

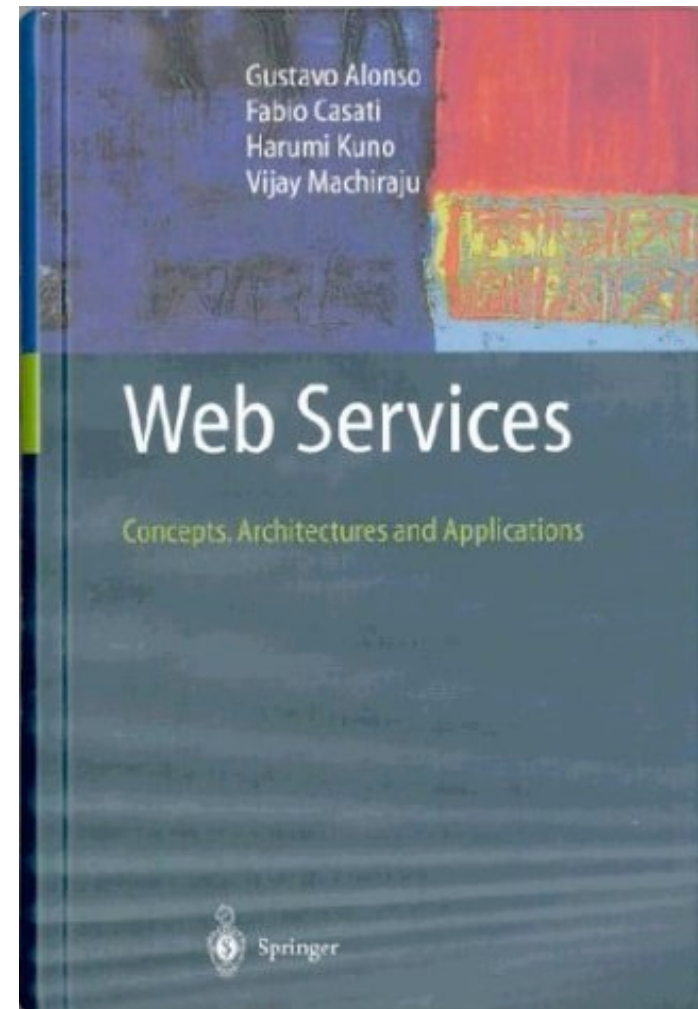
Standard Software for Enterprise Resource Planning

Lecturer: Prof. Dr. Ralf Möller
Lab classes: Rainer Marrone, Michael Wessel

Lecture: Thursdays (90 minutes)
Lab classes: Fridays (60 minutes)

Prerequisite:
Lecture on ECommerce

This lecture is based on:





ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Web Services - Concepts, Architecture and Applications

Part 2: Synchronous middleware

Gustavo Alonso and Cesare Pautasso
Computer Science Department
ETH Zürich
alonso@inf.ethz.ch
<http://www.inf.ethz.ch/~alonso>

Remote Procedure Call (RPC): the basics of all middleware

Basics of distribution

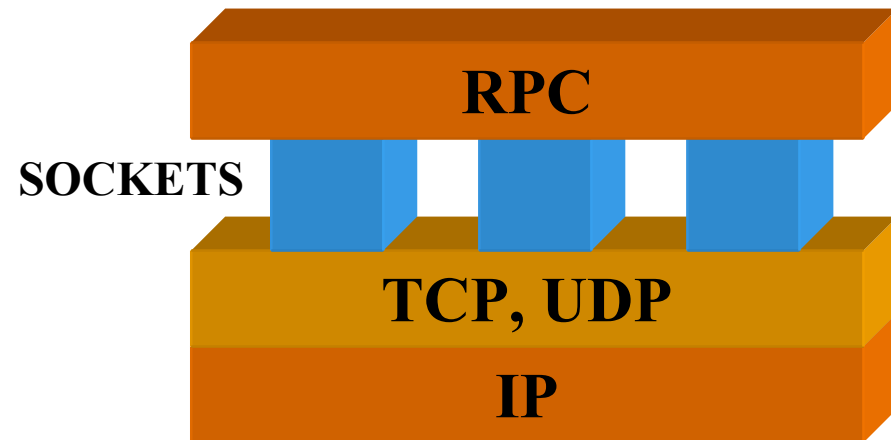


There are four basic problems to solve in a distributed information system. Everything else derives from these basic aspects:

- ❑ Use the appropriate abstraction for distribution in view of the communication infrastructure available. The abstraction must hide the network stack and provide high level primitives
- ❑ How to embed the service abstraction part of the programming language in a more or less transparent manner.
 - ➔ Don't forget this important aspect: whatever you design, others will have to program and use it
- ❑ How to exchange data between machines that might use different representations for different data types. This involves two aspects:
 - ➔ data type formats (e.g., byte orders in different architectures)
 - ➔ data structures (need to be flattened and then reconstructed)
- ❑ How to describe and find services:
 - ➔ so that clients can use them
 - ➔ to avoid tight integration

RPC as an abstraction

- ❑ The most accepted standard for network communication is IP (Internet Protocol) which provides unreliable delivery of single packets to one-hop distant hosts
- ❑ IP was designed to be hidden behind other software layers:
 - ➔ TCP (Transport Control Protocol) implements connected, reliable message exchange
 - ➔ UDP (User Datagram Protocol) implements unreliable datagram based message exchanges
- ❑ TCP/IP and UDP/IP are visible to applications through sockets. The purpose of the socket interface was to provide a UNIX-like abstraction
- ❑ Yet sockets are quite low level for many applications, thus, RPC (Remote Procedure Call) appeared as a way to
 - ➔ hide communication details behind a procedural call
 - ➔ bridge heterogeneous environments
- ❑ RPC is the standard for distributed (client-server) computing



RPC in programming languages

- ❑ The notion of distributed service invocation became a reality at the beginning of the 80's when procedural languages (mainly C) were dominant.
- ❑ In procedural languages, the basic module is the procedure. A procedure implements a particular function or service that can be used anywhere within the program.
- ❑ It seemed natural to maintain this same notion when talking about distribution: the client makes a procedure call to a procedure that is implemented by the server. Since the client and server can be in different machines, the procedure is remote.
- ❑ Client/Server architectures are based on Remote Procedure Calls (RPC)
- ❑ Once we are working with remote procedures in mind, there are several aspects that are immediately determined:
 - ➔ data exchange is done as input and output parameters of the procedure call
 - ➔ pointers cannot be passed as parameters in RPC, opaque references are needed instead so that the client can use this reference to refer to the same data structure or entity at the server across different calls. *The server is responsible for providing this opaque references.*

Describing services IDL

- ❑ All RPC systems have a language that allows to describe services in an abstract manner (independent of the programming language used). This language has the generic name of IDL (interface definition language) (e.g., the IDL of SUN RPC is called XDR)
- ❑ The IDL allows to define each service in terms of their names, and input and output parameters (plus maybe other relevant aspects).
- ❑ An interface compiler is then used to generate the stubs for clients and servers (*rpcgen* in SUN RPC). It might also generate procedure headings that the programmer can then use to fill out the details of the implementation.
- ❑ Given an IDL specification, the interface compiler performs a variety of tasks:
- ❑ It generates the client stub procedure for each procedure signature in the interface. The stub will be then compiled and linked with the client code
- ❑ It generates a server stub. It can also create a server *main*, with the stub and the dispatcher compiled and linked into it. This code can then be extended by the designer by writing the implementation of the procedures
- ❑ It might generate a *.h file for importing the interface and all the necessary constants and types

Machine independent representations

- ❑ Marshalling or serializing can be done by hand (although this is not desirable) using (in C) *sprintf* and *sscanf*:

Message= "Alonso" "ETHZ" "2001"

```
char *name="Alonso", place="ETHZ";
int year=2001;
```

```
sprintf(message, "%d %s %d %s %d",
        strlen(name), name, strlen(place), place,
        year);
```

Message after marshalling =
"6 Alonso 4 ETHZ 2001"

- ❑ Remember that the type and number of parameters is known, we only need to agree on the syntax ...

- ❑ SUN XDR follows a similar approach:

- ➔ messages are transformed into a sequence of 4 byte objects, each byte being in ASCII code
- ➔ it defines how to pack different data types into these objects, which end of an object is the most significant, and which byte of an object comes first
- ➔ the idea is to simplify computation at the expense of bandwidth

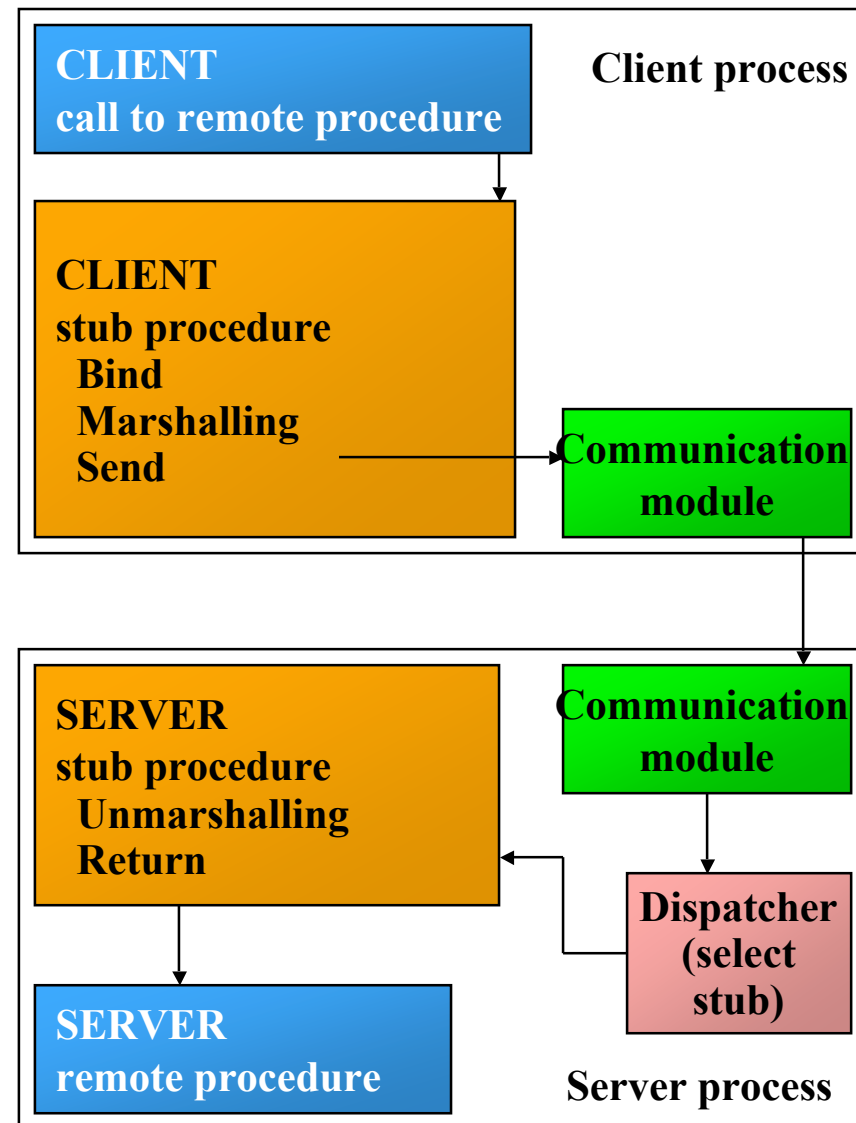
6	String length
A l o n	
s o	
4	String length
E T H Z	
2 0 0 1	cardinal

Discovering services

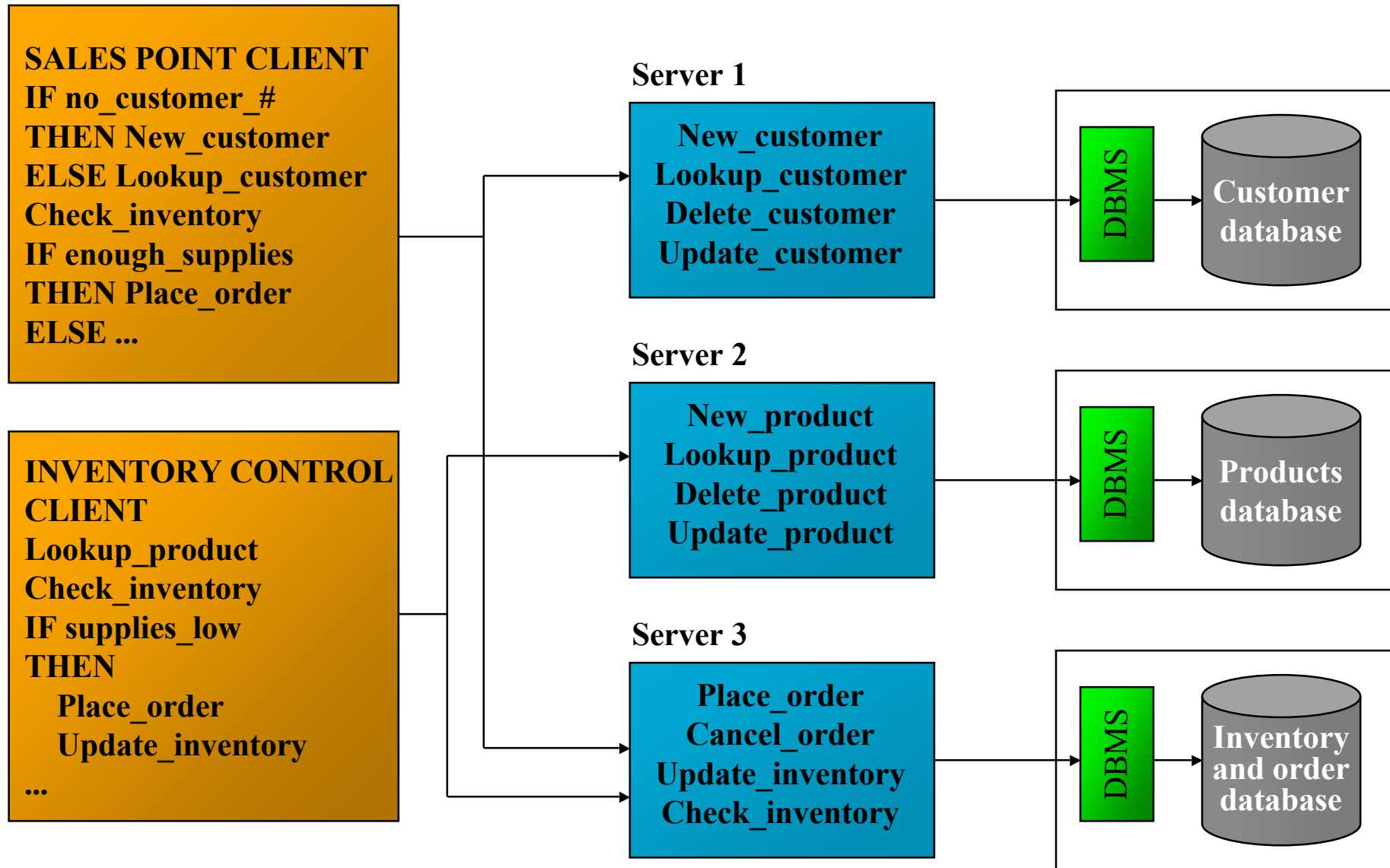
- ❑ A service is provided by a server located at a particular IP address and listening to a given port
- ❑ Binding is the process of mapping a service name to an address and port that can be used for communication purposes
- ❑ Binding can be done:
 - ➔ locally: the client must know the name (address) of the host of the server
 - ➔ distributed: there is a separate service (service location, name and directory services, etc.) in charge of mapping names and addresses. This service must be reachable to all participants
- ❑ With a distributed binder, several general operations are possible:
 - ➔ REGISTER (Exporting an interface): A server can register service names and the corresponding port
 - ➔ WITHDRAW: A server can withdraw a service
 - ➔ LookUP (Importing an interface): A client can ask the binder for the address and port of a given service
- ❑ There must also be a way to locate the binder (predefined location, environment variables, broadcasting to all nodes looking for the binder)

Making it work in practice

- ❑ One cannot expect the programmer to implement all these mechanisms every time a distributed application is developed. Instead, they are provided by a so called RPC system
- ❑ What does an RPC system do?
 - ➔ Provides an interface definition language (IDL) to describe the services
 - ➔ Generates all the additional codes 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



RPC Application



Transactions

- Acknowledgments to
Torben Weis
Andreas Ulbrich

Transaction properties

- ❑ Usually there isn't only one transaction at a time
 - ➔ Transactions need some way of ensuring concurrency control
- ❑ Typically transactions satisfy the “ACID-Properties”
 - ➔ **Atomicity**: “All or nothing”
 - ➔ **Consistency**: transition from one consistent state to another consistent state
 - ➔ **Isolation**: concurrent transactions do not interfere with each other
 - ➔ **Durability**: the final state is stored persistently and not lost in case of failures

Atomicity

- ❑ Transactions happen indivisibly to the outside world
 - ➔ Completely or not at all

Consistency

- ❑ Transition from one consistent state to another
 - ➔ In case the system has invariants that must always hold
 - ➔ If they held before the transaction they will hold afterwards
 - ➔ Depends on the application!
- ❑ Example: Banking system
 - ➔ Conservation of money
 - ➔ Total amount of money is the same after each internal transfer
 - ➔ Invariant may be violated for a short moment within a transaction

Isolation

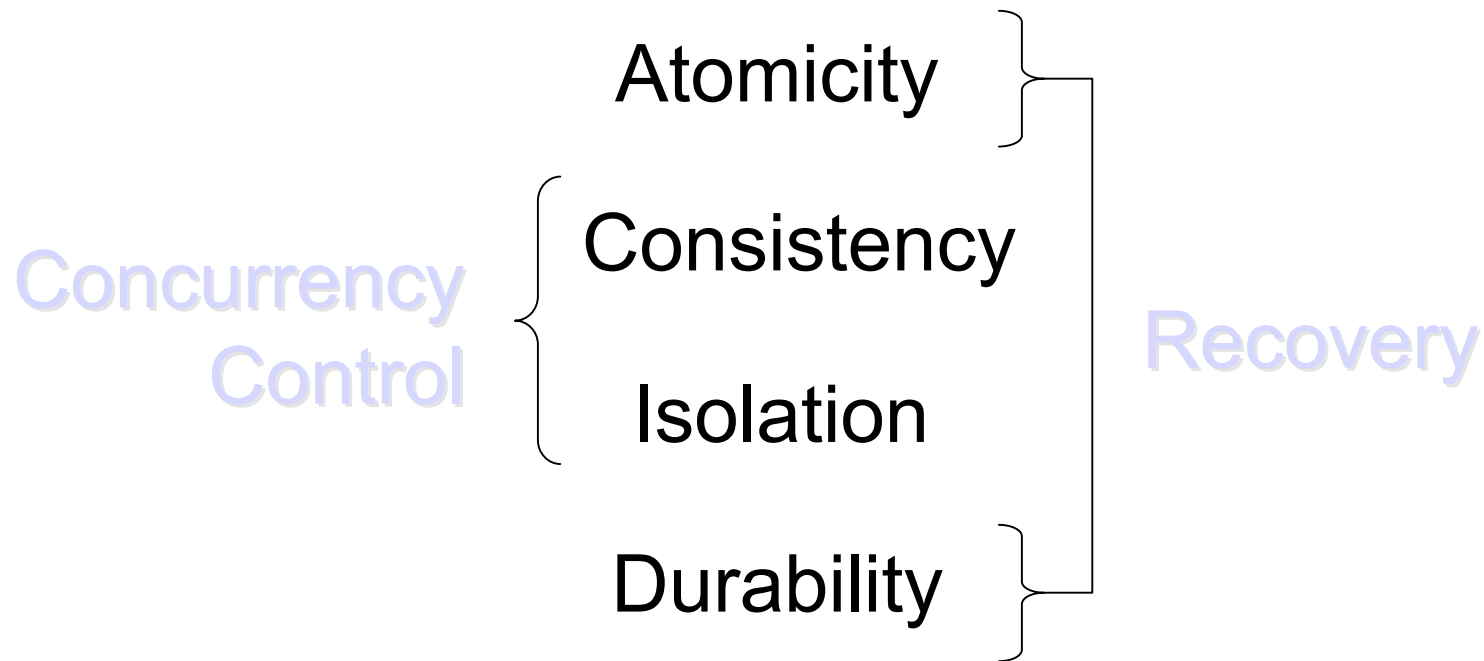
- ❑ Concurrent transactions do not interfere with each other
 - ➔ More precisely: they may interfere but the final result must look like the transaction ran in sequential order

Durability

- ❑ Final state is stored persistently
 - ➔ After a transaction commits the changes become permanent
 - ➔ No failure after commit can undo the changes

Concurrency control and recovery

- ❑ The ACID-properties are endangered by faulty environments and by interfering concurrent transactions



Concurrency: Banking example

```
TX1: transfer (in account_1, in account_2, in amount) {  
  // transfer money from bank account account_1 to bank // account account_2  
  balance_1 := READ(account_1);  
  WRITE(account_1, balance_1 - amount);  
  balance_2 := READ(account_2);  
  WRITE(account_2, balance_2 + amount);  
}
```

```
TX2: sum_up (in account_1, in account_2, out sum) {  
  // sum up the balances of two accounts  
  // does not write to the database  
  temp1 := READ(account_1);  
  temp2 := READ(account_2);  
  sum := temp1 + temp2;  
}
```

Example: serial schedules

- Initial state:
 $ac1=500\$, ac2=700\$\$
 - $TX1: transfer(ac1,ac2,200\$\)$
 - $TX2: X := sum_up(ac1,ac2)$
- Result of both possible serial schedules:
 $X=1200\$, ac1=300\$, ac2=900\$\.$

S1: $TX1 \rightarrow TX2$

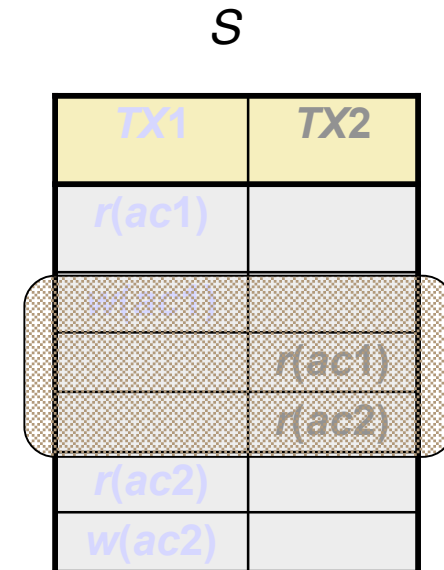
TX1	TX2
$r(ac1)$	
$w(ac1)$	
$r(ac2)$	
$w(ac2)$	
	$r(ac1)$
	$r(ac2)$

S2: $TX2 \rightarrow TX1$

TX1	TX2
	$r(ac1)$
	$r(ac2)$
$r(ac1)$	
$w(ac1)$	
$r(ac2)$	
$w(ac2)$	

Example: inconsistent view

- ❑ Initial state:
 $ac1=500\$$ $ac2=700\$$
 - ➔ $TX1: transfer(ac1, ac2, 200\$)$
 - ➔ $TX2: X := sum_up(ac1, ac2)$
- ❑ Schedule S:
 - ➔ TX2 has an inconsistent view on the data ($X=1000\$$) because it has read uncommitted data.
- ❑ Final state is OK ($ac1=300\$$, $ac2=900\$$)



Example: inconsistent data

- ❑ Initial state:
 $ac1=500\$, ac2=700\$, ac3=600\$\$
 - ➔ $TX1: transfer(ac1, ac2, 200\ \$)$
 - ➔ $TX2: transfer(ac3, ac1, 400\ \$)$
- ❑ Final state is inconsistent because $TX1$ overwrites uncommitted data
($ac1=300\$, ac2=900\$, ac3=200\ \$$)

S

TX1	TX2
$r(ac1)$	
	$r(ac3)$
	$w(ac3)$
	$r(ac1)$
	$w(ac1)$
$r(ac1)$	
$r(ac2)$	
$r(ac3)$	

Conflicting operations

- ❑ Conflicting operations:
 - ➔ Access the same data item
 - ➔ Belong to different transaction
 - ➔ At least one of them is a write operation
- ❑ A schedule is a partial order of the operations of the transaction
 - ➔ Respecting the order of the operations of a single transaction
 - ➔ Ordering conflicting operations of different transactions

Schedule properties

- ❑ Sequential
 - ➔ Example: $r1(x) w1(x) c1 r2(x) w2(x) c2$
- ❑ Serializable
 - ➔ Example: $r1(x) w1(x) r2(x) c1 w2(x) c2$
- ❑ Recoverable
 - ➔ Example: $w1(x) r2(x) w2(y) c2$
 - ➔ what if T1 aborts?
- ❑ Avoiding cascading aborts
 - ➔ Solution: read only committed data

Synchronization

- ❑ How to resolve conflicts between conflicting operations? → Synchronization
- ❑ Algorithms:
 - ↳ Locking
 - ↳ Timestamp ordering (pessimistic/optimistic)
 - ↳ ...

Two phase locking (2PL)

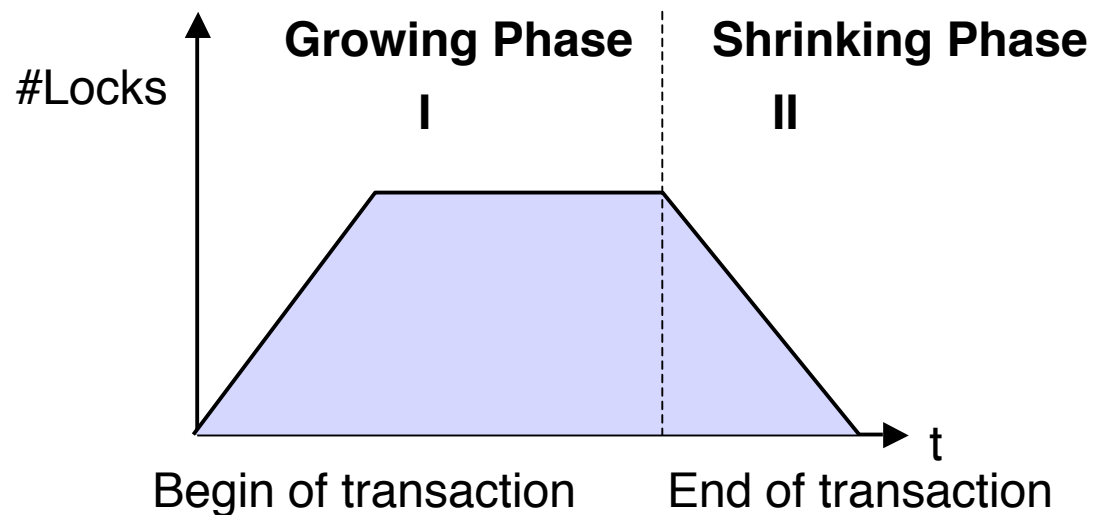
- ❑ Similar to critical sections
- ❑ Decreases concurrency
 - ➔ Transactions may have to wait until required locks are granted
- ❑ Usually two types of locks are used
 - ➔ Read lock: transactions can read data item after lock was granted
 - ➔ Write lock: transactions can read and write data item after lock was granted
- ❑ Locks are not granted by the programmer but automatically by the scheduler

Lock compatibility

- ❑ Shared lock
 - ➔ After a read lock was granted, only further read locks can be granted
- ❑ Exclusive lock
 - ➔ After a write lock was granted, no further locks can be granted
- ❑ Upgrading
 - ➔ A read lock can be upgraded to a write lock provided that no further read locks have been granted
- ❑ Downgrading
 - ➔ A write lock can be downgraded to a read lock if the transaction has not yet written the data item

Flow

- ❑ At the beginning the required locks are requested
- ❑ At the end of the transaction, all remaining locks are released



Properties of 2PL

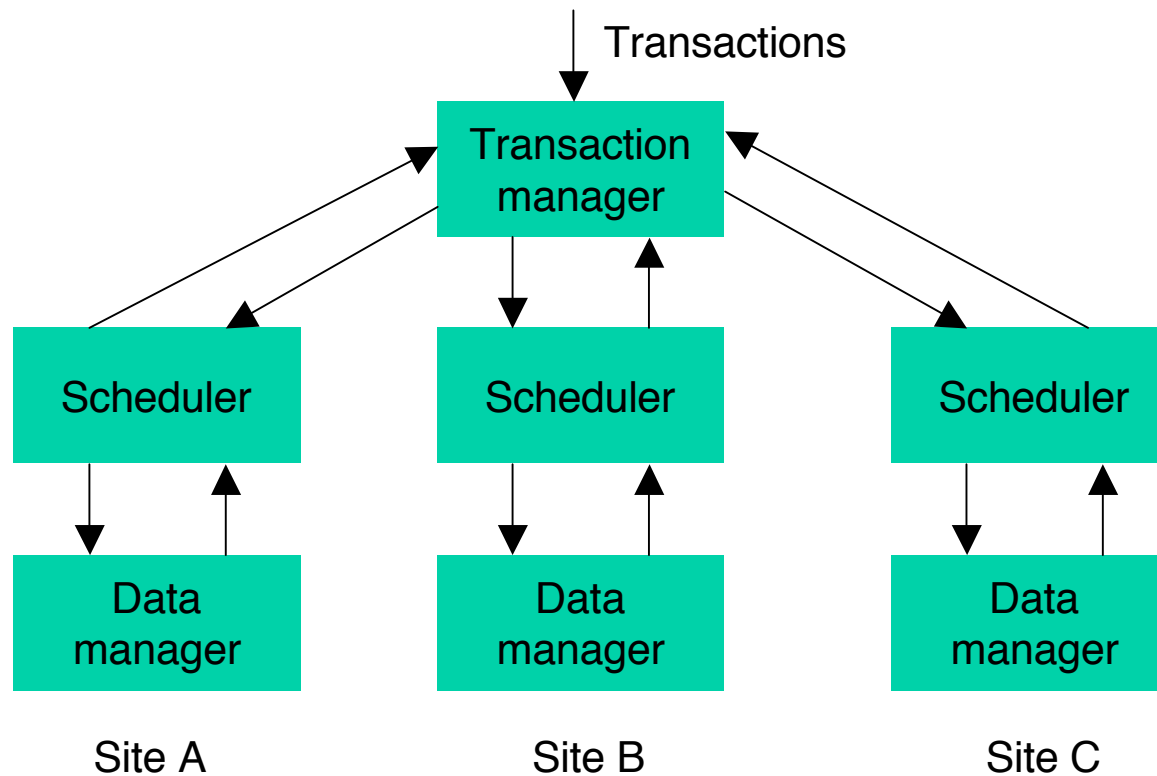
- ❑ Ensures serializability
- ❑ Missing:
 - ➔ Recoverability
 - ➔ Deadlock freeness
 - ➔ Avoidance of cascading aborts
- ❑ Solutions:
 - ➔ Canonical order of locks to avoid deadlocks
 - ➔ Deadlock detection via a dependency graph
 - ➔ ...

Distributed transactions

- ❑ Often local transactions are not sufficient
 - ➔ E.g., booking a journey requires atomic booking
 - Flight at the airline
 - Room at the hotel
 - Rental car at car rental service
- ❑ Distributed transactions allow transactions to span multiple independent participants on different nodes
 - ➔ Commit and abort of distributed transactions have to be coordinated among the participants

Design

- ❑ Each data item is stored at exactly one site
- ❑ Each site has a scheduler managing its local data items



Distributed two phase locking

- ❑ Decisions on granting a lock can be taken locally
 - ➔ Only depends on the locks currently active on an item
- ❑ Commit or abort operation is sent to all sites where the transaction has accessed data items
 - ➔ atomic commit protocol (ACP) needed

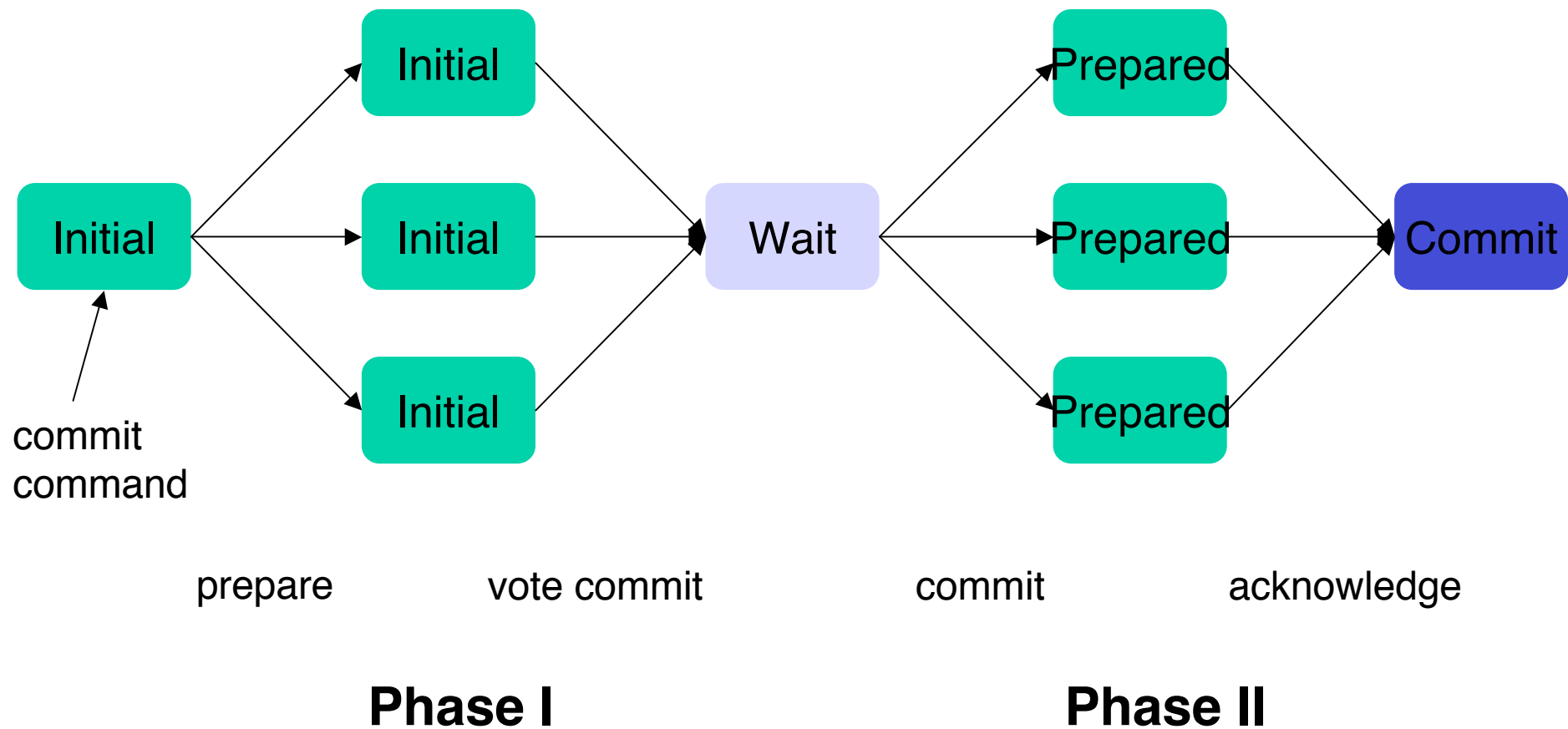
Commit protocols

- ❑ Protocols to make decisions (yes/no) with multiple participants
 - ➔ Commit a transaction or abort a transaction
- ❑ Various algorithms:
 - ➔ Two phase commit (2PC)
 - ➔ Three phase commit (3PC)
 - ➔ ...

Two phase commit I

- ❑ Participants go through two phases
 - ➔ Needed to allow unilateral aborts of participants
- 1. **Prepare** (to commit) - (voting phase)
 - ➔ Each participant votes to commit or to abort the transaction
 - Once a participant has voted to commit, it can no longer abort the transaction unilaterally
- 2. **Commit** - (completion phase)
 - ➔ Participants actually commit
 - After consensus has been reached that **all** participants are prepared
 - Otherwise all participants abort

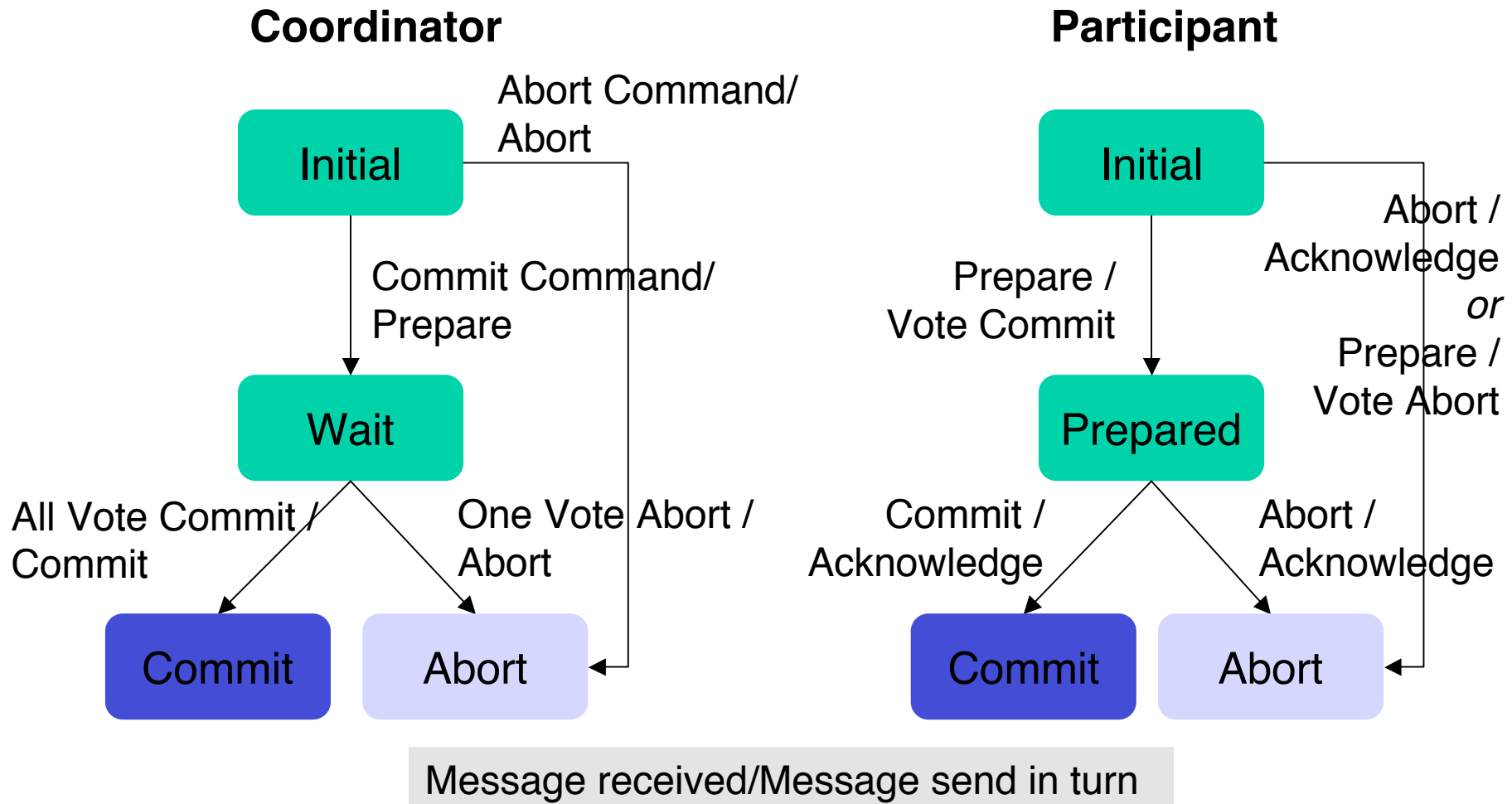
2PC: successful completion



Two phase commit II

- ❑ Any participant can initiate
 - ➔ To commit (commit command)
 - ➔ Or to abort (abort command) the transaction
- ❑ Coordinator is used to achieve consensus
 - ➔ All participants must have registered at the coordinator
 - ➔ Coordinator requests all participants to vote
 - ➔ Decides to commit if **all** participants have voted to commit
 - ➔ Decides to abort if **any** participant has voted to abort

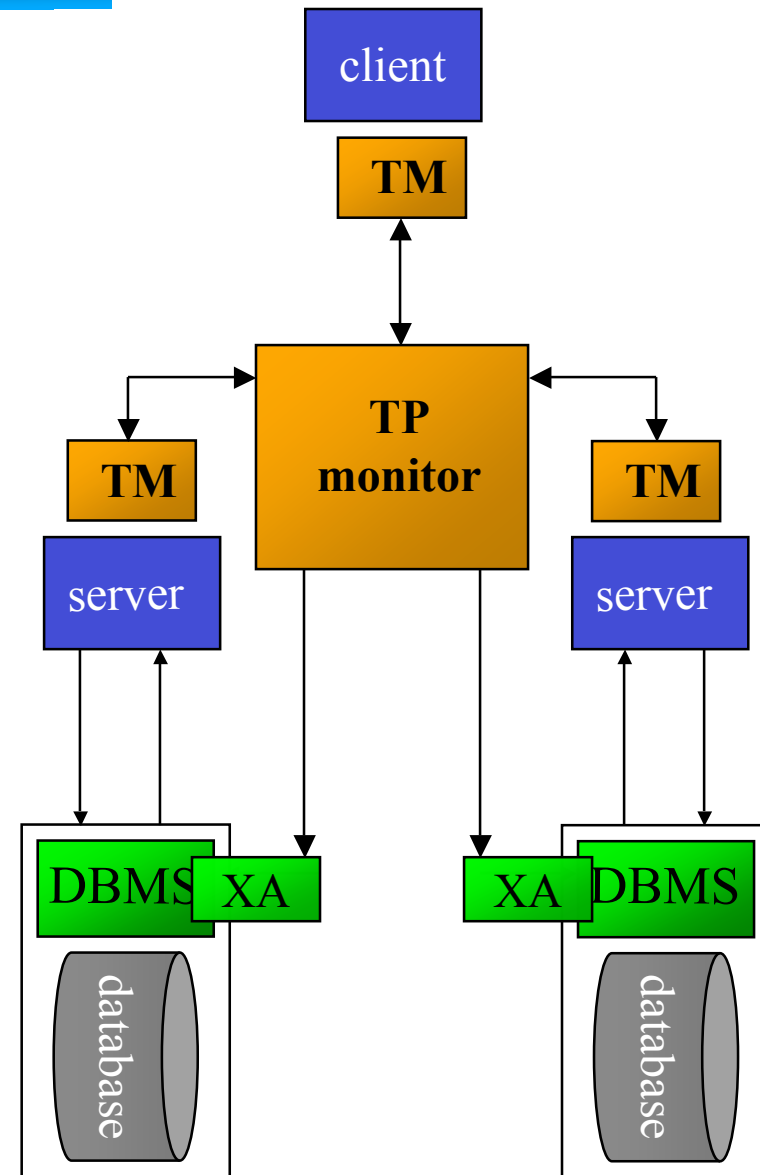
2PC state transitions



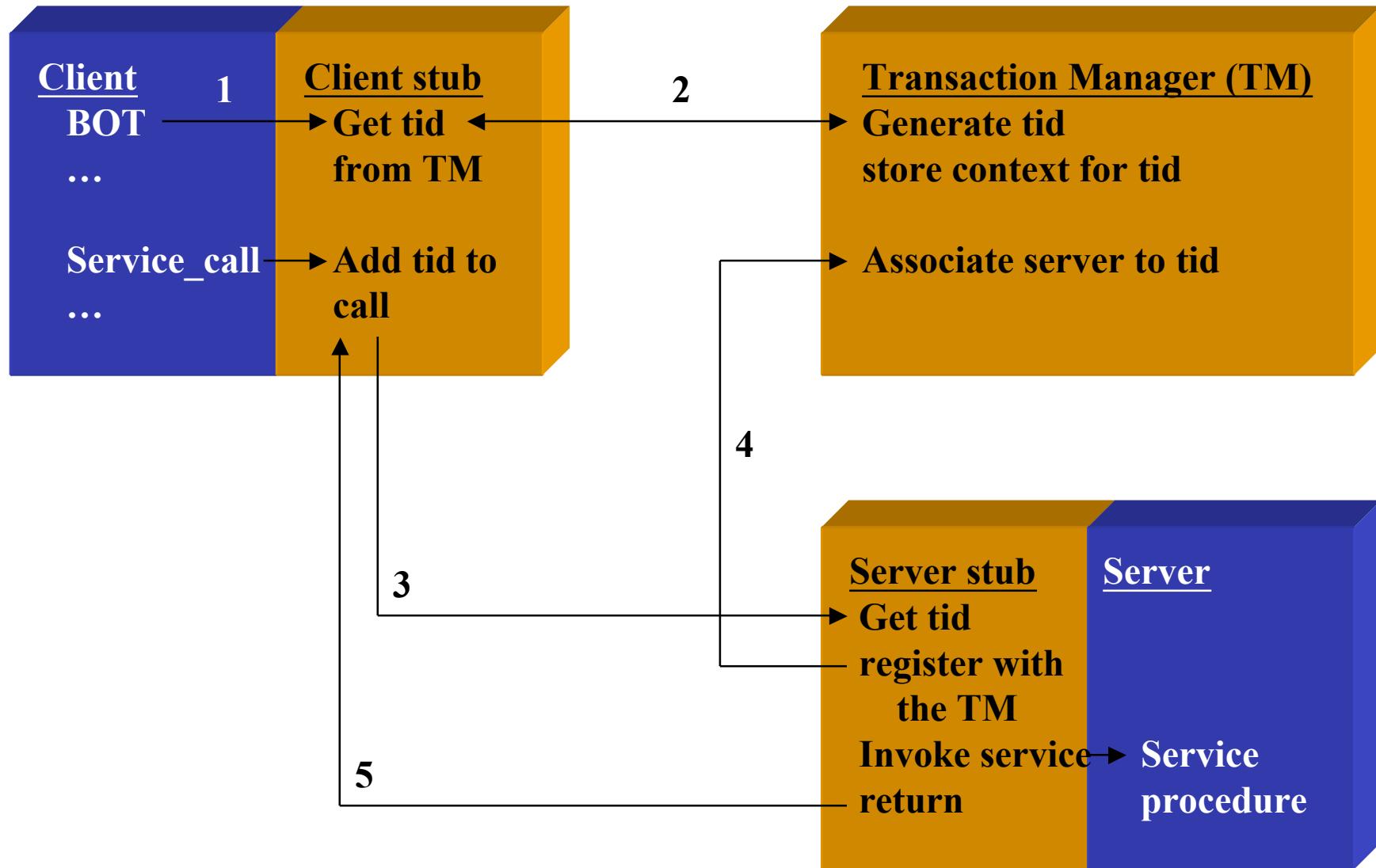
Building upon RPC: TP-monitors, object brokers, object monitors

Transactional RPC

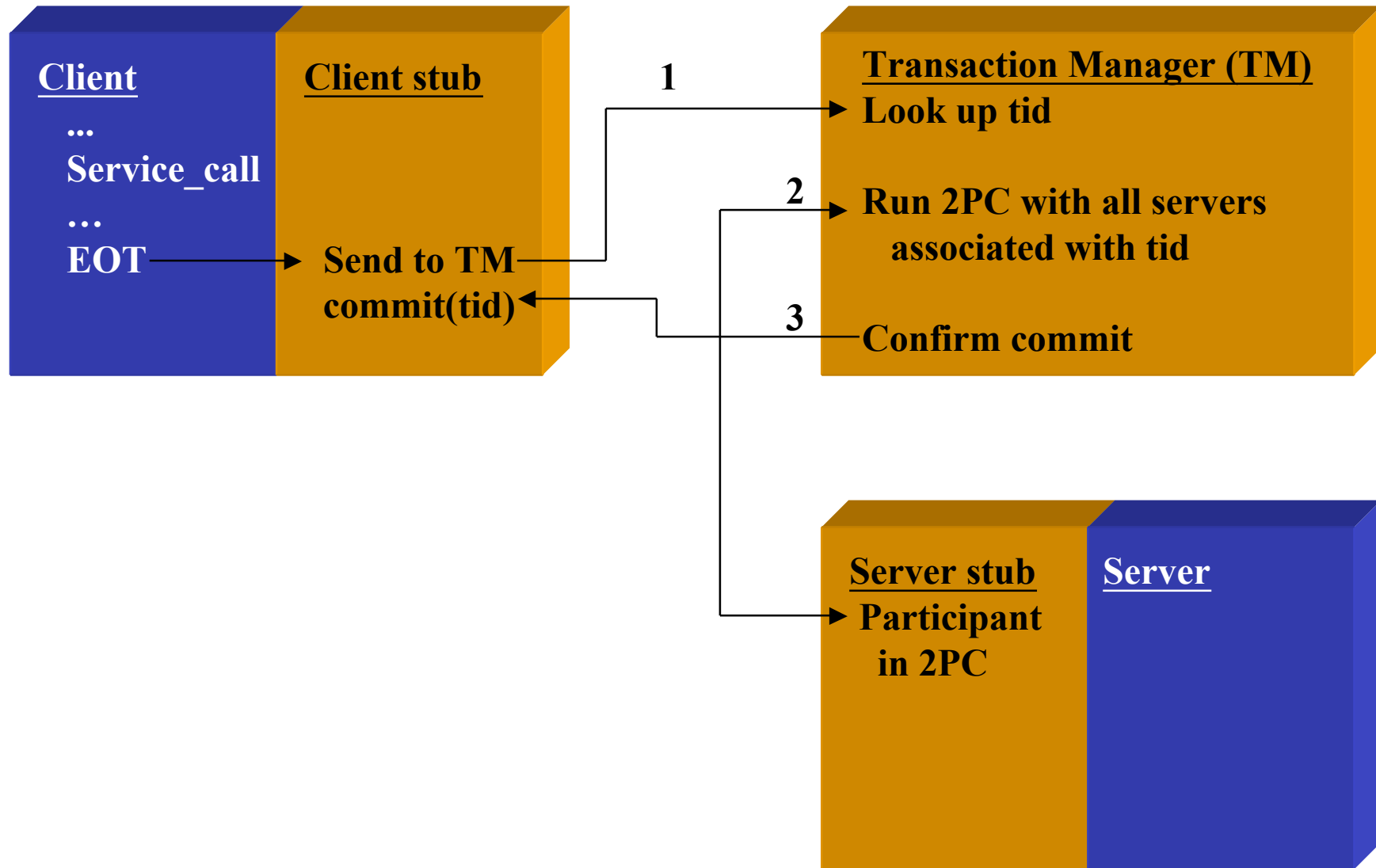
- ❑ The limitations of RPC in terms of reliability can be resolved by making RPC calls transactional. In practice, this means that they are controlled by a 2PC protocol
- ❑ An intermediate entity is needed to run 2PC (the client and server could do this themselves but it is neither practical nor generic enough)
- ❑ This intermediate entity is usually called a transaction manager (TM) and acts as intermediary in all interactions between clients, servers, and resource managers
- ❑ When all the services needed to support RPC, transactional RPC, and additional features are added to the intermediate layer, the result is a TP-Monitor



Basic TRPC (making calls)

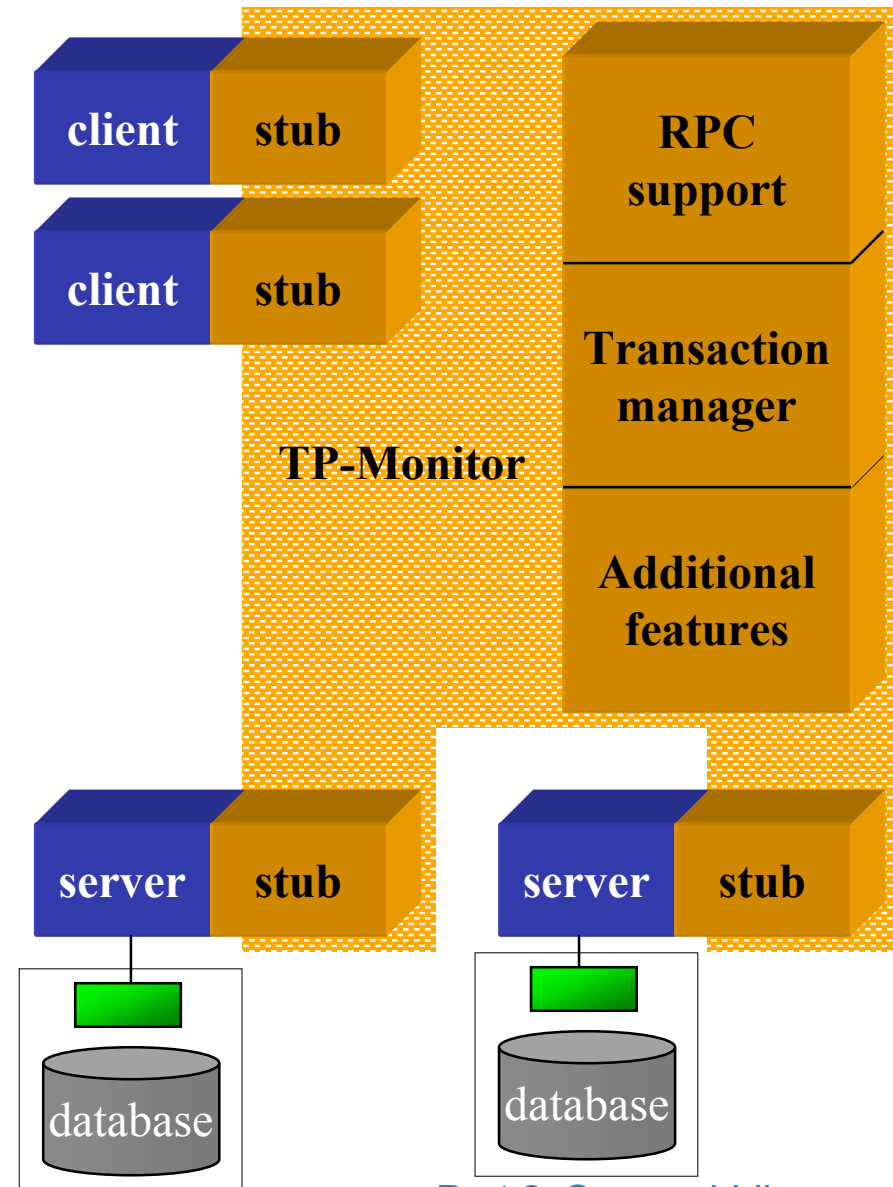


Basic TRPC (committing calls)



One step beyond ...

- ❑ The previous example assumes the server is transactional and can run 2PC. This could be, for instance, a stored procedure interface within a database. However, this is not the usual model
- ❑ Typically, the server invokes a resource manager (e.g., a database) that is the one actually running the transaction
- ❑ This makes the interaction more complicated as it adds more participants but the basic concept is the same:
 - ➔ the server registers the resource manager(s) it uses
 - ➔ the TM runs 2PC with the resources managers, instead of the server (see OTS at the end)



Component based design

- ❑ The notion of component based design is nowadays associated with object orientation. However, RPC, TRP, or TP-Monitors are just alternative (perhaps more primitive) forms of component based design
- ❑ The main difference is that with RPC, TRPC and conventional TP-Monitors, server and clients are typically developed together, as part of a single system and within the same developing (programming) effort
- ❑ Component based design tries to take this idea one step further: the clients or servers do not need to be developed together and, ultimately, can be developed by different teams (vendors) based on standard interfaces

