity problems, as there exists no intermediate layer that could provide abstractions hiding heterogeneities at all levels (data model, communication, schemas).

(cont'd on next page)

## Limitations of Two-Tier Architectures (2/2)



- Fat clients
  - Poor reusability
    - Monolithic application
  - Difficult maintenance
    - Monolithic application
  - Performance problems
    - Fine granularity for accessing databases
  - Heterogeneity problems
    - No intermediary to hide heterogeneity in data models, communication and schemas
- Thin clients
  - No transactions involving multiple databases
    - Stored procedures for DB1 cannot be used for DB2
  - Performance problems
    - One process per stored procedure
  - Vendor lock-in
    - Stored procedures are vendor specific

If the applications run on the server (thin clients, TP-Lite) other problems occur:

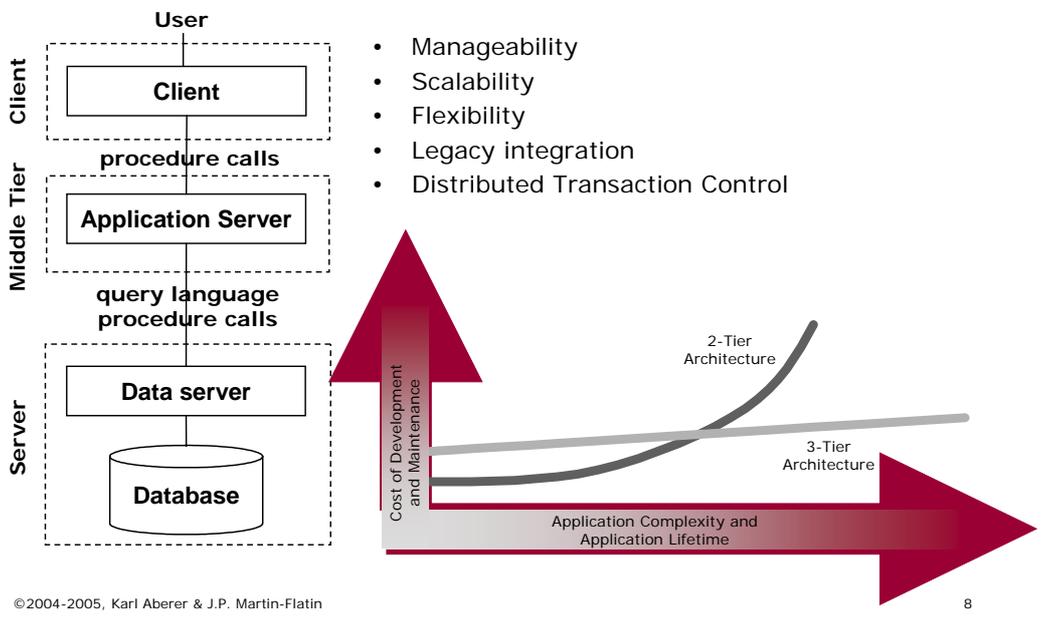
•*No transactions involving multiple databases*: If stored procedures are used for accessing a database DB1, they cannot be used to access another database DB2. This limits quite substantially the possibilities to implement applications using data from multiple databases. There are other limitations on using transactions; e.g., multiple stored procedures often cannot be executed within one transaction; if a stored procedure calls another stored procedure, they run in different transactions; etc.

•*Performance*: Usually, every call to a stored procedure initiates a new process. With multiple clients, the DB server quickly hits performance limits.

•*Vendor lock-in*: Stored procedures are only usable in vendor-specific environments.

In summary, neither of the two approaches to 2-tier architectures are satisfactory for large-scale applications.

## Solution: Introduce a Middle Layer for Applications



The solution to the aforementioned performance and maintenance problems is (again) introducing a middleware layer, so-called *application servers*. (In fact, we have already identified them as components that can handle distributed transactions.) As a result, we obtain what is called a *3-tier architecture*.

The advantages of 3-tier architectures are numerous and address many of the issues we have mentioned before.

- *Centralized management of applications*: It is possible now to have a special server where the applications can be put. This place (the application server) is neither bound to the existing DBs nor to the clients accessing them. Centralized management of applications improves manageability and application reuse.
- *Concentration and distribution of requests from the client*: If many clients and DB servers need to be bound via application programs, the number of necessary channels is much smaller. Assume that  $n$  clients need to access  $m$  DB servers. Without a middle tier,  $n*m$  connections must be supported. With a middle tier, only  $n+m$  connections are needed:  $n$  connections from the clients to the middle tier and  $m$  connections from the middle tier to the DB servers. Thus the architecture scales better.
- *Flexible hardware architecture*: Depending which of the components, specific applications or databases are needed most, they can be replicated or distributed accordingly and independently. Bottlenecks can thus be avoided.
- *Integration of legacy systems*: The middle tier can also be used in order to integrate legacy applications (similarly as legacy databases can be integrated using a federated DBMS, which is another kind of middle tier). Heterogeneities in accessing applications and databases can thus be hidden.
- *Distributed transaction control*: This we have already discussed when we introduced the distributed transaction concept.

Note that 3-tier architectures can be generalized to *n-tier architectures*, where  $n$  can be arbitrarily large. For instance, a tier can hide the heterogeneity of applications, while another tier above it can be used to manage distributed transactions when using these applications.

## 5.2 Overview of Transaction Monitors

- Historically, Transaction Monitors were among the first application servers to appear on the market
  - Used to manage client-server transactions in data-intensive business applications with many clients (e.g., banks, airlines)
  - TP-heavy transaction processing
- Main functions
  - Process Management
    - Pre-starting and sharing of server processes, load balancing
  - Communication Management
    - RPCs, P2P
    - Wrapping of heterogeneous resources (access protocols, authentication, message format)
  - Transaction Management
    - Control of distributed transactions accessing multiple databases (e.g., by implementing the 2PC protocol)
    - Providing access to transactional resources over the network
    - Providing transactional guarantees for transactions involving access to non-transactional resources
    - Synchronous communication (TRPC) or asynchronous communication (message queues)

©2004-2005, Karl Aberer & J.P. Martin-Flatin

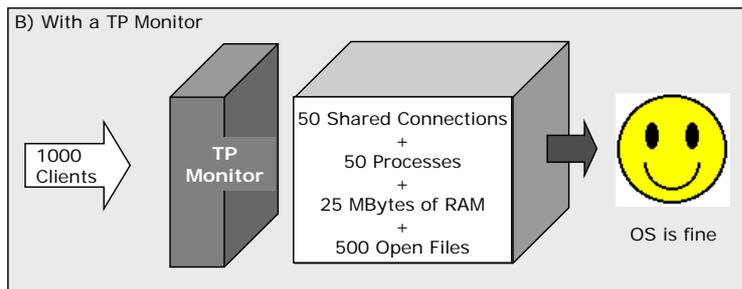
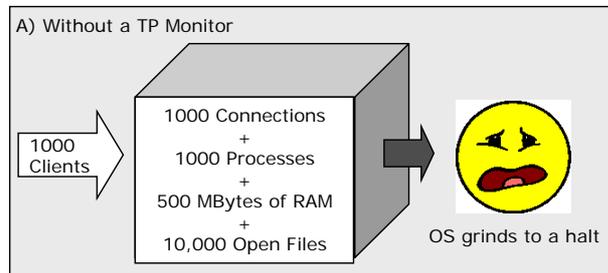
9

A transaction monitor does in fact more than "just" providing distributed transaction control. It provides among others:

- *Process Management*: TP monitors use multiple server processes to serve the same applications concurrently and improve thus performance. They can distribute the server processes over the network and provide load balancing mechanisms.
- *Communication management*: TP monitors provide basic communication facilities to exchange messages with distributed resources. Some are based on Remote Procedure Calls (RPCs), which match the client/server model well. Others support peer-to-peer types of communication. As part of their communication management, TP monitors also support interfaces to heterogeneous proprietary systems.
- *Transaction management*: One goal of a TP monitor is to guarantee global atomicity and isolation. A transaction running on a transaction monitor may run on different resources, including database systems that have their own transaction management. For non-transactional resources (e.g., the file system), the transaction monitor provides additional services to make them transactional (e.g., a lock manager) and thus supports synchronization (as DBMSes do for DB accesses). An important function of the TP monitor is to extend the local transaction control of individual resources to a global transaction control, e.g. by implementing the 2PC protocol. TP monitors use two approaches in order to support distributed transactional applications:
  - Synchronous communication and the transactional RPC communication mechanism;
  - Asynchronous communication and persistent message queuing.

## Process Management

---



©2004-2005, Karl Aberer & J.P. Martin-Flatin

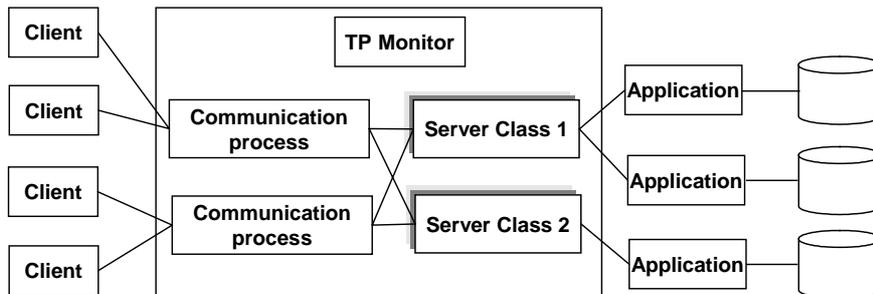
10

This figure illustrates the point why process management by TP monitors is so beneficial for performance. Assume 1000 clients are accessing a (database) application, and each of the requests is handled individually by the DBMS. Then the OS has to deal with the numbers given in the figure above.

By sharing resources using a TP monitor, in particular connections and processes, the requirements for the OS are substantially reduced and the OS is fine.

## Servers

- Terminology
  - Service: offered by a transactional program
  - Server: process that can execute a service
  - Server class: group of processes



- Administration of Transaction Monitors
  - Management of complete runtime environment
  - Administration more demanding than application development

© 2004-2005, Karl Aberer & J.P. Martin-Flatin

11

One of the key functions of a TP monitor is to control the usage of OS resources, in particular processes, and thus to increase overall performance in high-end transactional applications. Different products use different techniques to this end, but schematically the approach is generally as follows.

The TP Monitor provides a number of (shared) communication processes that implement the communication with the clients. Consequently, it is not necessary to start a new process for every request; instead, a process can be "pre-started" and taken from a process pool. These processes analyze the incoming requests and dispatch them to the server class that is providing the requested service. A server class consists of a number of processes (servers) that implement a specific service. Depending on the usage pattern, multiple processes can be running to implement a given service, possibly on different network nodes. A service is a transactional program that accesses one or more applications. For scheduling requests to servers, the TP monitor considers all required and available resources.

Regarding the management of TP monitors by system administrators, it is important to note that a TP Monitor manages the complete runtime environment: it performs the dispatching of requests, the load balancing, restarts in case of failure, etc. Doing so properly requires substantial configuration information, such as

- Existing services, servers, server classes, clients
- Performance characteristics of the servers
- Statistics on usage

Consequently, administering a TP monitor is often more complex than implementing the applications. A 5:1 ratio is generally assumed to hold between administration and development.

## 5.3 Transactional RPCs and Persistent Message Queues

- Problems
  - How can an application participate in a distributed transaction ?
  - How can the transaction monitor control the distributed transaction ?
  - How can this be done over the network ?
- Solution
  - Extend a communication mechanism with transactional control
  - communication + transactions = transaction monitor
- Two standard models
  - Synchronous: RPC + transactions = Transactional RPC (TRPC)
  - Asynchronous: message queues + transactions = persistent message queues

© 2004-2005, Karl Aberer & J.P. Martin-Flatin

12

**The key point about transaction monitors is that they combine two aspects of distributed transactions: communication and transaction control.** When executing a distributed transaction, the mechanism to control it depends on the chosen communication mechanism. Several communication mechanisms can be considered for realizing distributed transaction control, including:

- *RPCs (Remote Procedure Calls)*: This is the standard communication mechanism in client-server systems. In the Java world, RPCs are implemented by means of RMI (Remote Method Invocation). It is based on a request-response paradigm.
- *Message queues*: This mechanism is the asynchronous equivalent to RPCs. It is also based on the request-response paradigm.
- *Publish/Subscribe*: This is an asynchronous communication mechanism where messages are not sent to one specific host from which a response is expected, but are instead broadcast to a set of clients that have subscribed to a specific channel.
- *Peer-to-Peer (P2P)*: There is no specific distinction between requestors and responders. Messages can be sent in an arbitrary fashion among any nodes (peers).

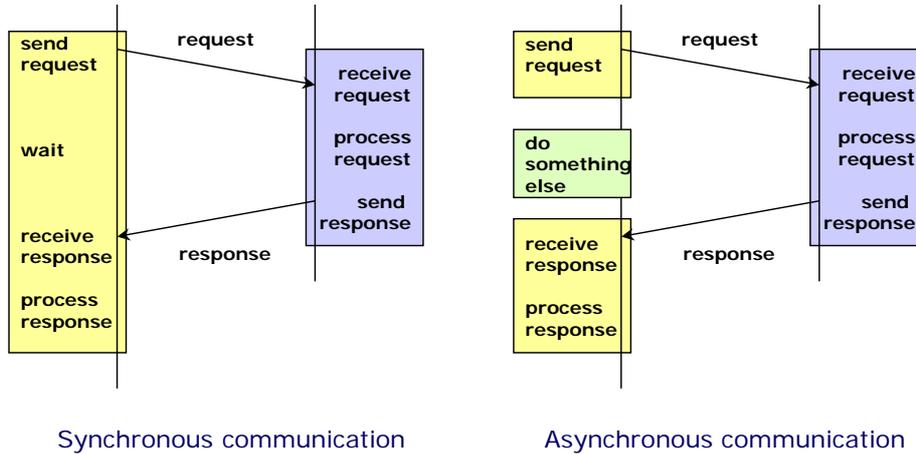
As transaction monitors have evolved in a client-server architectural framework, they have mostly been designed to support the client-server communication model. In this model, a client sends a request to server, which is then expected to generate a response and send it back to the client.

Depending on whether the communication is performed synchronously or asynchronously, we can distinguish two basic models of transactional support for distributed applications:

- *Transactional RPC (TRPC)*: the transaction is executed over an RPC communication mechanism;
- *Persistent Message Queues*: the transaction is executed over an asynchronous message queuing mechanism.

## Synchronous vs. Asynchronous Communication

- Commonality: request–response model
- Difference: requestor waits for response or not

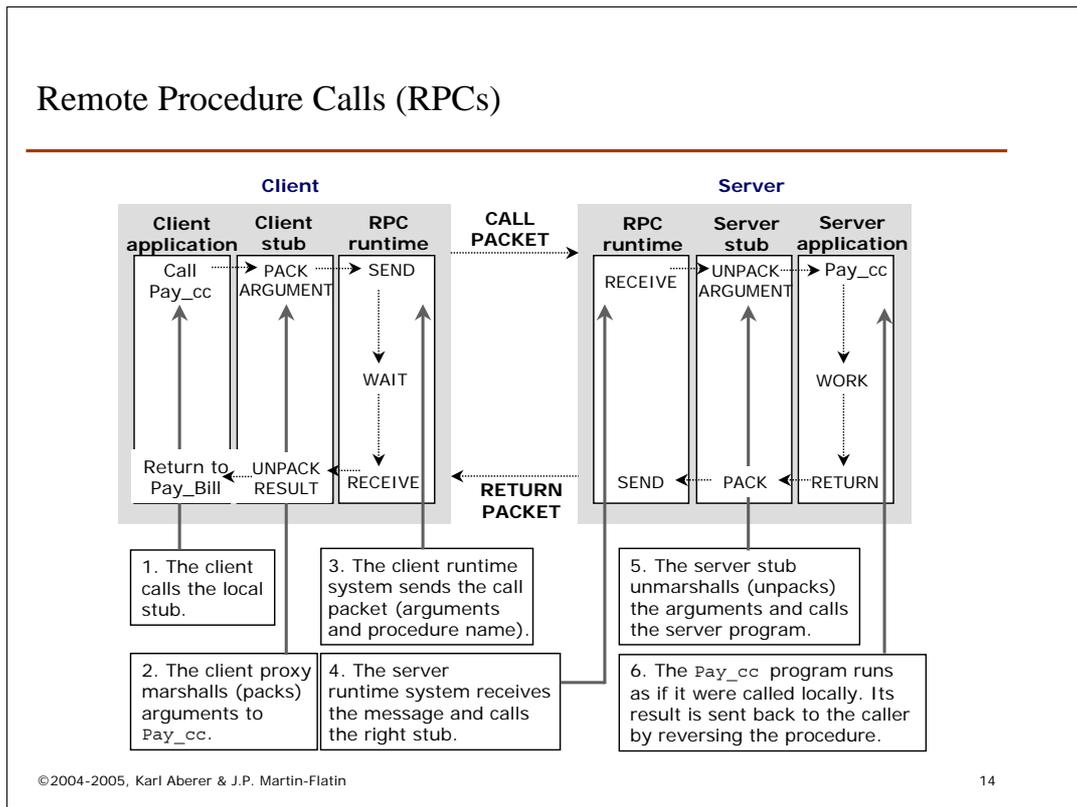


© 2004-2005, Karl Aberer & J.P. Martin-Flatin

13

The request-response model states that for every request message, there exists a response message. The difference between a synchronous and an asynchronous communication model is whether the requesting process (program) blocks and waits until it receives the response (synchronous model), or whether it continues with the processing without waiting for the response, and processes the response at a later time, once it has arrived.

## Remote Procedure Calls (RPCs)



This figure gives an overview of the details of processing an RPC call (as it is for example implemented for RMI in Java):

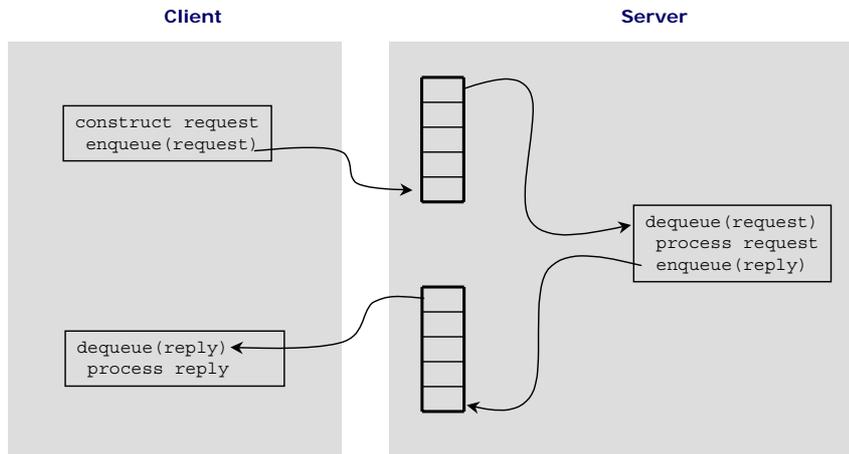
Between the applications running on the client and server, we find two intermediate software layers: the stubs and the runtime environment. The *stubs* are responsible for packing and unpacking the requests and responses from the program-internal representation into a packet that can be sent over the network. The main activity of this step is the so-called *marshalling* and *unmarshalling* of arguments, which is related to the fact that the arguments are packed into a binary format. The *runtime environment* is responsible for handling the network connections and sending/receiving packets.

The dashed arrows indicate the control flow within the client and the server. One can see that the client waits until it receives a response from the server, and that the server is performing the work during that time.

The comments describe the detailed steps when sending the request. The response processing is the exact inverse of this procedure.

## Message Queues

- Requests from client are sent asynchronously and stored in a message queue
- Multiple queues can be used between client and server (queue forwarding)



©2004-2005, Karl Aberer & J.P. Martin-Flatin

15

In order to enable asynchronous processing of requests, the server system maintains one (or more) queues where incoming requests are stored until they are processed by the server, and another queue where the results are stored until they are retrieved by the client.

The packing of requests and the sending of requests and responses can be handled as for RPCs. The difference is that the client is not waiting for a response, and the server is not necessarily immediately processing a request that is stored in the queue.

Queuing also offers a number of possibilities in order to optimize the execution of requests, e.g. by performing load balancing or forwarding requests to another queue (e.g., a queue running on a more appropriate server than the one that received the client's request).

## Comparison between synchronous and async. communication

	synchronous	asynchronous
Execution context	Requires a common session (data about the communication, context). In case of failure, reestablishing the context can be costly.	The two processes need no common execution context.
Programming paradigm	Corresponds to a procedural programming paradigm. Short function calls.	Corresponds to a message-based paradigm. Long-lasting processes.
Error handling	Errors can be immediately fixed in the application.	Slow responses are indistinguishable from lost messages.
Software development	Structured invocations: Generating requests and processing responses are implemented in a common module and therefore easy to follow.	Design of software can be more modular, as generating requests and processing responses can be implemented independently. More flexible communication models can be implemented.

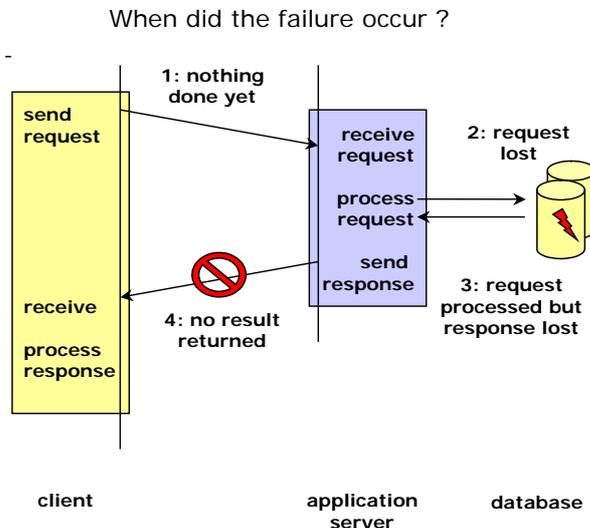
©2004-2005, Karl Aberer & J.P. Martin-Flatin

16

This table summarizes the main differences between synchronous and asynchronous communication. Which is the more suitable approach depends largely on the application requirements. Generally, asynchronous communication is more suitable in environments where the servers have to perform activities of long duration, and the clients cannot wait for their completion before doing something else.

## Failures

- Synchronous communication
  - perform all operations within the request as part of one transaction - transactional RPC
- Asynchronous communication
  - No timeout can be given: late message cannot be distinguished from lost message
  - Make message queues persistent



©2004-2005, Karl Aberer & J.P. Martin-Flatin

17

Both RPCs and message queuing encounter problems as soon as failures occur in processing and network communication. In particular, it is generally not possible to return to a consistent state after a failure. Why is this so? Consider a generic communication scenario as shown in the figure for the synchronous case (asynchronous is analogous). In case a client notices that it does not receive a response, e.g. after a timeout, it cannot know what was the reason for the failure. Four principal cases might have occurred:

1. The request never reached the server. Then it could be simply re-issued
2. The server received the request and forwarded it to another server (e.g., a database), but never received a response. Re-issuing the request would lead to two pending requests at the server, and both could eventually be processed. Depending on whether the database received the request, the processing may be doubled.
3. The request was forwarded by the server to a database system, but the server never received an answer. Re-issuing the request to the server could lead to a double processing, e.g. doing an update twice, but the server might be able to recognize that it had a lost message.
4. The server received a response from the database, but the response that it sent to the client never reached the client. Thus the server is not able to recognize that the processing of the request failed.

Therefore the client and the server are in general not able to return to a consistent state (easily). To do so, they would have to determine which of the cases occurred and what measures would be suitable for fixing the problem.

A better way to deal with this is to use transactions. Transactions support in particular the property of atomicity, which says that either all operations of a transaction are executed or none. If all the operations that are part of the request processing could be bundled into one transaction, the problem would be solved. However, we have to guarantee atomicity for a transaction that involves multiple systems, thus a distributed transaction mechanism needs to be implemented.

For synchronous communication, this leads to the concept of transactional RPC. In short, transactional RPC makes sure that all the operations that are invoked (by using RPCs) as part of a transaction become also part of that transaction.

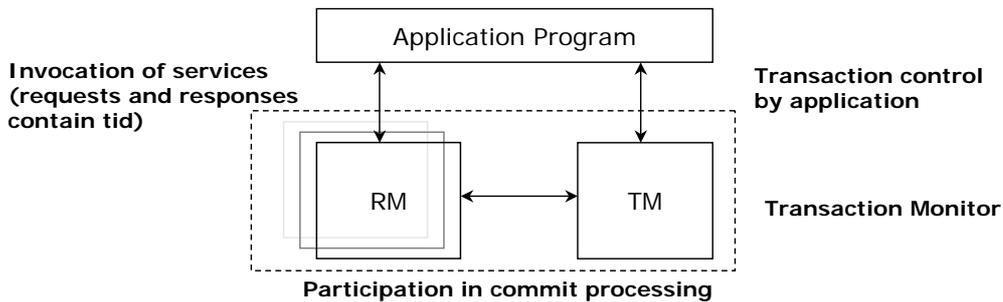
For asynchronous communication, the approach is a bit different because the late arrival of a response cannot be distinguished from a lost response (whereas in synchronous communication, we assume that in case a timeout is reached, an error must have occurred and some recovery action needs to be taken). Rather than performing the whole request processing atomically, one ensures that a failure can never occur in message passing. The solution is to make the message queues persistent (that is, the messages are stored durably) and to guarantee the following two properties:

1. *Guaranteed delivery*: a message that is sent from the client to the server and vice versa is never lost
2. *Exactly once delivery semantics*: a message is never duplicated.

As we will see, the implementation of persistent message queues in fact relies on transactional RPC.

## Transactional RPC

- Transaction Manager (TM)
  - Assigns transaction identifiers (tids)
  - Monitors transaction progress
  - Coordinates transaction commit or rollback
- Resource Manager (RM)
  - Manages application programs that provide services to the transaction manager
  - Provides ACID transactions for access to its resources and participates in 2PC
- Application Program
  - Calls the services provided by the resource managers and starts root transaction



How to make the operations that are invoked as part of a transaction via RPC really part of the transaction ?

First, these operations must know that they are part of a transaction. To receive this information, each transaction receives an ID (the transaction identifier, or *tid* for short), and all requests and responses contain the tid as part of their parameters. This is also called *transaction context propagation*. Note that a transactional RPC is different from an RPC call within a transaction! We still can invoke other programs via RPCs within a transaction, but if the call is not done via TRPC, e.g. if no tid is supplied, the operations of that program do not become part of the transaction.

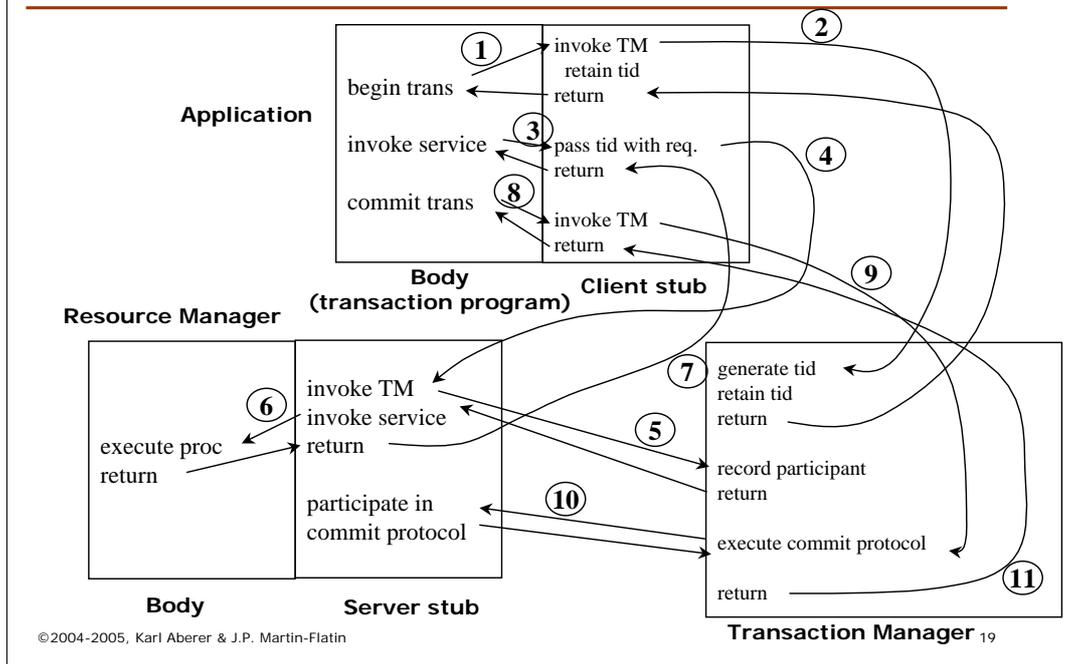
Second, a transaction manager is required that coordinates the overall execution of the distributed transaction (the coordinator of the distributed transaction). It assigns the tid, keeps track of who is participating in the transactions, and decides to commit or rollback the transaction.

Third, each resource taking part in the transaction (e.g., the database systems) are represented by a resource manager that provides access to the application programs and the necessary services to the transaction manager. In particular, the application program itself must be executed as part of a local ACID transaction, the transaction manager is notified about the start of local sub-transactions and the resource manager provides the necessary services in order to participate in the commit processing (e.g., by supporting prepared-to-commit messages).

Fourth, the applications themselves access the services they want to invoke via the resource managers, using requests extended by tids, and start transactions by accessing the transaction manager.

All the necessary functionalities for implementing transactional RPC constitute the key services provided by any transaction monitor.

## System View of TRPC



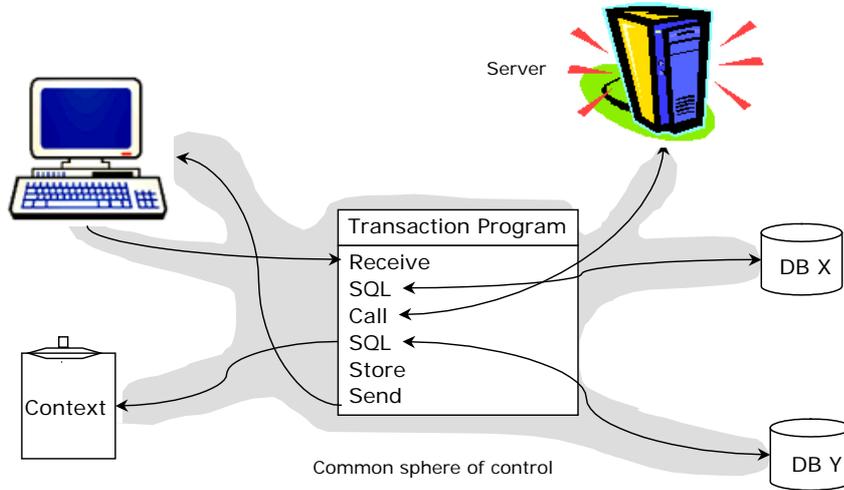
This figure gives a more detailed view of the processing steps that are taken with transactional RPC. We have an application program that starts a transaction and invokes one service via a resource manager. On the client side, as for RPCs, the application program has a client stub running that handles the communication with the server. On the server side, a server stub is running for the resource manager to handle accesses to the service.

One can observe that the stubs have more work to do for TRPC than for RPCs. They not only do the marshalling of parameters, but also have to i) translate abstract calls to the transaction and resource managers into concrete messages, and ii) manage the transaction information involved (e.g., keeping track of the tids). The steps are the following:

- Upon the invocation of a transaction on the client, the client stub generates a message to the transaction manager for starting a transaction.
- The transaction manager generates a new tid and returns it to the invoking application.
- The application invokes a service. The client stub recognizes that this invocation is part of a transaction and extends the call with the tid.
- The request is sent by the client stub to the resource manager of the server (more precisely to the server stub).
- The resource managers server stub notifies the transaction manager that this resource is now participating in the transaction, and the transaction manager records this information. This step is frequently called *resource enlistment*.
- The resource manager now calls the application program that is implementing the service and receives the results.
- The resource manager returns the results to the calling application.
- The application ends the transaction by issuing a commit command.
- When receiving the commit, the client stub asks the transaction manager to commit the transaction.
- The transaction manager starts to execute the commit protocol of the distributed transaction. It requests *prepared-to-commit* from all the resource managers that have been registered as participants in the transaction.
- Once the commit protocol is successfully executed, the transaction manager returns to the client stub the information of successful completion, which in turn returns to the application.

## Developer's View of TRPC

---

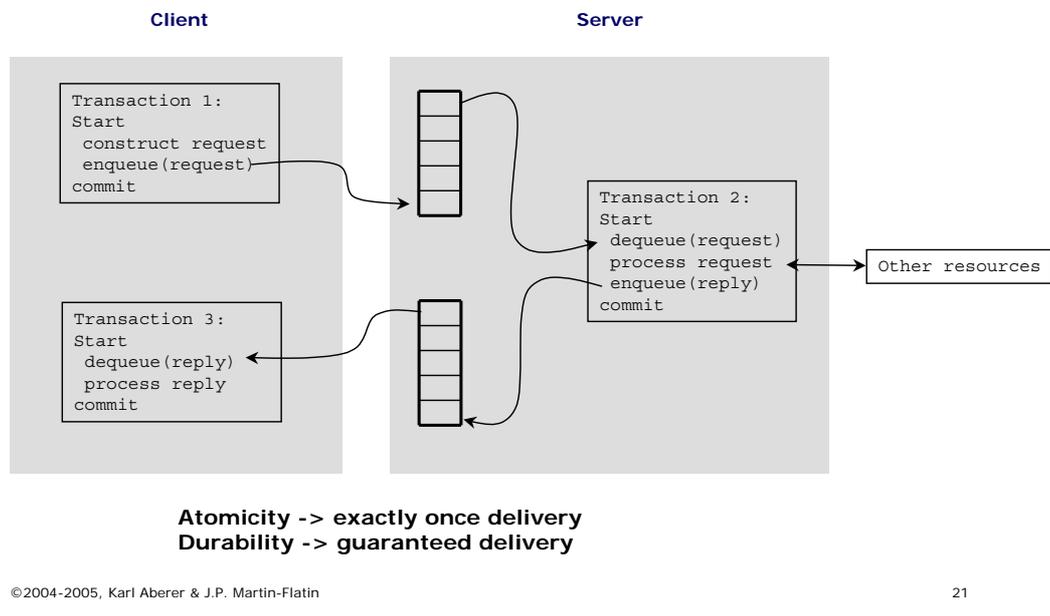


©2004-2005, Karl Aberer & J.P. Martin-Flatin

20

Once all the infrastructure (resource managers, transaction manager) is set up, the mechanism described on the previous slide provides the application developer with a simple means to implement distributed transactional applications. From his viewpoint, he can include distributed transactional resources as part of his application-defined transaction. It seems to him that all the resources are covered by a common "sphere of control" from the viewpoint of transaction management.

## Implementation of Persistent Message Queues



Now let us look at the implementation of persistent message queues. The approach takes advantage of a distributed transaction mechanism similar to what was described before. In fact, from an implementation viewpoint, it is a fairly straightforward thing to do because persistent message queues are typically a service provided by transaction monitors.

The idea is to consider message queues as persistent objects that are stored at the server, and to consider accesses to these queues as distributed transactions that are executed by the clients and access the server resources. Thus a message queue would have its own resource manager. Message passing is then implemented by performing a transaction at each of the three steps of sending the message from the client to the server, taking the request from the queue, processing it and putting the result in the response queue, and dequeuing the response by the client from the server.

As each of the operations that affect the passing of messages are executed as part of a transaction, they are executed atomically. This ensures both the "exactly once" delivery semantics, because the caller knows whether the transfer of the message was performed successfully (atomicity), and the "guaranteed-delivery" semantics, because once a message is passed to a queue, it is made persistent ( *durable* ).

Note also that in Transaction 2, when taking the request from the queue and processing it, all the accesses to transactional resources required for processing the request, e.g. database accesses, are also covered by this transaction.

## Comparison between TRPC and Persistent MQ

	TRPC Synchronous	Persistent MQ Asynchronous
Efficiency	More efficient (fewer transactions).	Load balancing: Multiple servers can receive messages from the same queue. Priority Scheduling: Extend first-come/first-serve scheduling of TRPC.
Error handling	Simpler Recovery. With persistent message queues, a client has to determine in which transaction the message crashed (1, 2 or 3).	No problem with servers and clients that are down. If a server is down, just leave the message and continue working. If client goes down, simply leave message and do not lose the work.

©2004-2005, Karl Aberer & J.P. Martin-Flatin

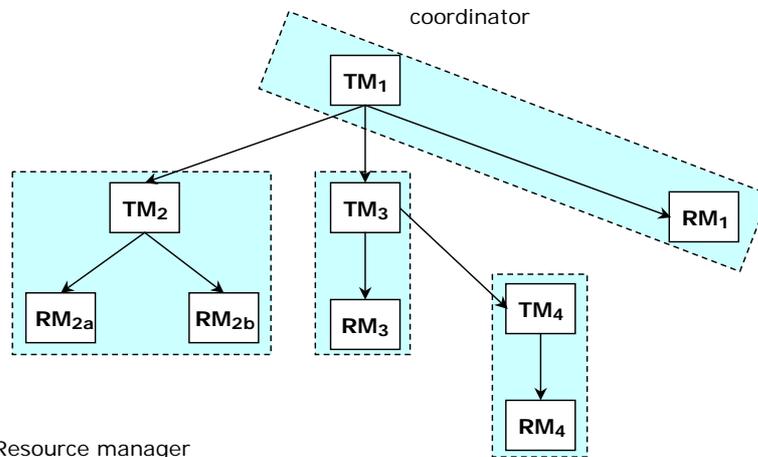
22

Let us extend now our comparison of synchronous and asynchronous communication mechanisms to their respective transactional extensions.

With respect to efficiency, message queuing introduces more overhead, but also more possibilities to increase performance. For example, as the requests need not be immediately processed, no decision needs to be made upon arrival of a message where it will be processed. This leaves room for implementing load balancing and priority strategies.

Regarding error handling, TRPC is simpler because the only case to be considered is essentially the abort of a transaction, whereas with persistent MQ, it is necessary to determine in which transaction the message crashed. On the other hand, persistent MQ is more robust (from the client's perspective) because the transaction can be processed even if one of the participating systems temporarily fails.

## 5.4 Distributed Transaction Processing



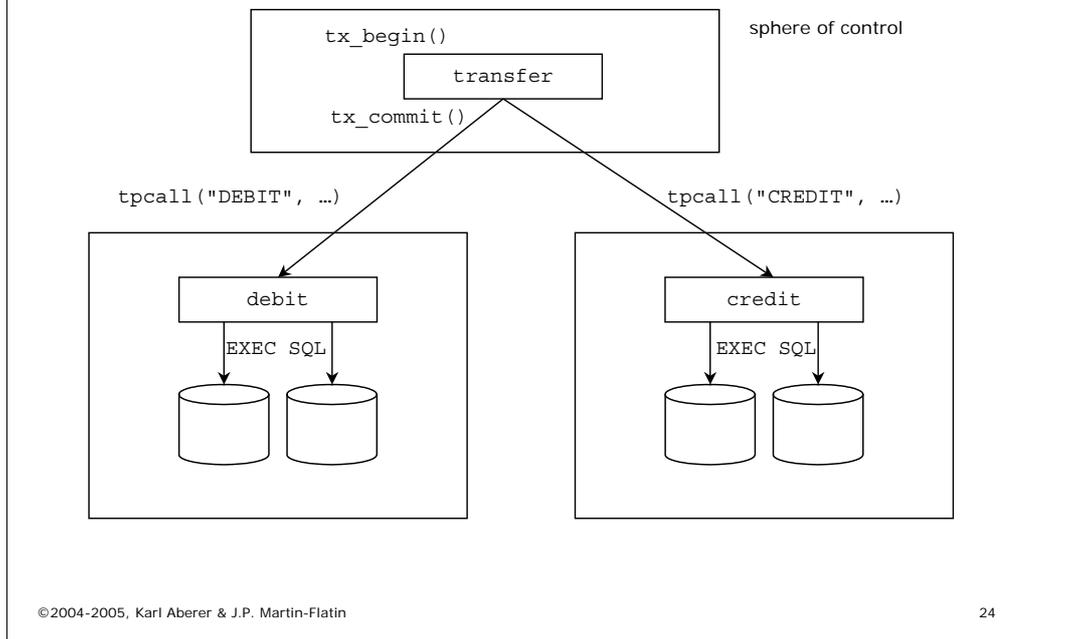
- RM = Resource manager
- TM = Transaction manager
- Dotted lines indicate spheres of control
- Arrows are from a coordinator to its participants.

©2004-2005, Karl Aberer & J.P. Martin-Flatin

23

Up to now, we assumed that a single transaction manager controls multiple resource managers. In practice, however, we can encounter the situation whereby transaction programs access resources that are controlled by different transaction managers. Therefore we need an extension of the model introduced so far, where different transaction managers can communicate with each other and provide access to each other's resources. This is called *distributed transaction processing*. In a distributed transaction involving multiple transaction managers, we always have one transaction manager that plays the role of the coordinator, and multiple TMs and RMs that are participating agents in the transaction.

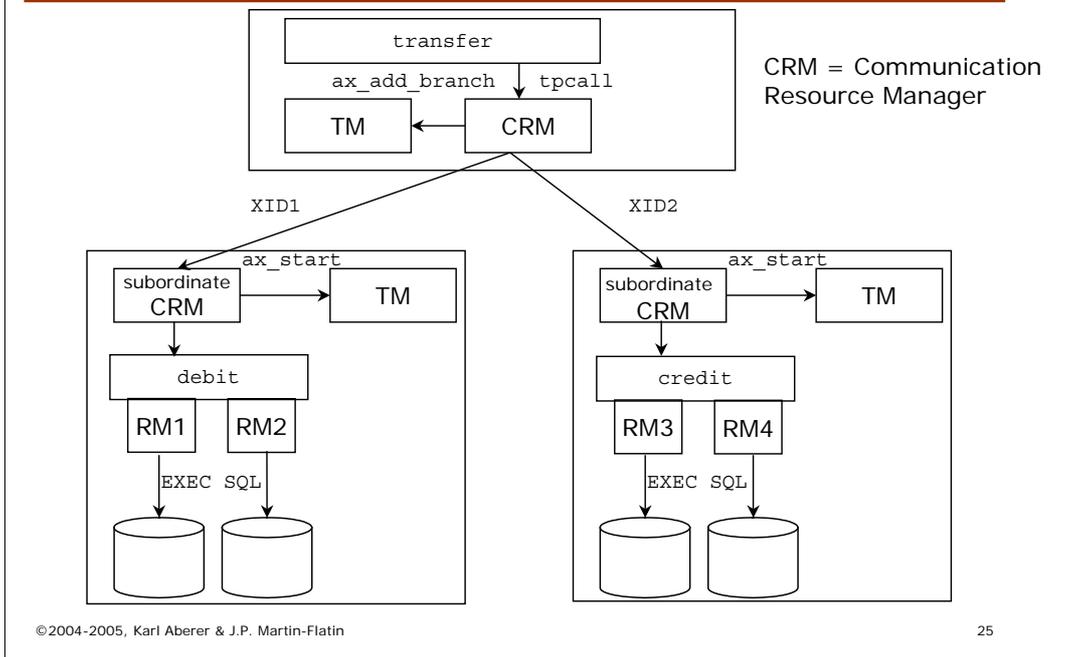
## Developer's View of a Distributed Transaction



For a developer, the fact that some of the resources he wishes to access are not controlled by the same transaction manager, should be transparent. It should thus be possible for him to access these resources in a transparent manner, as depicted in the figure.

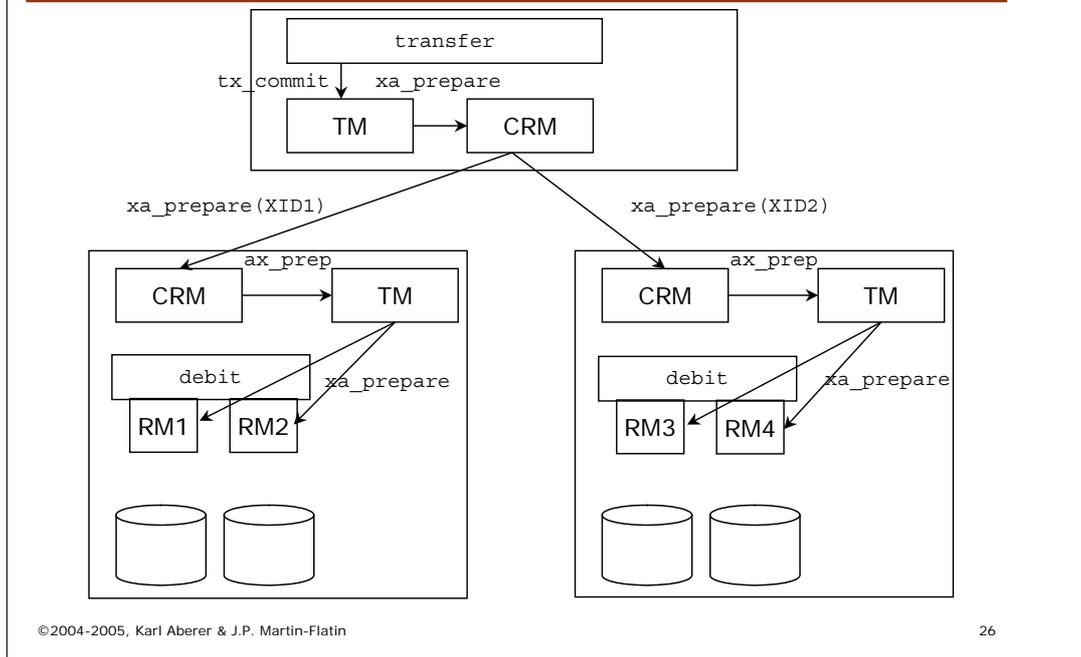
The transaction program "transfer" is executed under one sphere of control and accesses databases for subprograms "debit" and "credit", which run in two other spheres of control. Despite the different spheres of control, these subprograms are simply invoked by using a transactional RPC call "tpcall".

## System View of a Distributed Transaction - Transaction Call



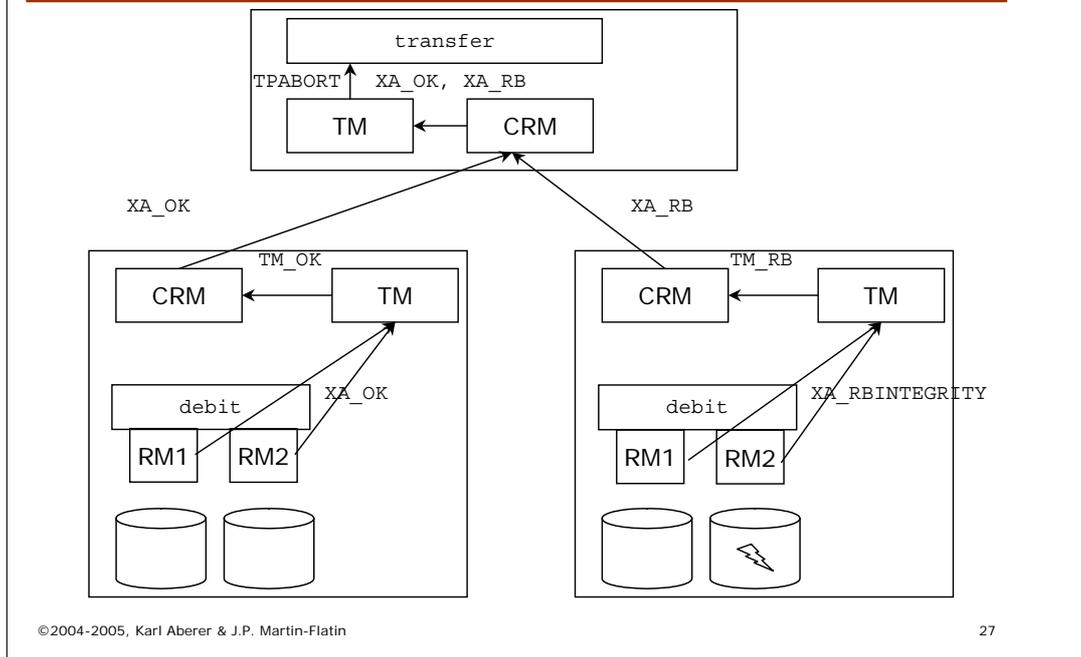
At the system level, the execution of such a distributed transaction is a bit more intricate than in the case where we had only one transaction manager involved. First of all, we need to use a so-called *Communication Resource Manager* (CRM), which enables access to resources that are under the control of different, distributed transaction managers. When starting the transaction, the CRM of the client's sphere of control inform its transaction manager of the start of the transaction. This TM then becomes the coordinator of the global transaction. As it is the root of the transaction tree, it uses a special command for that purpose (add\_branch). Then the CRM determines which other CRMs are responsible for the resources to which the debit and credit requests are sent, by passing them different tids for the respective sub-transactions. These become the subordinate CRMs responsible for executing the request. By doing so, they issue a start transaction command to their respective transaction managers, register the resources involved and perform the accesses to their resources (e.g., via SQL statements) through the resource managers.

## System View of a Distributed Transaction - Prepare to Commit



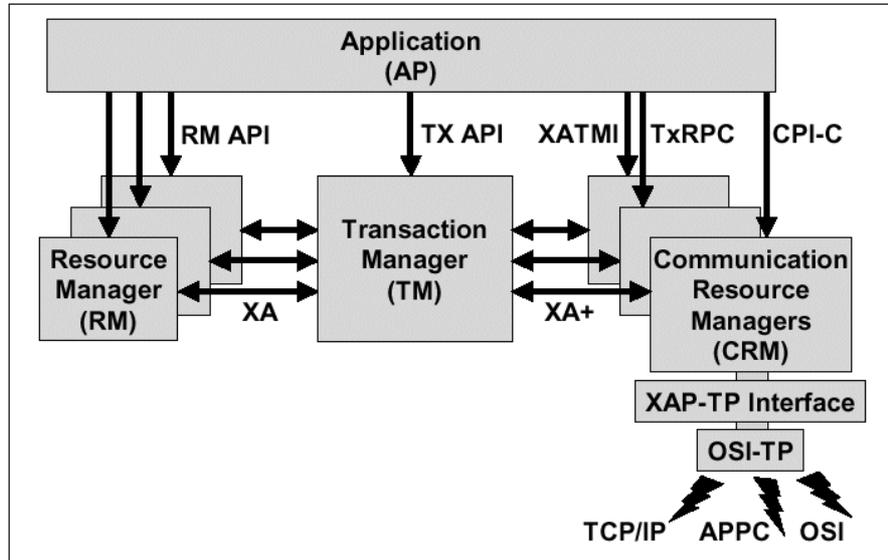
When committing the transaction from the calling client application, by calling the root TM, the root TM informs the local CRM to prepare-to-commit. It in turn forwards prepare-to-commit requests to the CRMs participating as subordinate CRMs in the transaction. The subordinate CRMs forward the prepare-to-commit message to their respective TMs. The TMs then ask the local resource managers to prepare-to-commit as in ordinary commit processing.

## System View of a Distributed Transaction - Abort



The resources respond to the prepare-to-commit messages and the result is propagated all the way back through the TMs and CRMs to the root TM, which then decides whether to commit or rollback the transaction. This completes the walkthrough of how a distributed transaction is processed.

## 5.5 An Architecture for Transaction Monitors: X/Open DTP



©2004-2005, Karl Aberer & J.P. Martin-Flatin

28

In order to enable the interoperability of different transaction monitor products, an industrial consortium (X/Open) standardized the architecture and interfaces for Distributed Transaction Processing (DTP). The architecture of the X/Open DTP standard corresponds to the components that we have introduced so far: AP, RM, TM and CRM. It underlies most of today's implementations of distributed transaction processing environments and transaction monitors.

X/Open DTP standardizes also all the interfaces among the different components of a distributed transaction processing environment, so that different components developed by different vendors remain interoperable (at least in theory). The most important interfaces are the main interfaces among the components, that is:

- RM API, TX API and XATMI API for the applications (and thus used by application program developers);
- XA and XA+ for the communication among transaction managers and resource managers (or communication resource managers).

The main functions of each of those interfaces are explained in the following slides.

## Interfaces in X/Open DTP

---

- RM API
  - Interface for applications to access resources (e.g. ESQL, JDBC for database systems)
- TX API
  - Interface for application programs to access the transaction manager
  - Main functions: tx\_begin, tx\_commit, tx\_rollback
- XA API
  - Bidirectional interface between transaction manager and resource manager
  - Main functions: xa\_start, xa\_prepare, xa\_commit, xa\_rollback, xa\_end
  - Enables a resource manager to participate in a distributed transaction
  - Enlist transactions at the TM: ax\_ref, ax\_unreg
- XATMI
  - Transactional communication between application and communication resource managers
  - Main functions: tpcall(), tpacall()
- XA+ API
  - Bidirectional interface between transaction manager and communication resource manager
  - Main functions: ax\_prepare, ax\_commit, ax\_rollback

©2004-2005, Karl Aberer & J.P. Martin-Flatin

29

The RM API defines how an application program can access the resources. In case of a DBMS, this means essentially the DB gateways (e.g., JDBC). However only the part of JDBC for DB access, such as for issuing queries, belongs in this category.

The TX API specifies how an application communicates with the transaction manager. These are the functions for starting, committing or aborting transactions. In the Java world, they are also part of the JDBC standard.

The XATMI API allows transactional calls to applications managed by communication resource managers. The main function *tpcall()* allows the synchronous invocation of a program. Its asynchronous counterpart is *tpacall()*.

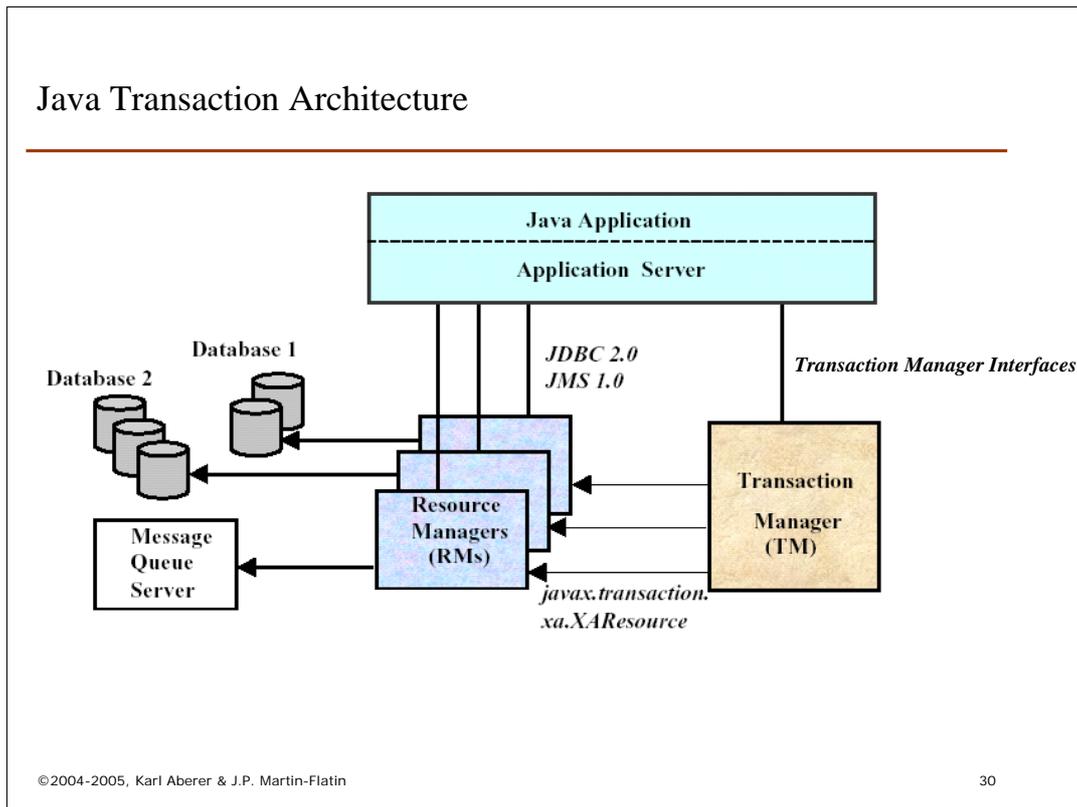
The XA and XA+ APIs are required by developers who want to enable a resource to become interoperable with a transaction manager.

What should a resource do in order to participate in a distributed transaction, i.e. become a resource manager? It has to:

- provide an interface for applications, e.g. JDBC;
- implement the XA API for the transaction management in order to participate in the commit protocol;
- use the *ax\_XXX()* functions to enlist the transactions at the TM.

Most of today's transaction monitor products, e.g. Encina (IBM) and Tuxedo (BEA), support the standard (Microsoft Transaction Server does not). Most commercial DB products and message queuing products support the XA API, e.g. Oracle, Sybase, Microsoft SQLServer, and IBM MQSeries.

## Java Transaction Architecture



Sun proposes for Java a complete architecture for transaction management and message queues as part of their J2EE platform (J2EE = Java Enterprise Edition).

The components of this architecture are

JTA: Java Transaction Application

JTS: Java Transaction Service

JMS: Java Message Service

JDBC: Java Database Connectivity

JNDI: Java Naming Service

JTA is an X/Open-compliant XA API that can be used by application developers. More precisely, it extends the XA API to support the development of application servers. This will be discussed later in the context of EJBs (Enterprise Java Beans).

JTS is an architecture for building transaction managers. It builds on the CORBA OTS service (see next lecture).

JMS specifies a generic messaging interface. Thus the J2EE approach to persistent message queuing is to consider message queuing objects as persistent resources that are treated as other persistent resources. The difference lies in the interface provided to the applications for accessing them, which is JDBC for relational databases and JMS for message queues.

For accessing the transaction manager from the applications, the Transaction Manager Interfaces are provided, which correspond to the TX API from the X/Open. For accessing resources from applications, the JDBC and JMS interfaces are supplied. For interactions between TMs and resources, the XAResource interface is defined, which corresponds to the X/Open XA interface.

## Java Transaction Programming (JTA)

---

- Two possibilities
  - *explicit* transaction control (UserTransaction interface): application programs explicitly control transaction boundaries
  - *implicit* transaction control: application servers manage transactions on behalf of the user (see lecture on EJBs)

- Methods for explicit transaction control

```
javax.transaction.UserTransaction
//object represents transaction
begin                // start transaction
commit              // end transaction
rollback            // abort transaction
getStatus           // return status object
setTransactionTimeout
setRollbackOnly    // transaction can no more be
                  // committed but is not yet aborted
```

©2004-2005, Karl Aberer & J.P. Martin-Flatin

31

The UserTransaction interface supports the following methods (self-explaining ones are omitted):

- *getStatus()*: the values that can be returned are defined as part of the Status object.
- *setRollbackOnly()*: the application can enforce rollback and the transaction can no more commit.

The transaction object can be used to obtain explicitly the status of ongoing transactions.

JTA provides another interface used for implicit transaction control used by application server implementers. We will see later how this comes into play when introducing EJBs.

## Example: Implementation of a User Transaction

---

```
import javax.sql.*;
import javax.naming.*           // import the JNDI API
import javax.transaction.*     // import the JTA
try {
    Context initialContext = new InitialContext();
    UserTransaction ut =
        (UserTransaction)initialContext.lookup("javax.transaction.UserTransaction");
        // obtain object implementing the
        // javax.transaction.UserTransaction interface
        // using the JNDI service
    ut.begin()                 //start transaction
}
DataSource dataSource = (DataSource) context.lookup("jdbc/x");
//Open data source for DB accesses
Connection connection = dataSource.getConnection();
// perform DB operations ...

if(...) { ut.setRollbackOnly() }

//Rollback if certain conditions are violated before closing connection
connection.close();
ut.commit(); // Ending the transaction
```

©2004-2005, Karl Aberer & J.P. Martin-Flatin

32

This example gives a walkthrough of how the `UserTransaction` interface is used. First the object implementing the `UserTransaction` interface needs to be accessed. To do this, we use the Java naming service (JNDI). Then the transaction can be started.

Similarly, access to a data source is obtained using the JNDI service and a connection is opened. Rather than explicitly rolling back a transaction when an integrity constraint is violated, the transaction manager is informed that the transaction cannot commit any more by using `setRollbackOnly()`. It is left to the transaction manager to rollback the transaction when the connection to the database is closed.

By using the `commit()` method, the transaction can be committed.

## Java Message Queues

---

- JMS (Java Message Service) implements both non-persistent and persistent message queues
  - JMS providers can optionally support distributed transactions via JTA: persistent message queues
- Example (non-persistent message queue)

```
// omitted: setup of context and declarations
stockQueue = (Queue) messaging.lookup("StockQueue");
QueueConnection queueConnection;
queueConnection =
    queueConnectionFactory.createQueueConnection();
QueueSession session;
session = queueConnection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
sender = session.createSender(queue);
receiver = session.createReceiver(queue);
queueConnection.start();
message = session.createTextMessage();
message.setText(stockData);
sender.send(message);
stockMessage = (StreamMessage) receiver.receive();
```

Persistent message queues are implemented in Java in two steps. First, Java provides interfaces for general (transient) message queues. In order to set up a communication through message queues, a connection object is created. This connection object can support sessions that allow sequences of message exchanges. For sending and receiving messages, corresponding objects are created. Messages are also represented as objects. Through methods of the sender and receiver objects, messages can be read from and sent to the message queue.

## JMS and Persistent Message Queues

---

- JMS API includes XAQueueSession interface
  - Vendors supporting JMS sessions as XA-compliant resources must implement this interface
  - Then a JMS XAQueueSession object can participate as a resource in a JTA transaction

```
XAQueueConnection queueConnection;
queueConnection = queueConnectionFactory.createQueueConnection();
XAQueueSession session;
session = queueConnection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
UserTransaction xact =
ctx.lookup("javax.transaction.UserTransaction");
xact.begin();
// Perform desired operations
// operations on the persistent message queue now participate in
// the transaction

// commit
xact.commit();
// or rollback
xact.rollback();
```

©2004-2005, Karl Aberer & J.P. Martin-Flatin

34

In order to implement persistent message queues, we use the same interfaces as with transient JMS, except that we create the connection and session objects as instances of XAQueueConnection and XAQueueSession. These interfaces are XA-enabled, i.e. they can participate in distributed transactions by communicating with the transaction manager through the XA interfaces.

This requires that the provider of an implementation of these classes has integrated the necessary functionality. Typically, this is part of an application server implementation. Once the queue-related objects are declared this way, any operations performed on them become part of a transaction. In this example, we show how this allows to include these operations into transactions when using explicit transaction control.

## Summary

---

- Three main approaches to implement transaction programs
  - Within database servers
  - Using transaction monitors
  - Using persistent message queues
- Three-tier architectures provide better scalability than two-tier
  - Development
  - Runtime
- Transaction monitors were the first kind of middleware
  - High-end applications
  - Well standardized (X/Open)
  - Widely used in many disguises (Java/J2SE, CORBA, EJBs/J2EE)
  - Provide also communication and resource management

## Integration of Transactional Resources Checklist

---

- Task: integrate distributed (and probably heterogeneous and autonomous) transactional resources (like databases)
- Abstract Model ✓
  - TRPC, Messages (persistent)
- Embedding ✓
  - RM, Message queuing interface
- Architectures and Systems ✓
  - Transaction Monitors, Message Queuing Systems
- Methodology ✗
  - Developers/System administrators

## References

---

- Books
  - R. Orfali, D. Harkey and J. Edwards, *Client/Server Survival Guide*, 3rd edition, Wiley, 1999.
  - J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1992.
  - P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.
  - P.A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kauffman, 1997.
- Websites
  - <http://www-306.ibm.com/software/data/informix/pubs/library/datablade/dbdk/start.htm> (Informix UDR)
  - <http://edocs.bea.com/wls/docs61/jms/trans.html> (WebLogic JMS and transactions)
  - <http://e-docs.bea.com/tuxedo/tux80/interm/over.htm> (BEA Tuxedo)
  - <http://www.opengroup.org/public/tech/europe/ACTS96V2.htm> (EU ActTrans Project on Open Distributed Transaction Processing)

*(URLs last updated in April 2005)*