# Distributed Systems in practice
HS 2007

Gustavo Alonso
Institute for Pervasive Computing
Computer Science Department
Swiss Federal Institute of Technology (ETHZ)
alonso@inf.ethz.ch
http://www.iks.inf.ethz.ch/

## Distributed systems in practice

- Motivation and examples
  - Enterprise computing / Enterprise Architecture
  - Modern distributed systems
- 2 Phase Commit – Transactions (Dec. 3, 2007)
  - Transactional exchanges
  - 2PC
  - 3PC
- Data Replication (Dec. 10, 2007)
  - Data replication models
  - Data replication systems
- Web services (Dec. 17, 2007)
  - SOAP, WSDL, UDDI / Service Oriented Architecture



## and as a complement ...

- Building a distributed system with embedded devices and sensors
  - René Müller (Dec 7., 2007)
- Modular architectures and distribution
  - Jan Rellermeyer (Dec. 14., 2007)
- Exercises (paper) will be distributed during the lecture - due one week later

## References

- References to use (and read):
  - For 2PC and 3PC
    - Concurrency Control and Recovery in Database Systems (Bernstein, Hadzilacos, Goodman) http://research.microsoft.com/~philbe/ccontrol/
  - For replication: same & slides
  - For web services: slides and supporting material

©IKS, ETH Zürich.


## A prelude to

- Courses in the Distributed Systems Master Track:
  - Enterprise Application Integration
  - Web Services and Service Oriented Architectures
  - Distributed algorithms
  - Sensor networks
  - P2P systems
  - ...

©IKS, ETH Zürich.

## Motivation and examples
## Enterprise Architecture

©IKS, ETH Zürich.

## Enterprise architecture at Credit Suisse

Multiple backends, multiple frontends, flexible composition

Graphic courtesy of Claus Hagen, Stephen Murer and Hanspeter Uebelbacher of Credit Suisse



## Understanding the layers

Presentation logic
Application Logic
Resource Manager

1-2 years Clients and External interface (presentation, access channels)

2-5 years Application (system'slogic)

~10 years Data management systems (operational and strategic data)

- To support a client, the system needs to have a presentation layer through which the user can submit operations and obtain a result.
- The application logic establishes what operations can be performed over the system and how they take place. It takes care of enforcing the business rules and establish the business processes.
- The resource manager deals with the organization (storage, indexing, and retrieval) of the data necessary to support the application logic. This is typically a database but it can also be any other data management system.



web client, wap client, java client — CLIENT TIER

ACCESS TIER — api, api, api

business object, business object, business object — APP TIER

INTEGRATION TIER

wrapper, wrapper, wrapper — db, db, db — RESOURCE TIER

CLIENT: web and wap browsers, specialized clients (Java, Notes), SMS ...

HTML, SOAP, XML

ACCESS: web servers, J2EE, CGI, JAVA Servlets API

MOM, HTML, IIOP, RMI-IIOP, SOAP, XML

APP: TP-Monitors, stored procedures, programs, scripts, beans

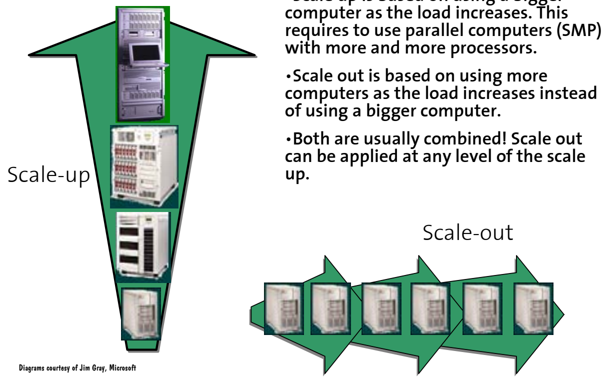
MOM, IIOP, RMI-IIOP, XML

INTEGRATION: system federations, filters, object monitors, MOM

ODBC, JDBC, RPC, MOM, IIOP, RMI-IIOP

RESOURCE: databases, multi-tier systems, backends, mainframes

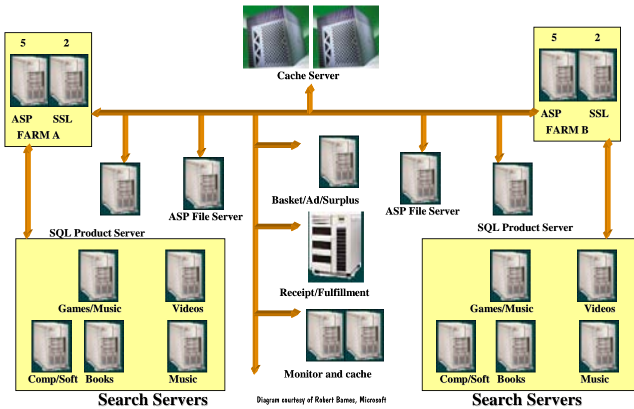
## Understanding the context



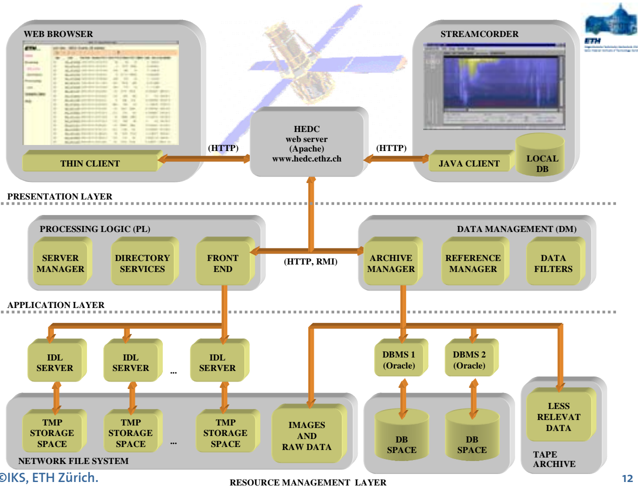

- Scale up is based on using a bigger computer as the load increases. This requires to use parallel computers (SMP) with more and more processors.
- Scale out is based on using more computers as the load increases instead of using a bigger computer.
- Both are usually combined! Scale out can be applied at any level of the scale up.



## Understanding the applications



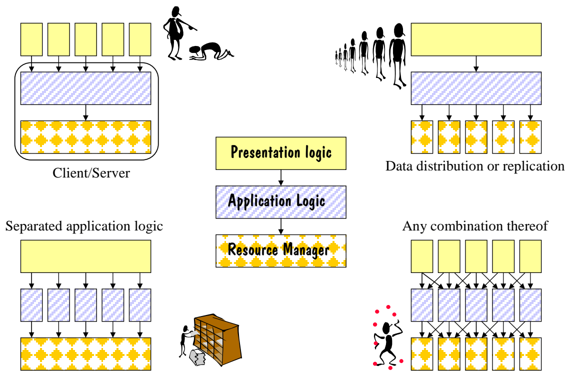


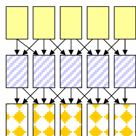



# Motivation and examples
## Modern distributed systems



# Distribution at the different layers



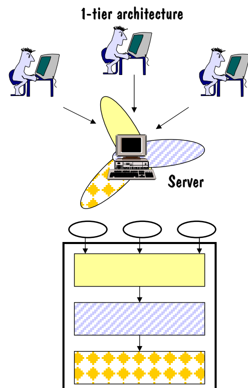


# A game of boxes and arrows

There is no problem in system design that cannot be solved by adding a level of indirection. There is no performance problem that cannot be solved by removing a level of indirection.

- Each box represents a part of the system.
- Each arrow represents a connection between two parts of the system.
- The more boxes, the more modular the system: more opportunities for distribution and parallelism. This allows encapsulation, component based design, reuse.
- The more boxes, the more arrows: more sessions (connections) need to be maintained, more coordination is necessary. The system becomes more complex to monitor and manage.
- The more boxes, the greater the number of context switches and intermediate steps to go through before one gets to the data. Performance suffers considerably.
- System designers try to balance the capacity of the computers involved and the advantages and disadvantages of the different architectures.

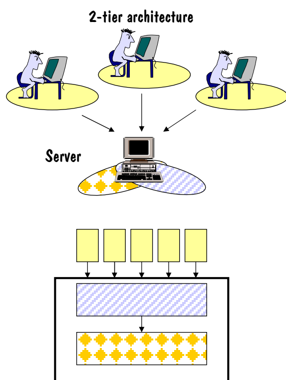
## Architectures (1): 1 tier architectures




- The presentation layer, application logic and resource manager are built as a monolithic entity.
- Users/programs access the system through display terminals but what is displayed and how it appears is controlled by the server. (This are the "dumb" terminals).
- This was the typical architecture of mainframe applications, offering several advantages:
  - no forced context switches in the control flow (everything happens within the system),
  - all is centralized, managing and controlling resources is easier,
  - the design can be highly optimized by blurring the separation between layers.
- This is not as unfashionable as one may think: network computing is based on similar ideas!

©IKS, ETH Zürich.

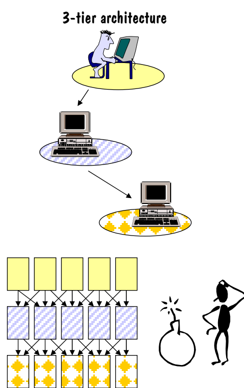

## Architecture (2): 2 tier architectures




- As computers became more powerful, it was possible to move the presentation layer to the client. This has several advantages:
  - Clients are independent of each other: one could have several presentation layers depending on what each client wants to do.
  - One can take advantage of the computing power at the client machine.
  - It introduces the concept of API (Application Program Interface). An interface to invoke the system from the outside. It also allows to think about federating these systems by linking several of them.
  - The resource manager only sees one client: the application logic. This greatly helps with performance since there are no connections/sessions to maintain.

©IKS, ETH Zürich.

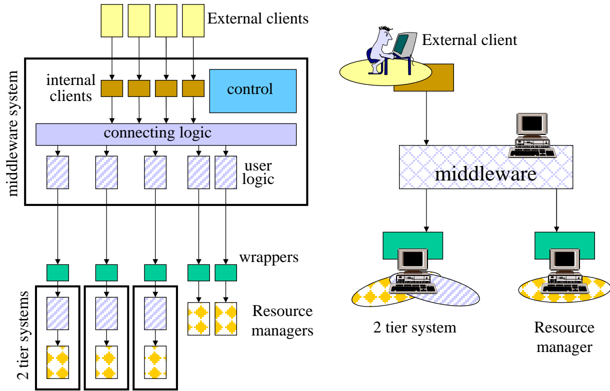

## Architecture (3): 3 tier architectures




- In a 3 tier system, the three layers are fully separated.
- For some people, a middleware based system is a 3 tier architecture. This is a bit oversimplified but conceptually correct since the underlying systems can be treated as black boxes. In fact, 3 tier makes only sense in the context of middleware systems (otherwise the client has the same problems as in a 2 tier system!).
- We will see examples of this architecture when concrete middleware systems are discussed.
- A 3 tier systems has the same advantages as a middleware system and also its disadvantages.
- In practice, things are not as simple as they seem ... there are several hidden layers that are not necessarily trivial: the wrappers.

©IKS, ETH Zürich.

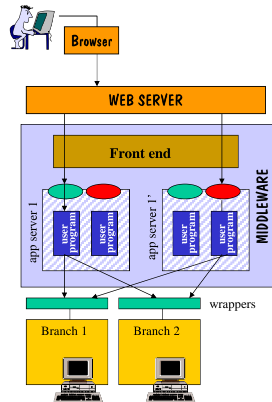
# A real 3 tier middleware based system ...





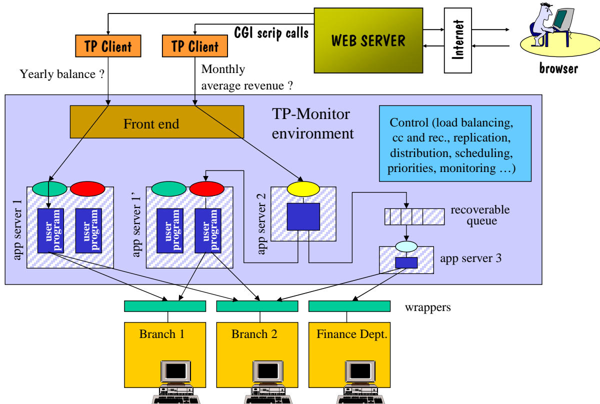

# The Web as software layer ...

- The WWW suddenly opened up software systems that had remained hidden within the IT organization of a company
- It is not that new types of interactions were possible. Behind the WWW there is the same client/server model as in basic RPC. However, the WWW made everything much easier, cheaper and efficient
  - integration at the level of user interface became possible
  - services could be accessed from anywhere in the world
  - the clients could now be not just an internal or selected user but anybody with a browser

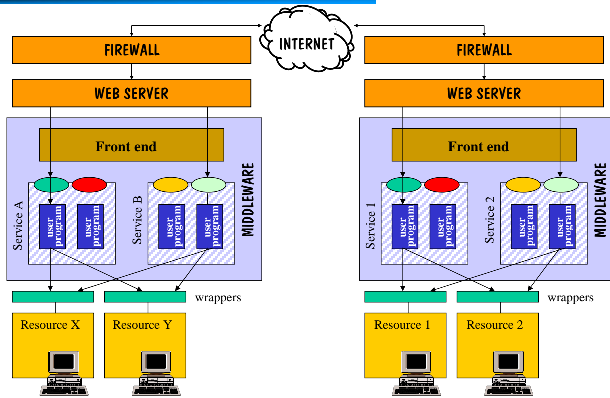





# ... on top of existing systems



# Business to Business (B2B)

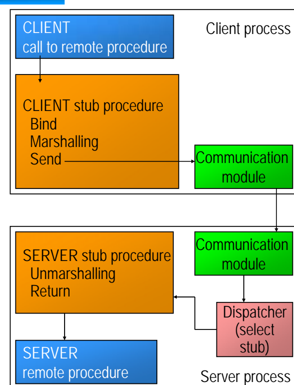


# Motivation and examples



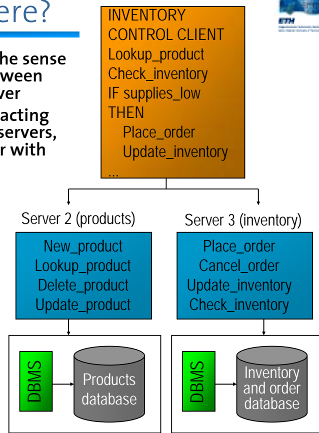
# Basic middleware: RPC

- One cannot expect the programmer to implement a complete infrastructure for every distributed application. Instead, one can use an RPC system (our first example of low level middleware)
- What does an RPC system do?
  - Hides distribution behind procedure calls
  - Provides an interface definition language (IDL) to describe the services
  - Generates all the additional code necessary to make a procedure call remote and to deal with all the communication aspects
  - Provides a binder in case it has a distributed name and directory service system



## What can go wrong here?

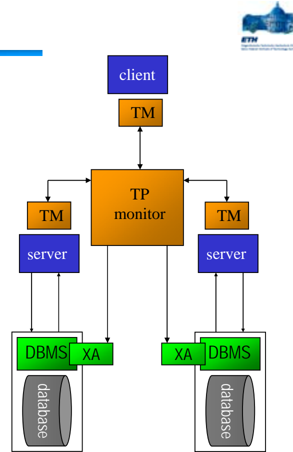
- RPC is a point to point protocol in the sense that it supports the interaction between two entities: the client and the server
- When there are more entities interacting with each other (a client with two servers, a client with a server and the server with a database), RPC treats the calls as independent of each other. However, the calls are not independent
- Recovering from partial system failures is very complex. For instance, the order was placed but the inventory was not updated, or payment was made but the order was not recorded …
- Avoiding these problems using plain RPC systems is very cumbersome





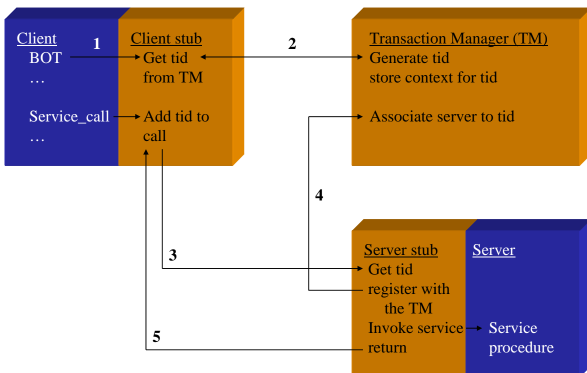

## Transactional RPC

- The limitations of RPC can be resolved by making RPC calls transactional. In practice, this means that they are controlled by a 2PC protocol
- As before, an intermediate entity is needed to run 2PC (the client and server could do this themselves but it is neither practical nor generic enough)
- This intermediate entity is usually called a transaction manager (TM) and acts as intermediary in all interactions between

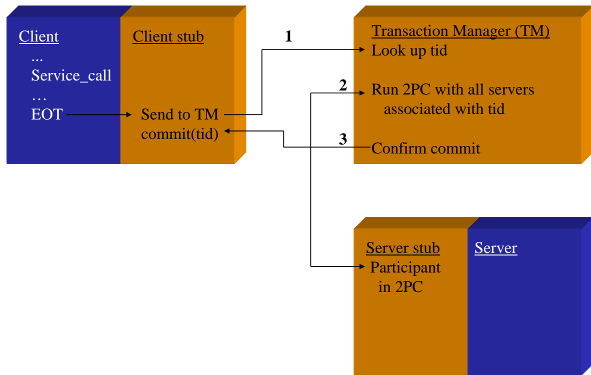





## Basic TRPC (making calls)



## Basic TRPC (committing calls)






## What we will see next

- 2 Phase Commit
- Consistency across a distributed system (data replication)
- Extending RPC/RMI to Internet scale systems

## 2PC-3PC: Basics of transaction processing

# Transaction Processing

Why is transaction processing relevant?

- ➢ Most of the information systems used in businesses are transaction based (either databases or TP-Monitors). The market for transaction processing is many tens billions of dollars per year
- ➢ Not long ago, transaction processing was used mostly in large companies (both users and providers). This is no longer the case (CORBA, WWW, Commodity TP-Monitors, Internet providers, distributed computing)
- ➢ Transaction processing is not just database technology, it is core distributed systems technology

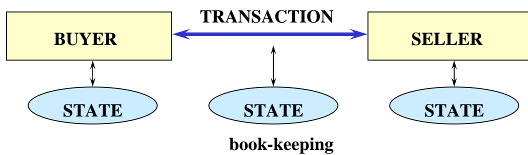
Why distributed transaction processing?

- ➢ It is an accepted, proven, and tested programming model and computing paradigm for complex applications
- ➢ The convergence of many technologies (databases, networks, workflow management, ORB frameworks, clusters of workstations ...) is largely based on distributed transactional processing



# From business to transactions

- ❑ A business transaction usually involves an exchange between two or more entities (selling, buying, renting, booking ...).
- ❑ When computers are considered, these business transactions become electronic transactions:




- ❑ The ideas behind a business transaction are intuitive. These same ideas are used in electronic transactions.
- ❑ Electronic transactions open up many possibilities that are unfeasible with traditional accounting systems.



# The problems of electronic transactions

Transactions are a great idea:

- ❑ Hack a small, cute program and that's it.

Unfortunately, they are also a complex idea:

- ❑ From a programming point of view, one must be able to encapsulate the transaction (not everything is a transaction).
- ❑ One must be able to run high volumes of these transactions (buyers want fast response, sellers want to run many transactions cheaply).
- ❑ Transactions must be correct even if many of them are running concurrently (= at the same time over the same data).
- ❑ Transactions must be atomic. Partially executed transactions are almost always incorrect (even in business transactions).
- ❑ While the business is closed, one makes no money (in most business). Transactions are "mission critical".
- ❑ Legally, most business transactions require a written record. So do electronic transactions.

## What is a transaction?

Transactions originated as “spheres of control” in which to encapsulate certain behavior of particular pieces of code.

- A transaction is basically a set of service invocations, usually from a program (although it can also be interactive).
- A transaction is a way to help the programmer to indicate when the system should take over certain tasks (like semaphores in an operating system, but much more complicated).
- Transactions help to automate many tedious and complex operations:
  - record keeping,
  - concurrency control,
  - recovery,
  - durability,
  - consistency.
- It is in this sense that transactions are considered ACID (Atomic, Consistent, Isolated, and Durable).



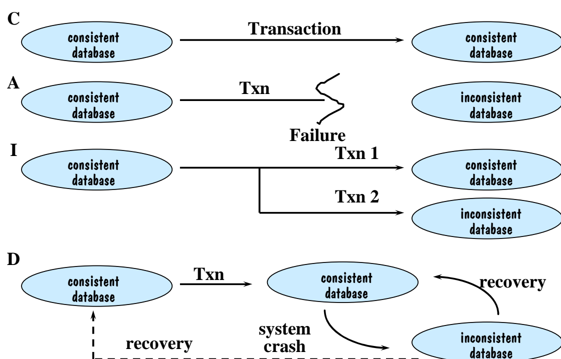
## Transactional properties

These systems would have been very difficult to build without the concept of transaction. To understand why, one needs to understand the four key properties of a transaction:

- ATOMICITY: necessary in any distributed system (but also in centralized ones). A transaction is atomic if it is executed in its entirety or not at all.
- CONSISTENCY: used in database environments. A transactions must preserve the data consistency.
- ISOLATION: important in multi-programming, multi-user environments. A transaction must execute as if it were the only one in the system.
- DURABILITY: important in all cases. The changes made by a transaction must be permanent (= they must not be lost in case of failures).

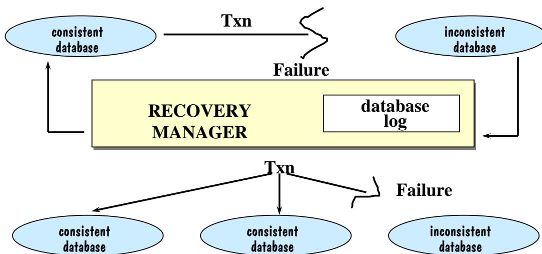


## Transactional properties



## Transactional atomicity

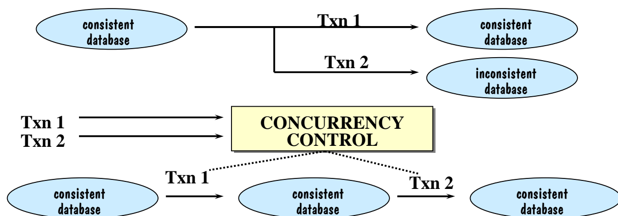
- Transactional atomicity is an "all or nothing" property: either the entire transaction takes place or it does not take place at all.
- A transaction often involves several operations that are executed at different times (control flow dependencies). Thus, transactional atomicity requires a mechanism to eliminate partial, incomplete results (a recovery protocol).






## Transactional isolation

- Isolation addresses the problem of ensuring correct results even when there are many transactions being executed concurrently over the same data.
- The goal is to make transactions believe there is no other transaction in the system (isolation).
- This is enforced by a concurrency control protocol, which aims at guaranteeing serializability.






## Transactional consistency

- Concurrency control and recovery protocols are based on a strong assumption: the transaction is always correct.
- In practice, transactions make mistakes (introduce negative salaries, empty social security numbers, different names for the same person ...). These mistakes violate database consistency.
- Transaction consistency is enforced through integrity constraints:
  - Null constrains: when an attribute can be left empty.
  - Foreign keys: indicating when an attribute is a key in another table.
  - Check constraints: to specify general rules ("employees must be either managers or technicians").
- Thus, integrity constraints acts as filters determining whether a transaction is acceptable or not.
- NOTE: integrity constraints are checked by the system, not by the transaction programmer.

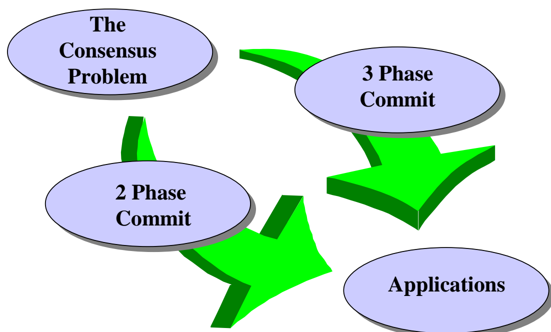
## Transactional durability

- Transactional system often deal with valuable information. There must be a guarantee that the changes introduced by a transaction will last.
- This means that the changes introduced by a transaction must survive failures (if you deposit money in your bank account, you don't want the bank to tell you they have lost all traces of the transaction because there was a disk crash).
- In practice, durability is guaranteed by using replication: database backups, mirrored disks.
- Often durability is combined with other desirable properties such as availability:
  - Availability is the percentage of time the system can be used for its intended purpose (common requirement: 99.86% or 1 hour a month of down time).
  - Availability plays an important role in many systems. Consider, for instance, the name server used in a CORBA implementation.

## Atomic commitment: 2PC-3PC



## Atomic Commitment



## Atomic Commitment

Properties to enforce:

- AC1 = All processors that reach a decision reach the same one (agreement, consensus).
- AC2 = A processor cannot reverse its decision.
- AC3 = Commit can only be decided if all processors vote YES (no imposed decisions).
- AC4 = If there are no failures and all processors voted YES, the decision will be to commit (non triviality).
- AC5 = Consider an execution with normal failures. If all failures are repaired and no more failures occur for sufficiently long, then all processors will eventually reach a decision (liveness).



## Simple 2PC Protocol and its correctness

PROTOCOL:

- Coordinator send VOTE-REQ to all participants.
- Upon receiving a VOTE-REQ, a participant sends a message with YES or NO (if the vote is NO, the participant aborts the transaction and stops).
- Coordinator collects all votes:
  - All YES = Commit and send COMMIT to all others.
  - Some NO = Abort and send ABORT to all which voted YES.
- A participant receiving COMMIT or ABORT messages from the coordinator decides accordingly and stops.

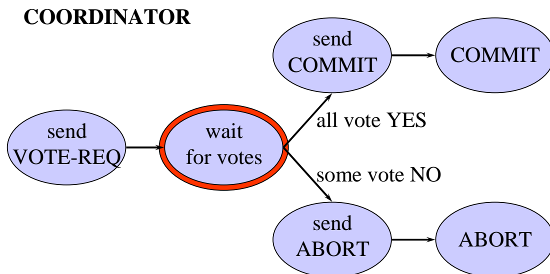
CORRECTNESS:

The protocol meets the 5 AC conditions (I - V):

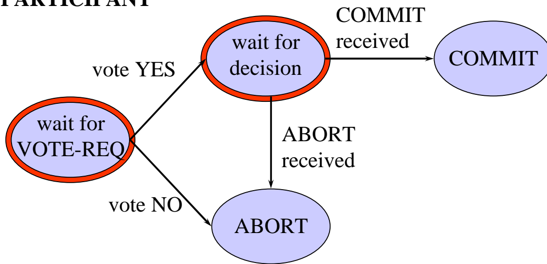
- ACI = every processor decides what the coordinator decides (if one decides to abort, the coordinator will decide to abort).
- AC2 = any processor arriving at a decision "stops".
- AC3 = the coordinator will decide to commit if all decide to commit (all vote YES).
- AC4 = if there are no failures and everybody votes YES, the decision will be to commit.
- AC5 = the protocol needs to be extended in case of failures (in case of timeout, a site may need to "ask around").



## Timeout Possibilities



## Timeout Possibilities






## Timeout and termination

In those three waiting periods:

- If the coordinator times-out waiting for votes: it can decide to abort (nobody has decided anything yet, or if they have, it has been to abort)
- If a participant times-out waiting for VOTE-REQ: it can decide to abort (nobody has decided anything yet, or if they have, it has been to abort)
- If a participant times-out waiting for a decision: it cannot decide anything unilaterally, it must ask (run a Cooperative Termination Protocol). If everybody is in the same situation no decision can be made: all processors will block. This state is called uncertainty period

When in doubt, ask. If anybody has decided, they will tell us what the decision was:

- There is always at least one processor that has decided or is able to decide (the coordinator has no uncertainty period). Thus, if all failures are repaired, all processors will eventually reach a decision
- If the coordinator fails after receiving all YES votes but before sending any COMMIT message: all participants are uncertain and will not be able to decide anything until the coordinator recovers. This is the blocking behavior of 2PC (compare with the impossibility result discussed previously)



## Recovery and persistence

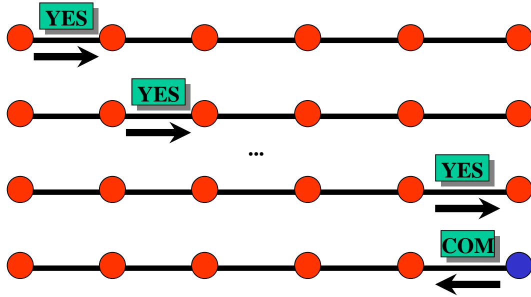
Processors must know their state to be able to tell others whether they have reached a decision. This state must be persistent:

- Persistence is achieved by writing a log record. This requires flushing the log buffer to disk (I/O).
- This is done for every state change in the protocol.
- This is done for every distributed transaction.
- This is expensive!

- When sending VOTE-REQ, the coordinator writes a START-2PC log record (to know the coordinator).
- If a participant votes YES, it writes a YES record in the log BEFORE it send its vote. If it votes NO, then it writes a NO record.
- If the coordinator decides to commit or abort, it writes a COMMIT or ABORT record before sending any message.
- After receiving the coordinator's decision, a participant writes its own decision in the log.

## Linear 2PC

- Linear 2PC commit exploits a particular network configuration to minimize the number of messages:

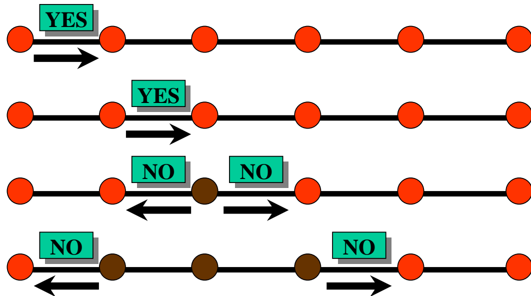



©IKS, ETH Zürich.



## Linear 2PC

- The total number of messages is 2n instead of 3n, but the number of rounds is 2n instead of 3




©IKS, ETH Zürich.



## 3 Phase Commit Protocol

2PC may block if the coordinator fails after having sent a VOTE-REQ to all processes and all processes vote YES. It is possible to reduce the window of vulnerability even further by using a slightly more complex protocol (3PC).

In practice 3PC is not used. It is too expensive (more than 2PC) and the probability of blocking is considered to be small enough to allow using 2PC instead.

But 3PC is a good way to understand better the subtleties of atomic commitment

We will consider two versions of 3PC:

- One capable of tolerating only site failures (no communication failures). Blocking occurs only when there is a total failure (every process is down). This version is useful if all participants reside in the same site.
- One capable of tolerating both site and communication failures (based on quorums). But blocking is still possible if no quorum can be formed.

©IKS, ETH Zürich.

## Blocking in 2PC

Why does a process block in 2PC?

- If a process fails and everybody else is uncertain, there is no way to know whether this process has committed or aborted (NOTE: the coordinator has no uncertainty period. To block the coordinator must fail).
- Note, however, that the fact that everybody is uncertain implies everybody voted YES!
- Why, then, uncertain processes cannot reach a decision among themselves?

The reason why uncertain process cannot make a decision is that being uncertain does not mean all other processes are uncertain. Processes may have decided and then failed. To avoid this situation, 3PC enforces the following rule:

- NB rule: No operational process can decide to commit if there are operational processes that are uncertain.

How does the NB rule prevent blocking?

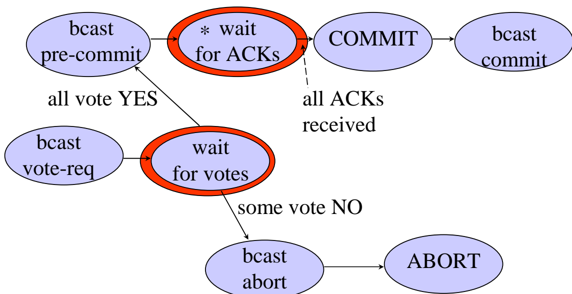
## Avoiding Blocking in 3PC

The NB rule guarantees that if anybody is uncertain, nobody can have decided to commit. Thus, when running the cooperative termination protocol, if a process finds out that everybody else is uncertain, they can all safely decide to abort.

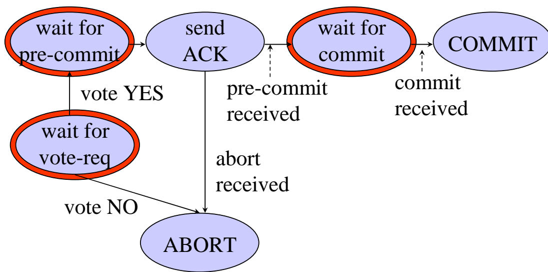
- The consequence of the NB rule is that the coordinator cannot make a decision by itself as in 2PC. Before making a decision, it must be sure that everybody is out of the uncertainty area. Therefore, the coordinator, must first tell all processes what is going to happen: (request votes, prepare to commit, commit). This implies yet another round of messages!



## 3PC Coordinator



## 3PC Participant




Possible time-out actions



## 3PC and Knowledge (using the NB rule)

3PC is interesting in that the processes know what will happen before it happens:

- Once the coordinator reaches the "bcast pre-commit", it knows the decision will be to commit.
- Once a participant receives the pre-commit message from the coordinator, it knows that the decision will be to commit.

Why is the extra-round of messages useful?

- The extra round of messages is used to spread knowledge across the system. They provide information about what is going on at other processes (NB rule).

The NB rule is used when time-outs occur (remember, however, that there are no communication failures):

- If coordinator times out waiting for votes = ABORT.
- If participant times out waiting for vote-req = ABORT.
- If coordinator times out waiting for ACKs = ignore those who did not sent the ACK! (at this stage everybody has agreed to commit).
- If participant times out waiting for pre-commit = still in the uncertainty period, ask around.
- If participant times out waiting for commit message = not uncertain any more but needs to ask around!



## Persistence and recovery in 3PC

Similarly to 2PC, a process has to remember its previous actions to be able to participate in any decision. This is accomplished by recording every step in the log:

- Coordinator writes "start-3PC" record before doing anything. It writes an "abort" or "commit" record before sending any abort or commit message.
- Participant writes its YES vote to the log before sending it to the coordinator. If it votes NO, it writes it to the log after sending it to the coordinator. When reaching a decision, it writes it in the log (abort or commit).

Processes in 3PC cannot independently recover unless they had already reached a decision or they have not participated at all:

- If the coordinator recovers and finds a "start 3PC" record in its log but no decision record, it needs to ask around to find out what the decision was. If it does not find a "start 3PC", it will find no records of the transaction, then it can decide to abort.
- If a participant has a YES vote in its log but no decision record, it must ask around. If it has not voted, it can decide to abort.

## Termination Protocol

- Elect a new coordinator.
- New coordinator sends a “state req” to all processes. participants send their state (aborted, committed, uncertain, committable).
- TR1 = If some “aborted” received, then abort.
- TR2 = If some “committed” received, then commit.
- TR3 = If all uncertain, then abort.
- TR4 = If some “committable” but no “committed” received, then send “PRE-COMMIT” to all, wait for ACKs and send commit message.

TR4 is similar to 3PC, have we actually solved the problem?

- Yes, failures of the participants on the termination protocol can be ignored. At this stage, the coordinator knows that everybody is uncertain, those who have not sent an ACK have failed and cannot have made a decision. Therefore, all remaining can safely decide to commit after going over the pre-commit and commit phases.
- The problem is when the new coordinator fails after asking for the state but before sending any pre-commit message. In this case, we have to start all over again.



## Partition and total failures

This protocol does not tolerate communication failures:

- A site decides to vote NO, but its message is lost.
- All vote YES and then a partition occurs. Assume the sides of the partition are A and B and all processes in A are uncertain and all processes in B are committable. When they run the termination protocol, those in A will decide to abort and those in B will decide to commit.
- This can be avoided is quorums are used, that is, no decision can be made without having a quorum of processes who agree (this reintroduces the possibility of blocking, all processes in A will block).

Total failures require special treatment, if after the total failure every process is still uncertain, it is necessary to find out which process was the last on to fail. If the last one to fail is found and is still uncertain, then all can decide to abort.

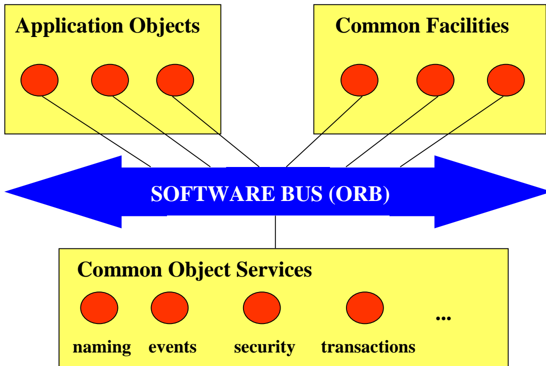
- Why? Because of partitions. Everybody votes YES, then all processes in A fail. Processes in B will decide to commit once the coordinator times out waiting for ACKs. Then all processes in B fail. Processes in A recover. They run the termination protocol and they are all uncertain. Following the termination protocol will lead them to abort.



## 2PC in Practice

- 2PC is  a protocol used in many applications from distributed systems to Internet environments
- 2PC is not only a database protocol, it is used in many systems that are not necessarily databases but, traditionally, it has been associated with transactional systems
- 2PC appears in a variety of forms: distributed transactions, transactional remote procedure calls, Object Transaction Services, Transaction Internet Protocol ...
- In any of these systems, it is important to remember the main characteristic of 2PC: if failures occur the protocol may block. In practice, in many systems, blocking does not happen but the outcome is not deterministic and requires manual intervention

## ORB




©Gustavo Alonso, ETH Zurich.

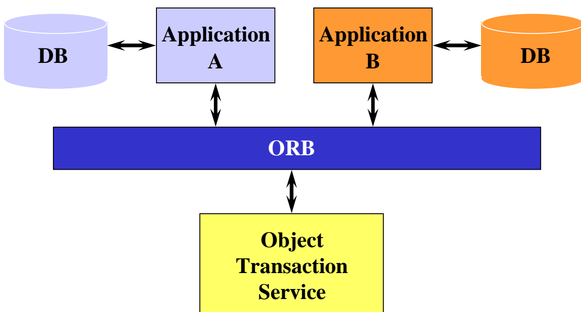

## Object Transaction Service

- The OTS provides transactional guarantees to the execution of invocations between different components of a distributed application built on top of the ORB
- The OTS is fairly similar to a TP-Monitor and provides much of the same functionality discussed before for RPC and TRPC, but in the context of the CORBA standard
- Regardless of whether it is a TP-monitor or an OTS, the functionality needed to support transactional interactions is the same:
  - transactional protocols (like 2PC)
  - knowing who is participating
  - knowing the interface supported by each participant

©Gustavo Alonso, ETH Zurich.

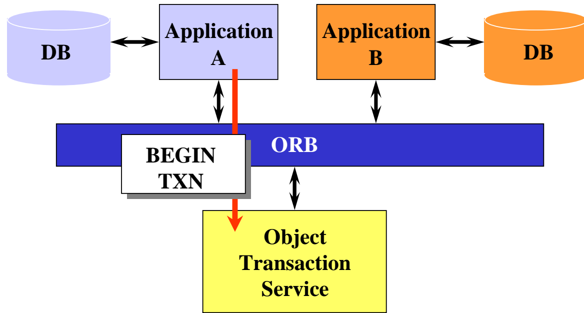

## Object Transaction Service

**Assume App A wants to update its database and also that in B**



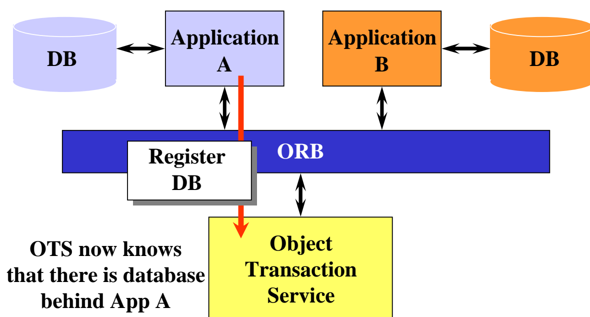

©Gustavo Alonso, ETH Zurich.

## Object Transaction Service



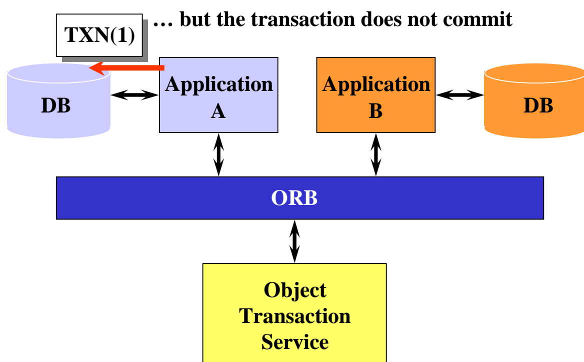



## Object Transaction Service

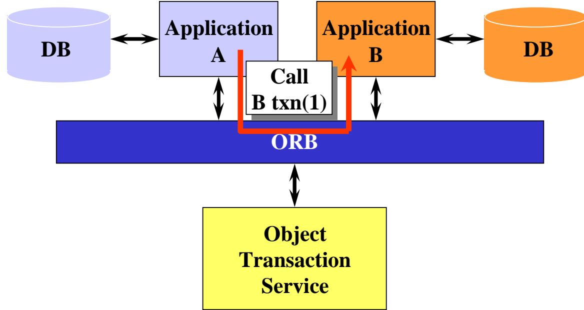





## Object Transaction Service

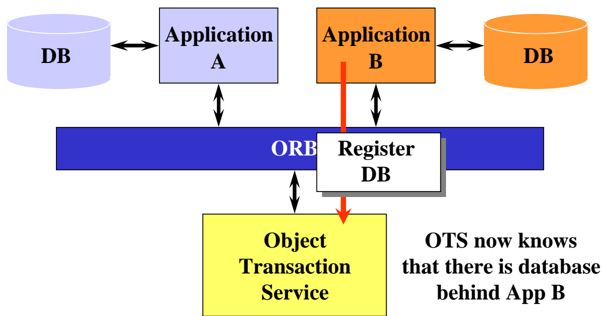


## Object Transaction Service






## Object Transaction Service

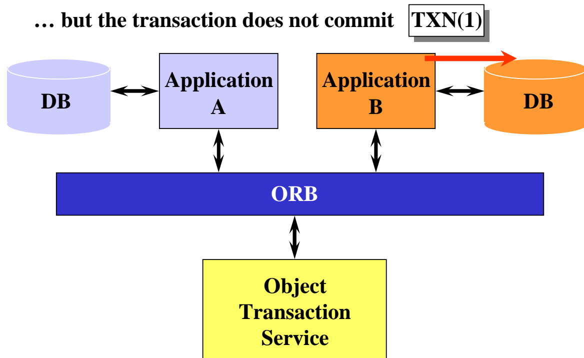



OTS now knows that there is database behind App B

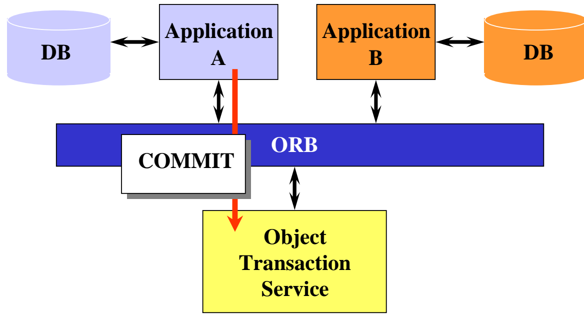


## Object Transaction Service

… but the transaction does not commit

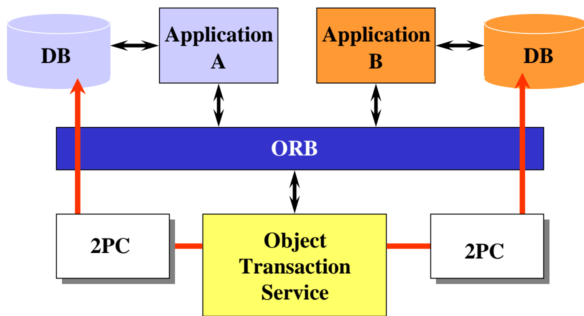


## Object Transaction Service



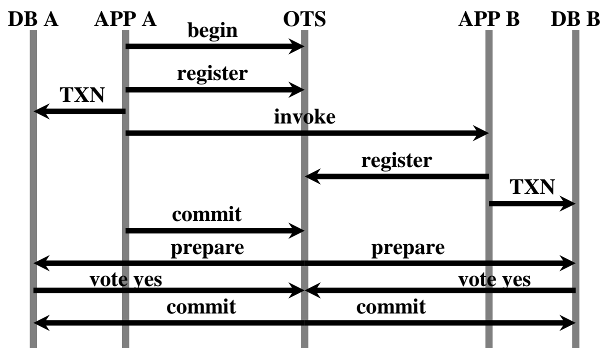



## Object Transaction Service






## OTS Sequence of Messages



# Data replication:
# Replication models

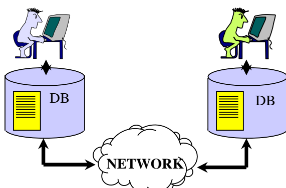


## Introduction to Database Replication

- What is database replication
- The advantages of database replication
- A taxonomy of replication strategies:
  - Synchronous
  - Asynchronous
  - Update everywhere
  - Primary copy
- Discussion on the various replication strategies.



## Database Replication

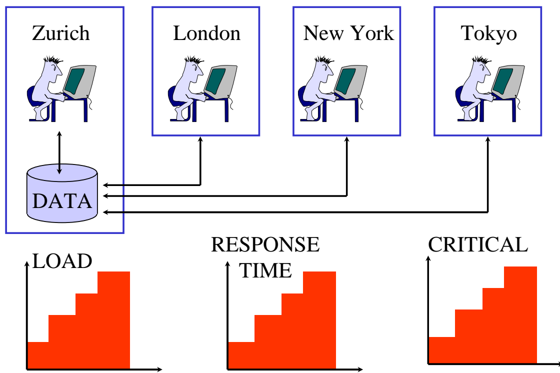



- Replication is a common strategy in data management: RAID technology (Redundant Array of Independent Disks), Mirror sites for web pages, Back up mechanisms (1-safe, 2-safe, hot/cold stand by)
- Here we will focus our attention on replicated databases but many of the ideas we will discuss apply to other environments as well.

Why replication?

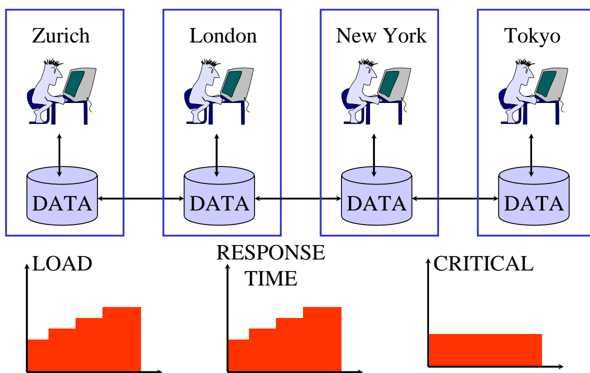
- PERFORMANCE: Location transparency is difficult to achieve in a distributed environment. Local accesses are fast, remote accesses are slow. If everything is local, then all accesses should be fast.
- FAULT TOLERANCE: Failure resilience is also difficult to achieve. If a site fails, the data it contains becomes unavailable. By keeping several copies of the data at different sites, single site failures should not affect the overall availability.
- APPLICATION TYPE: Databases have always tried to separate queries form updates to avoid interference. This leads to two different application types OLTP and OLAP, depending on whether they are update or read intensive.

## Remote access to data?






## Replication





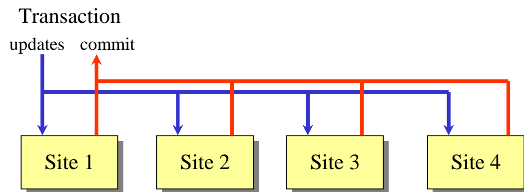

## How to replicate data?

- There are two basic parameters to select when designing a replication strategy: where and when.
- Depending on when the updates are propagated:
  - Synchronous (eager)
  - Asynchronous (lazy)
- Depending on where the updates can take place:
  - Primary Copy (master)
  - Update Everywhere (group)

| | master | group |
|---|---|---|
| Sync | | |
| Async | | |

## Synchronous Replication

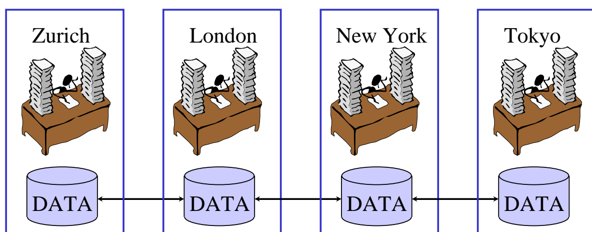
- Synchronous replication propagates any changes to the data immediately to all existing copies. Moreover, the changes are propagated within the scope of the transaction making the changes. The ACID properties apply to all copy updates.




©IKS, ETH Zürich.



## Synchronous Replication



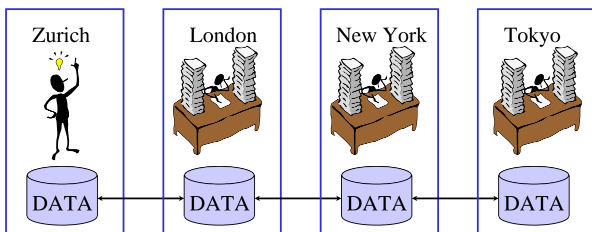

**Price = $ 50** **Price = $ 50** **Price = $ 50** **Price = $ 50**

**DATA IS CONSISTENT AT ALL SITES**

©IKS, ETH Zürich.



## Synchronous Replication

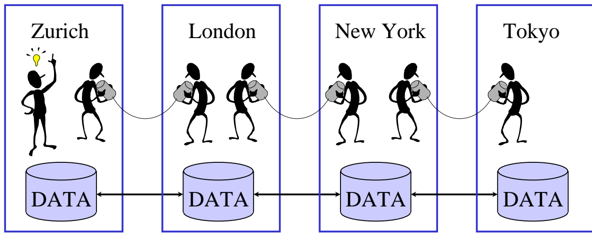



**Price = $ 50** **Price = $ 50** **Price = $ 50** **Price = $ 50**

**A SITE WANTS TO UPDATE THE PRICE ...**

©IKS, ETH Zürich.

## Synchronous Replication

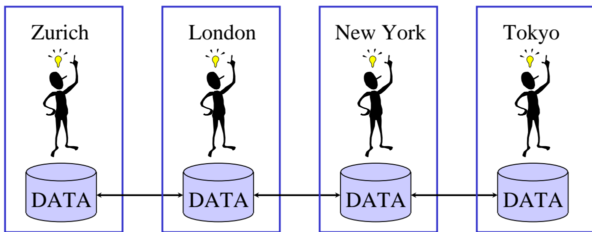



**Price = $ 50** **Price = $ 50** **Price = $ 50** **Price = $ 50**

**… IT FIRST CONSULTS WITH EVERYBODY ELSE ...**



## Synchronous Replication

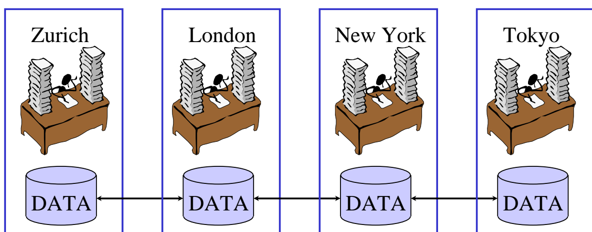



**Price = $ 50** **Price = $ 50** **Price = $ 50** **Price = $ 50**

**… AN AGREEMENT IS REACHED ...**



## Synchronous Replication

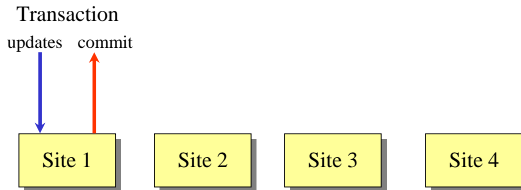



**Price = $ 100** **Price = $ 100** **Price = $ 100** **Price = $ 100**

**… THE PRICE IS UPDATED AND PROCESSING CONTINUES.**

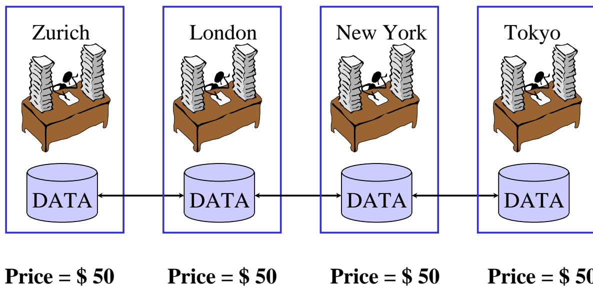
## Asynchronous Replication

- Asynchronous replication first executes the updating transaction on the local copy. Then the changes are propagated to all other copies. While the propagation takes place, the copies are inconsistent (they have different values).
- The time the copies are inconsistent is an adjustable parameter which is application dependent.






## Asynchronous Replication

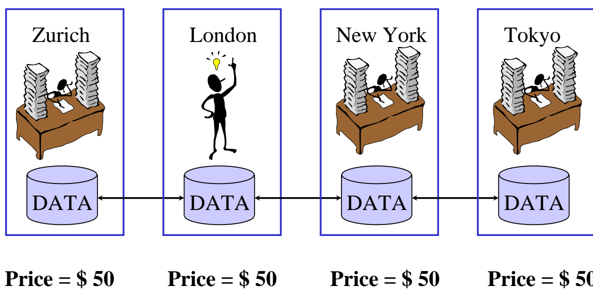



**Price = $ 50 Price = $ 50 Price = $ 50 Price = $ 50**

**DATA IS CONSISTENT AT ALL SITES**



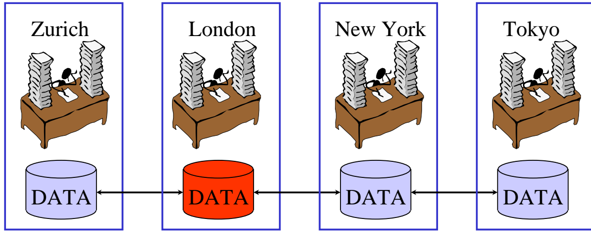
## Asynchronous Replication




**Price = $ 50 Price = $ 50 Price = $ 50 Price = $ 50**

**A SITE WANTS TO UPDATE THE PRICE ...**

## Asynchronous Replication

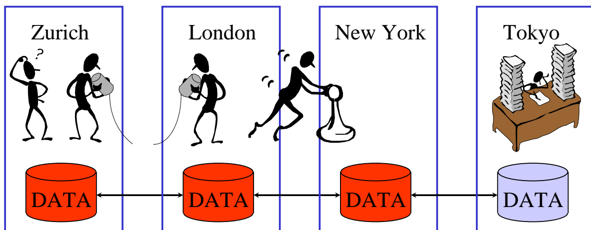



**Price = $ 50** **Price = $ 100** **Price = $ 50** **Price = $ 50**

**THEN IT UPDATES THE PRICE LOCALLY AND CONTINUES PROCESSING (DATA IS NOT CONSISTENT!)...**



## Asynchronous Replication



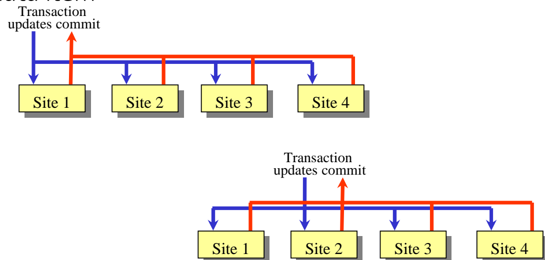

**Price = $ 100** **Price = $ 100** **Price = $ 100** **Price = $ 50**

**THE UPDATE IS EVENTUALLY PROPAGATED TO ALL SITES (PUSH, PULL MODELS)**

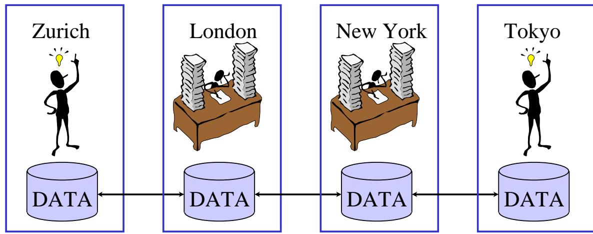


## Update Everywhere

- With an update everywhere approach, changes can be initiated at any of the copies. That is, any of the sites which owns a copy can update the value of the data item



## Update Everywhere



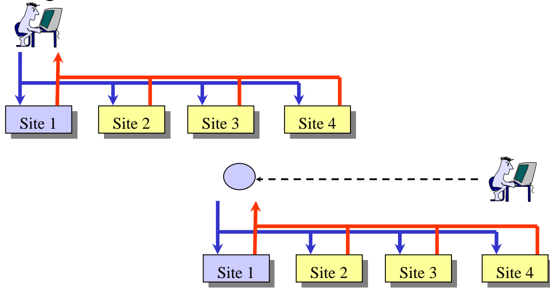

**Price = $ 50** **Price = $ 50** **Price = $ 50** **Price = $ 50**

**ALL SITES ARE ALLOWED TO UPDATE THEIR COPY**



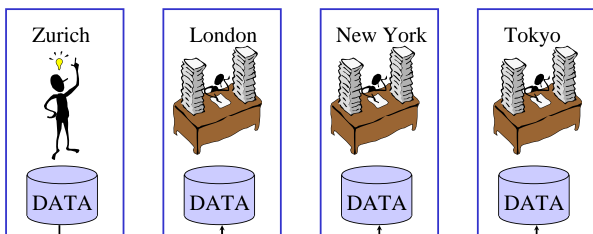
## Primary Copy

- With a primary copy approach, there is only one copy which can be updated (the master), all others (secondary copies) are updated reflecting the changes to the master.






## Primary Copy




**Price = $ 50** **Price = $ 50** **Price = $ 50** **Price = $ 50**

**ONLY ONE SITE IS ALLOWED TO DO UPDATES, THE OTHER ARE READ ONLY COPIES**

## Forms of replication

**Synchronous**
- Advantages:
  - No inconsistencies (identical copies)
  - Reading the local copy yields the most up to date value
  - Changes are atomic
- Disadvantages: A transaction has to update all sites (longer execution time, worse response time)

**Asynchronous**
- Advantages: A transaction is always local (good response time)
- Disadvantages:
  - Data inconsistencies
  - A local read does not always return the most up to date value
  - Changes to all copies are not guaranteed
  - Replication is not transparent

**Update everywhere**
- Advantages:
  - Any site can run a transaction
  - Load is evenly distributed
- Disadvantages:
  - Copies need to be synchronized

**Primary Copy**
- Advantages:
  - No inter-site synchronization is necessary (it takes place at the primary copy)
  - There is always one site which has all the updates
- Disadvantages:
  - The load at the primary copy can be quite large
  - Reading the local copy may not yield the most up to date value



## Replication Strategies

The previous ideas can be combined into 4 different replication strategies:

| | Primary copy | Update everywhere |
|---|---|---|
| **Synchronous (eager)** | synchronous primary copy | synchronous update everywhere |
| **Asynchronous (lazy)** | asynchronous primary copy | asynchronous update everywhere |



## Replication Strategies

| | Primary copy | Update everywhere |
|---|---|---|
| **Synchronous** | Advantages:<br>Updates not coordinated<br>No inconsistencies<br>Disadvantages:<br>Longest response time<br>Only useful with few updates<br>Local copies can only be read | Advantages:<br>No inconsistencies<br>Elegant (symmetrical solution)<br>Disadvantages:<br>Long response times<br>Updates need to be coordinated |
| **Asynchronous** | Advantages:<br>No coordination necessary<br>Short response times<br>Disadvantages:<br>Local copies are not up to date<br>Inconsistencies | Advantages:<br>No centralized coordination<br>Shortest response times<br>Disadvantages:<br>Inconsistencies<br>Updates can be lost (reconciliation) |

## Replication (Ideal)

| | Primary copy | Update everywhere |
|---|---|---|
| Synchronous (eager) | Globally correct Remote writes | Globally correct Local writes |
| Asynchronous (lazy) | Inconsistent reads | Inconsistent reads Reconciliation |



## Replication (Practical)

| | Primary copy | Update everywhere |
|---|---|---|
| Synchronous (eager) | Too Expensive (usefulness?) | Too expensive (does not scale) |
| Asynchronous (lazy) | Feasible | Feasible in some applications |



## Summary - I

- Replication is used for performance and fault tolerant purposes.
- There are four possible strategies to implement replication solutions depending on whether it is synchronous or asynchronous, primary copy or update everywhere.
- Each strategy has advantages and disadvantages which are more or less obvious given the way they work.
- There seems to be a trade-off between correctness (data consistency) and performance (throughput and response time).
- The next step is to analyze these strategies in more detail to better understand how they work and where the problems lie.

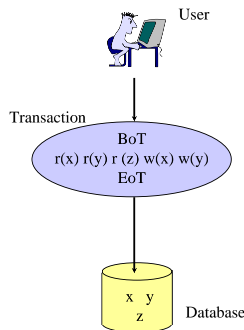
## Database Replication Strategies

- Database environments
- Managing replication
- Technical aspects and correctness/performance issues of each replication strategy:
  - Synchronous - primary copy
  - Synchronous - update everywhere
  - Asynchronous - primary copy
  - Asynchronous - update everywhere



## Basic Database Notation

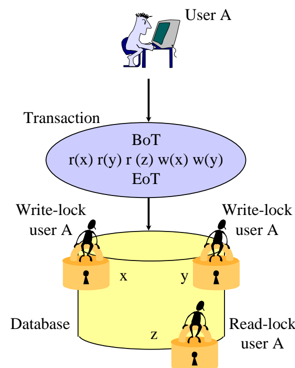
- A user interacts with the database by issuing read and write operations.
- These read and write operations are grouped into transactions with the following properties:
  - Atomicity: either all of the transaction is executed or nothing at all.
  - Consistency: the transaction produces consistent changes.
  - Isolation: transactions do not interfere with each other.
  - Durability: Once the transaction commits, its changes remain.





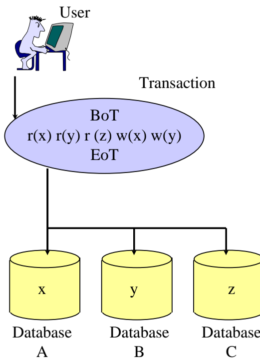

## Isolation

- Isolation is guaranteed by a concurrency control protocol.
- In commercial databases, this is usually 2 Phase Locking (2PL):
  - conflicting locks cannot coexist (writes conflict with reads and writes on the same item)
  - Before accessing an item, the item must be locked.
  - After releasing a lock, a transaction cannot obtain any more locks.



## Atomicity

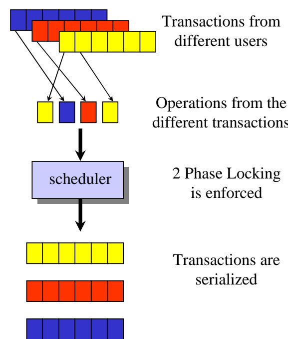
- A transaction must commit all its changes.
- When a transaction executes at various sites, it must execute an atomic commitment protocol, i.e., it must commit at all sites or at none of them.
- Commercial systems use 2 Phase Commit:
  - A coordinator asks everybody whether they want to commit
  - If everybody agrees, the coordinator sends a message indicating they can all commit






## Transaction Manager

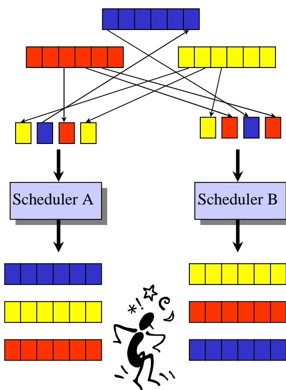
- The transaction manager takes care of isolation and atomicity.
- It acquires locks on behalf of all transactions and tries to come up with a serializable execution, i.e., make it look like the transactions were executed one after the other.
- If the transactions follow 2 Phase Locking, serializability is guaranteed. Thus, the scheduler only needs to enforce 2PL behaviour.





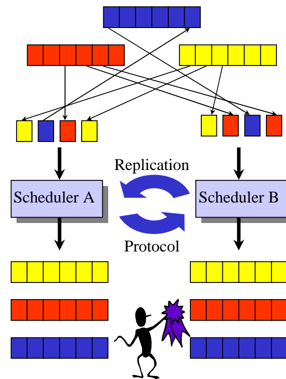

## Managing Replication

- When the data is replicated, we still need to guarantee atomicity and isolation.
- Atomicity can be guaranteed by using 2 Phase Commit. This is the easy part.
- The problem is how to make sure the serialization orders are the same at all sites, i.e., make sure that all sites do the same things in the same order (otherwise the copies would be inconsistent).



## Managing Replication

- To avoid this, replication protocols are used.
- A replication protocol specifies how the different sites must be coordinated in order to provide a concrete set of guarantees.
- The replication protocols depend on the replication strategy (synchronous, asynchronous, primary copy, update everywhere).





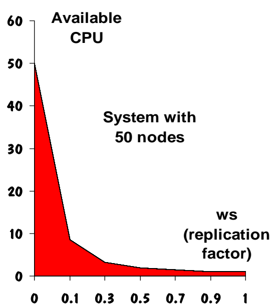

## Replication Strategies

Now we can analyze the advantages and disadvantages of each strategy:

| | Primary copy | Update everywhere |
|---|---|---|
| **Synchronous (eager)** | synchronous primary copy | synchronous update everywhere |
| **Asynchronous (lazy)** | asynchronous primary copy | asynchronous update everywhere |



## Cost of Replication




- Assume a 50 node replicated system where a fraction s of the data is replicated and w represents the fraction of updates made (ws = replication factor)
- Overall computing power of the system:

$$\frac{N}{1 + w \cdot s \cdot (N - 1)}$$

- No performance gain with large ws factor (many updates or many replicated data items)
- Reads must be local to get performance advantages.

## Synchronous - update everywhere

Assume all sites contain the same data.

READ ONE-WRITE ALL

- Each sites uses 2 Phase Locking.
- Read operations are performed locally.
- Write operations are performed at all sites (using a distributed locking protocol).

This protocol guarantees that every site will behave as if there were only one database. The execution is serializable (correct) and all reads access the latest version.

This simple protocol illustrates the main idea behind replication, but it needs to be extended in order to cope with realistic environments:

- Sites fail, which reduces the availability (if a site fails, no copy can be written).
- Sites eventually have to recover (a recently recovered site may not have the latest updates).



## Dealing with Site Failures

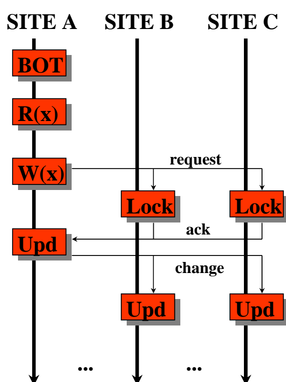
Assume, for the moment, that there are no communication failures. Instead of writing to all copies, we could

WRITE ALL AVAILABLE COPIES

- READ = read any copy, if time-out, read another copy.
- WRITE = send Write(x) to all copies. If one site rejects the operation, then abort. Otherwise, all sites not responding are "missing writes".
- VALIDATION = To commit a transaction
  - Check that all sites in "missing writes" are still down. If not, then abort the transaction.
  - Check that all sites that were available are still available. If some do not respond, then abort.

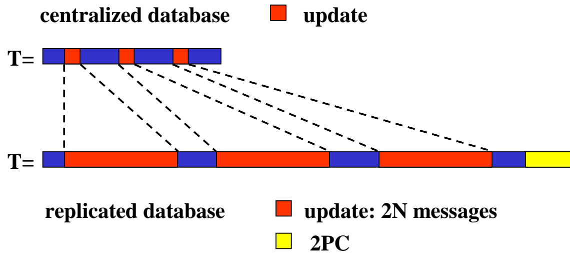


## Synchronous - Update Everywhere Protocol




- Each site uses 2PL
- Read operations are performed locally
- Write operations involve locking all copies of the data item (request a lock, obtain the lock, receive an acknowledgement)
- The transaction is committed using 2PC
- Main optimizations are based on the idea of quorums (but all we will say about this protocol also applies to quorums)

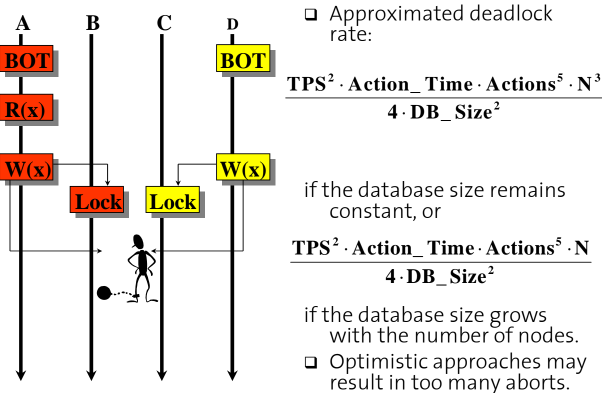
## Response Time and Messages




**The way replication takes place (one operation at a time), increases the response time and, thereby, the conflict profile of the transaction. The message overhead is too high (even if broadcast facilities are available).**



## The Deadlock Problem




- Approximated deadlock rate:

$$\frac{TPS^2 \cdot Action_Time \cdot Actions^5 \cdot N^3}{4 \cdot DB_Size^2}$$

if the database size remains constant, or

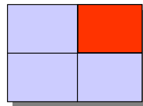
$$\frac{TPS^2 \cdot Action_Time \cdot Actions^5 \cdot N}{4 \cdot DB_Size^2}$$

if the database size grows with the number of nodes.

- Optimistic approaches may result in too many aborts.



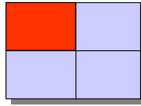
## Synchronous - update everywhere



- Advantages:
  - No inconsistencies
  - Elegant (symmetrical solution)
- Disadvantages:
  - Very high number of messages involved
  - Transaction response time is very long
  - The system will not scale because of deadlocks (as the number of nodes increases, the probability of getting into a deadlock gets too high)

Data consistency is guaranteed. Performance may be seriously affected with this strategy. The system may also have scalability problems (deadlocks). High fault tolerance.

## Synchronous - primary copy

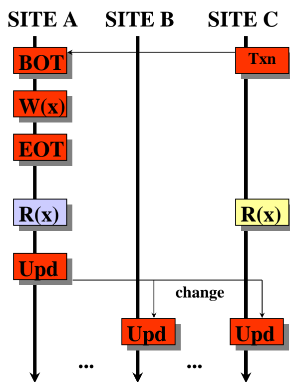


- Advantages:
  - Updates do not need to be coordinated
  - No inconsistencies, no deadlocks.
- Disadvantages:
  - Longest response time
  - Only useful with few updates (primary copy is a bottleneck)
  - Local copies are almost useless
  - Not used in practice

Similar problems to those of Sync - update everywhere. Including scalability problems (bottlenecks). Data consistency is guaranteed. Fault tolerant.



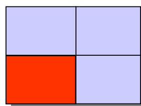
## Async - primary copy protocol




- Update transactions are executed at the primary copy site
- Read transactions are executed locally
- After the transaction is executed, the changes are propagated to all other sites
- Locally, the primary copy site uses 2 Phase Locking
- In this scenario, there is no atomic commitment problem (the other sites are not updated until later)



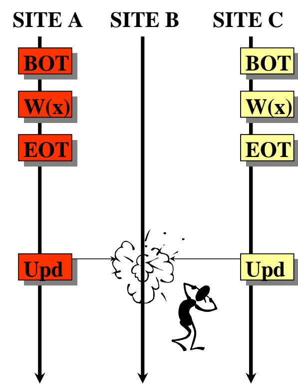
## Asynchronous - primary copy



- Advantages:
  - No coordination necessary
  - Short response times (transaction is local)
- Disadvantages:
  - Local copies are not up to date (a local read will not always include the updates made at the local copy)
  - Inconsistencies (different sites have different values of the same data item)

Performance is good (almost same as if no replication). Fault tolerance is limited. Data inconsistencies arise.

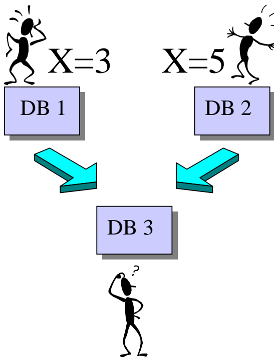
## Async - update everywhere protocol




- All transactions are executed locally
- After the transaction is executed, the changes are propagated to all other sites
- Locally, a site uses 2 Phase Locking
- In this scenario, there is no atomic commitment problem (the other sites are not updated until later)
- However, unlike with primary copy, updates need to be coordinated



## Async / Update Everywhere




- Probability of needing reconciliation:

$$\frac{TPS^2 \cdot Action_time \cdot Actions^3 \cdot N^3}{2 \cdot DB_Size}$$

- What does it mean to commit a transaction locally? There is no guarantee that a committed transaction will be valid (it may be eliminated if "the other value" wins).

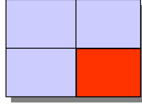


## Reconciliation

- Such problems can be solved using pre-arranged patterns:
  - Latest update win (newer updates preferred over old ones)
  - Site priority (preference to updates from headquarters)
  - Largest value (the larger transaction is preferred)
- or using ad-hoc decision making procedures:
  - identify the changes and try to combine them
  - analyze the transactions and eliminate the non-important ones
  - implement your own priority schemas

## Asynchronous - update everywhere

- Advantages:
  - No centralized coordination
  - Shortest response times
- Disadvantages:
  - Inconsistencies
  - Updates can be lost (reconciliation)



Performance is excellent (same as no replication). High fault tolerance. No data consistency. Reconciliation is a tough problem (to be solved almost manually).



## Summary - II

- We have seen the different technical issues involved with each replication strategy
- Each replication strategy has well defined problems (deadlocks, reconciliation, message overhead, consistency) related to the way the replication protocols work
- The trade-off between correctness (data consistency) and performance (throughput and response time) is now clear
- The next step is to see how these ideas are implemented in practice

## Data replication: Data replication systems

## Replication in Practice

- Replication scenarios
- On Line Transaction Processing (OLTP)
- On Line Analytical Processing (OLAP)
- Replication in Sybase
- Replication in IBM
- Replication in Oracle
- Replication in Domino (Lotus Notes)

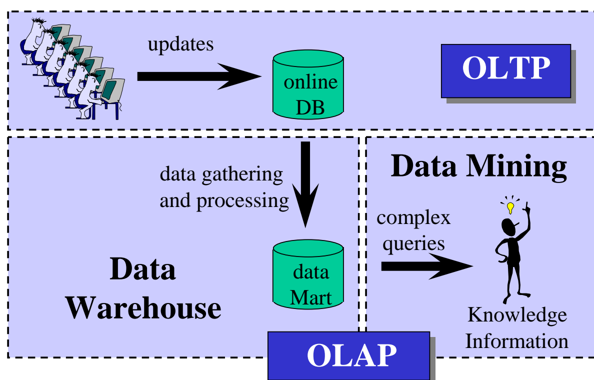


## Replication Scenarios

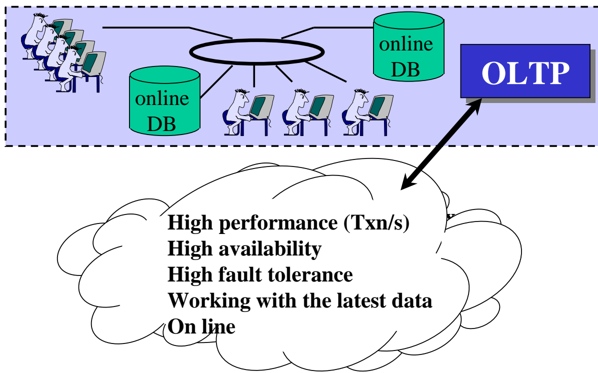
- In practice, replication is used in many different scenarios. Each one has its own demands. A commercial system has to be flexible enough to implement several of these scenarios, otherwise it would not be commercially viable.
- Database systems, however, are very big systems and evolve very slowly. Most were not designed with replication in mind. Commercial solutions are determined by the existing architecture, not necessarily by a sound replication strategy. Replication is fairly new in commercial databases!
- The focus on OLTP and OLAP determines the replication strategy in many products.
- From a practical standpoint, the trade-off between correctness and performance seems to have been resolved in favor of performance.
- It is important to understand how each system works in order to determine whether the system will ultimately scale, perform well, require frequent manual intervention ...



## OLTP vs. OLAP

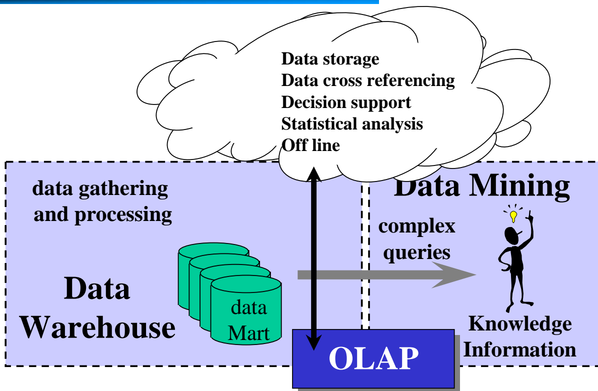


## OLTP






## OLAP






## Commercial replication

When evaluating a commercial replication strategy, keep in mind:

- The customer base (who is going to use it?).
- The underlying database (what can the system do?).
- What competitors are doing (market pressure).
- There is no such a thing as a “better approach”.
- The complexity of the problem.

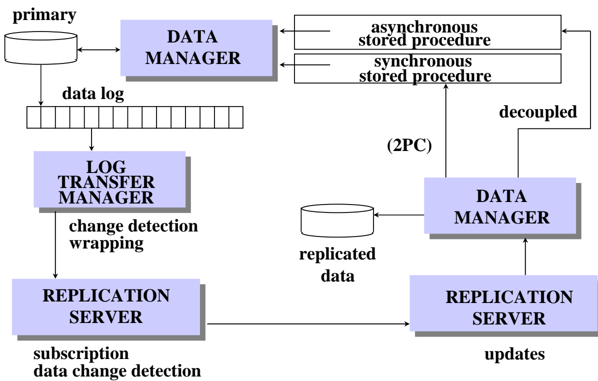
Replication will keep evolving in the future, current systems may change radically.

## Sybase Replication Server

- Goal of replication: Avoid server bottlenecks by moving data to the clients. To maintain performance, asynchronous replication is used (changes are propagated only after the transaction commits). The changes are propagated on a transaction basis (get the replicas up-to-date as quickly as possible). Capture of changes is done "off-line", using the log to minimize the impact on the running server.
- Applications: OLTP, client/server architectures, distributed database environments.



## Sybase Replication Architecture






## Sybase Replication (basics)

- Loose consistency (= asynchronous). Primary copy.
- PUSH model: replication takes place by "subscription". A site subscribes to copies of data. Changes are propagated from the primary as soon as they occur. The goal is to minimize the time the copies are not consistent but still within an asynchronous environment (updates are sent only after they are committed).
- Updates are taken from the log in stable storage (only committed transactions).
- Remote sites update using special stored procedures (synchronous or a synchronous).
- Persistent queues are used to store changes in case of disconnection.
- The Log Transfer Manager monitors the log of Sybase SQL Server and notifies any changes to the replication server. It acts as a light weight process that examines the log to detect committed transactions (a wrapper). It is possible to write your own Log Transfer Manager for other systems. When a transaction is detected, its log records are sent to the:
- The Replication Server usually runs on a different system than the database to minimize the load. It takes updates, looks who is subscribed to them and send them to the corresponding replication servers at the remote site. Upon receiving these changes, a replication server applies them at the remote site.

## Sybase Replication (updates)

Primary copy. All updates must be done at the primary using either :

- Synchronous stored procedures, which reside at the primary and are invoked (RPC) by any site who wants to update. 2 Phase Commit is used.
- Stored procedures for asynchronous transactions: invoked locally, but sent asynchronously to the primary for execution. If the transaction fails manual intervention is required to fix the problem.
- It is possible to fragment a table and make different sites the primary copy for each fragment.
- It is possible to subscribe to selections of tables using WHERE clauses.

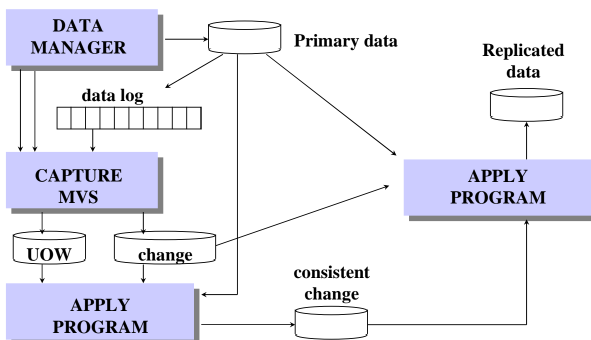


## IBM Data Propagator

- Goal: Replication is seen as part of the "Information Warehousing" strategy. The goal is to provide complex views of the data for decision-support. The source systems are usually highly tuned, the replication system is designed to interfere as less as possible with them: replication is asynchronous and there are no explicit mechanisms for updating.
- Applications: OLAP, decision-support, data warehousing, data mining.



## IBM Replication (architecture)



## IBM Data Propagator (basics)

- Asynchronous replication.
- No explicit update support (primary copy, if anything).
- PULL MODEL: (smallest interval 1 minute) the replicated data is maintained by querying either the primary data, the change table, the consistent change table, or any combination of the three. The goal is to support sophisticated views of the data (data warehousing). Pull model means replication is driven by the recipient of the replica. The replica must "ask" for updates to keep up-to-date.
- Updates are taken from the main memory buffer containing log entries (both committed and uncommitted entries; this is an adjustable parameter).
- Updates are sent to the primary (updates converted into inserts if tuple has been deleted, inserts converted into updates if tuple already exists, as in Sybase). The system is geared towards decision support, replication consistency is not a key issue.
- Sophisticated data replication is possible (base aggregation, change aggregation, time slices ...)
- Sophisticated optimizations for data propagation (from where to get the data).
- Sophisticated views of the data (aggregation, time slicing).
- Capture/MVS is a separate address space monitor, to minimize interference it captures log records from the log buffer area



## IBM Data Propagator

There are two key components in the architecture:

- Capture: analyzes raw log information from the buffer area (to avoid I/O). It reconstructs the logical log records and creates a "change table" and a "transaction table" (a dump of all database activity).
- Apply Program: takes information from the database, the change table and the transaction table to built "consistent change table" to allow consistent retrieval and time slicing. It works by "refreshing" data (copies the entire data source) or "updating" (copies changes only). It allows very useful optimizations (get the data from the database directly, reconstruct, etc.).

The emphasis is on extracting information:

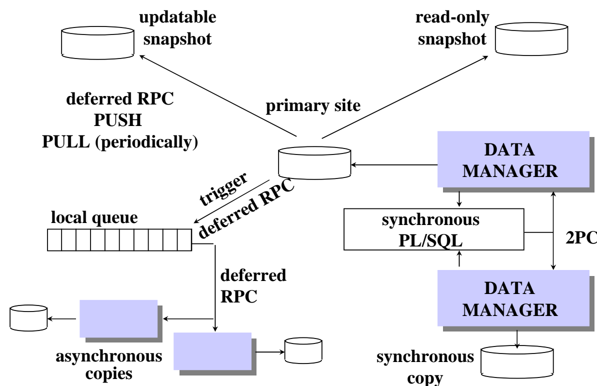
- Data Propagator/2 is used to subscribe and request data.
- It is possible to ask for the state of data at a given time (time slicing or snapshots).
- It is possible to ask for changes:
  - how many customers have been added?
  - how many customers have been removed?
  - how many customers were between 20 and 30 years old?
- This is not the conventional idea of replication!



## Oracle Symmetric Replication

- Goals: Flexibility. It tries to provide a platform that can be tailored to as many applications as possible. It provides several approaches to replication and the user must select the most appropriate to the application. There is no such a thing as a "bad approach", so all of them must be supported (or as many as possible)
- Applications: intended for a wide range of applications.

## Oracle Replication (architecture)






## Oracle Replication

- “DO-IT-YOURSELF” model supporting almost any kind of replication (push model, pull model), Dynamic Ownership (the site designated as the primary can change over time), and Shared Ownership (update anywhere, asynchronously).
- One of the earliest implementations: Snapshot. This was a copy of the database. Refreshing was done by getting a new copy.
- Symmetric replication: changes are forwarded at time intervals (push) or on demand (pull).
- Asynchronous replication is the default but synchronous is also possible.
- Primary copy (static / dynamic) or update everywhere.
- Readable Snapshots: A copy of the database. Refresh is performed by examining the log records of all operations performed, determining the changes and applying them to the snapshot. The snapshot cannot be modified but they are periodically refreshed (complete/fast refreshes)
- Writable Snapshots: fast-refreshable table snapshots but the copy can be updated (if changes are sent to the master copy, it becomes a form of asynchronous - update everywhere replication).



## Oracle Replication (basics)

Replication is based on these two ideas:

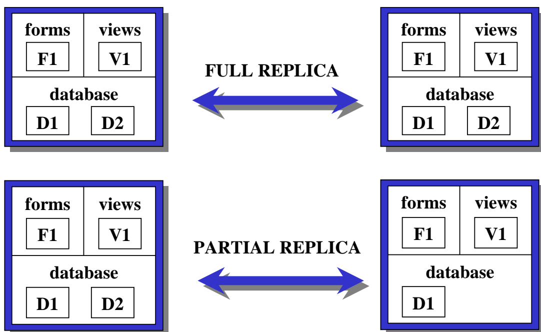
- Triggers: changes to a copy are captured by triggers. The trigger executes a RPC to a local queue and it inserts the changes in the queue. These changes take the form of an invocation to a stored procedure at the remote site. These triggers are “deferred” in the sense that they work asynchronously with respect to the transaction
- Queues: queues follow a FIFO discipline and 2PC is used to guarantee the call makes it to the queue at the remote site. At the remote site, the queue is read and the call made in the order they arrive.
- Dynamic ownership: It is possible to dynamically reassign the “master copy” to different sites. That is, the primary copy can move around (doing it well, it is then possible to always read and write locally)
- Shared ownership: (= update everywhere!). Conflicts are detected by propagating both the before and the after image of data. When a conflict is detected, there are several predefined routines that can be automatically called or the user can write and ad-hoc routine to resolve the conflict
- Synchronous, update everywhere: using the sync -update everywhere protocol previously discussed

## Replication in Lotus Notes (Domino)

- Lotus Notes implements asynchronous (lazy), update every-where replication in an epidemic environment.
- Lotus Notes distinguishes between a replica and a copy (a snapshot). All replicas have the same id. Each copy has its own id.
- Lotus allows to specify what to replicate (in addition to replica stubs and field level replication) to minimize overhead.
- Replication conflicts are detected and some attempt is made at reconciliation (user intervention is usually required).
- Lotus Notes is a cooperative environment, the goal is data distribution and sharing. Consistency is largely user defined and not enforced by the system.



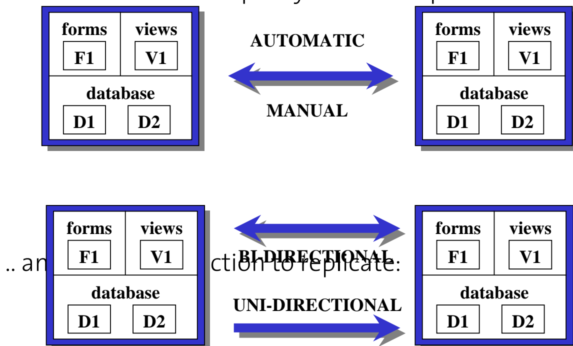
## Replication in Lotus Notes






## Replication in Lotus Notes

Notes also allows to specify when to replicate ...




.. an ... ction to replicate:

# Data replication: Additional protocols



## Token Passing Protocol

Replication is used in many applications other than databases. For these applications, there is a large number of protocols and algorithms that can be used to guarantee "correctness":

- The token based protocol is used as an example of replication in distributed systems to illustrate the problems of fault-tolerance and starvation.



## Distributed Mutual Exclusion

- The original protocol was proposed for distributed mutual exclusion. It can be used, however, to maintain replicated data and to implement the notion of dynamic ownership (Oracle replication).

In here, it will be used for the following:

- Asynchronous, master copy (dynamic ownership)
- The protocol will be used to locate the master copy
- Requirements:
  - there is only one master copy at all times
  - deadlock free
  - fault-tolerant
  - starvation free

## Token Passing (model)

Working assumptions

- Communications are by message passing
- Sites are fail-stop or may fail to send and receive messages
- Failed sites eventually recover (failure detection by time-out)
- Network partitions may occur
- No duplicate messages and FIFO delivery
- Causality enforced by logical clocks (Lamport)

| Happen Before Relation → | Clock condition |
|---|---|
| (1) events in a process are ordered<br>(2) sending(m) → receiving(m)<br>(3) if a → b and b → c, then a → c | (1) each event has a timestamp<br>(2) succesive events have increasing timestamps<br>(3) receiving(m) has a higher timestamp than sending(m) |

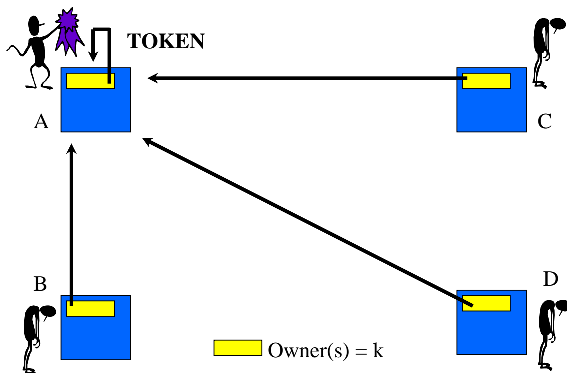


## Basic Protocol (no failures)

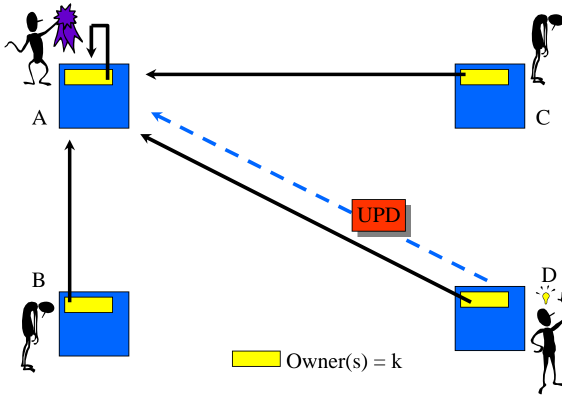
- Assume no communication or site failures
- A node with the token is the master copy
- Each site, s, has a pointer, Owner(s), indicating where that site believes the master copy is located
- The master copy updates locally
- Other sites sent their updates following the pointer
- When the master copy reassigns the token (the master copy moves to another site), the ex-master copy readjusts its pointer so it points towards the new master copy
- For correctness reasons, assume the master copy is never reassigned while updates are taking place.



## Basic Protocol (owner)

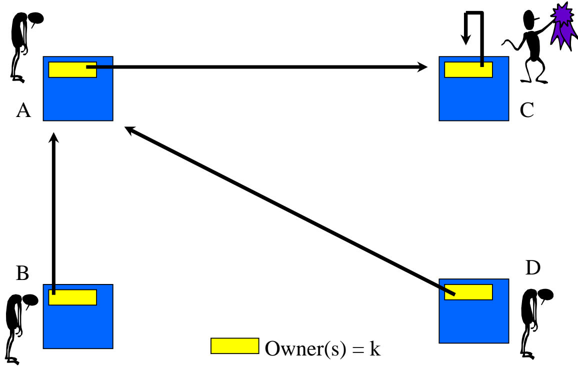


## Basic Protocol (update)



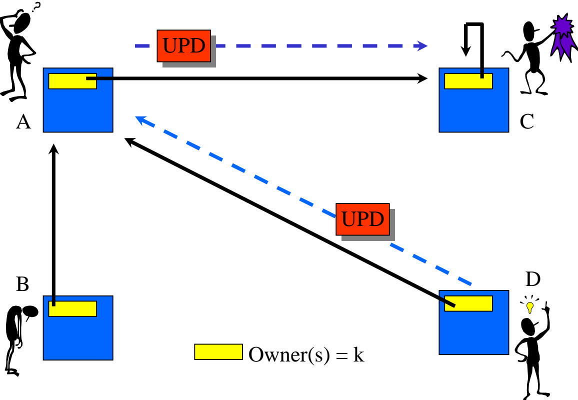

©IKS, ETH Zürich.


## Basic Protocol (token change)



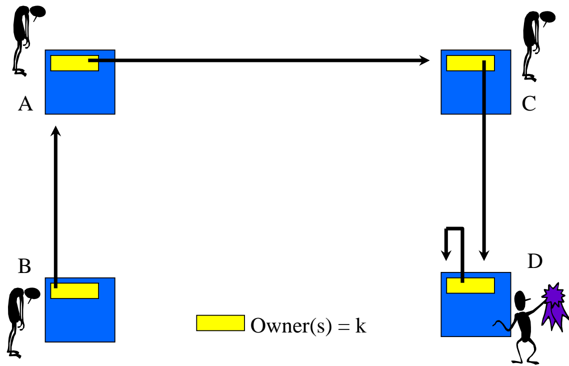

©IKS, ETH Zürich.


## Basic Protocol (update)



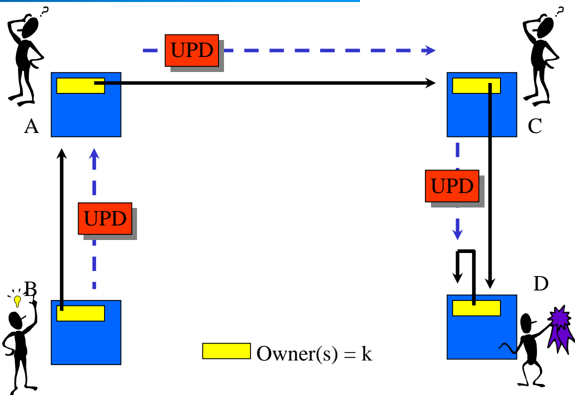

©IKS, ETH Zürich.

## Basic Protocol (token change)




©IKS, ETH Zürich.


## Basic Protocol (update)




©IKS, ETH Zürich.


## Basic Protocol (algorithms)

| Requesting the master copy (s) | Receiving a request (q) |
|---|---|
| IF Owner(s) = s THEN<br>  master copy already in s<br>ELSE<br>  SEND(request) to Owner(s)<br>  RECEIVE(Token)<br>  Owner(s) = s<br>END (*IF*) | Receive (request(s))<br>IF Owner(q) = q THEN<br>  Owner(q) = s<br>  SEND(Token) to s<br>ELSE<br>  SEND(request(s)) to Owner(q)<br>END (*IF*) |

©IKS, ETH Zürich.

## Failures

If communication failures occur, the token may disappear while in transit (message is lost).

- First, the loss of the token must be detected
- Second, the token must be regenerated
- Third, after the regeneration, there must be only one token in the system (only one master copy)

To do this, logical clocks are used:

- OwnerTime(s) is a logical clock associated with the token, it indicates when site s sent or received the token
- TokenState(s) is the state of the shared resource (values associated with the token itself)

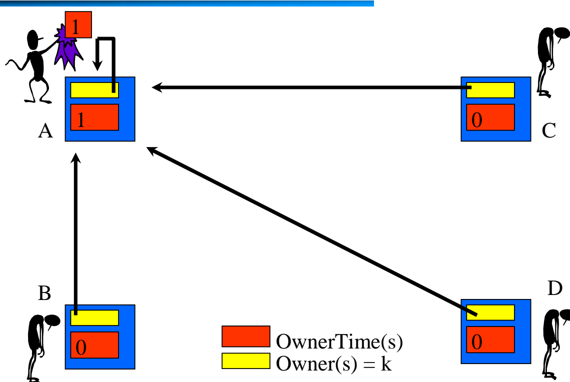


## Token Loss Protocol

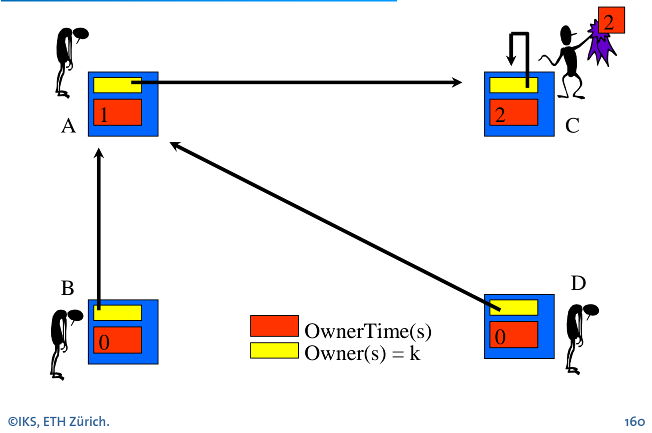
- Assume bounded delay (if a message does not arrive after time t, it has been lost). Sites do not fail
- When a site sends the token, it sends along its own OwnerTime
- When a site receives the token, it sets its OwnerTime to a value greater than that received with the token
- From here, it follows that the values of the OwnerTime variables along the chain of pointers must increase
- If, along the chain of pointers, there is a pair of values that is not increasing, the token has been lost between these two sites and must be regenerated



## Token Loss Protocol

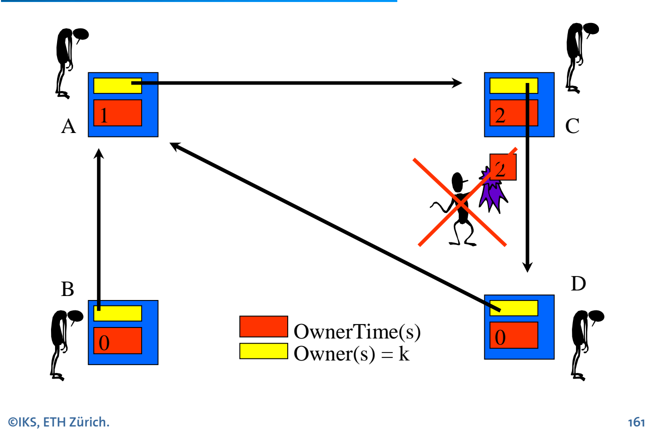


## Detecting Token Loss



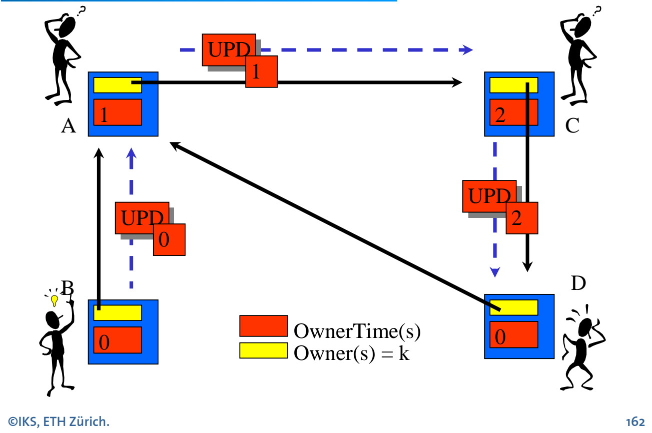



## Detecting Token Loss

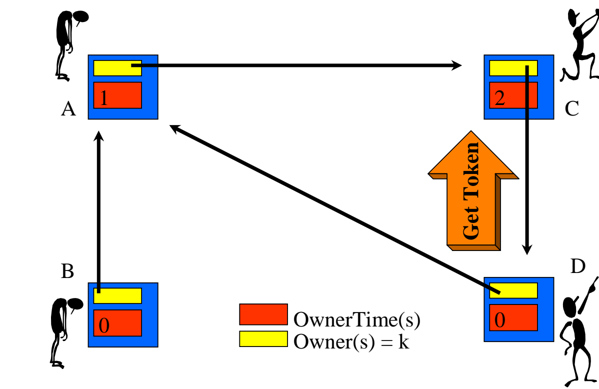





## Detecting Token Loss

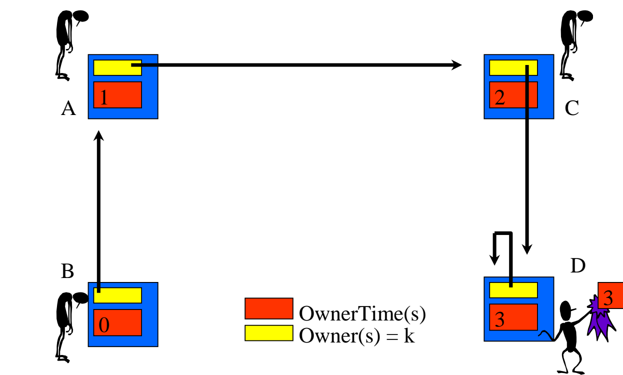


## Regenerating the Token






## Token Recovered






## Token Loss (algorithm 1)

```
Request (s)
IF Owner(s) = s THEN
    already master copy
ELSE
    SEND(request(s),OwnerTime(s)) to Owner(s)
    Receive(Token,TTime) on Timeout(ReqDelay) ABORT
    Owner(s) = s
    OwnerTime(s) = TTime + 1
    TokenState = Token
END (*IF*)
```

## Token Loss (algorithm 2)

```
Receive (request(s),timestamp) FROM p
IF timestamp > OwnerTime(q) THEN (* TOKEN IS LOST *)
   SEND(GetToken) TO p
   Receive(Token,TTime) FROM p ON Timeout ABORT
   Owner(q) = q
   OwnerTime(q) = TTime + 1
   TokenState = Token
END (*IF*)
IF Owner(q) <> q THEN
    SEND(request(s),timestamp) TO Owner(q)
ELSE
   Owner(q) = s
   SEND(Token, OwnerTime(q)) TO s
END (*IF*)
```

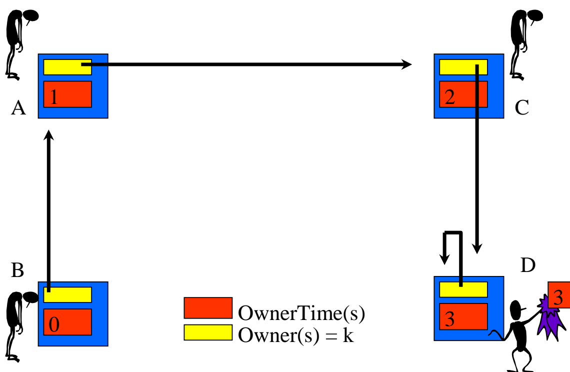


## Site Failures

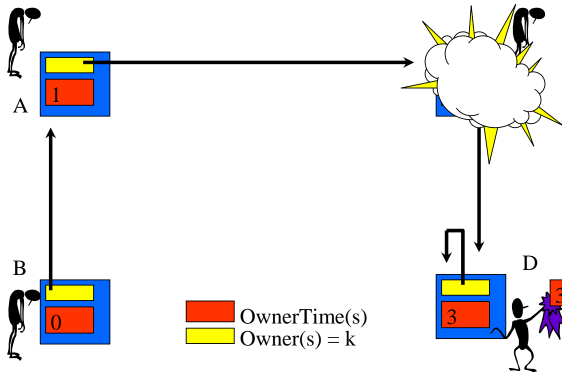
- Sites failures interrupt the chain of pointers (and may also result in the token being lost, if the failed site had the token)
- In this case, the previous algorithm ABORTs the protocol
- Instead of aborting, and to tolerate site failures, a broadcast algorithm can be used to ask everybody and find out what has happened in the system
- Two “states” are used
  - TokenReceived: the site has received the token
  - TokenLoss: a site determines that somewhere in the system there are p,q such that Owner(p) = q and OwnerTime(p) > OwnerTime(q)



## Chain Loss due to Site Failure

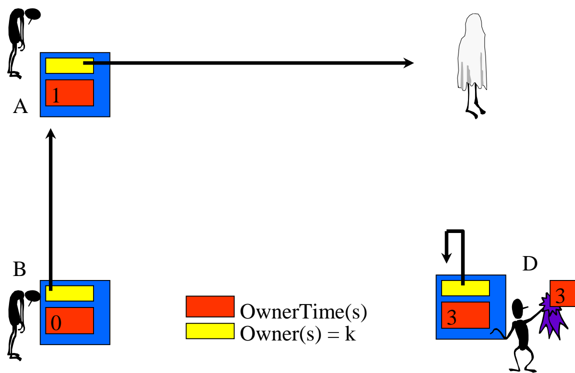


## Chain Loss due to Site Failure



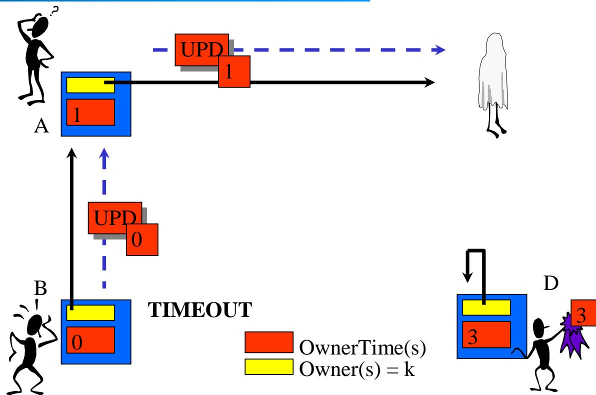



## Chain Loss due to Site Failure

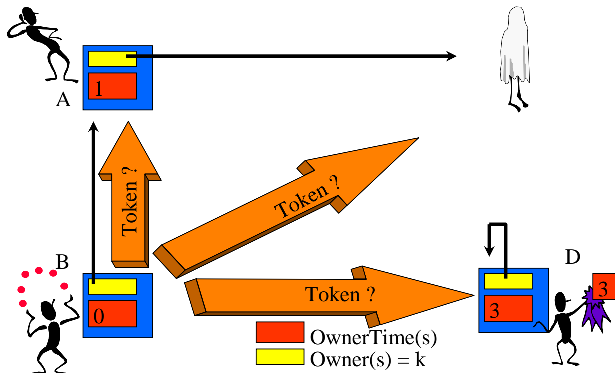





## Chain Loss due to Site Failure

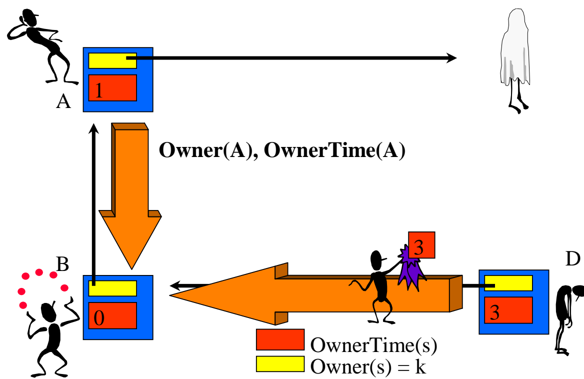


## Token Loss due to Site Failure



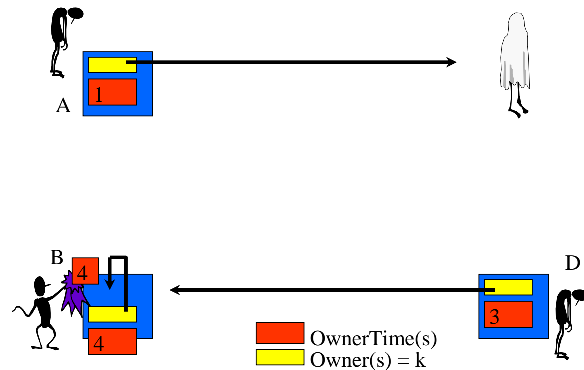



## Token Loss due to Site Failure

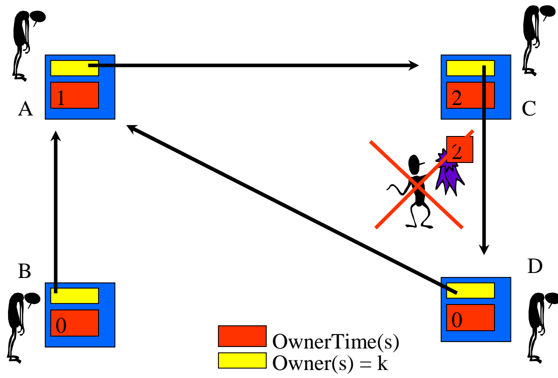





## Chain Loss due to Site Failure

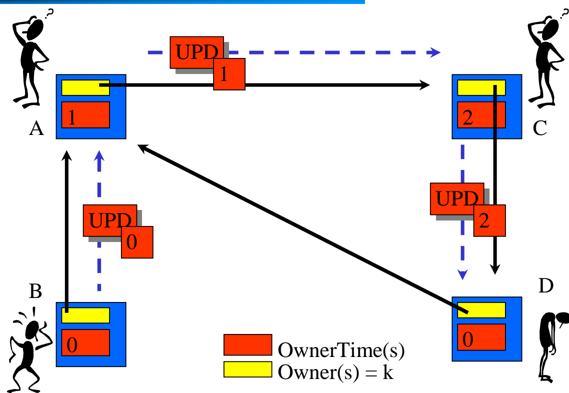


## Detecting Token Loss in Others



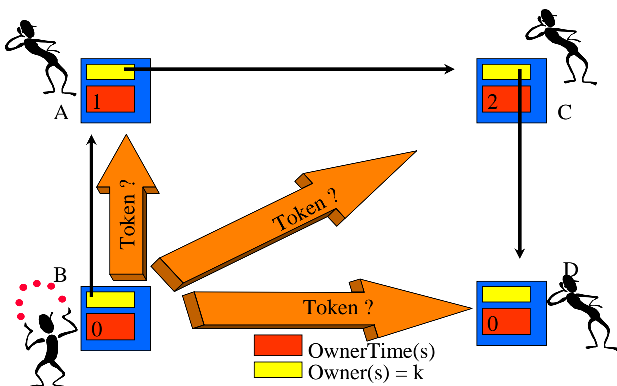



## Detecting Token Loss in Others

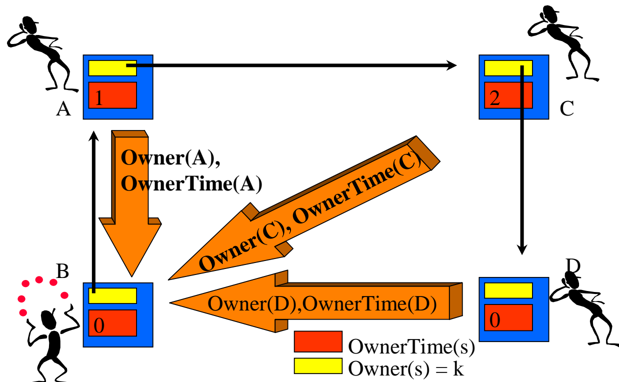





## Detecting Token Loss in Others

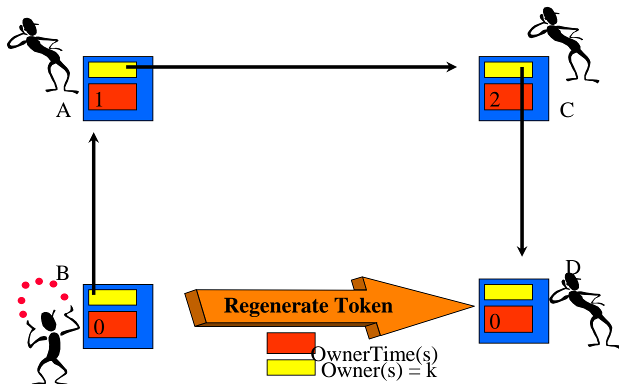


## Detecting Token Loss in Others



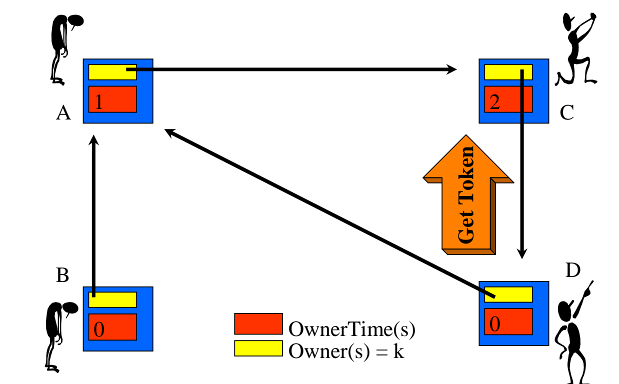



## Regenerating Token in Others

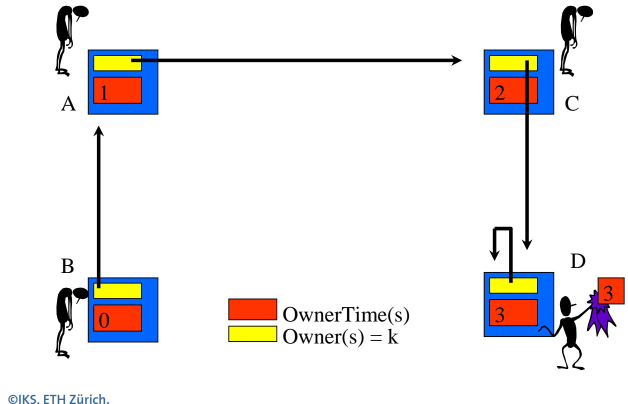





## Regenerating the Token



## Token Recovered






## Broadcast (algorithm)

SITE s: SEND (Bcast) TO all sites
COLLECT replies UNTIL TokenReceived OR TokenLoss
IF TokenReceived THEN
    Owner(s) = s
    OwnerTime = TTime + 1
    TokenState = Token
END (*IF*)
IF TokenLoss THEN
    DetectionTime = OwnerTime(q)
    SEND(Regenerate, DetectionTime, p) TO q
    RESTART
END (*IF*)



## Broadcast Request (algorithm)

Broadcast Request arrives at q from s:
Receive(Bcast)
IF Owner(q) = q THEN
    Owner(q) = s
    SEND(Token,OwnerTime(q)) TO s
ELSE
    SEND(Owner(q),OwnerTime(q)) TO s
END (*IF*)

## Regenerate Token (algorithm)

```
A request to regenerate the token arrives at q:
Receive(Regenerate, DetectionTime, p)
IF OwnerTime(q) = DetectionTime THEN
   SEND(GetToken) TO p
   Receive(Token,TTime) FROM p ON Timeout ABORT
   Owner(q) = q
   OwnerTime(q) = TTime + 1
   TokenState = Token
END (*IF*)
```

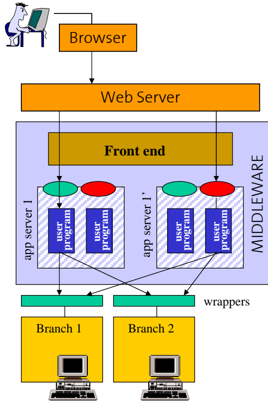


## Starvation

- Starvation can occur if a request for the token keeps going around the system behind the token but it always arrives after another request
- One way to solve this problem is to make a list of all requests, order the requests by timestamp and only grant a request when it is the one with the lowest timestamp in the list
- The list can be passed around with the token and each site can keep a local copy of the list that will be merged with that arriving with the token (thereby avoiding that requests get lost in the pointer chase)

## Web services
### Background

## The Web as software layer (N-tier)

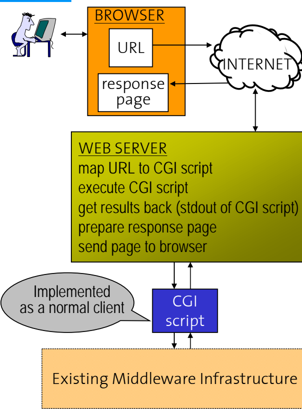



- N-tier architectures result from connecting several three tier systems to each other and/or by adding an additional layer to allow clients to access the system through a Web server
- The Web layer was initially external to the system (a true additional layer); today, it is slowly being incorporated into a presentation layer that resides on the server side (part of the middleware infrastructure in a three tier system, or part of the server directly in a two tier system)
- The addition of the Web layer led to the notion of "application servers", which was used to refer to middleware platforms supporting access through the Web



## WWW basics

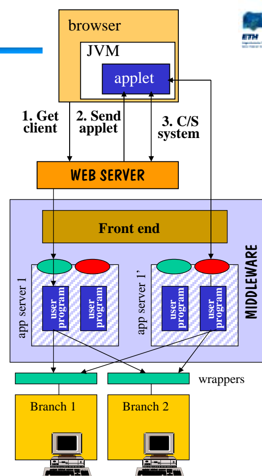
- The earliest implementations were very simple and built directly upon the existing systems (client/server based on RPC, TP-Monitors, or any other form of middleware which allowed interaction through a programmable client)
  - the CGI script (or program) acted as client in the traditional sense (for instance using RPC)
  - the user clicked in a given URL and the server invoked the corresponding script
  - the script executed, produced the results and passed them back to the server (usually as the address of a web page)
  - the server retrieved the page and send it to the browser





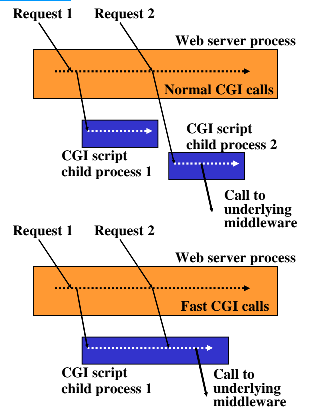

## Applets and clients

- The problem of the using a web browser as universal client is that it does not do much beyond displaying data (it is a thin client):
  - multiple interactions are needed to complete complex operations
  - the same operations must be done over and over again for all clients
  - the processing power at the client is not used
- By adding a JVM (Java Virtual Machine) to the browser, now it becomes possible to dynamically download the client functionality (an applet) every time it is needed
- The client becomes truly independent of the operating system and is always under the control of the server



# Web server as a client of a EAI system

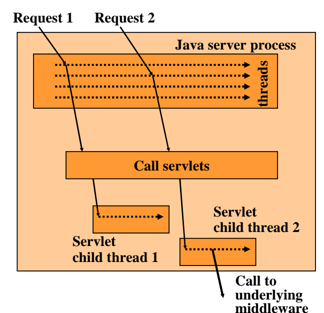
- CGI scripts were initially widely used as there was no other way of connecting the web server with the IT system so that it could do something beyond sending static documents
- However, CGI scripts have several problems that are not easy to solve:
  - CGI scripts are separate processes, requiring additional context switches when a call is made (and thereby adding to the overall delay)
  - Fast-CGI allows calls to be made to a single running process but it still requires two context switches
  - CGI is really a quick hack not designed for performance, security, scalability, etc.





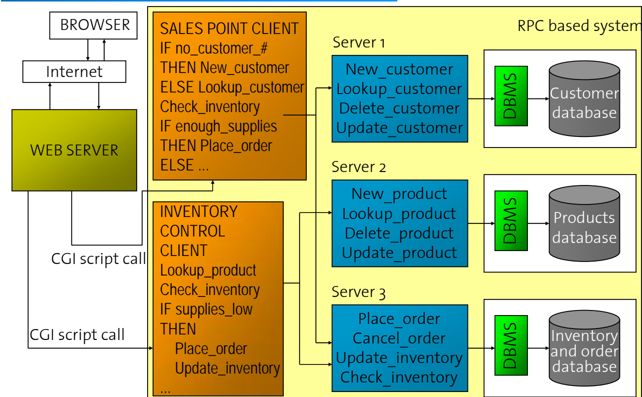

# Servlets

- Servlets fulfill the same role as CGI scripts: they provide a way to invoke a program in response to an http request.
- However:
  - Servlets run as threads of the Java server process (not necessarily the web server) not as separate OS processes
  - unlike CGI scripts, that can be written in any language, Servlets are always written in Java (and are, therefore, portable)
  - can use all the mechanisms provided by the JVM for security purposes

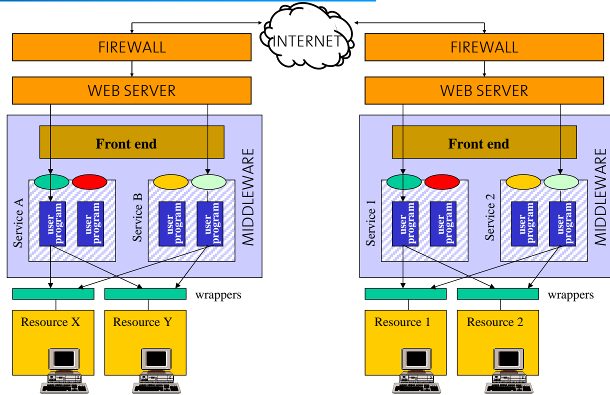





# Just one more layer ...



# Business to Business (B2B)






# Limitations of the WWW

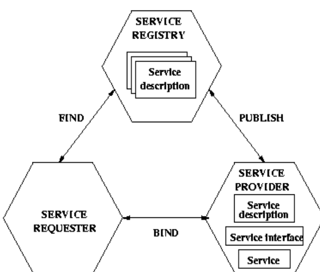
- HTTP was originally designed as a document exchange protocol (request a document, get the document, display the document). It lacked support for client side parameters
- Its architecture was originally designed with human users in mind. The document format (HTML) was designed to cope with GUI problems not with semantics. In EAI, the goal is almost always to remove humans from the business processes (mostly to reduce costs and to speed the process up). Strict formatting rules and tagging are key to exchanging messages across heterogeneous systems
- Interaction through document exchange can be very inefficient when the two sides of the interaction are programs (documents must be created, sent, parsed on arrival, information extracted, etc.). Unfortunately, http does not directly support any other form of interaction
- The initial WWW model was heavily biased towards the server side: the client (the browser) does not do much beyond displaying the document. For complex applications that meant
  - much more traffic between client and server
  - high loads at the server as the number of users increases



# Web Services Architecture

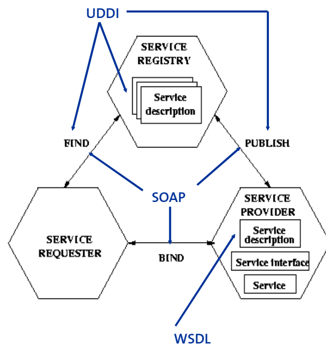
- A popular interpretation of Web services is based on IBM's *Web service architecture* based on three elements:

1. Service requester: The potential user of a service (the client)
2. Service provider: The entity that implements the service and offers to carry it out on behalf of the requester (the server)
3. Service registry: A place where available services are listed and that allows providers to advertise their services and requesters to lookup and query for services



# Main Web Services Standards

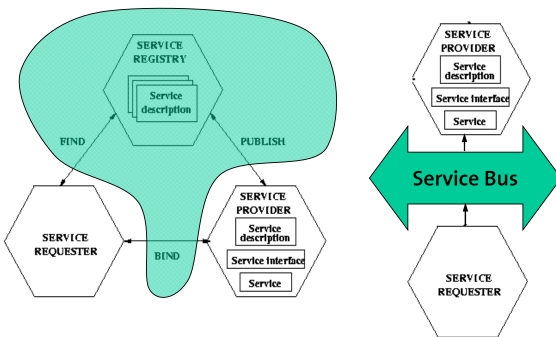
- The Web service architecture proposed by IBM is based on two key concepts:
  - architecture of existing synchronous middleware platforms
  - current specifications of SOAP, UDDI and WSDL
- The architecture has a remarkable client/server flavor
- It reflects only what can be done with
  - SOAP (Simple Object Access Protocol)
  - UDDI (Universal Description and Discovery Protocol)
  - WSDL (Web Services Description Language)






# The Service Bus

- The service bus can be seen as a refactoring of the basic Web service architecture, where a higher degree of loose coupling has been added.




SERVICE PROVIDER
Service description
Service interface
Service
Service Bus
SERVICE REQUESTER



# Benefits of Web services

- One important difference with conventional middleware is related to the standardization efforts at the W3C that should guarantee:
  - Platform independence (Hardware, Operating System)
  - Reuse of existing networking infrastructure (HTTP has become ubiquitous)
  - Programming language neutrality (.NET talks with Java, and vice versa)
  - Portability across Middleware tools of different Vendors
  - Web services are “loosely coupled” components that foster software reuse
  - WS technologies should be composable so that they can be adopted incrementally

## WS Standards and Specifications

| | | | |
|---|---|---|---|
| Transport | HTTP, IIOP, SMTP, JMS | | |
| Messaging | XML, **SOAP** | | WS-Addressing |
| Description | XML Schema, **WSDL** | | WS-Policy, SSDL |
| Discovery | **UDDI** | | WS-MetadataExchange |
| Choreography | WSCL | WSCI | **WS-Coordination** |
| Business Processes | **WS-BPEL** | BPML | WSCDL |
| Stateful Resources | **WS-Resource Framework** | | |
| Transactions | WS-CAF | **WS-Transactions** WS-Business Activities | |
| Reliable Messaging | **WS-Reliability** | | **WS-ReliableMessaging** |
| Security | **WS-Security SAML, XACML** | | WS-Trust, WS-Privacy **WS-SecureConversation** |
| Event Notification | WS-Notification | | WS-Eventing |
| Management | **WSDM** | | **WS-Management** |
| Data Access | OGSA-DAI | | SDO |



## Web services SOAP



## What is SOAP?

- The W3C started working on SOAP in 1999. SOAP 1.0 was entirely based on HTTP. The current specification is SOAP 1.1 (May 2000) is more generic by including other transport protocols. Version 1.2 is at the working draft stage.
- SOAP 1.1 covers the following four main areas:
  - A message format for one-way communication describing how a message can be packed into an XML document
  - A description of how a SOAP message (or the XML document that makes up a SOAP message) should be transported using HTTP (for Web based interaction) or SMTP(for e-mail based interaction)
  - A set of rules that must be followed when processing a SOAP message and a simple classification of the entities involved in processing a SOAP message. It also specifies what parts of the messages should be read by whom and how to react in case the content is not understood
  - A set of conventions on how to turn an RPC call into a SOAP message and back as well as how to implement the RPC style of interaction (how the client makes an RPC call, this is translated into a SOAP message, forwarded, turned into an RPC call at the server, the reply of the server converted into a SOAP message, sent to the client, and passed on to the client as the return of the RPC call)

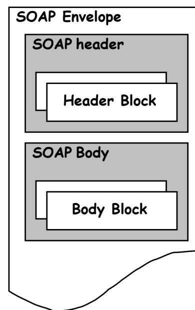
## The background for SOAP

- SOAP was originally conceived as the minimal possible infrastructure necessary to perform RPC through the Internet:
  - use of XML as intermediate representation between systems
  - very simple message structure
  - mapping to HTTP for tunneling through firewalls and using the Web infrastructure
- The idea was to avoid the problems associated with CORBA's IIOP/GIOP (which fulfilled a similar role but using a non-standard intermediate representation and had to be tunneled through HTTP any way)
- The goal was to have an extension that could be easily plugged on top of existing middleware platforms to allow them to interact through the Internet rather than through a LAN as it is typically the case. Hence the emphasis on RPC from the very beginning (essentially all forms of middleware use RPC at one level or another)
- Eventually SOAP started to be presented as a generic vehicle for computer driven message exchanges through the Internet and then it was open to support interactions other than RPC and protocols other then HTTP. This process, however, is only in its very early stages.



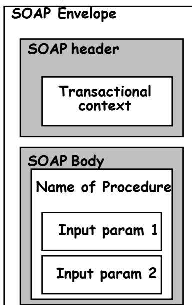
## SOAP messages

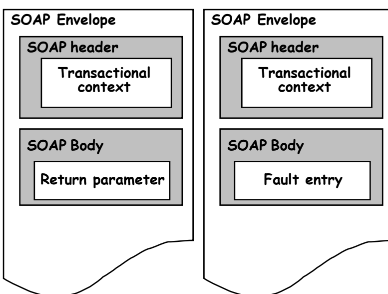
- SOAP is based on message exchanges
- Messages are seen as envelops where the application encloses the data to be sent
- A message has two main parts:
  - header: which can be divided into blocks
  - body: which can be divided into blocks
- SOAP does not say what to do with the header and the body, it only states that the header is optional and the body is mandatory
- Use of header and body, however, is implicit. The body is for application level data. The header is for infrastructure level data





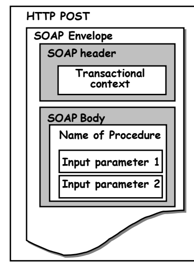

## From TRPC to SOAP messages



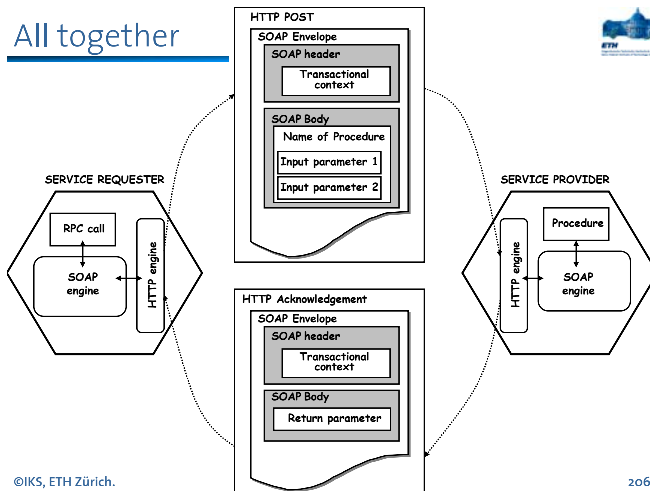
## SOAP and HTTP

- A binding of SOAP to a transport protocol is a description of how a SOAP message is to be sent using that transport protocol
- The typical binding for SOAP is HTTP
- SOAP can use GET or POST. With GET, the request is not a SOAP message but the response is a SOAP message, with POST both request and response are SOAP messages (in version 1.2,






## All together



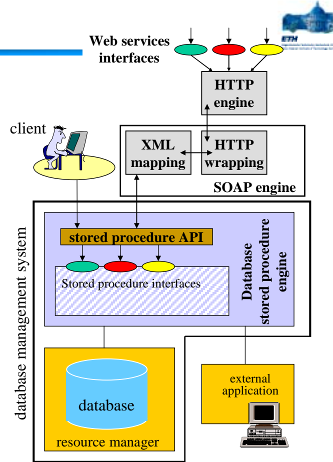



## SOAP and the client server model

- The close relation between SOAP, RPC and HTTP has two main reasons:
  - SOAP has been initially designed for client server type of interaction which is typically implemented as RPC or variations thereof
  - RPC, SOAP and HTTP follow very similar models of interaction that can be very easily mapped into each other (and this is what SOAP has done)
- The advantages of SOAP arise from its ability to provide a universal vehicle for conveying information across heterogeneous middleware platforms and applications. In this regard, SOAP will play a crucial role in enterprise application integration efforts in the future as it provides the standard that has been missing all these years

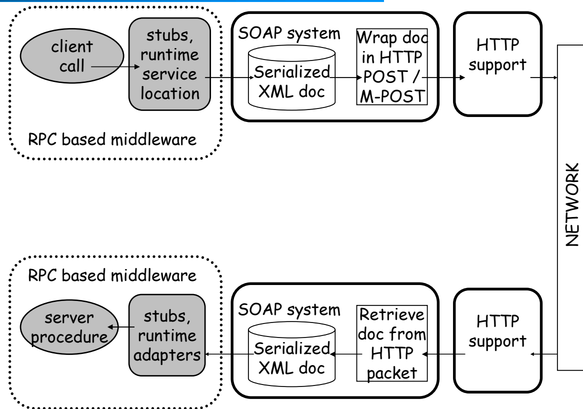
# A first use of SOAP

- Some of the first systems to incorporate SOAP as an access method have been databases. The process is extremely simple:
  - a stored procedure is essentially an RPC interface
  - Web service = stored procedure
  - IDL for stored procedure = translated into WSDL
  - call to Web service = use SOAP engine to map to call to stored procedure
- This use demonstrates how well SOAP fits with conventional middleware architectures and interfaces. It is just a natural extension to them




©IKS, ETH Zürich.


# Automatic conversion RPC - SOAP




©IKS, ETH Zürich.


# Web services WSDL

©IKS, ETH Zürich.

## What is WSDL?

- The Web Services Description Language specification is in version 1.1 (March 2001) and currently under revision (v1.2 is in the working draft stage)
- WSDL 1.1 discusses how to describe the different parts that comprise a Web service:
  - the type system used to describe the interfaces (based on XML)
  - the messages involved in invoking the service
  - the individual operations that make up the service
  - the sets of operations that constitute a service
  - the mapping to a transport protocol for the messages
  - the location where the service resides
  - groups of locations that can be used to access the same service

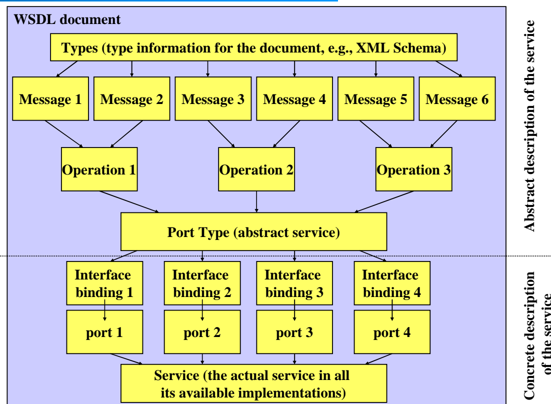


## WSDL vs IDL

- WSDL can be best understood when we approach it as an XML version of an IDL that also covers the aspects related to integration through the Internet and the added complexity of Web services
- An IDL in conventional middleware and enterprise application integration platforms has several purposes:
  - description of the interfaces of the services provided (e.g., RPC)
  - serve as an intermediate representation for bridging heterogeneity by providing a mapping of the native data types to the intermediate representation associated to the IDL in question
  - serve as the basis for development through an IDL compiler that produces stubs and libraries that can be use to develop the application



## Elements of WSDL



# Web services
UDDI



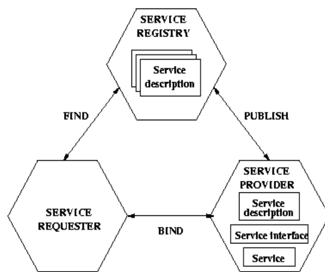
## What is UDDI?

- The UDDI specification is probably the one that has evolved the most from all specifications we have seen so far. The latest version is version 3 (July 2002):
  - version 1 defined the basis for a business service registry
  - version 2 adapted the working of the registry to SOAP and WSDL
  - version 3 redefines the role and purpose of UDDI registries, emphasizes the role of private implementations, and deals with the problem of interaction across private and public UDDI registries
- Originally, UDDI was conceived as an "Universal Business Registry" similar to search engines (e.g., Google) which will be used as the main mechanism to find electronic services provided by companies



## Role of UDDI

- Services offered through the Internet to other companies require much more information that a typical middleware service
- In many middleware and EAI efforts, the same people develop the service and the application using the service
- This is obviously no longer the case and, therefore, using a service requires much

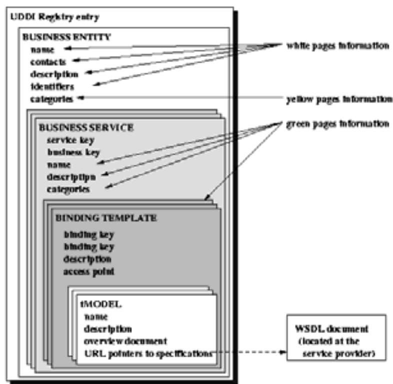


## UDDI data

- An entry in an UDDI registry is an XML document composed of different elements (labeled as such in XML), the most important ones being:
  - *businessEntity* : is a description of the organization that provides the service.
  - *businessService*: a list of all the Web services offered by the business entity.
  - bindingTemplate: describes the technical aspects of the service being offered.
  - *tModel*: ("technical model")is a generic element that can be used to store addotional information about the service, typically additional technical information on how to use the service, conditions for use, guarantees, etc.
- Together, these elements are used to provide:
  - white pages information: data about the service provider (name, address, contact person, etc.)
  - yellow pages information: what type of services are offered and a list of the different services offered
  - green pages information: technical information on how to use each one of the services offered, including pointers to WSDL descriptions of the services (which do not reside in the UDDI registry)

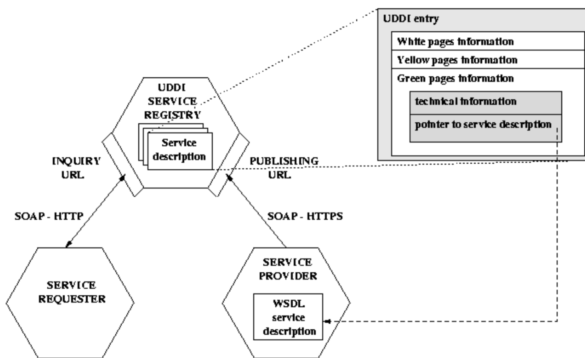


## Summary of the data in UDDI






## UDDI and WSDL



## Summary UDDI

- The UDDI specification is rather complete and encompasses many aspects of an UDDI registry from its use to its distribution across several nodes and the consistency of the data in a distributed registry
- Most UDDI registries are private and typically serve as the source of documentation for integration efforts based on Web services
- UDDI registries are not necessarily intended as the final repository of the information pertaining Web services. Even in the "universal" version of the repository, the idea is to standardize basic functions and then built proprietary tools that exploit the basic repository. That way it is possible to both tailor the design and maintain the necessary compatibility across repositories
- While being the most visible part of the efforts around Web services, UDDI is perhaps the least critical due to the complexities of B2B interactions (establishing trust, contracts, legal constrains and procedures, etc.) . The ultimate goal is, of course, full automation, but until that happens a long list of problems need to be resolved and much more standardization is necessary.

## Web services
## Service Oriented Architectures



## What is SOA

- SOA = Services Oriented Architecture
  - Services = another name for large scale components wrapped behind a standard interface (Web services although not only)
  - Architecture = SOA is intended as a way to build applications and follows on previous ideas such as software bus, IT backbone, or enterprise bus
- The part that it is not in the name
  - Loosely-coupled = the services are independent of each other, heterogeneous, distributed
  - Message based = interaction is through message exchanges rather than through direct calls (unlike Web services, CORBA, RPC, etc.)

## The novelty behind SOA

- The concept of SOA is not new:
  - Message oriented middleware
  - Message brokers
  - Event based architectures
- The current context is different
  - Emergence of standard interfaces (Web services)
  - Emphasis on simplifying development (automatic)
  - Use of complex underlying infrastructure (containers, middleware stacks, etc.)
- Interest in SOA arises from a number of reasons:
  - Basic technology in place
  - More clear understanding of distributed applications
  - The key problem is integration not programming



## The need for SOA

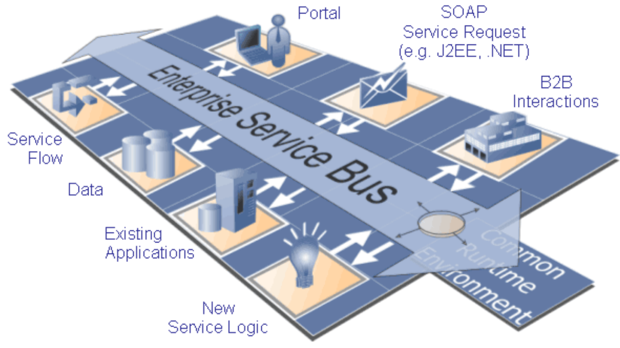
- Most companies today have a large, heterogeneous IT infrastructure that:
  - Keeps changing
  - Needs to evolve to adopt new technology
  - Needs to be connected of that of commercial partners
  - Needs to support an increasing amount of purposes and goals
- This was the field of Enterprise Application Integration using systems like CORBA or DCOM. However, solutions until now suffered from:
  - Tightly integrated systems
  - Vendor lock-in (e.g., vendor stacks)
  - Technology lock-in (e.g., CORBA)
  - Lack of flexibility and limitations when new technology arises (e.g., Internet)
- SOA is an attempt to build on standards (web services) to reduce the cost of integration
- It introduces very interesting possibilities:
  - Development by composition
  - Large scale reuse
  - Frees developers from “lock-in” effects of various kinds



## SOA vs. Web services

- Web services are about
  - Interoperability
  - Standardization
  - Integration across heterogeneous, distributed systems
- Service Oriented Architectures are about:
  - Large scale software design
  - Software Engineering
  - Architecture of distributed systems
- SOA is possible but more difficult without Web services
- SOA introduces some radical changes to software:
  - Language independence (what matters is the interface)
  - Event based interaction (no longer synchronous models)
  - Message based exchanges (no RPC)
  - Composition and orchestration

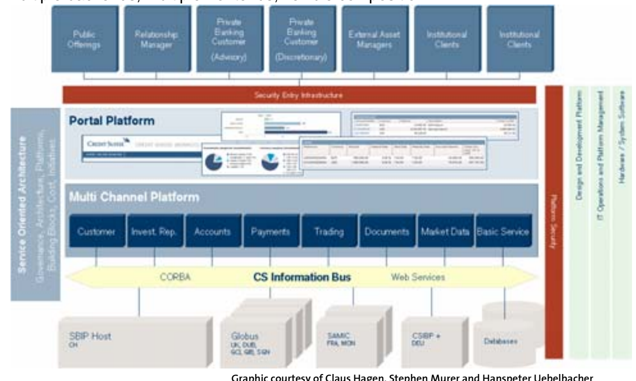
## An integration backbone






## Enterprise architecture at Credit Suisse

Multiple backends, multiple frontends, flexible composition




Graphic courtesy of Claus Hagen, Stephen Murer and Hanspeter Uebelbacher of Credit Suisse