

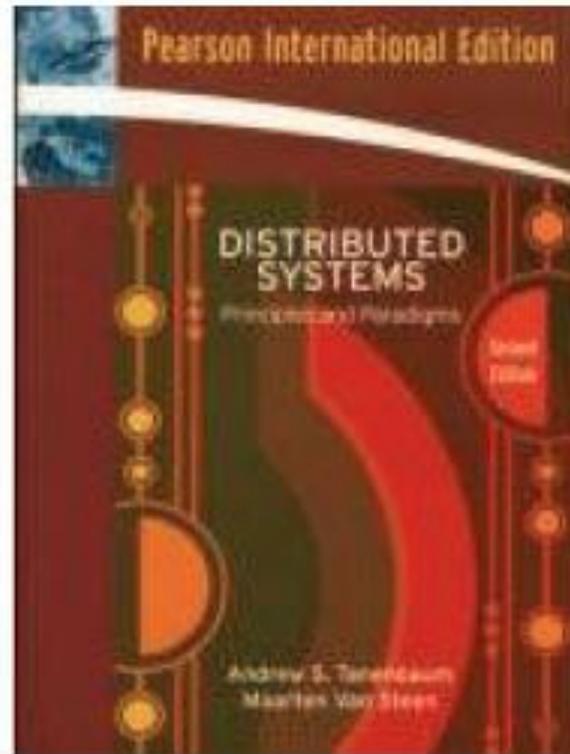
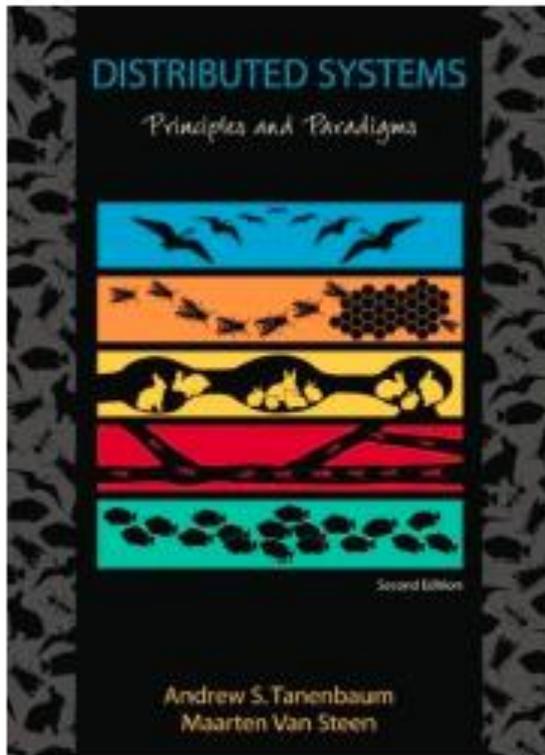
# Distributed Systems

Prof. Dr. Schahram Dustdar  
Distributed Systems Group  
Vienna University of Technology

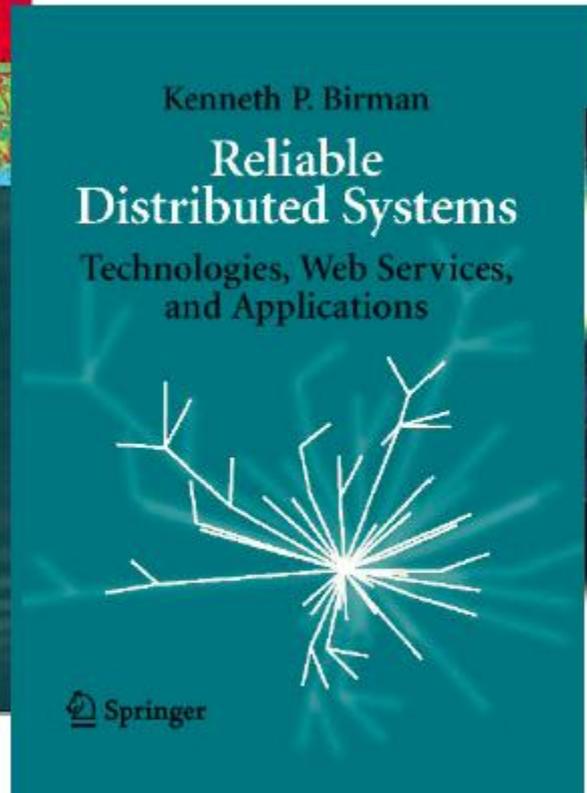
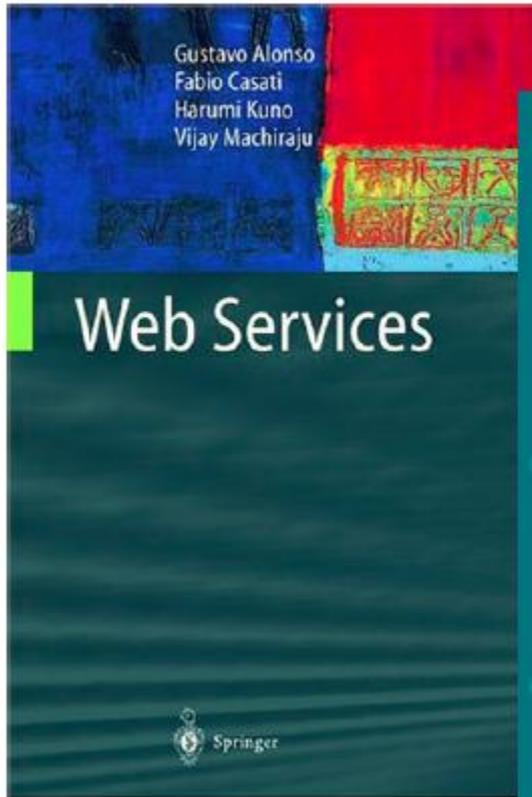
[dustdar@dsg.tuwien.ac.at](mailto:dustdar@dsg.tuwien.ac.at)  
[dsg.tuwien.ac.at](http://dsg.tuwien.ac.at)

1. History
2. What is a distributed system?
3. Key concepts and design goals
4. Architectural styles

- Slides available for download, but not sufficient for self-study! Please read on...



# Recommended additional reading



fourth edition

## DISTRIBUTED SYSTEMS

### CONCEPTS AND DESIGN

George Coulouris  
Jean Dollimore  
Tim Kindberg

ADDISON  
WESLEY



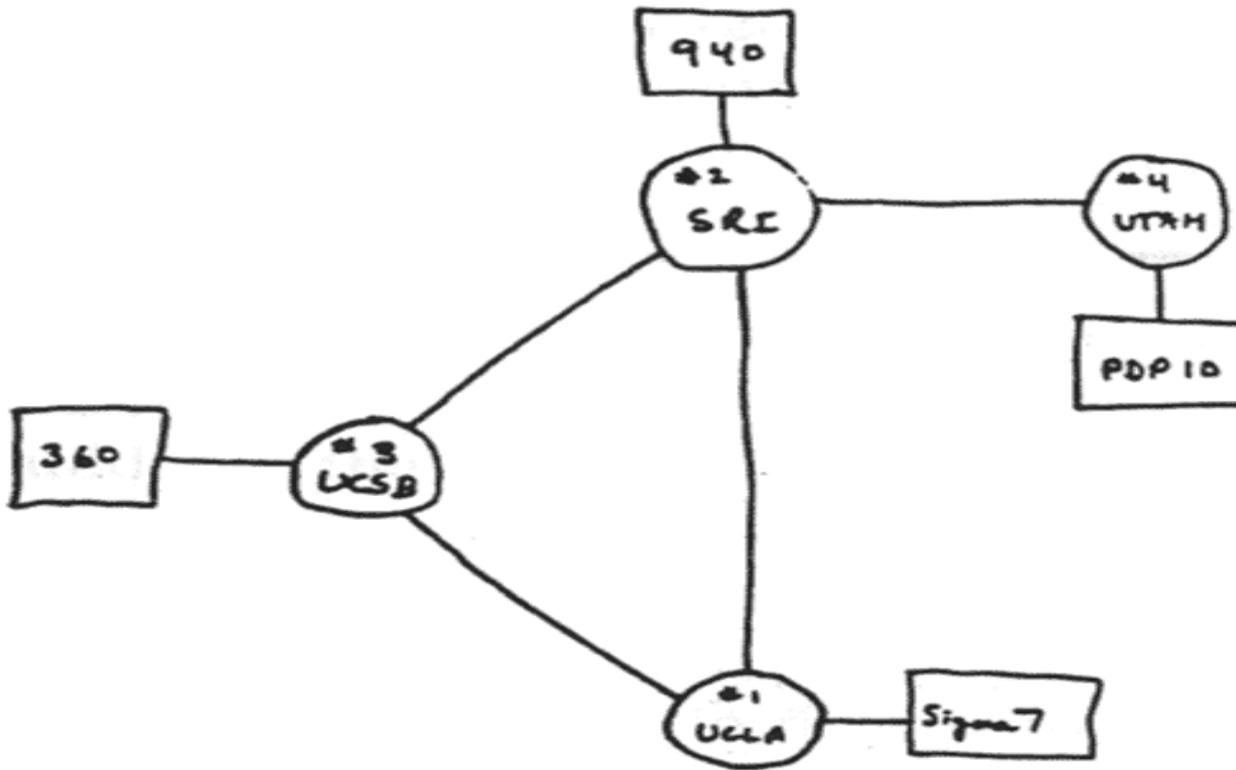
# Prerequisites

- Data structures and algorithms (sequential)
- Operating systems / Systems programming
- Software engineering concepts
- Object-oriented programming
  
- For the **lab**: Java's support for modularity (packages and interfaces), object orientation, exceptions, distribution (RMI), code mobility (applets, class loader), and concurrency (threads and synchronization)

# OVERVIEW AND INTRODUCTION

- Until 1985 large and expensive stand-alone computers
  - Powerful microprocessors (price/performance gain  $10^{12}$  in 50 years)
  - High-speed computer networks (LAN/WAN)
- > composition of computing systems of large numbers of computers connected by a highspeed network increase

# The complete Internet 1969

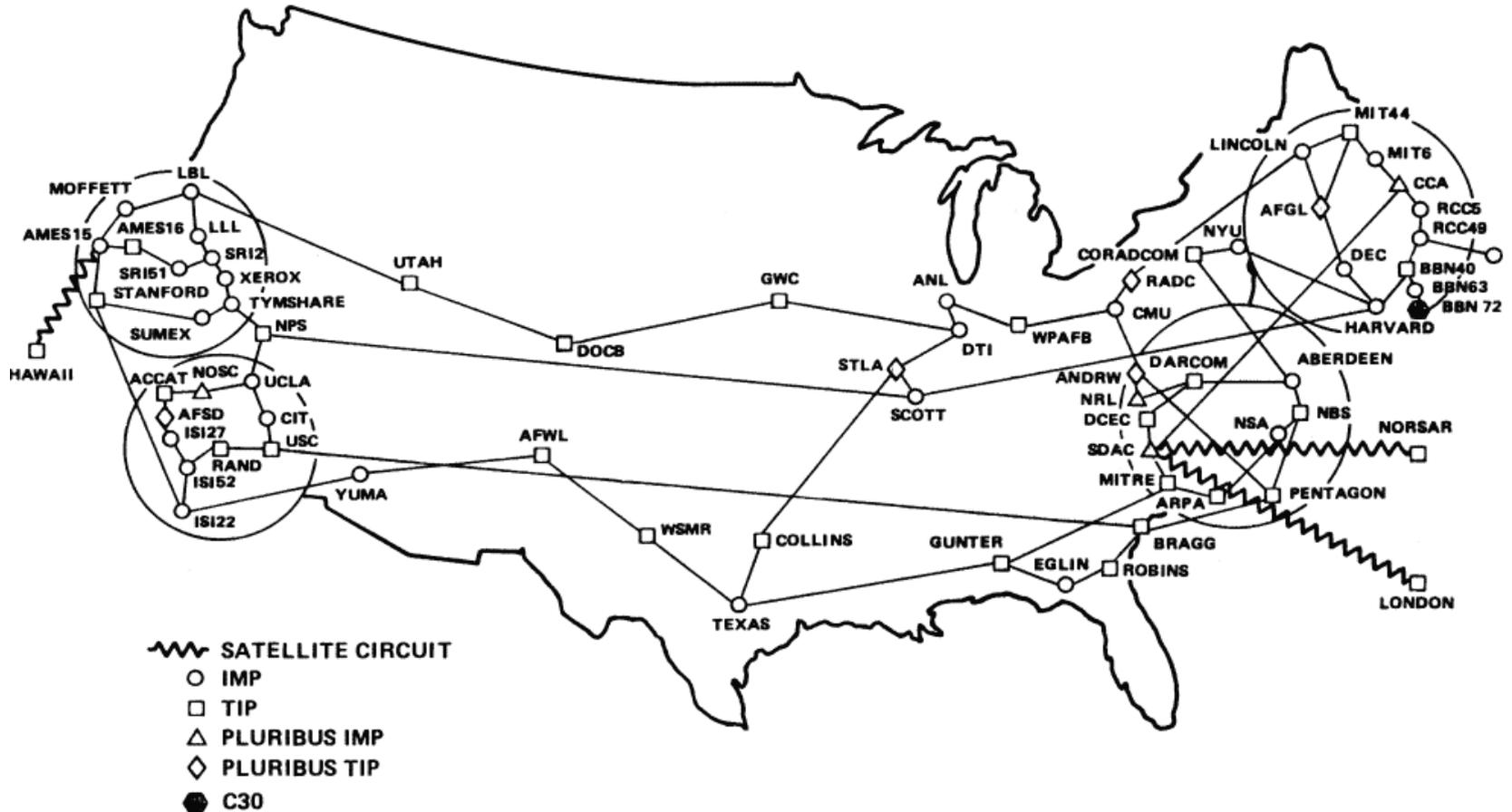


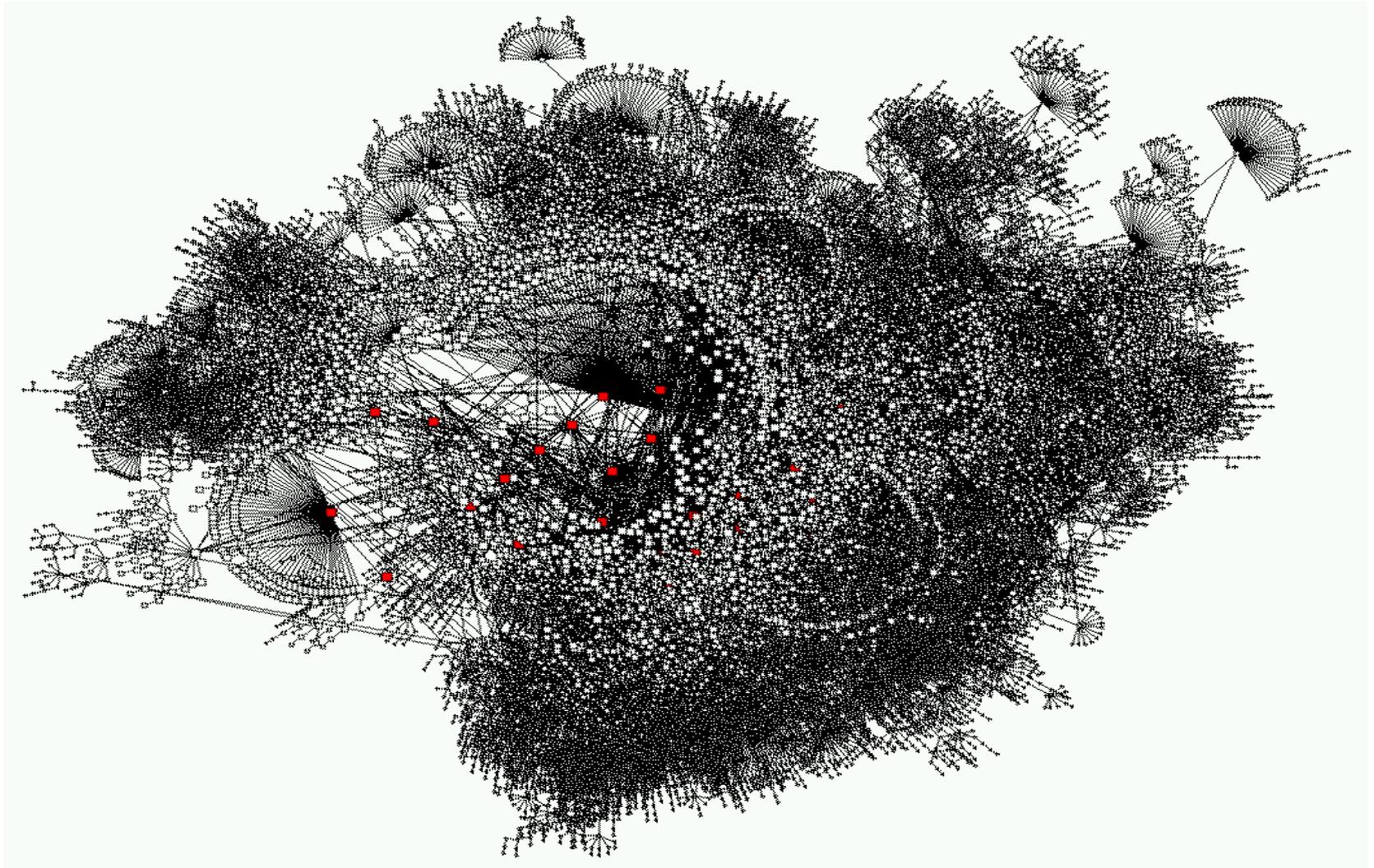
THE ARPA NETWORK

DEC 1969

# Part of the US Internet 1980

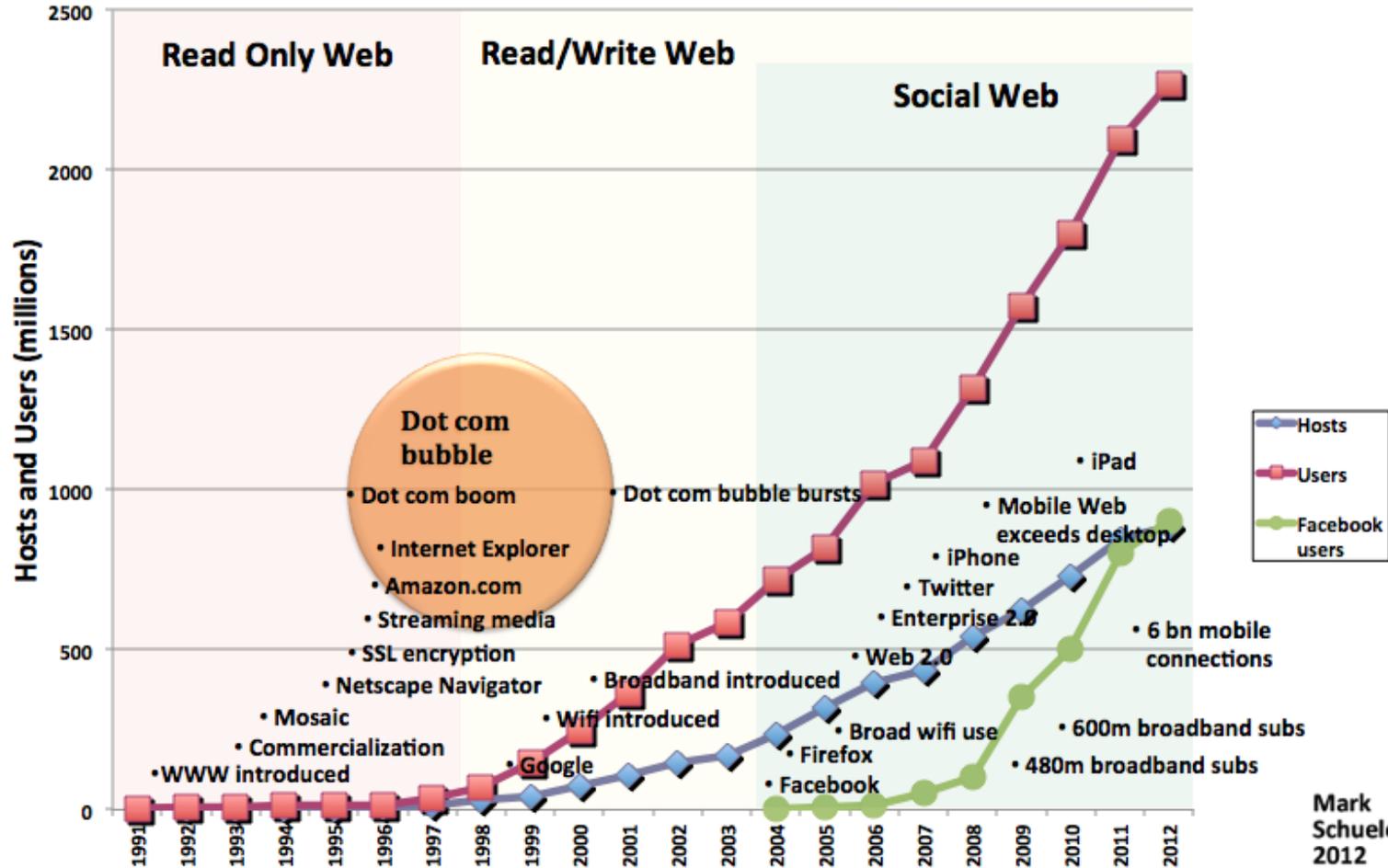
ARPANET GEOGRAPHIC MAP, OCTOBER 1980





# Growth of the Internet

## Internet Growth - Usage Phases - Tech Events

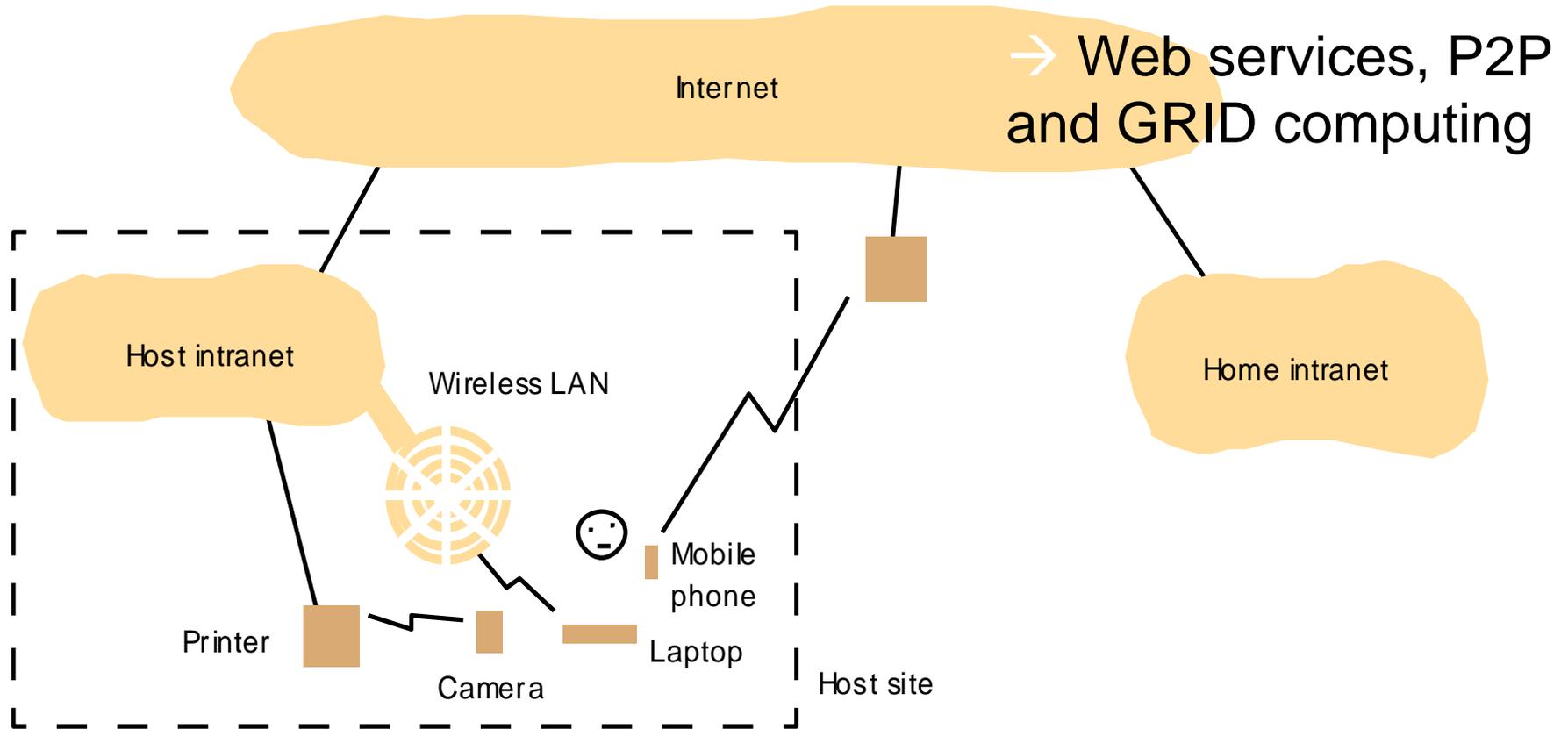


Note - events shown relate to the time axis only.

Mark Schueler 2012



# Portable and handheld devices in a distributed system





# Evolution of Distribution technologies

- Mainframe computers
- Workstations and local networks
- Client-server systems
- Internet-scale systems and the Web
- Sensor/actor networks in automation
- Mobile, ad-hoc, and adaptive systems
- Pervasive (ubiquitous) systems
- Today, less than 2% of processors go into personal computers!



# 24x7 Direct Alarm System

# Emergency Hub



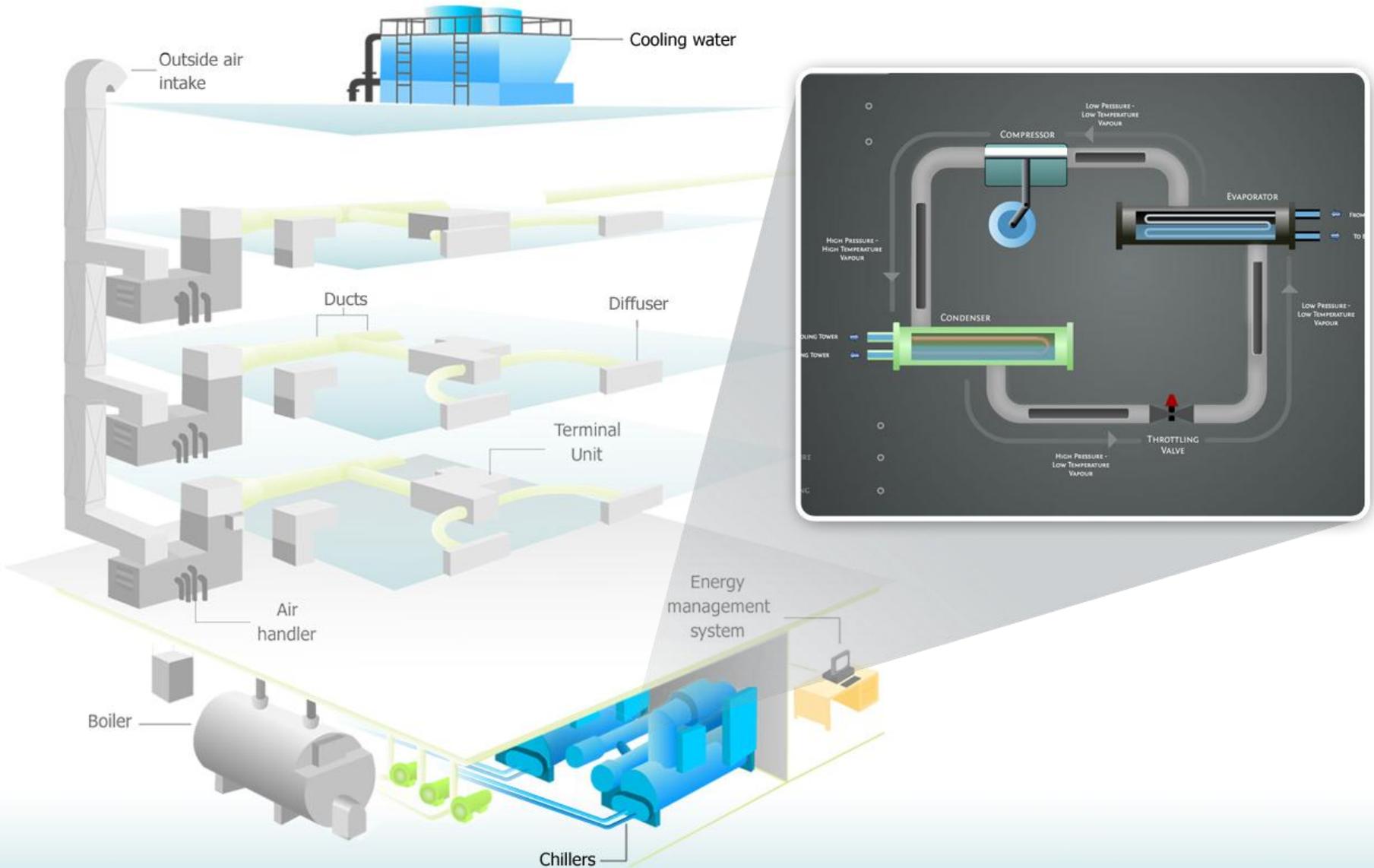
Vehicle tracking system

Logistics Management

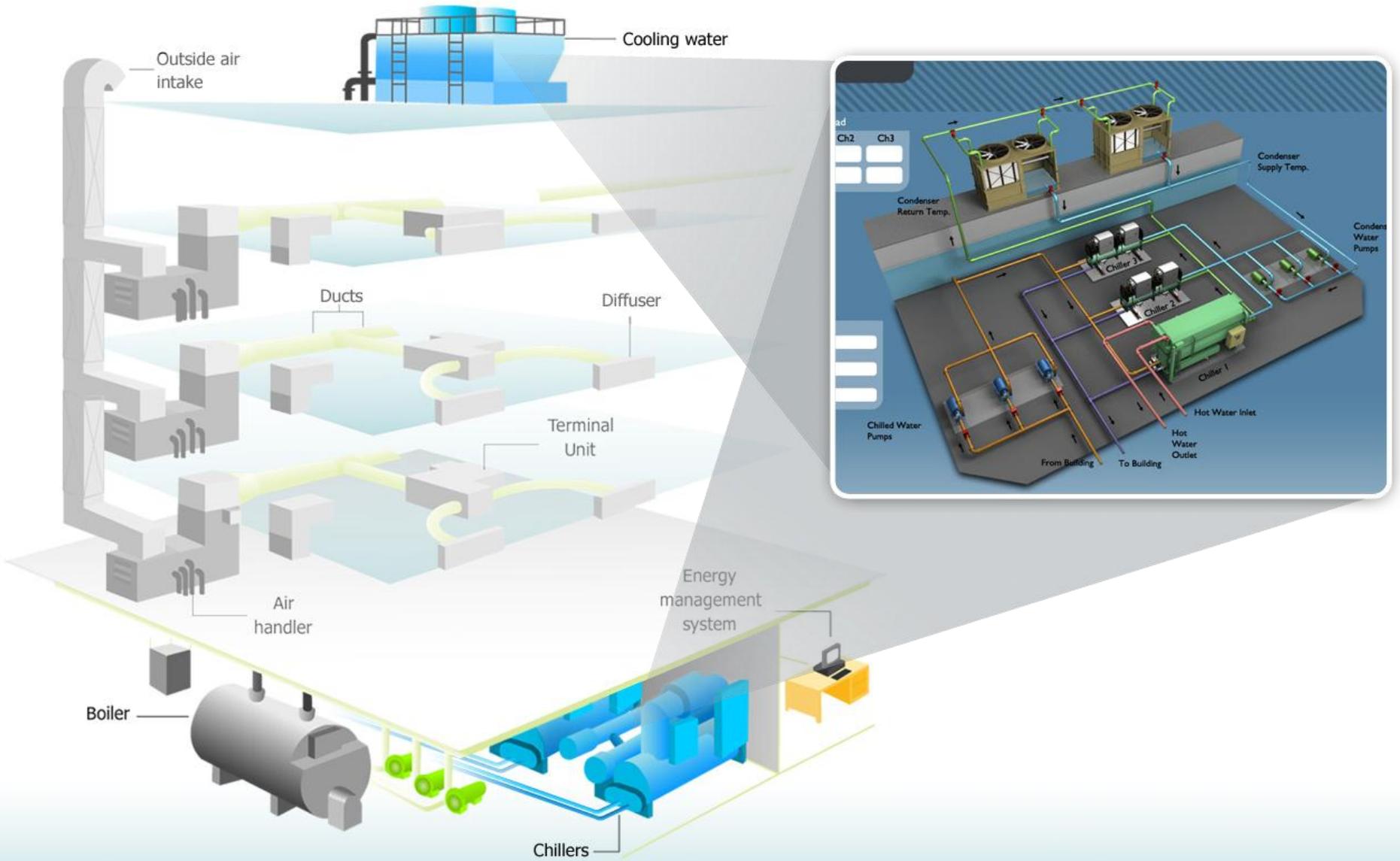




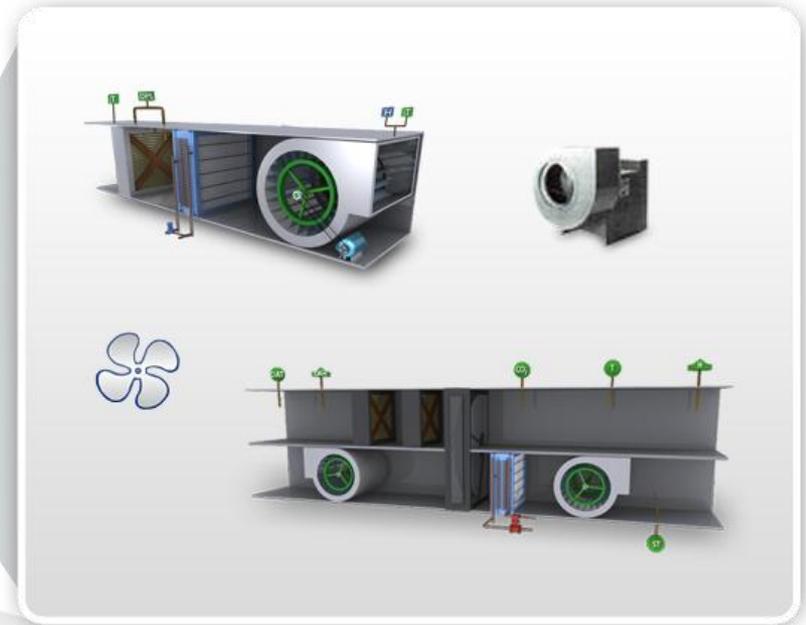
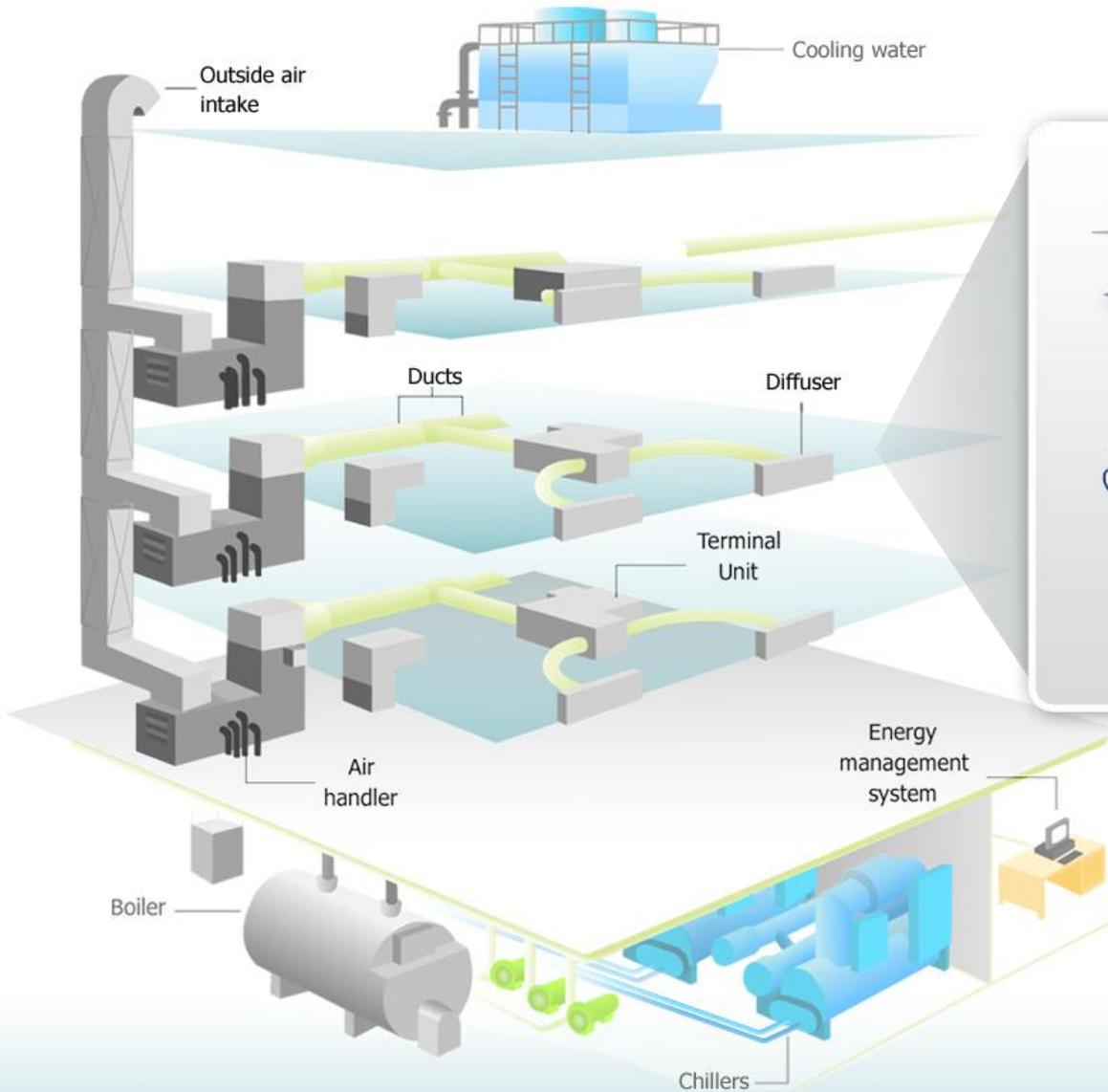
# HVAC (Heating, Ventilation, Air Conditioning) Ecosystem



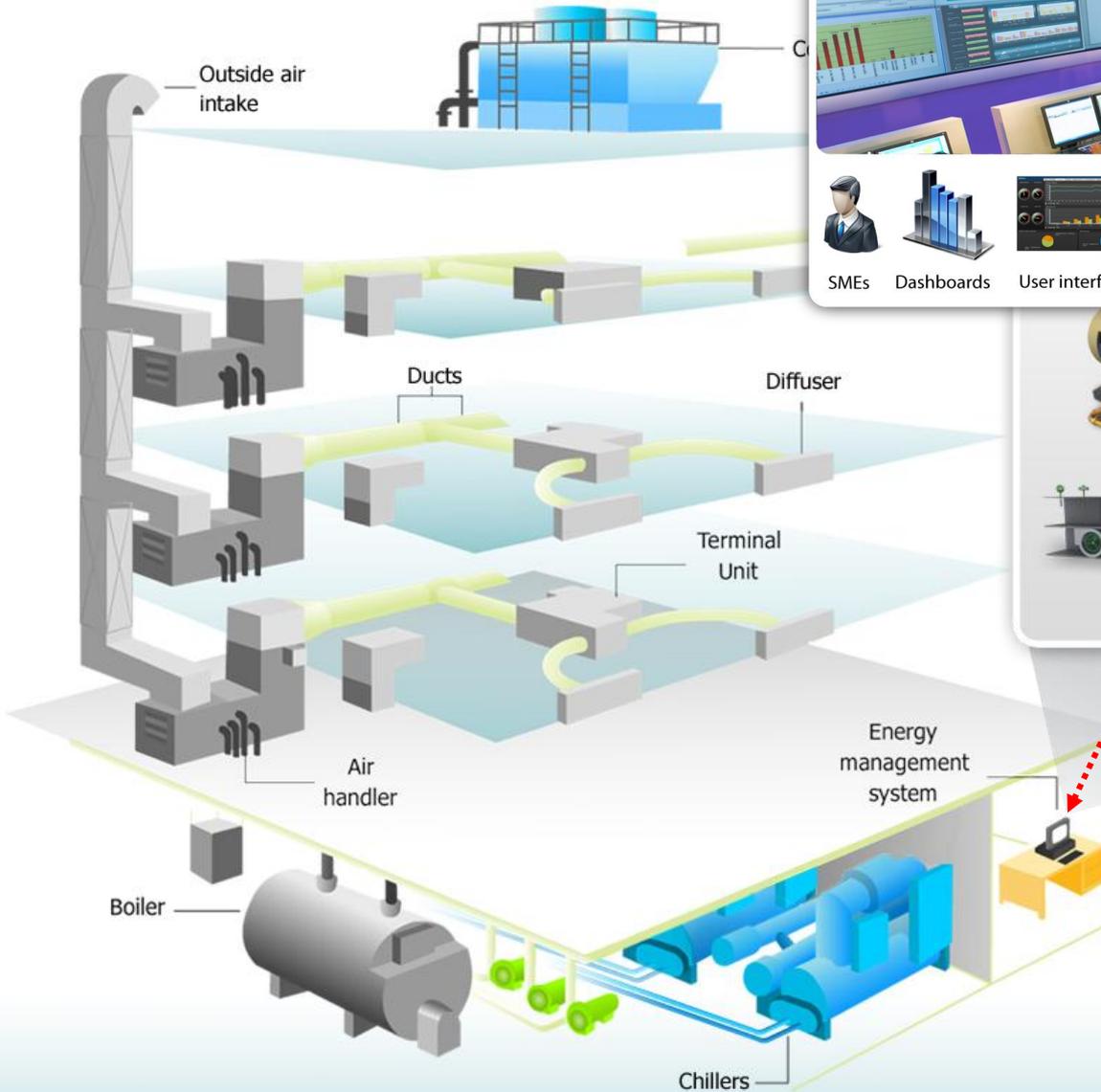
# Water Ecosystem



# Air Ecosystem



# Monitoring

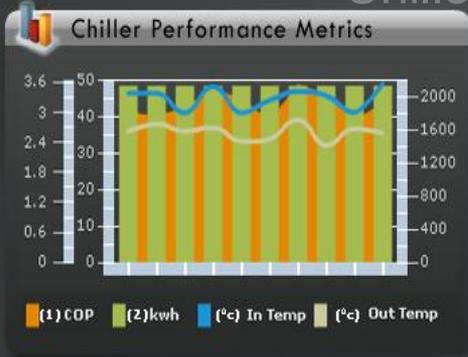


## Galaxy

A monitoring center with multiple computer workstations and large display screens showing data visualizations. Below the center is a row of icons representing system features: SMEs (person icon), Dashboards (bar chart), User interfaces (control panel), Reports (document with 'Report' text), Carbon footprint measurement (CO2 footprint icon), Benchmarking (line graph), Remote monitoring (warning cone), and Engineers (person with hard hat icon).

Measurement & Verification components including a large industrial motor, a server rack with green lights, and a server cabinet with a circular gauge. A red dashed arrow points from the Energy management system in the main diagram to this section.

# Chiller Plant Analysis Tool



43 C Outside Air Temperature

78 % Humidity

detailed analysis

refrigeration cycle

Electrical Load 66.5 kW

Energy Consumption 1312.4 kWh

Comp A ●

Run Hrs 4892.0 hrs

Percentage Load 70.0%

Comp B ●

Run Hrs 5179.0 hrs

Percentage Load 100.0%

**COMPRESSOR B**

- MOTOR CURRENT 100.0 A
- MOTOR TEMPERATURE 87.4 °C
- DISCHARGE GAS TEMPERATURE 53.5 °C
- DISCHARGE GAS PRESSURE 51.2 psi
- SUCTION PRESSURE 43.7 psi
- SATURATED SUCTION TEMPERATURE 5.3 °C
- OIL PRESSURE 45.9 psi
- OIL PRESSURE DIFFERENCE 2.5 psi
- SATURATED CONDENSING TEMPERATURE 36.1 °C

**COMPRESSOR A**

- MOTOR CURRENT 99.0 A
- MOTOR TEMPERATURE 90.3 °C
- DISCHARGE GAS TEMPERATURE 46.7 °C
- DISCHARGE GAS PRESSURE 117.6 psi
- SUCTION PRESSURE 44.0 psi
- SATURATED SUCTION TEMPERATURE 9.8 °C
- OIL PRESSURE 106.9 psi
- OIL PRESSURE DIFFERENCE 51.4 psi
- SATURATED CONDENSING TEMPERATURE 10.2 °C

FROM BUILDING 11.1 °C

TO BUILDING 7.7 °C

FROM COOLING TOWER 30.9 °C

TO COOLING TOWER 33.6 °C

# ICT for energy savings in buildings

## Command Control Center

- SMEs
- Dashboards
- User interfaces
- Reports
- Carbon footprint measurement
- Benchmarking
- Remote monitoring
- Engineers



Villas

- Fire
- Safety & security
- Energy
- HVAC
- CCTV
- Carbon footprint



Factories

- Fire
- Lift
- Safety & security
- Energy
- Chiller / HVAC
- Boiler
- CCTV
- Carbon footprint



Schools

- Fire
- Safety & security
- Energy
- Chiller / HVAC
- CCTV
- Carbon footprint



Commercial & residential buildings

- Fire
- Lift
- Safety & security
- Energy
- Chiller / HVAC
- Boiler
- CCTV
- Carbon footprint



Utilities

- Sewage pumps
- Water treatment plants
- Irrigation



Hospitals

- Fire
- Lift
- Safety & security
- Energy
- Chiller / HVAC
- Boiler
- CCTV
- Carbon footprint

# Remote Service Maintenance

RETAIL  
VERTICAL

INDUSTRIAL  
VERTICAL

LIFE & SAFETY  
VERTICAL

BUILDINGS

SECURITY  
& SURVEILLANCE

TRANSPORT  
VERTICAL

DATA CENTER  
VERTICAL

HEALTH  
VERTICAL

HOTELS  
VERTICAL

AIRPORT  
VERTICAL

EDUCATIONAL  
VERTICAL

ENERGY  
VERTICAL



Hybrid Level

A **collection** of  
**independent computers**  
that appears to its users  
as a **single coherent**  
**system.**

A collection of **autonomous** computers linked by a computer **network** and supported by **software** that enables the collection to operate as an **integrated** facility.

You know you have one  
when the **crash** of a computer  
you have **never heard of**  
**stops you** from getting any  
work done. (Leslie Lamport)

# Types of Distributed Systems (1)

- Object/component based (CORBA, EJB, COM)
- File based (NFS)
- Document based (WWW, Lotus Notes)
- Coordination (or event-) based (Jini, JavaSpaces, publish/subscribe, P2P)
- Resource oriented (GRID, Cloud, P2P, MANET)
- Service oriented (Web services, Cloud, P2P)

# Types of Distributed Systems (2)

- Distributed **Computing** (cluster, GRID, cloud)
- Distributed **Information Systems** (EAI, TP, SOA)
- Distributed **Pervasive Systems** (often P2P, UPnP in home systems, sensor networks, ...)

- Communication
- Concurrency and operating system support (competitive, cooperative)
- Naming and discovery
- Synchronization and agreement
- Consistency and replication
- Fault-tolerance
- Security

# KEY CONCEPTS AND DESIGN GOALS



# Why distribute at all?

- Connecting users to resources and services
  - Basic function of a distributed system
- Dependability and Security
  - Availability, Fault Tolerance (FT), Intrusion Tolerance, ...
- Performance
  - Latency, throughput, ...

**Otherwise: Don't distribute, its far more complex hence expensive, error-prone, ...**

- Resource sharing (collaborative, competitive)
- Transparency
- Hiding internal structure, complexity
  - Openness, Portability, Interoperability, ...
- Services provided by standard rules
- Scalability
- Ability to expand the system easily
- Concurrency
  - inherently parallel (not just simulated)
- Fault Tolerance (FT), availability



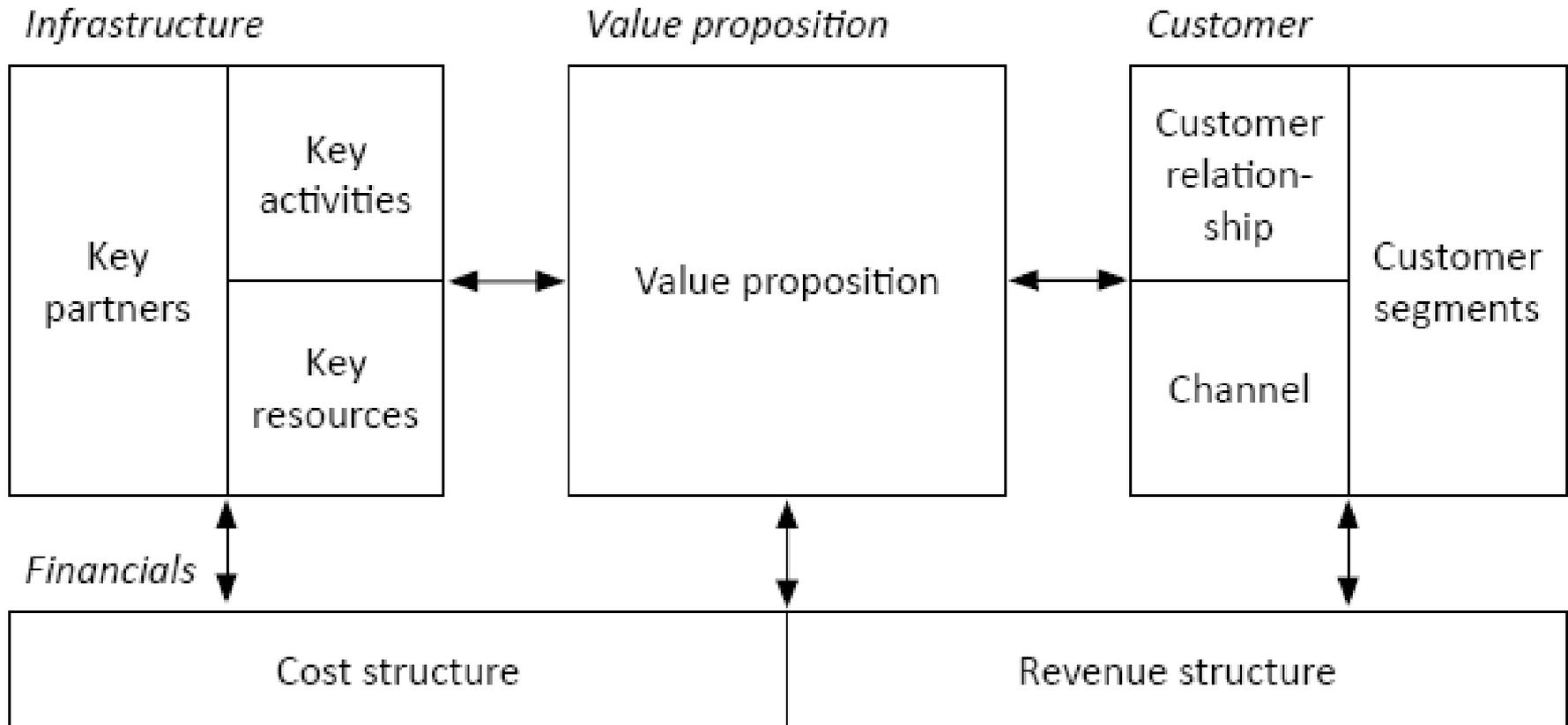
# The 8 Fallacies of Distributed Computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

*Essentially everyone, when they first build a distributed application, makes the above eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences. (Peter Deutsch)*



- Access and share (remote) resources
- Business Models and policies (see next slide)
- Collaboration by information exchange
- Communication (Convergence, VoIP)
- Groupware and virtual organizations
- Electronic and mobile commerce
- Sensor/actor networks in automation and pervasive computing (fine grained distribution)
- May compromise security (tamper proof HW) and privacy (tracking, spam)



# Quality of Service (QoS)

- QoS is a concept with which clients can indicate the level of service (SLA) they require

Examples:

- For real-time voice communication, the client prefers reliable delivery times over guaranteed delivery
- In financial applications, a client may prefer encrypted communication in favor of faster communication
- You can't have it all -> **Trade-offs!**



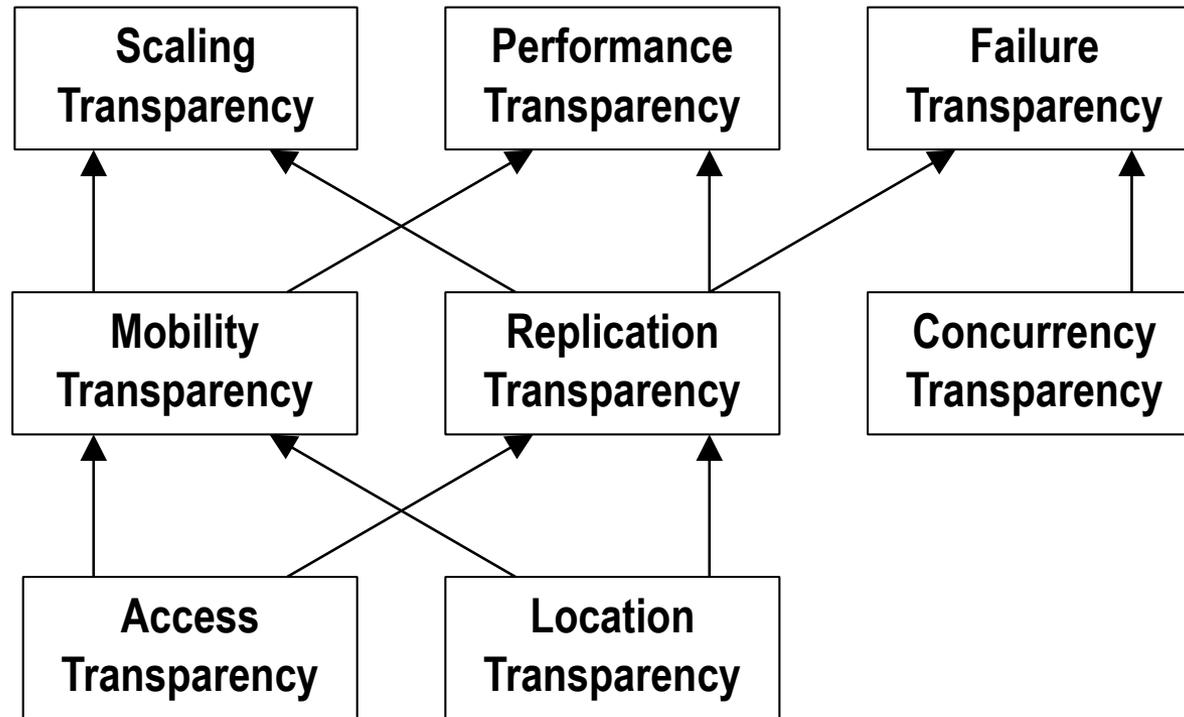
# Transparency

- Concept: **Hide** different aspects of distribution from the client. It is the ultimate goal of many distributed systems.
- It can be achieved by providing lower-level (system) services (i.e. use another **layer**).
- The client uses these services instead of hardcoding the information.
- The **service layer provides** a service with a certain Quality of Service.

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Different forms of transparency in a distributed system (ISO, 1995).

## Transparency: Information Hiding Applied to Distributed Systems



- Not blindly try to hide every aspect of distribution
- Performance transparency difficult (LAN/WAN)
- **Trade-off transparency/performance**
  - Failure masking
  - Replica consistency
- → Transparency is an important goal, but has to be considered together with all other non-functional requirements and with respect to particular demands

- Offer services according to standard rules (syntax and semantics: format, contents, and meaning)
- Formalized in protocols
- Interfaces (IDL): semantics often informal
  - Complete → Interoperability: Communication between processes
  - Neutral → Portability: Different implementations of interface
- Flexibility: composition, configuration, replacement, extensibility (CBSE)

# Separating Policy from Mechanism

- Granularity: objects vs. applications?
- Component interaction and composition standards (instead of closed/monolithic)
- E.g. Web browser provides facility to store cached documents, but caching policy can be plugged in arbitrarily (parameters or algorithmic).

## Web examples

- Different Web servers and Web browsers interoperate
- New browsers may be introduced to work with existing servers (and vice versa)
- Plugin interface allows new services to be added

- A distributed system's **ability to grow** to meet increasing demands along several dimensions:
  1. Size (users and resources)
  2. Geographically (topologically)
  3. Administratively (independent organizations/domains)
- System remains effective
- System and application software should not need to change
- Trade-Off scalability/security

# Scalability Challenges (size)

- **Controlling the cost of physical resources:** The quantity required should be  $O(n)$ , i.e., the algorithm's performance is directly proportional to the size of the data set being processed
- **Controlling the performance loss:** In hierarchical system should be no worse than  $O(\log n)$ , i.e., the algorithm deals with a data set that is iteratively partitioned, like a balanced binary tree.
- **Preventing software resources running out**, but over-compensation may be even worse: Internet Addresses or Oracle7 2TB restriction
- **Avoiding performance bottlenecks** (centralized services, data, or algorithms)

<b>Concept</b>	<b>Example</b>
Centralized services	A single server for all users
Centralized data	A single on-line telephone book, central DNS
Centralized algorithms	Doing routing based on complete information

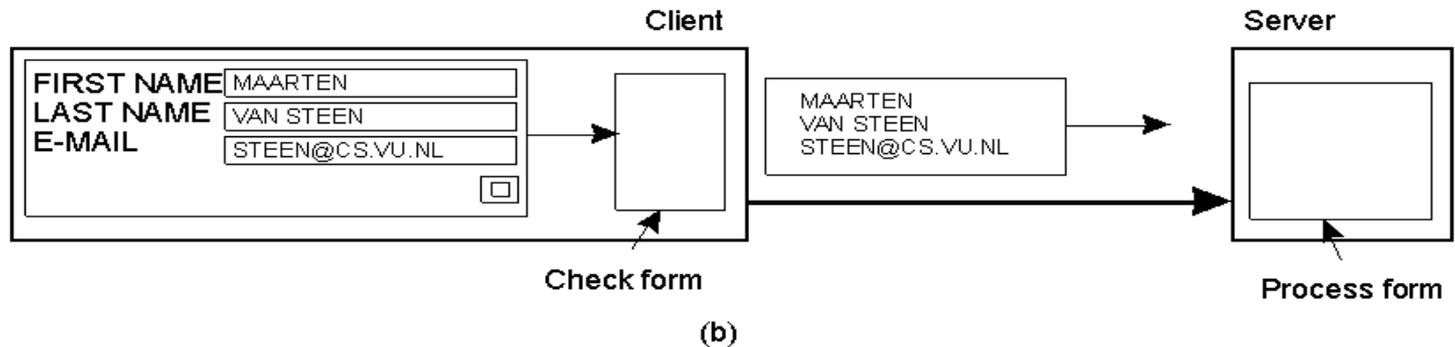
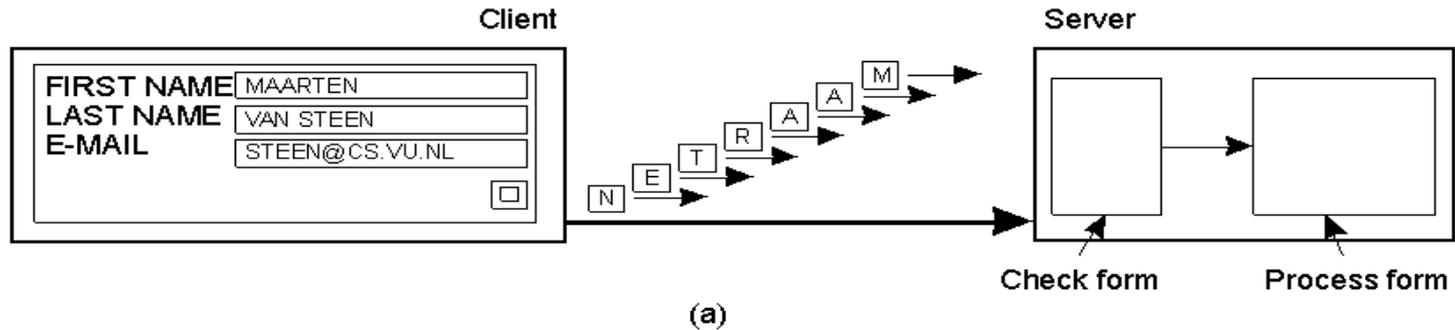
1. No machine has complete system state information
2. Machines make decisions based only on local (surrounding) information
3. Failure of one machine does not ruin the algorithm (no single point of failure)
4. No implicit assumption that a global clock exists

- LAN:
  - Synchronous communication
  - Fast
  - Broadcast
  - Highly reliable
- WAN:
  - Asynchronous communication
  - Slow
  - Point to point (e.g. problems with location service)
  - Unreliable

- e.g., Mobile Number portability, conflicting (orthogonal) policies:
  1. Resource usage
  2. Billing
  3. Management
  4. Security: Protection between the administrative domains
    - trusted domains – enforced limitations

- **Hiding communication latencies**  
Asynchronous communication (batch processing, parallel applications)  
Reduce overall communication (HMI)
- **Distribution**  
Hierarchies, domains, zones, ... → split
- **Replication**  
Availability, load balance, reduce communication  
Caching: proximity, client decision  
Consistency issues may adverse scalability!

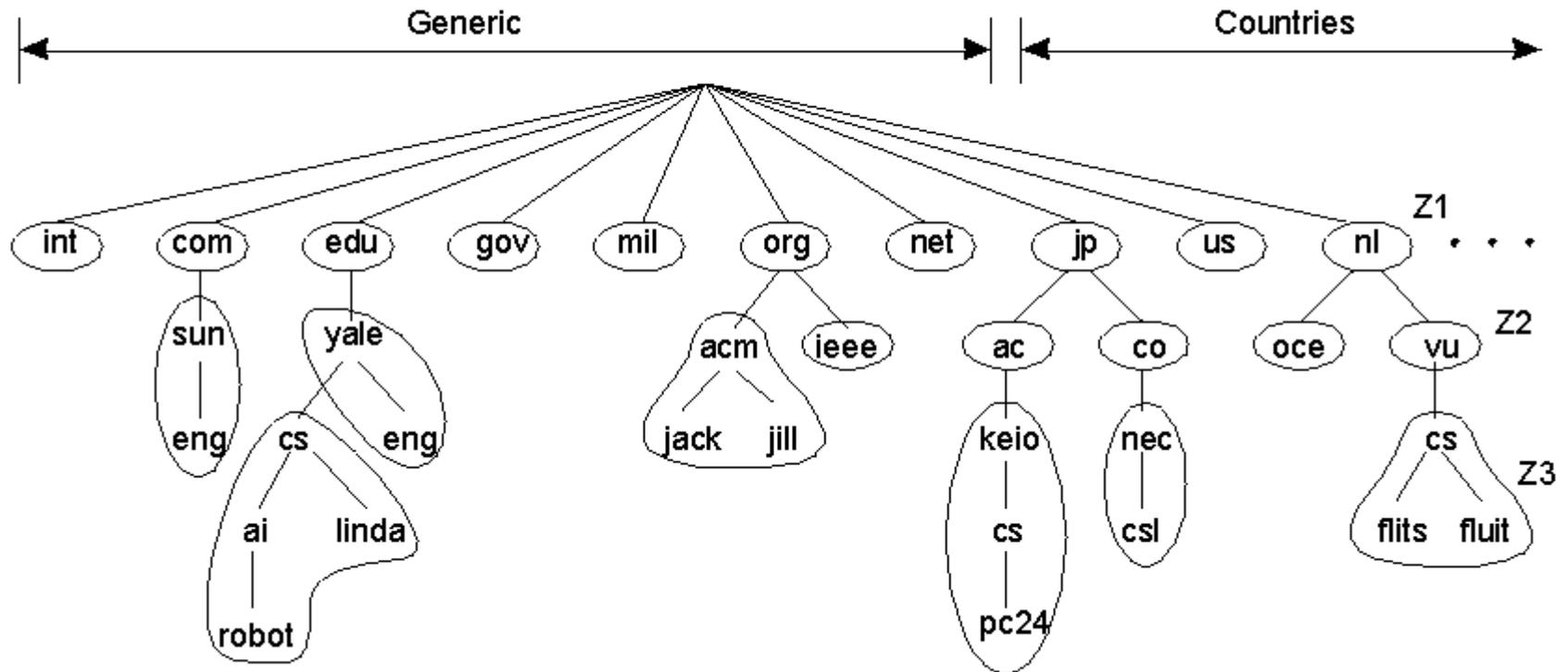
# Scaling Techniques (2)



The difference between letting:

(a) a server or (b) a client check forms as they are being filled

# Scaling Techniques (3)



# ARCHITECTURAL STYLES

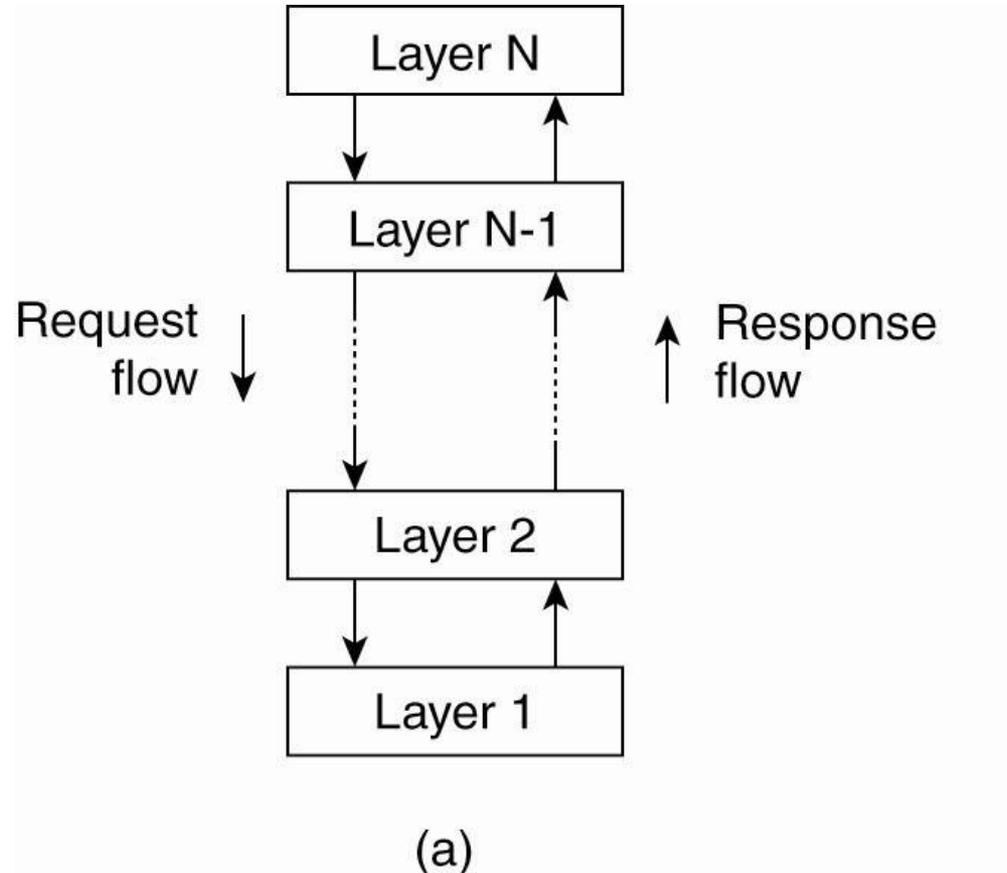
- **Abstraction (and modeling)**
  - Client, server, service
  - Interface versus implementation
- **Information hiding (encapsulation)**
  - Interface design
- **Separation of concerns**
  - Layering (filesystem example: bytes, disc blocks, files)
  - Client and server
  - Components (granularity issues)

- **Multiprocessors: shared memory** (requires protection against concurrent access)
- **Multicomputers: message passing**
- **Synchronization in shared memory**
  - Semaphores (atomic mutex variable)
  - Monitors — an abstract data type whose operations may be invoked by concurrent threads; different invocations are synchronized
- **Synchronization in multicomputers: blocking in message passing**

## Important styles of architecture for distributed systems

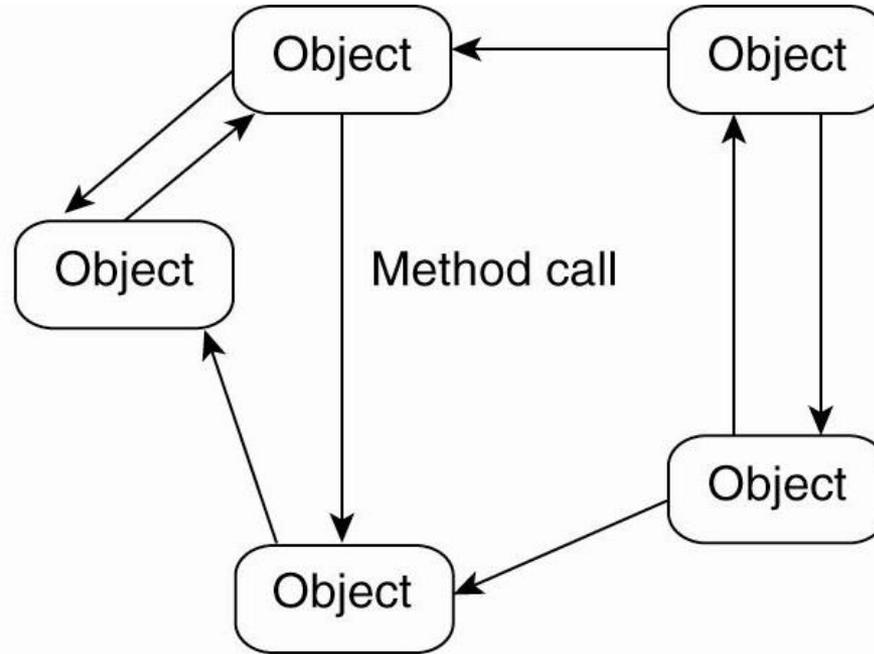
- Layered architectures
- Object-based architectures
- Data-centered architectures
- Event-based architectures

# Architectural Styles (2)



The **layered** architectural style

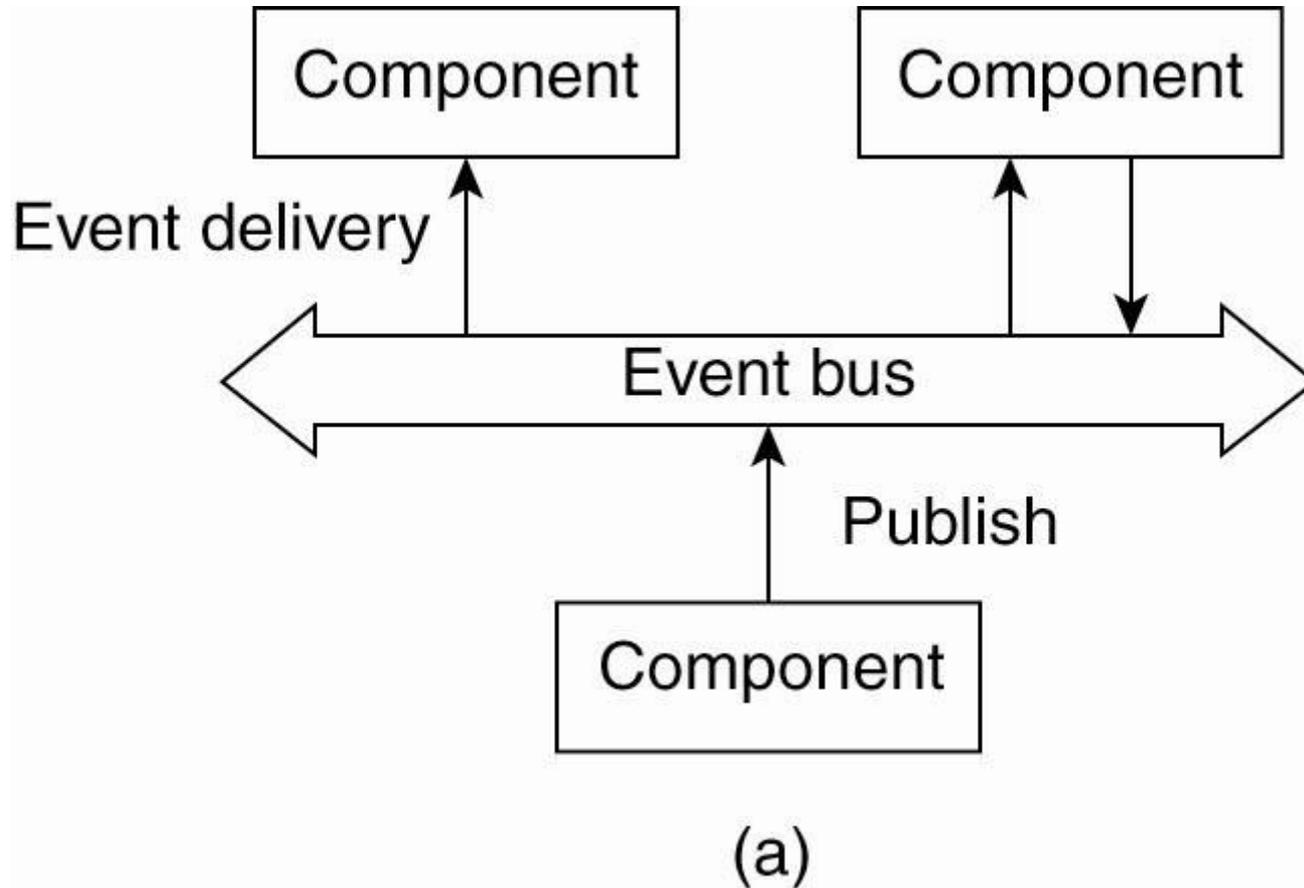
# Architectural Styles (3)



(b)

The **object-based** architectural style

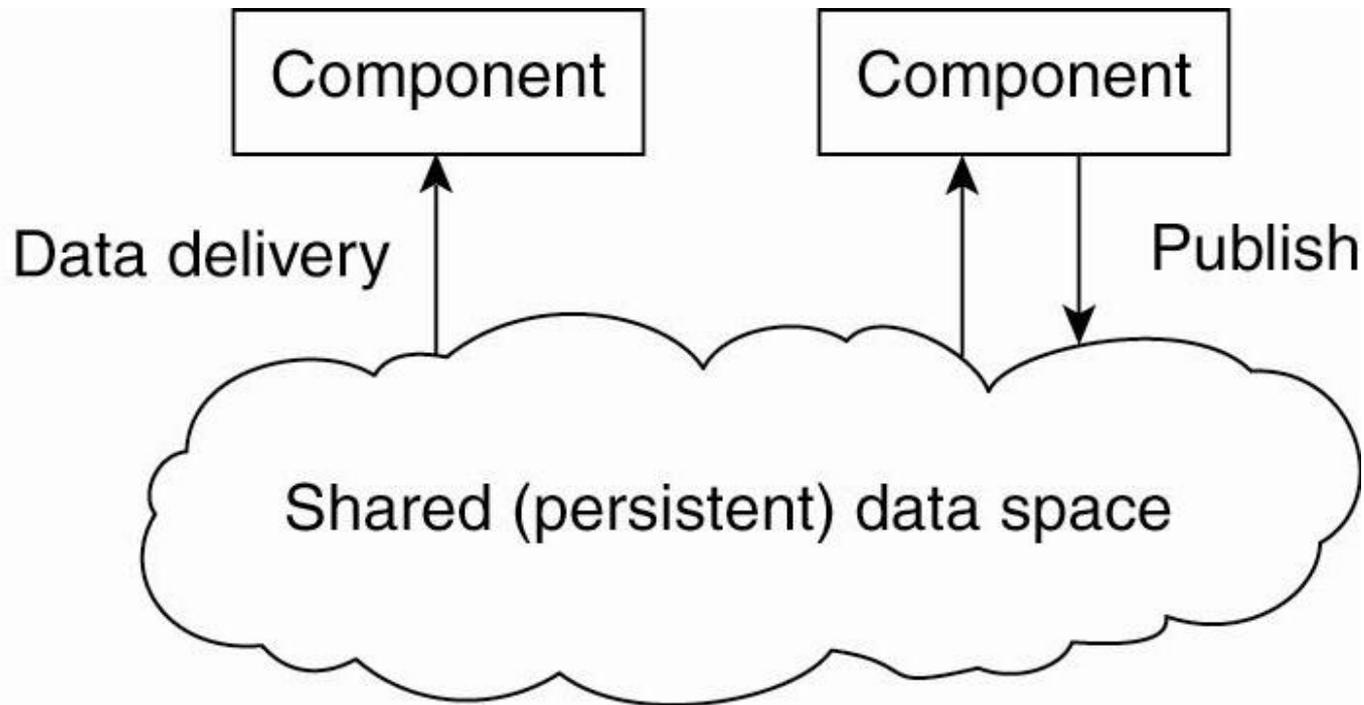
# Architectural Styles (4)



The **event-based** architectural style

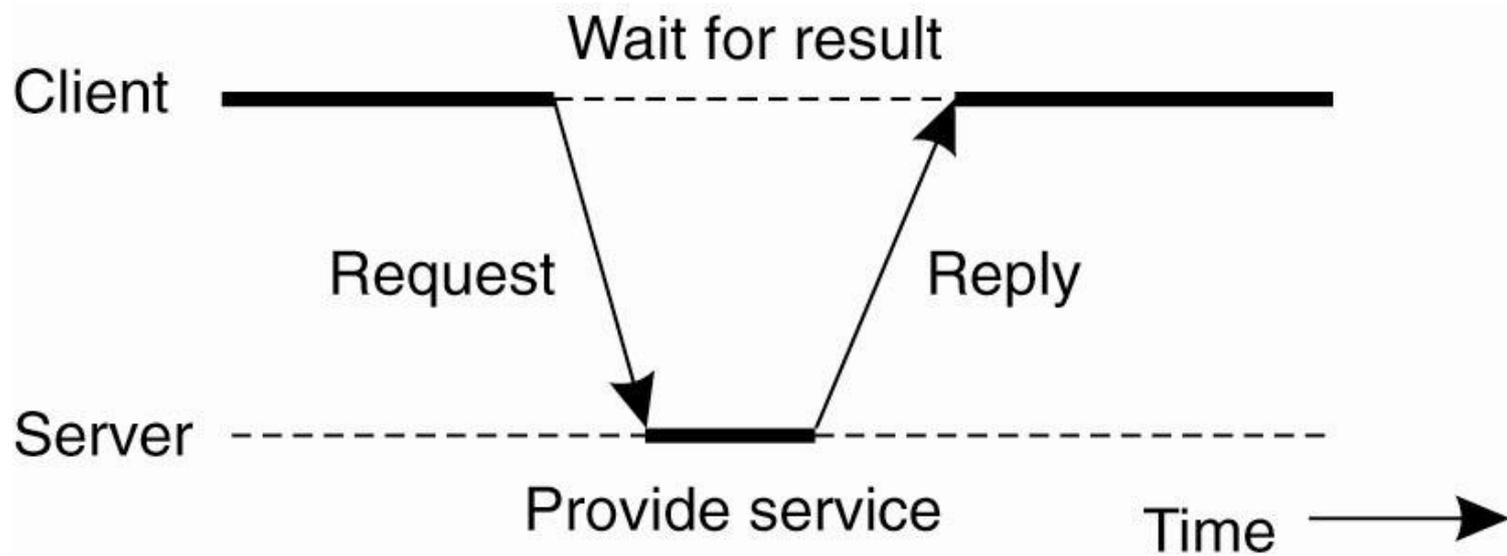
# Architectural Styles (5)

The **shared data-space** architectural style.



(b)

General interaction between a client and a server.



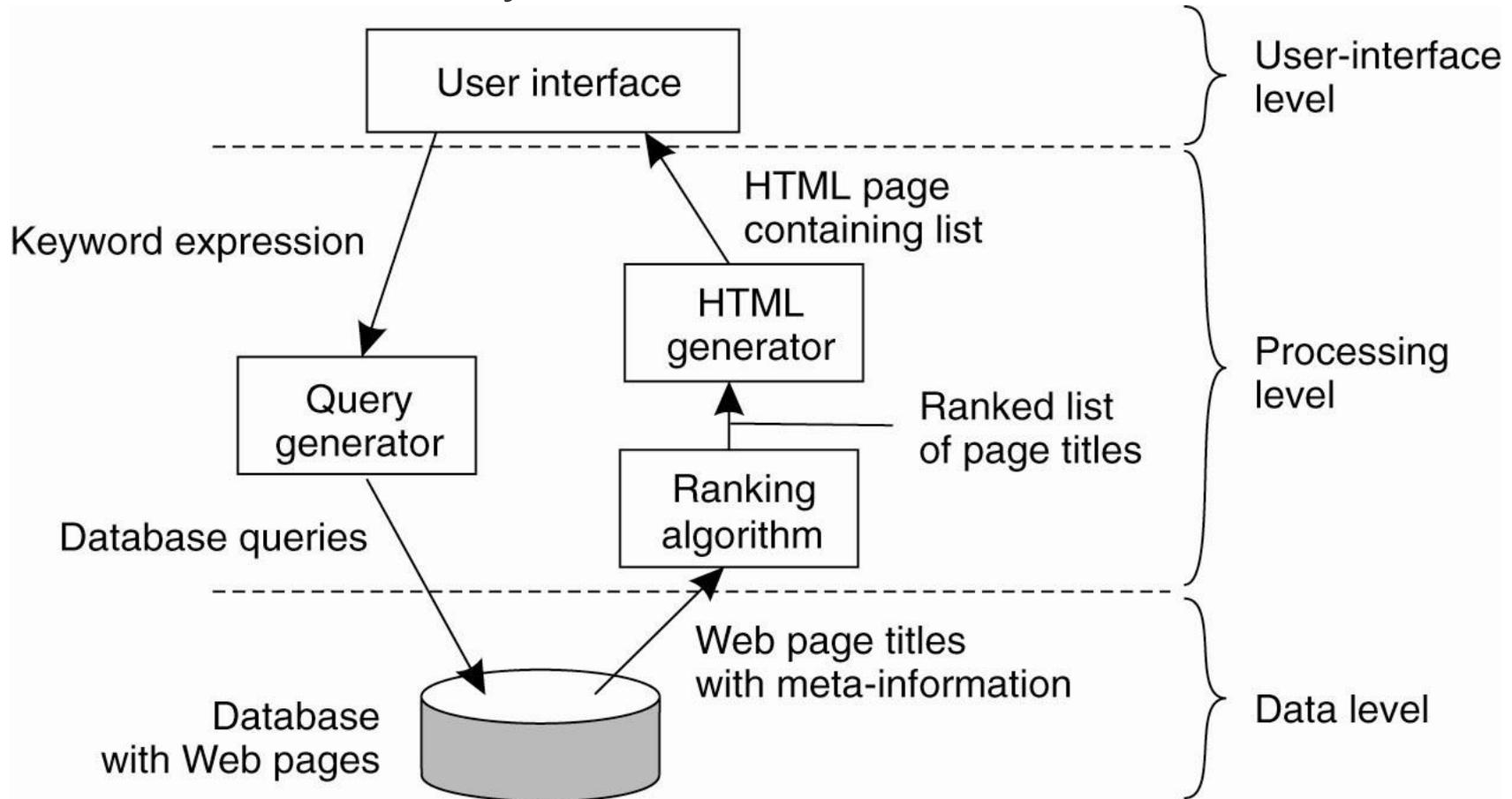
# Application Layering (1)

Recall previously mentioned layers of architectural style

- The user-interface level
- The processing level
- The data level

# Application Layering (2)

The simplified organization of an Internet search engine into three different layers.





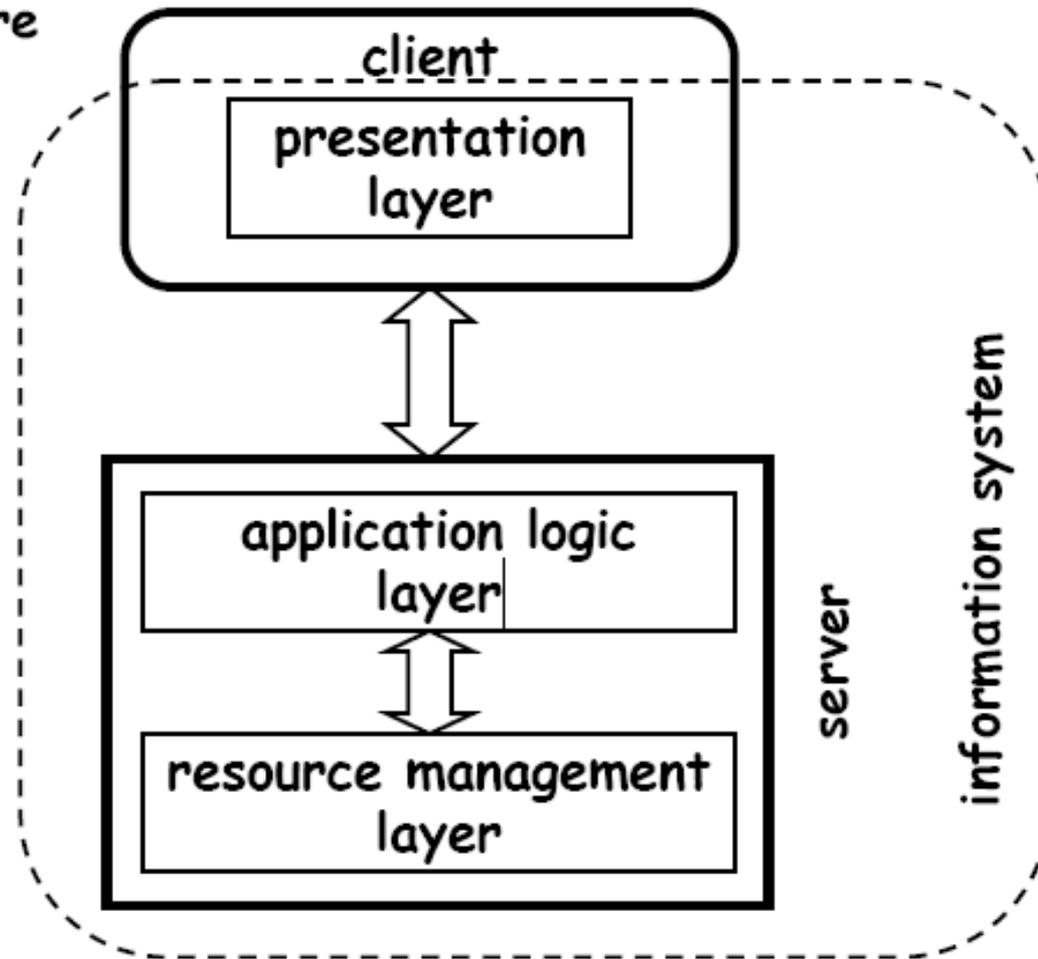
# Multitiered Architectures (1)

The **simplest organization is to have only two types of machines:**

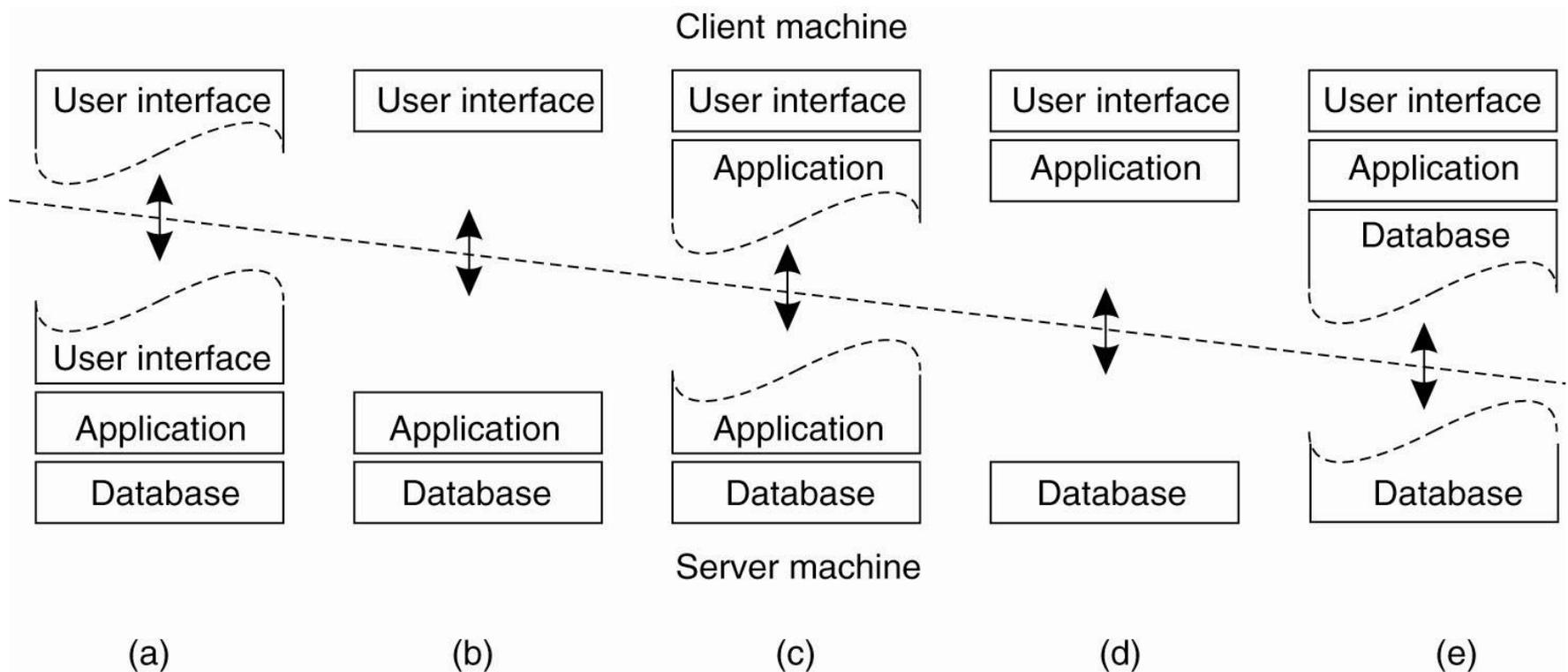
- A **client** machine containing only the programs implementing (part of) the user-interface level
- A **server** machine containing the rest, the programs implementing the processing and data level

# Two-tier Architecture

2-tier architecture

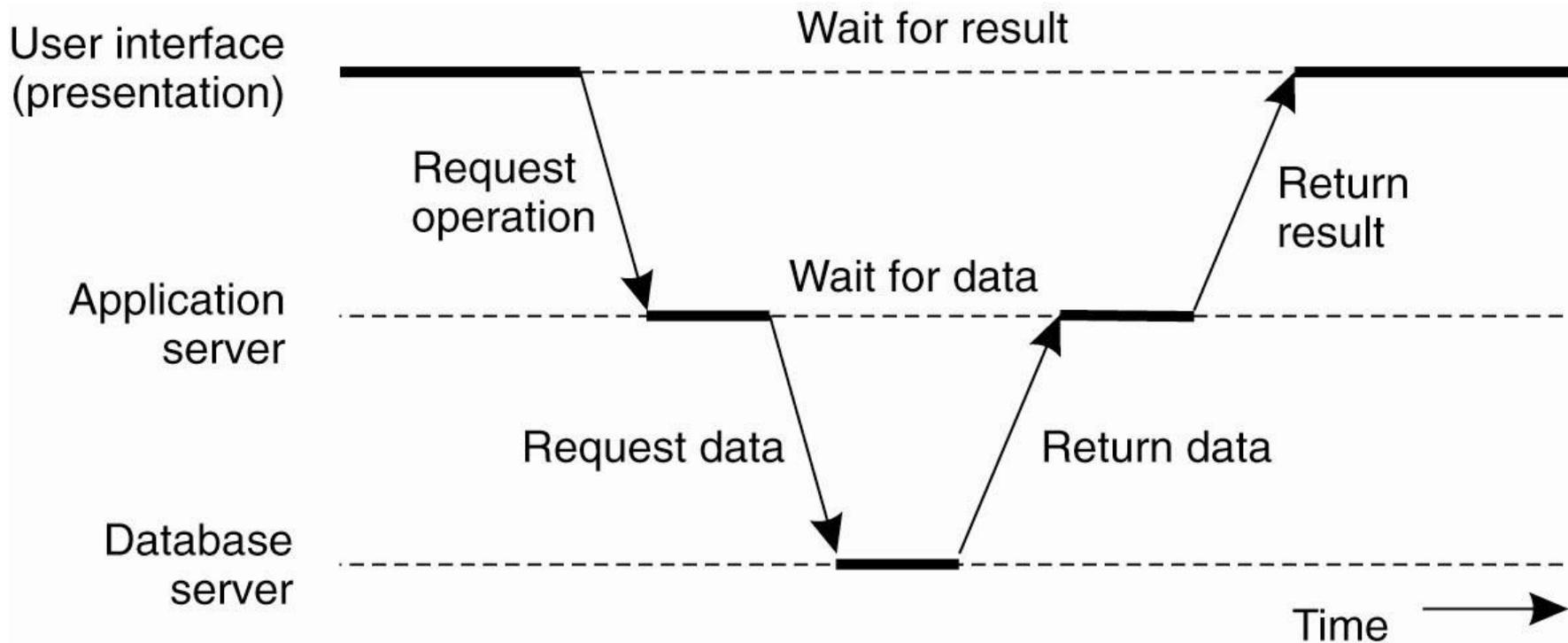


**Vertical Distribution:** Alternative client-server organizations (a)–(e).



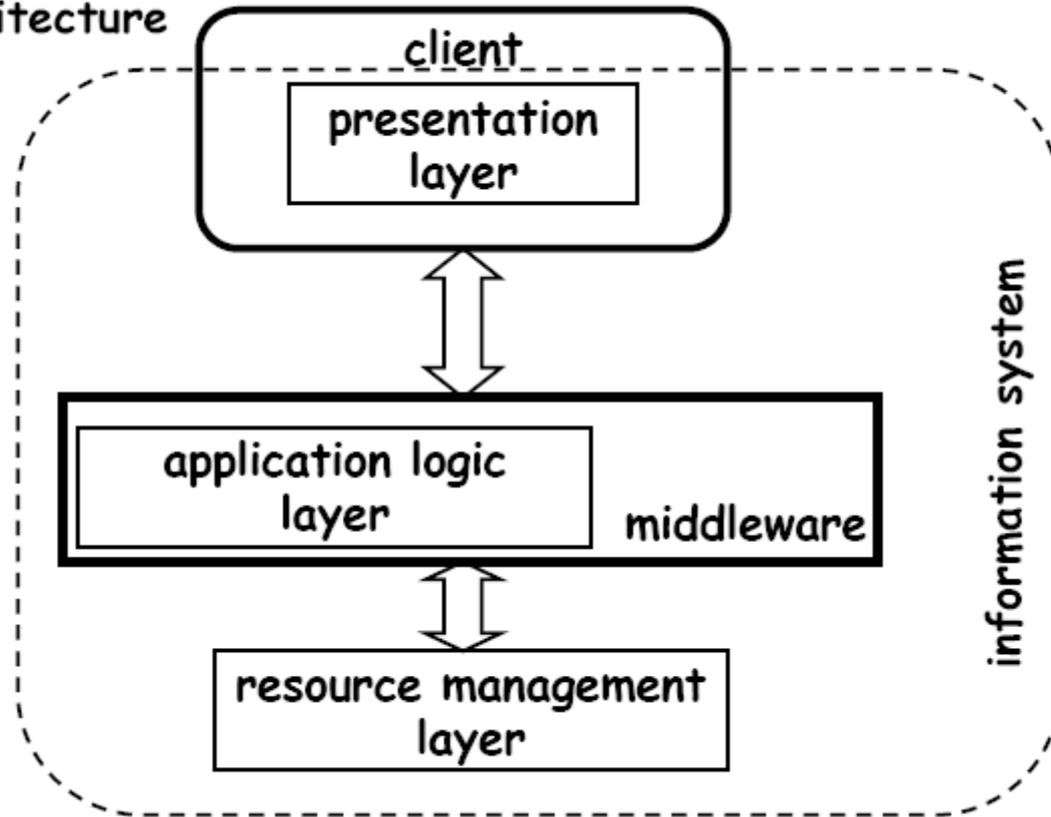
# Multi-tiered Architectures (3)

An example of a server acting as client.

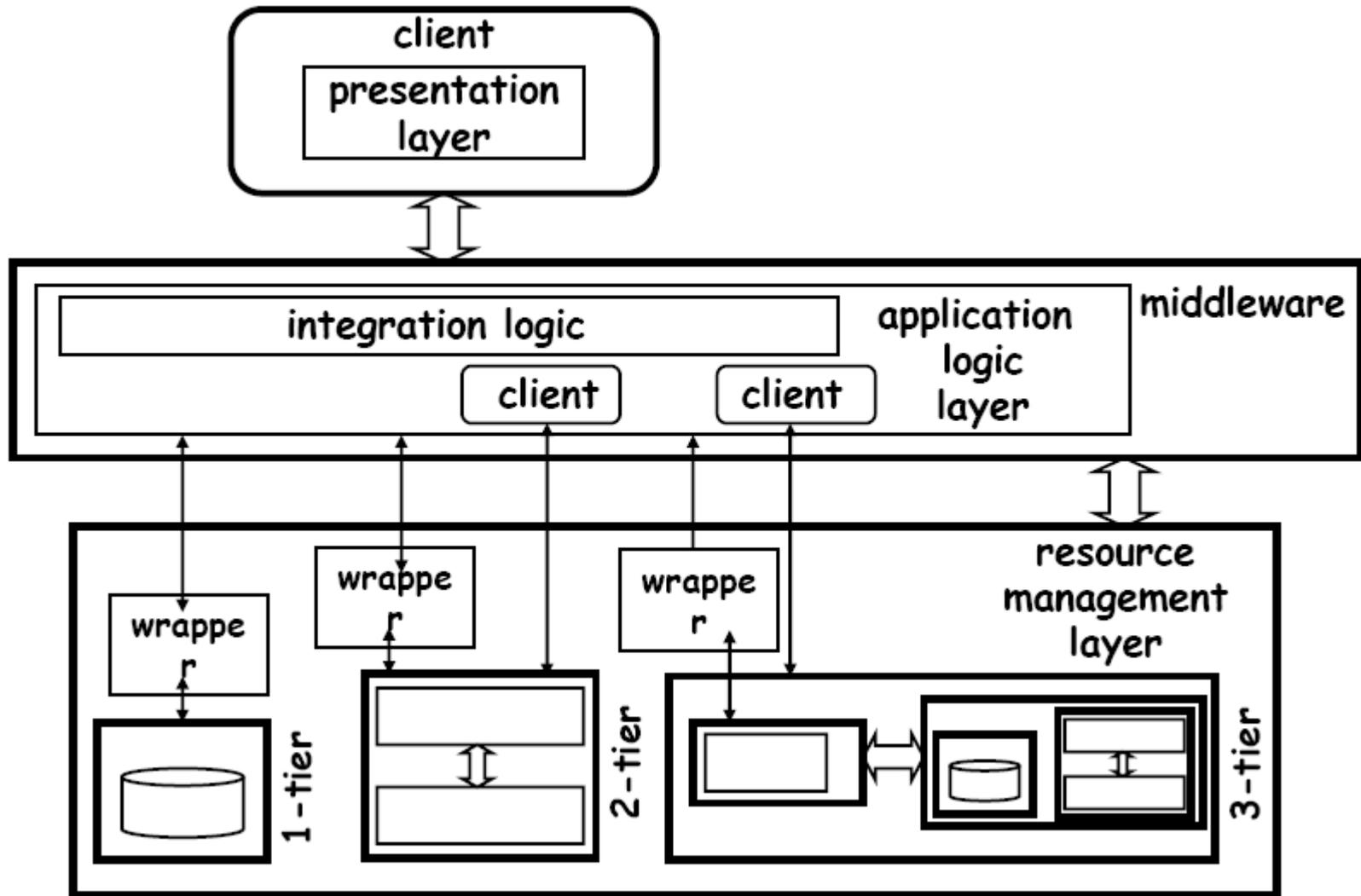


# Three-tier Architecture

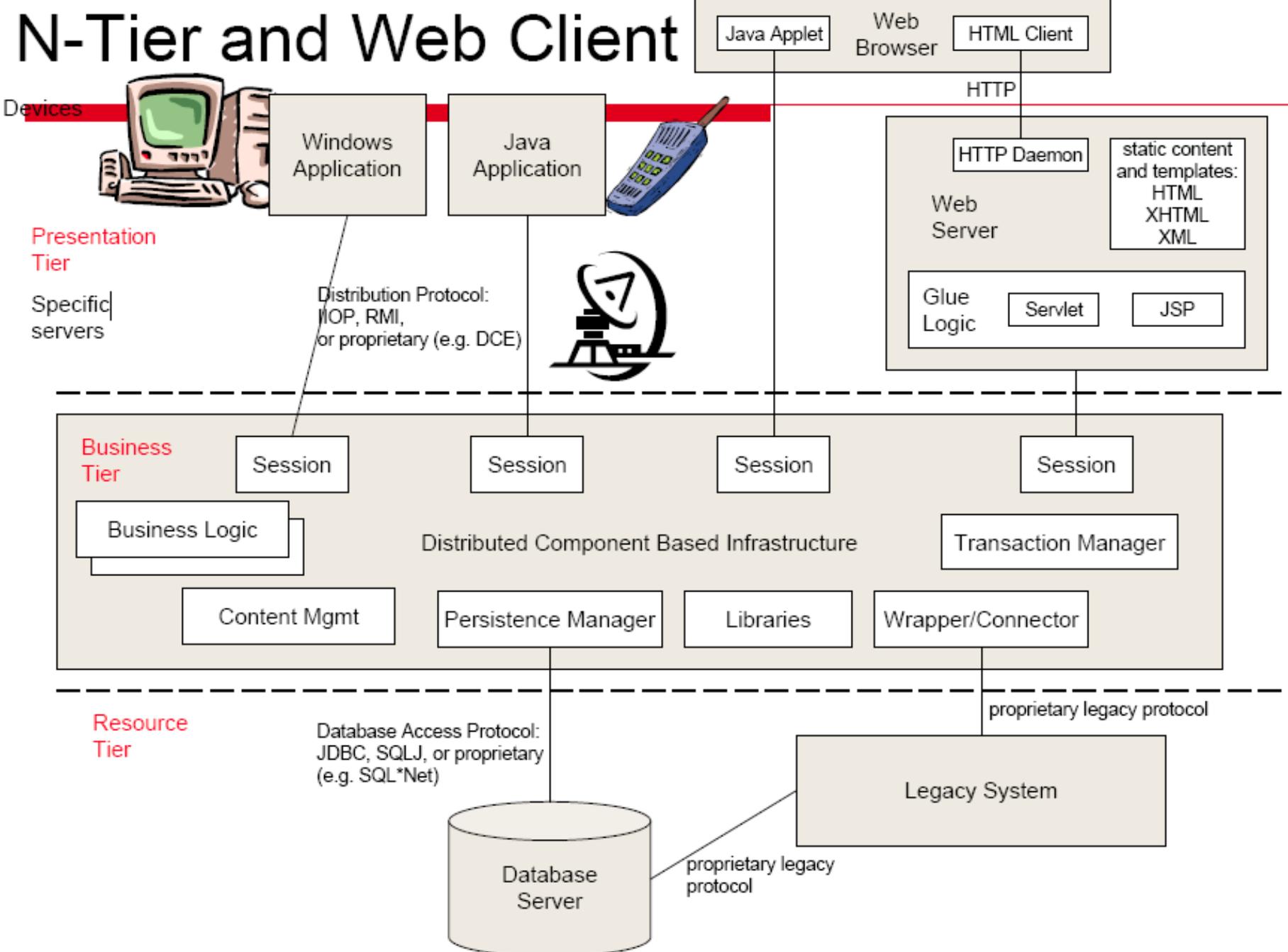
3-tier architecture



# Application (System) Integration



# N-Tier and Web Client



- A distributed system is a collection of computers working seamlessly together (single-system image – pro/con!)
- Distributed systems have evolved to be pervasive
- Principles and techniques are needed to cope with the complexity of distributed systems (openness, scalability, architectural styles, ...)
- Basic abstractions and concepts for distributed systems: client/server, layering (multitier), middleware, service, QoS, ...

# Thanks for your attention

Prof.Dr. Schahram Dustdar  
Distributed Systems Group  
Vienna University of Technology

[dustdar@dsg.tuwien.ac.at](mailto:dustdar@dsg.tuwien.ac.at)  
[dsg.tuwien.ac.at](http://dsg.tuwien.ac.at)

# Communication in Distributed Systems – Fundamental Concepts

Hong-Linh Truong  
Distributed Systems Group,  
Vienna University of Technology

[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
[dsg.tuwien.ac.at/staff/truong](http://dsg.tuwien.ac.at/staff/truong)

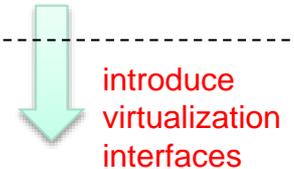
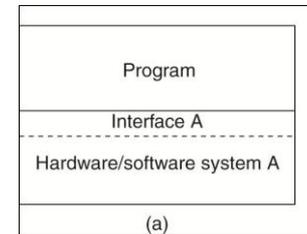
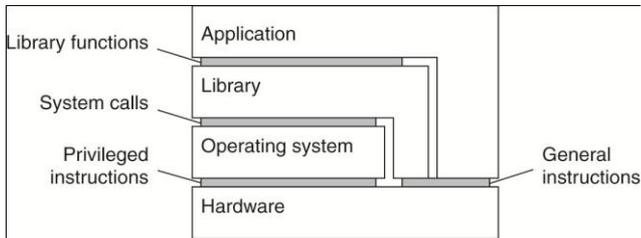
# Learning Materials

- Main reading:
  - Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall
    - Chapters 3 & 4
- Others
  - George Coulouris, Jean Dollimore, Tim Kindberg, „Distributed Systems – Concepts and Design“, 2nd Edition
    - Chapters 3,4, 6.
  - Craig Hunt, TCP/IP Network Administration, 3edition, 2002, O'Reilly.
- Test the examples in the lecture

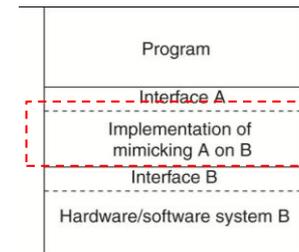
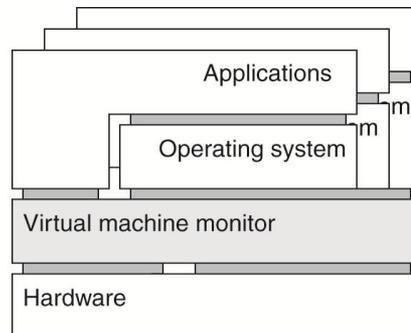
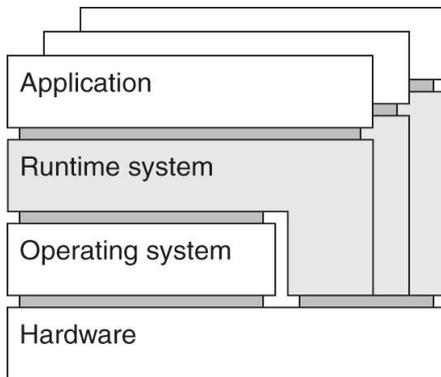
- Programs, interfaces, systems, and communication
- Key issues in communication in distributed systems
- Protocols
- Processing requests
- Summary

# Programs, interfaces, systems and communication (1)

## Program, interface and systems in typical machines



## Program, interface and systems in typical virtual machines



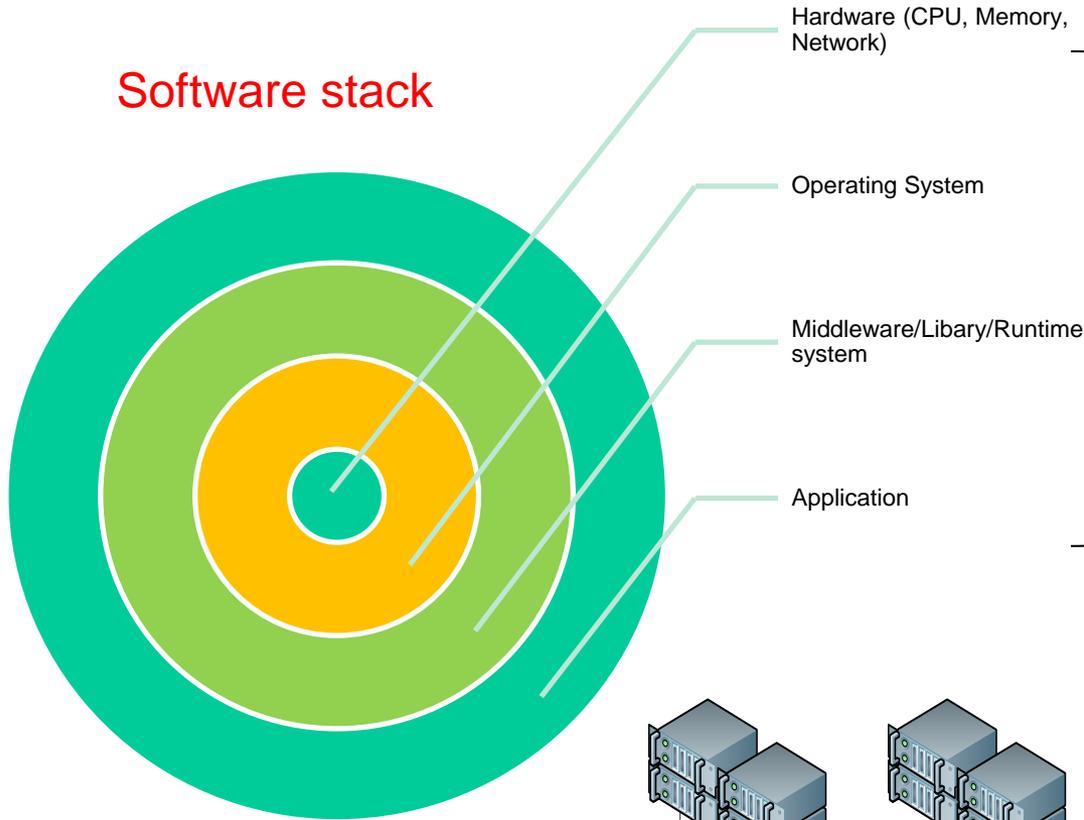
Figures source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Q: Why virtualization techniques are useful?



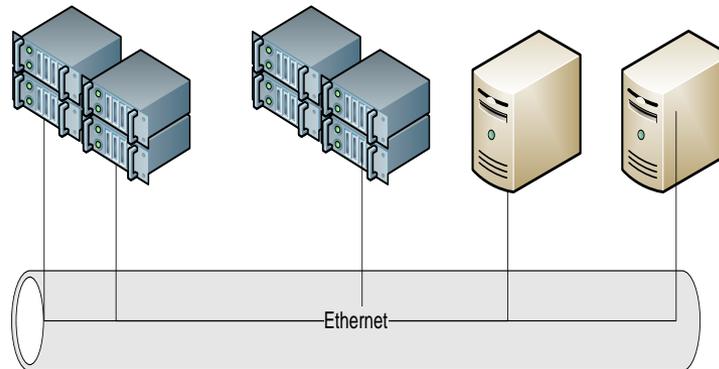
# Programs, interfaces, systems and communication (2)

## Software stack



## Programs

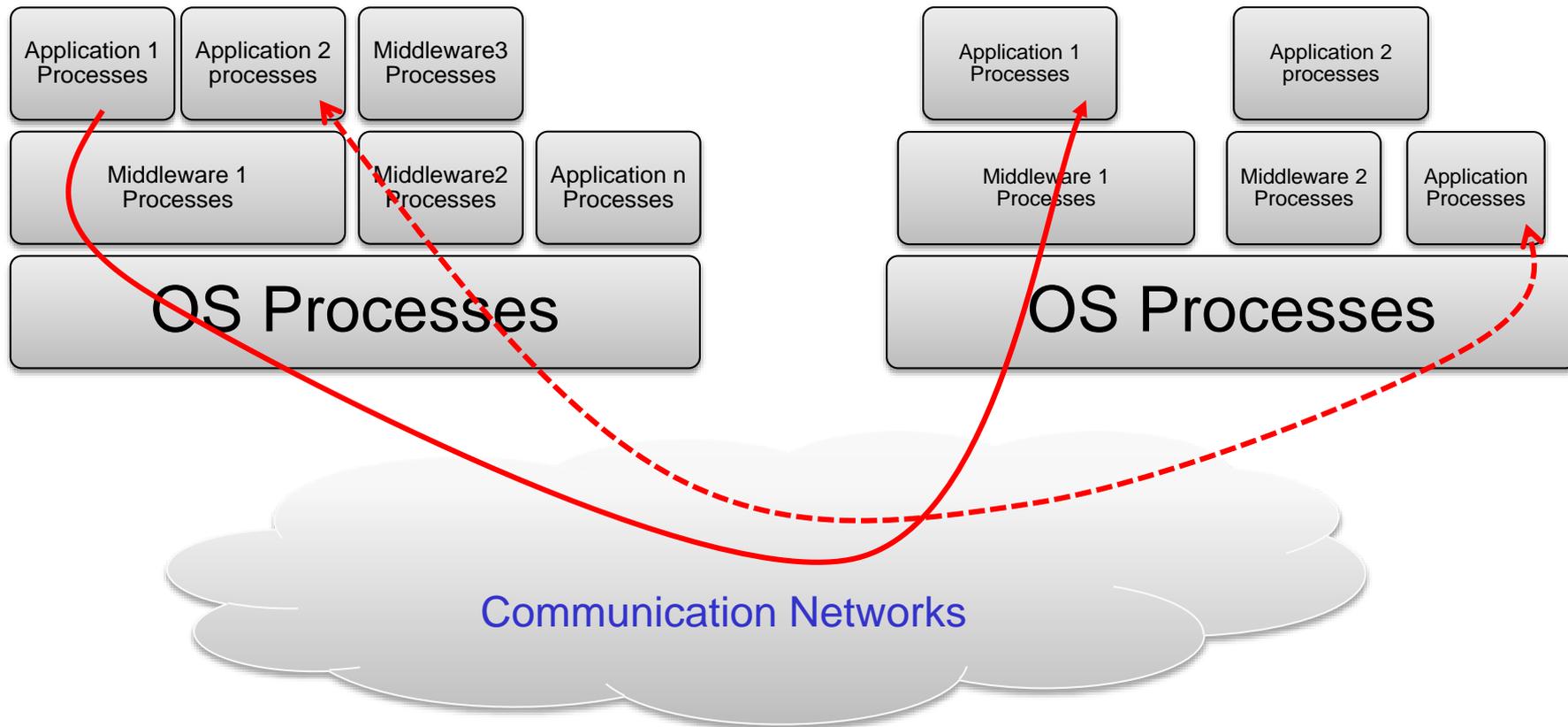
- C/C++/Java, Python, ...
- Different types of programs: systems versus applications; sequential versus parallel ones; clients versus servers/services



# Programs, interfaces, systems and communication (3)

Communication in distributed systems

- **between processes within a single** application/middleware/service
- **among processes belonging to different** applications/middleware/services



# KEY ISSUES

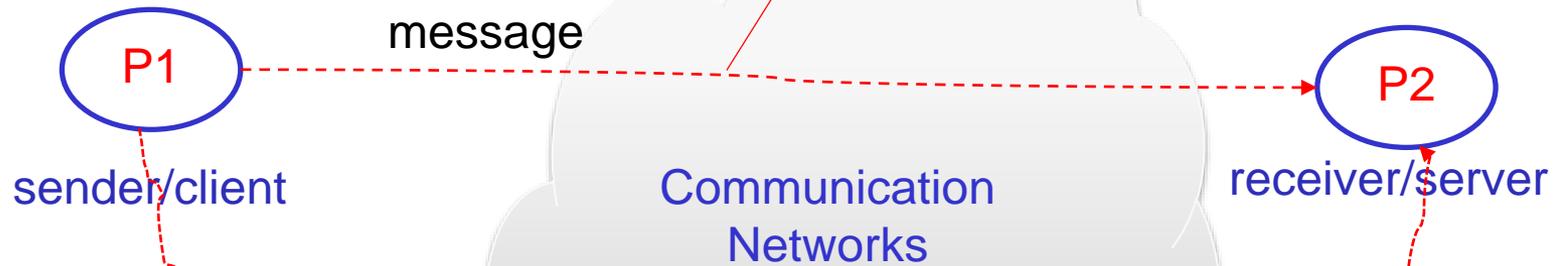
# Communication networks in distributed systems

- Maybe designed for specific **types of environments**
  - High performance computing, M2M, building/home, etc.
  - Voices, documents, sensory data, etc.
- Distributed, different **network spans**
  - Personal area networks (PANs), local area networks (LANs), campus area networks (CANs), metropolitan area networks (MANs), and wide area networks (WANs)
- Different **layered networks** for distributed systems
  - Physical versus overlay network topologies (virtual network topologies atop physical networks)

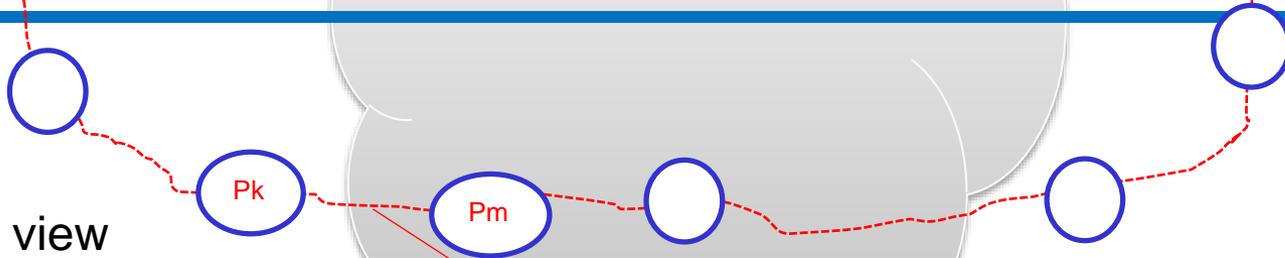
# Layered communication

In the view of P1 and P2

End-to-end process to process communication  
 e.g., email [abc@tuwien.ac.at](mailto:abc@tuwien.ac.at) to [ab@gmail.com](mailto:ab@gmail.com)



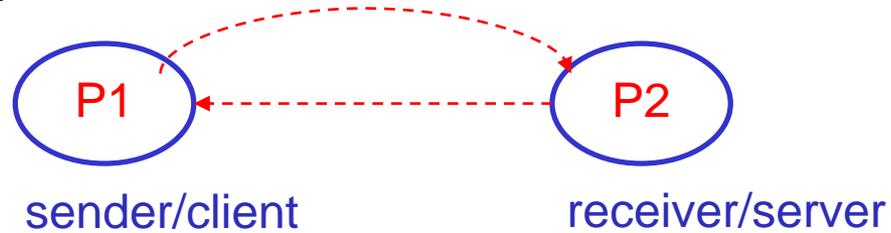
Holistic system view



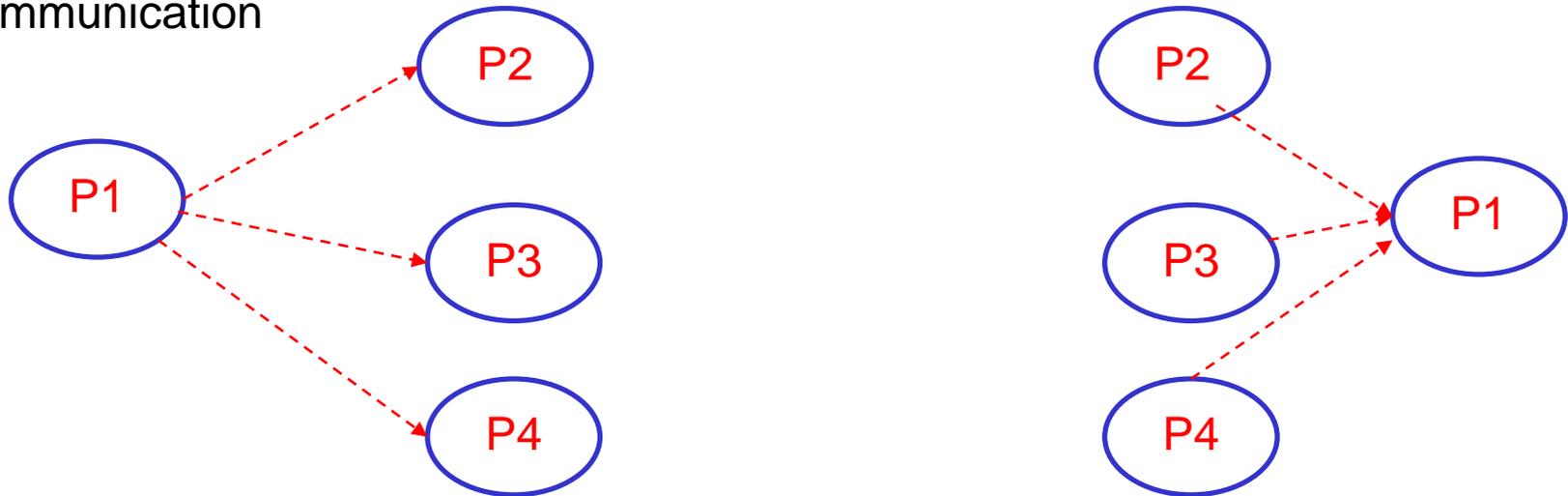
End-to-end process to process communication

# Communication Patterns

One-to-one/client-server



Group communication



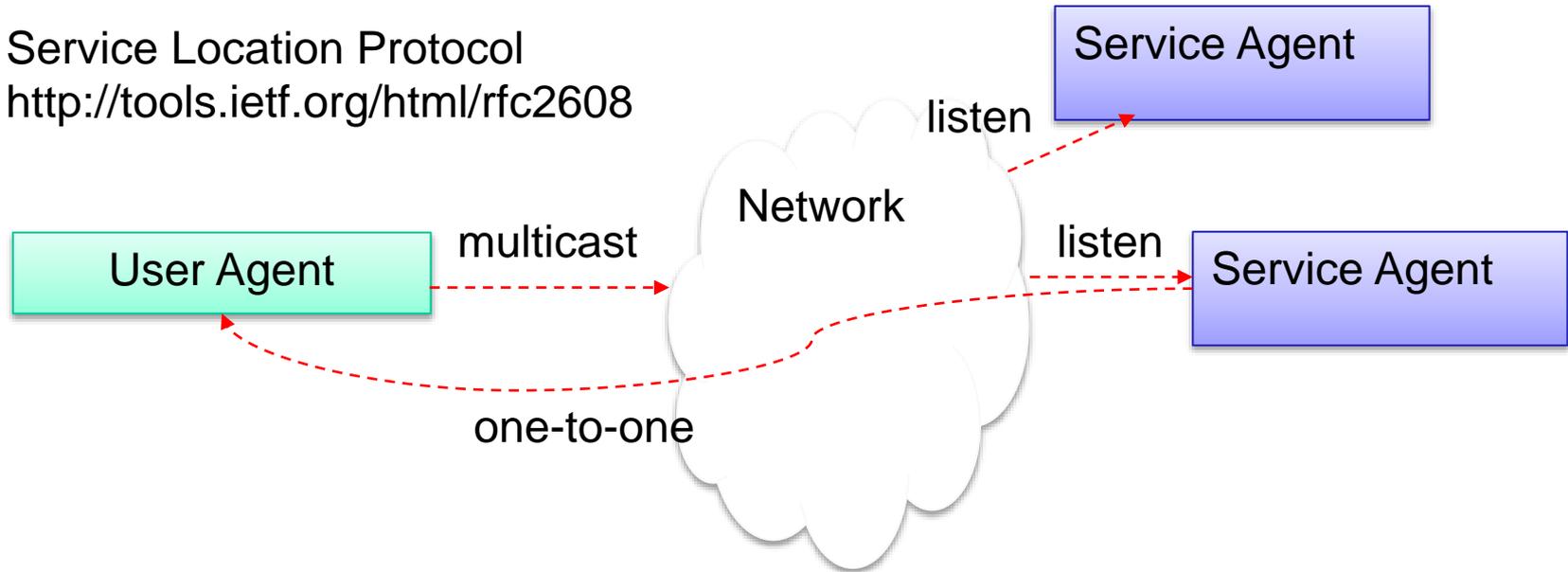
Q: What are the benefits of group communication, give some examples?

# Identifiers of entities participating in communication

- Communication cannot be done without knowing identifiers (names) of participating entities
  - Local versus global identifier
  - Individual versus group identifier
- Multiple layers/entities → different forms of identifiers
  - Process ID in an OS
  - Machine ID: name/IP address
  - Access point: (machine ID, port number)
  - A unique communication ID in a communication network
  - Emails for humans
  - Group ID

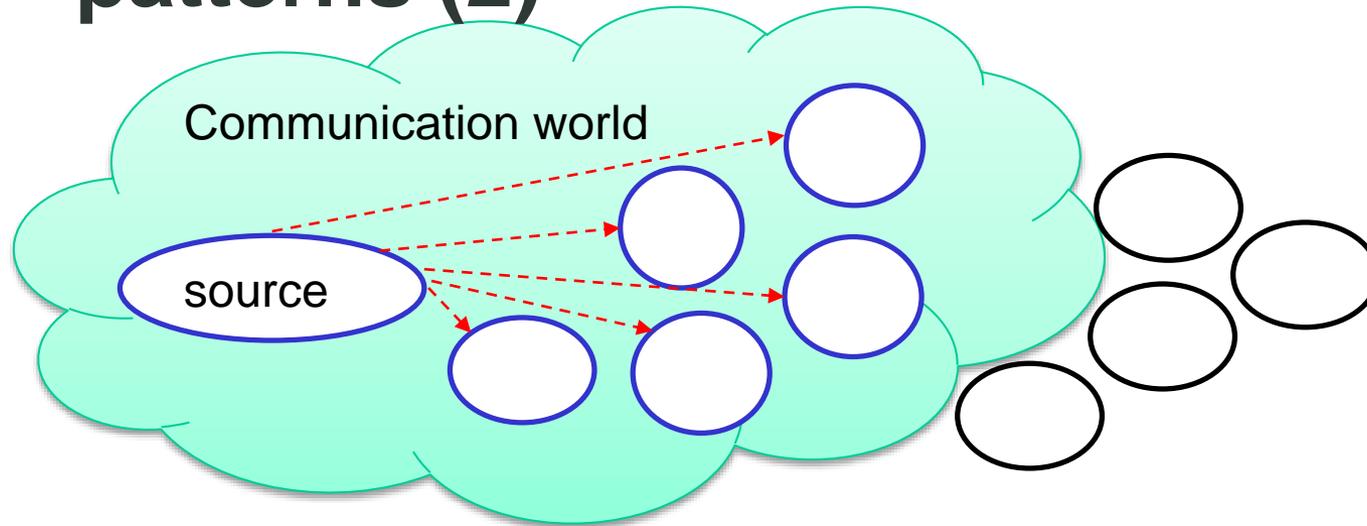
# Examples of communication patterns (1)

Service Location Protocol  
<http://tools.ietf.org/html/rfc2608>



- A User Agent wants to find a Service Agent
- Different roles and different communication patterns
- Get <http://jslp.sourceforge.net/> and play samples to see how it works

# Examples of communication patterns (2)



- MPI (Message Passing Interface)

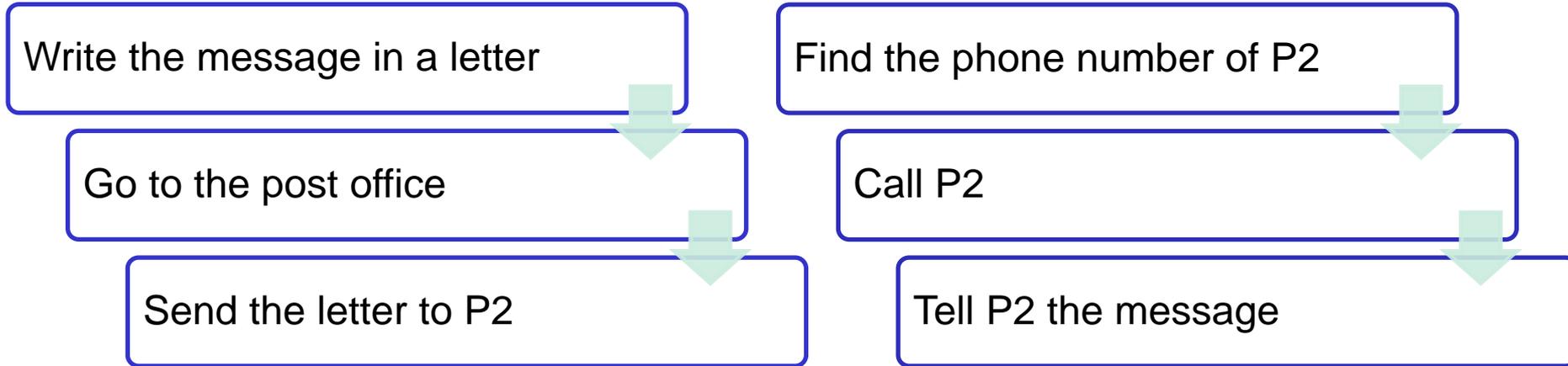
```
$sudo apt-get install mpich
$mpicc c_ex04.c
$mpirun -np 4 ./a.out
```

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
source=0;
count=4;
if(myid == source){
    for(i=0;i<count;i++)
        buffer[i]=i;
}
MPI_Bcast(buffer,count,MPI_INT,source,MPI_COMM_WORLD);
```

[http://geco.mines.edu/workshop/class2/examples/mpi/c\\_ex04.c](http://geco.mines.edu/workshop/class2/examples/mpi/c_ex04.c)

# Connection-oriented or connectionless communication

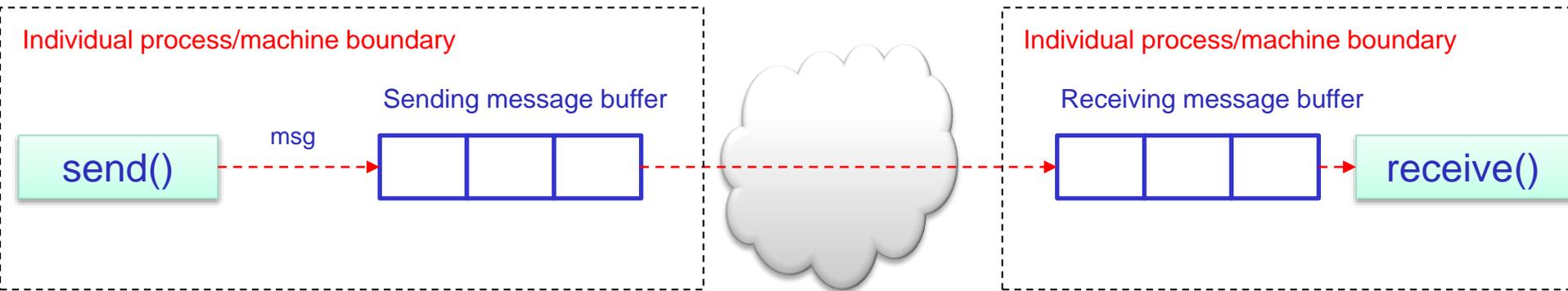
The message: „there is a party tonight“



Connection-oriented communication between **P1 and P2** requires the setup of communication connection between **them** first – no setup in connectionless communication

Q: What are the pros/cons of connection-oriented/connectionless communications? Is it possible to have a connectionless communication between (P1,P2) through some connection-oriented connections?

# Blocking versus non-blocking communication calls



**Send: transmitting a message is finished, it does not necessarily mean that the message reaches its final destination.**

- Blocking: the process execution is suspended until the message transmission finishes
- Non-blocking: the process execution continues without waiting until the finish of the message transmission

Q: Analyze the benefits of non-blocking communication. How non-blocking `receive()` works?

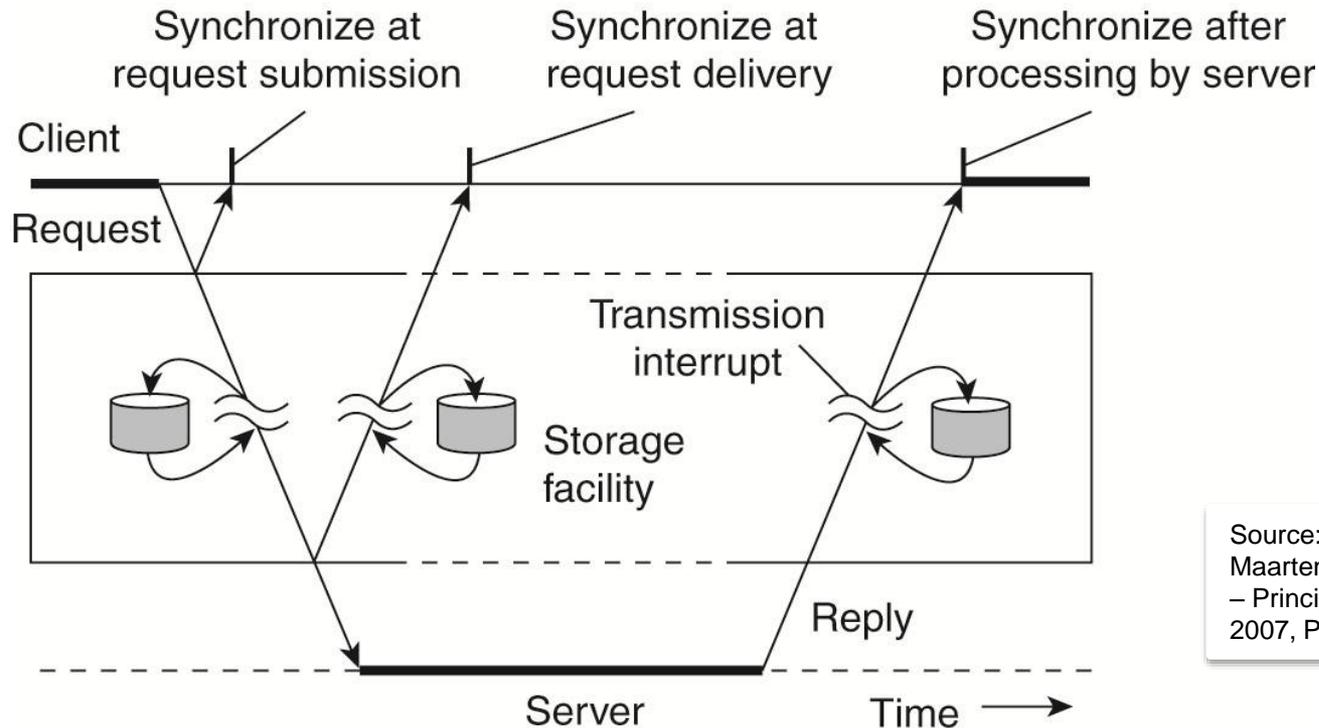
# Persistent and transient communication

- Persistent communication
  - **Messages are kept** in the communication system **until** they are delivered to the receiver
  - Often storage is needed
- Transient communication
  - **Messages are kept** in the communication temporary **only if** both the sender and receiver are live

# Asynchronous versus synchronous communication

- Asynchronous: continues after sending messages
  - Non blocking send
  - Receive may/may not be blocking
  - Callback mechanisms
- Synchronous: the sender **waits until it knows** the messages **delivered** to the receiver
  - Blocking send/blocking receive
  - Typically utilize connection-oriented and keep-alive connection
  - Blocking request-reply styles

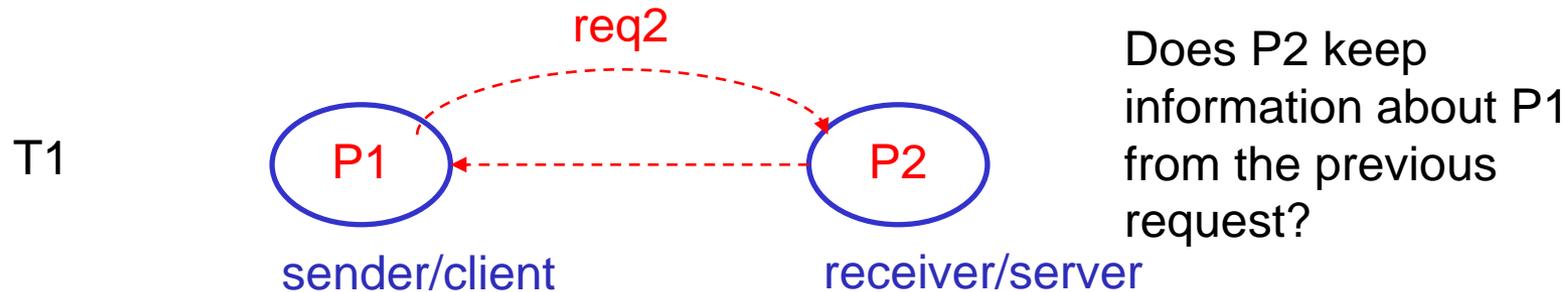
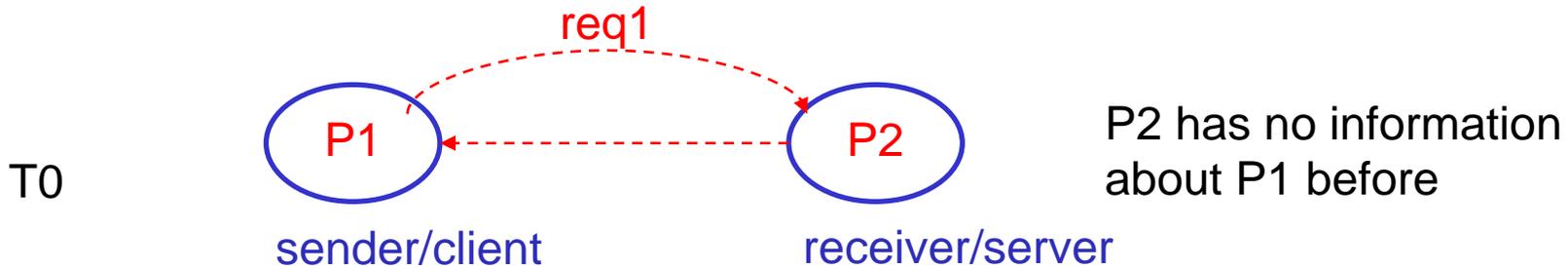
# Different forms of communication



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

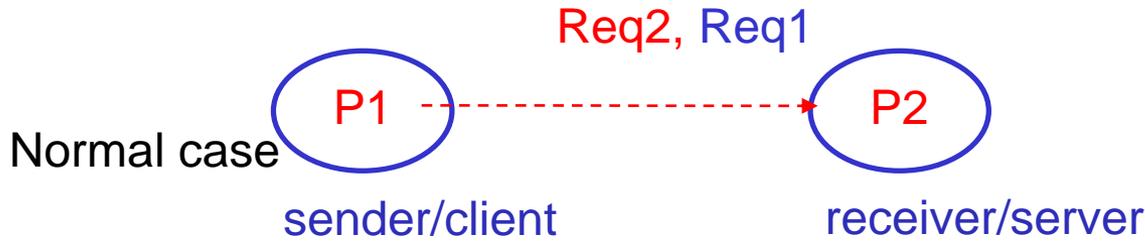
Q: How can we achieve the „persistence“? What are possible problems if a server sends a accepted/replied/ACK message before processing the request?

# Stateful versus Stateless Server

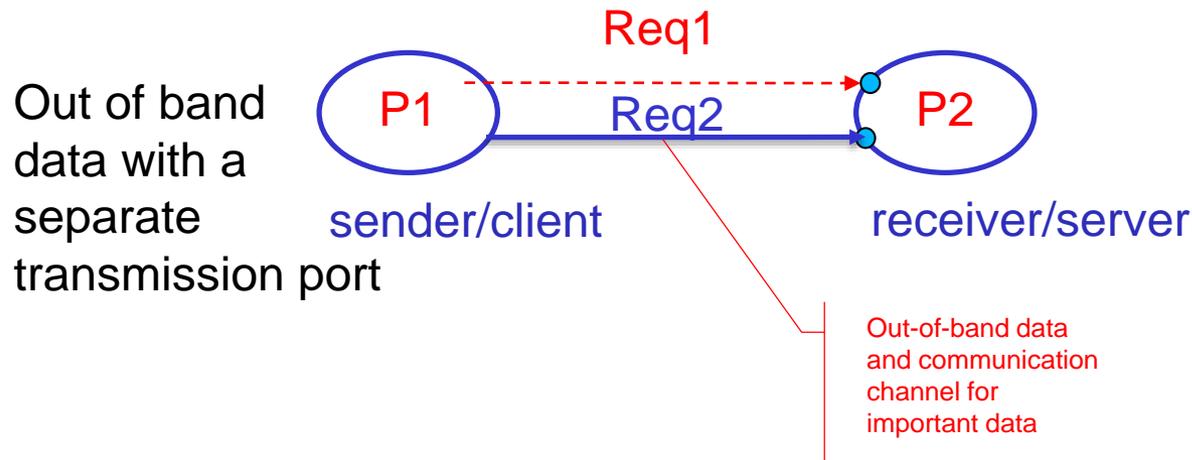


Stateless server	Soft State	Stateful Server
Does not keep client's state information	Keep some limited client's state information in a limited time	Maintain client's state information permanently

# Handling out of band data



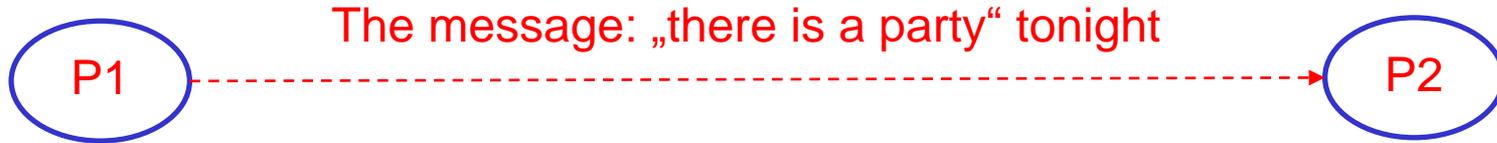
All messages come to P2 in the same port, no clear information about priority



Q: How can out-of-band data and normal data be handled by using the same transmission channel?

# COMMUNICATION PROTOCOLS

# Some key questions – Protocols

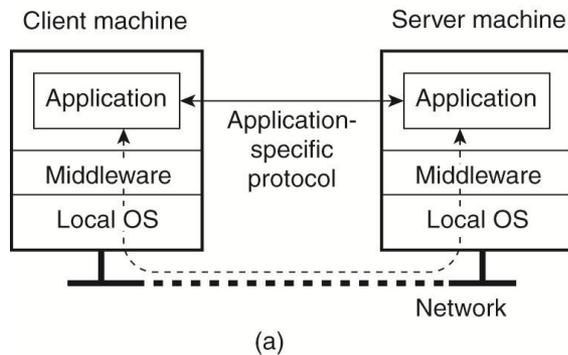


- **Communication patterns**
  - Can I use a single sending command to send the message to multiple people?
- **Identifier/Naming/Destination**
  - How do I identify the guys I need to send the message
- **Connection setup**
  - Can I send the message without setting up the connection
- **Message structure**
  - Can I use German or English to write the message
- **Layered communication**
  - Do I need other intermediators to relay the message?
- ...

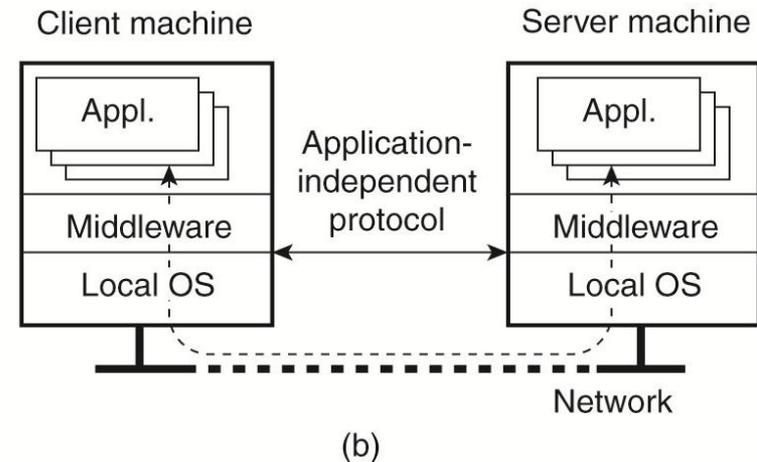
A communication protocol will describe rules addressing these issues

# Applications and Protocols

## Application-specific protocols



## Application-independent protocols

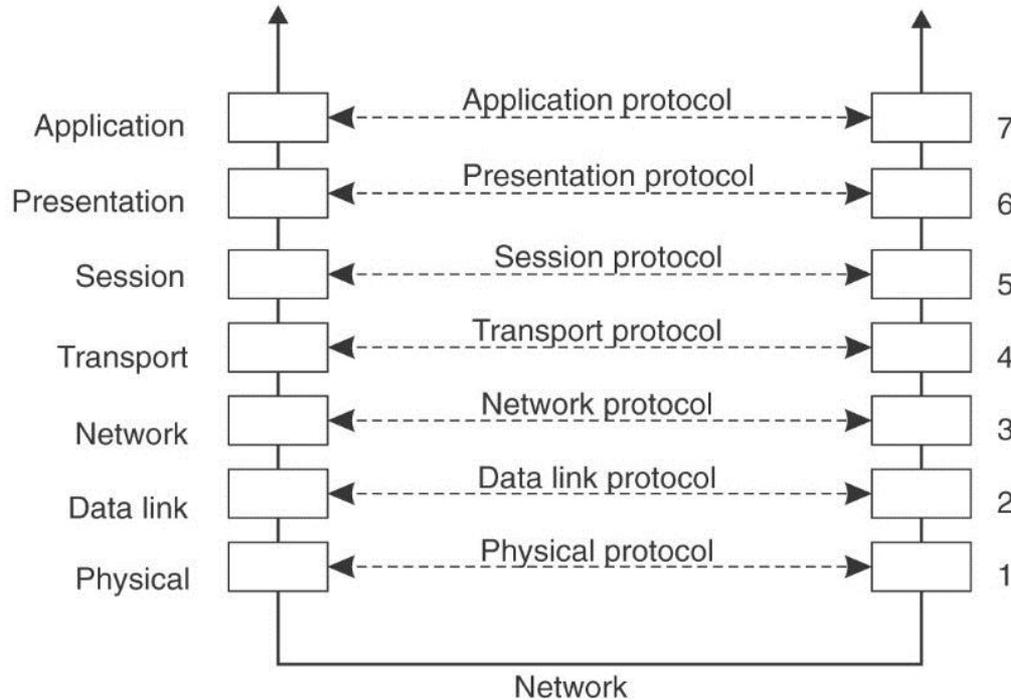


Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Layered Communication Protocols

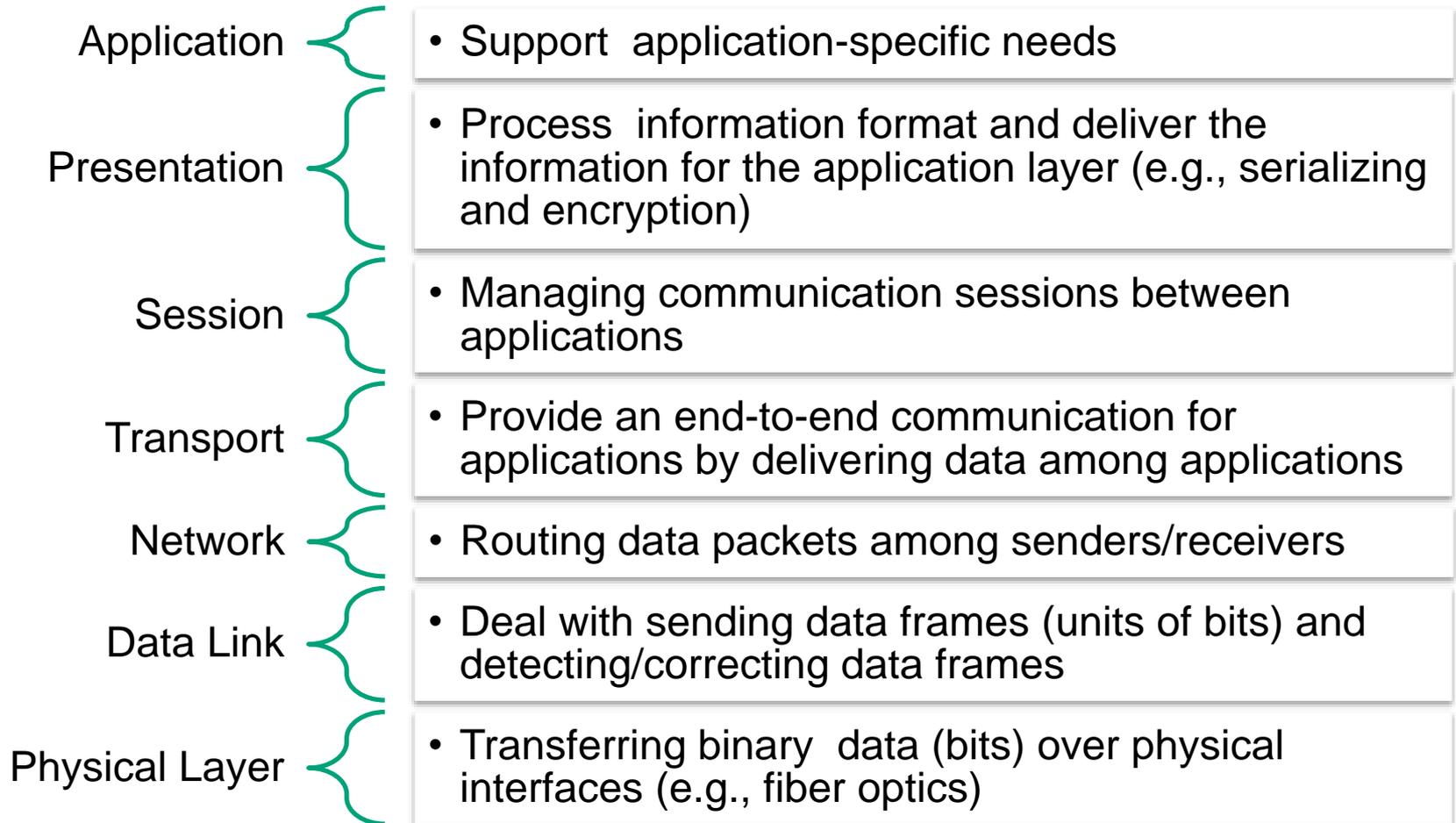
- Complex and open communication requires **multiple communication protocols**
- Communication protocols are typically organized into different layers: layered protocols/protocol stacks
- Conceptually: each layer has a set of different **protocols for certain communication functions**
  - Different protocols are designed for different environments/criteria
- **A protocol suite**: usually a set of protocols used together in a layered model

# OSI – Open Systems Interconnection **Reference** Model



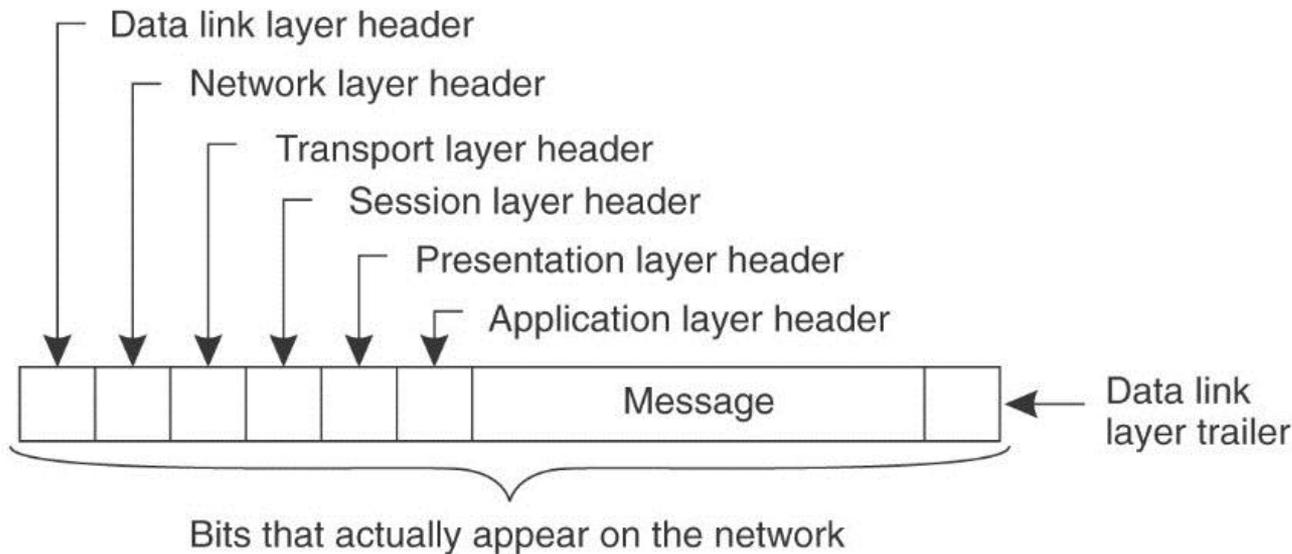
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# OSI Layers



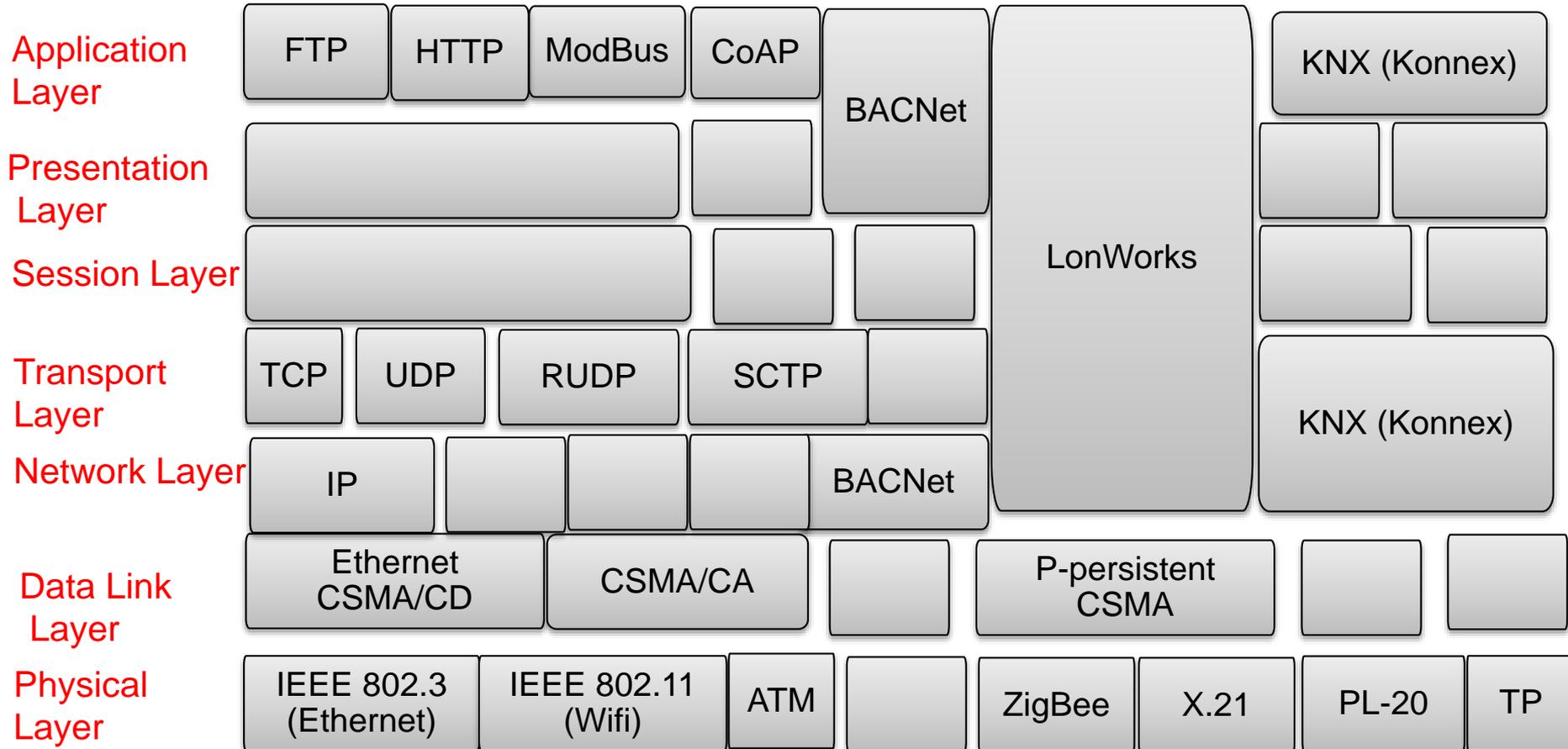
# How layered protocols work – message exchange

- Principles of constructing messages/data encapsulation

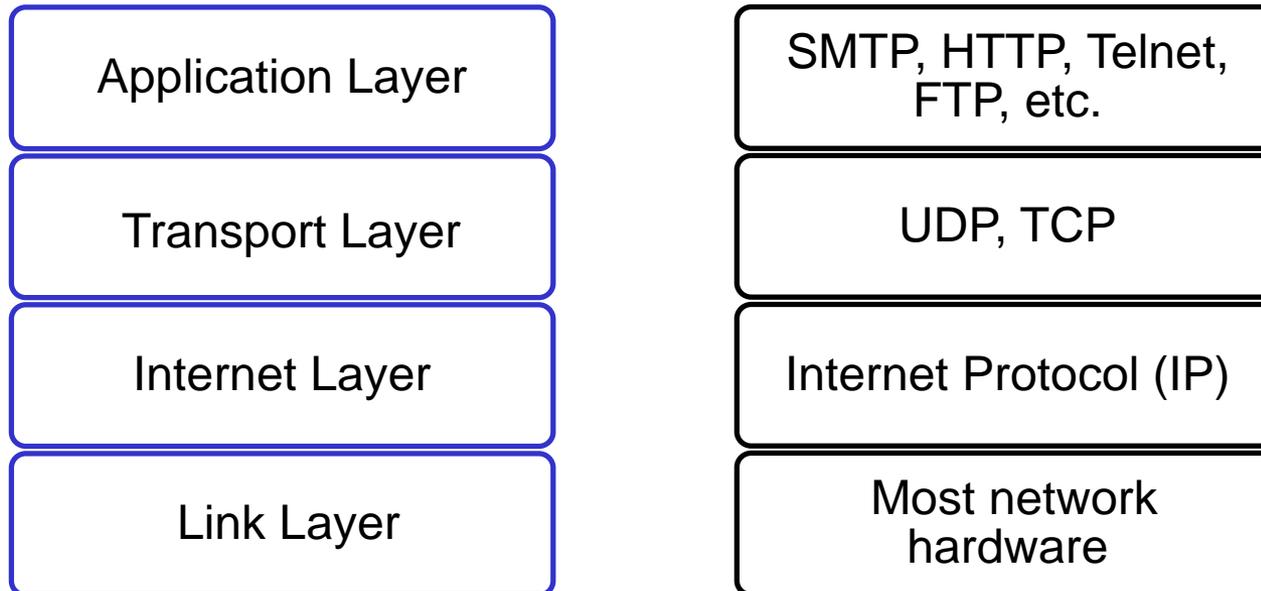


Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Examples of Layered Protocols



- The most popular protocol suite used in the Internet
- Four layers



<http://tools.ietf.org/html/rfc1122>

# Internet Protocol (IP)

- Define the datagram as the basic data unit
- Define the Internet address scheme
- Transmit data between the Network Access Layer and Transport Layer
- Route datagrams to destinations
- Divide and assemble datagrams

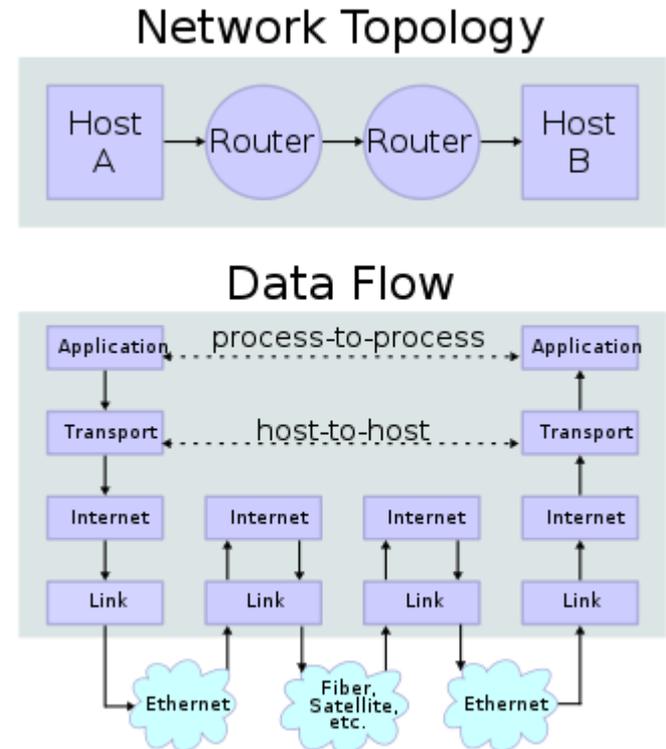


Figure source:  
[http://en.wikipedia.org/wiki/Internet\\_protocol\\_suite](http://en.wikipedia.org/wiki/Internet_protocol_suite)

# TCP/IP – Transport Layer

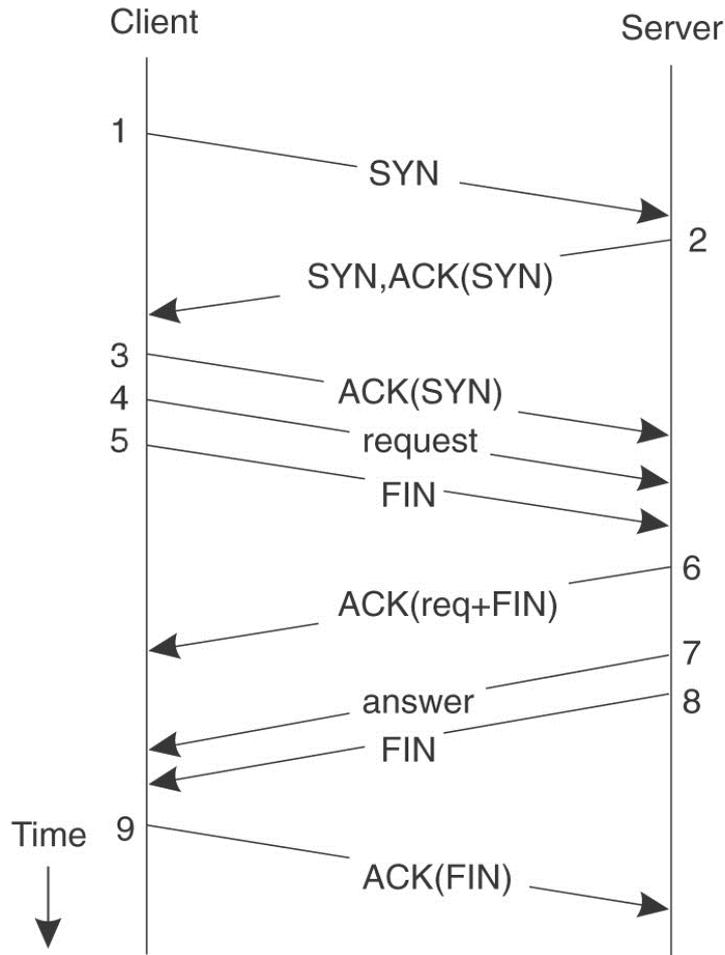
- Host-to-host transport features
- Two main protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol)

Layer\Protocol	TCP	UDP
Application layer	Data sent via Streams	Data sent in Messages
Transport Layer	Segment	Packet
Internet Layer	Datagram	Datagram
Link Layer	Frame	Frame

# TCP operations

```
$sudo nst -d -T iptest >ip.out
```

```
$wget www.tuwien.ac.at
```



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2002, Prentice-Hall, Inc.

```

--[ TCP ]-----
192.168.1.7:46023(unknown) -> 128.130.35.76:80(http)
TTL: 64 Window: 14600 Version: 4 Length: 60
FLAGS: -S----- SEQ: 3308581872 - ACK: 0
Packet Number: 16

--[ TCP ]-----
128.130.35.76:80(http) -> 192.168.1.7:46023(unknown)
TTL: 54 Window: 14480 Version: 4 Length: 60
FLAGS: -S-A-- SEQ: 3467332359 - ACK: 3308581873
Packet Number: 17

--[ TCP ]-----
192.168.1.7:46023(unknown) -> 128.130.35.76:80(http)
TTL: 64 Window: 115 Version: 4 Length: 52
FLAGS: ---A-- SEQ: 3308581873 - ACK: 3467332360
Packet Number: 18

--[ TCP ]-----
192.168.1.7:46023(unknown) -> 128.130.35.76:80(http)
TTL: 64 Window: 115 Version: 4 Length: 166
FLAGS: ---PA-- SEQ: 3308581873 - ACK: 3467332360
Packet Number: 19

--[ TCP Data ]-----
GET / HTTP/1.1

--[ TCP ]-----
128.130.35.76:80(http) -> 192.168.1.7:46023(unknown)
TTL: 54 Window: 114 Version: 4 Length: 52
FLAGS: ---A-- SEQ: 3467332360 - ACK: 3308581987
Packet Number: 20

--[ TCP ]-----
128.130.35.76:80(http) -> 192.168.1.7:46023(unknown)
TTL: 54 Window: 114 Version: 4 Length: 1500
FLAGS: ---A-- SEQ: 3467332360 - ACK: 3308581987
Packet Number: 21

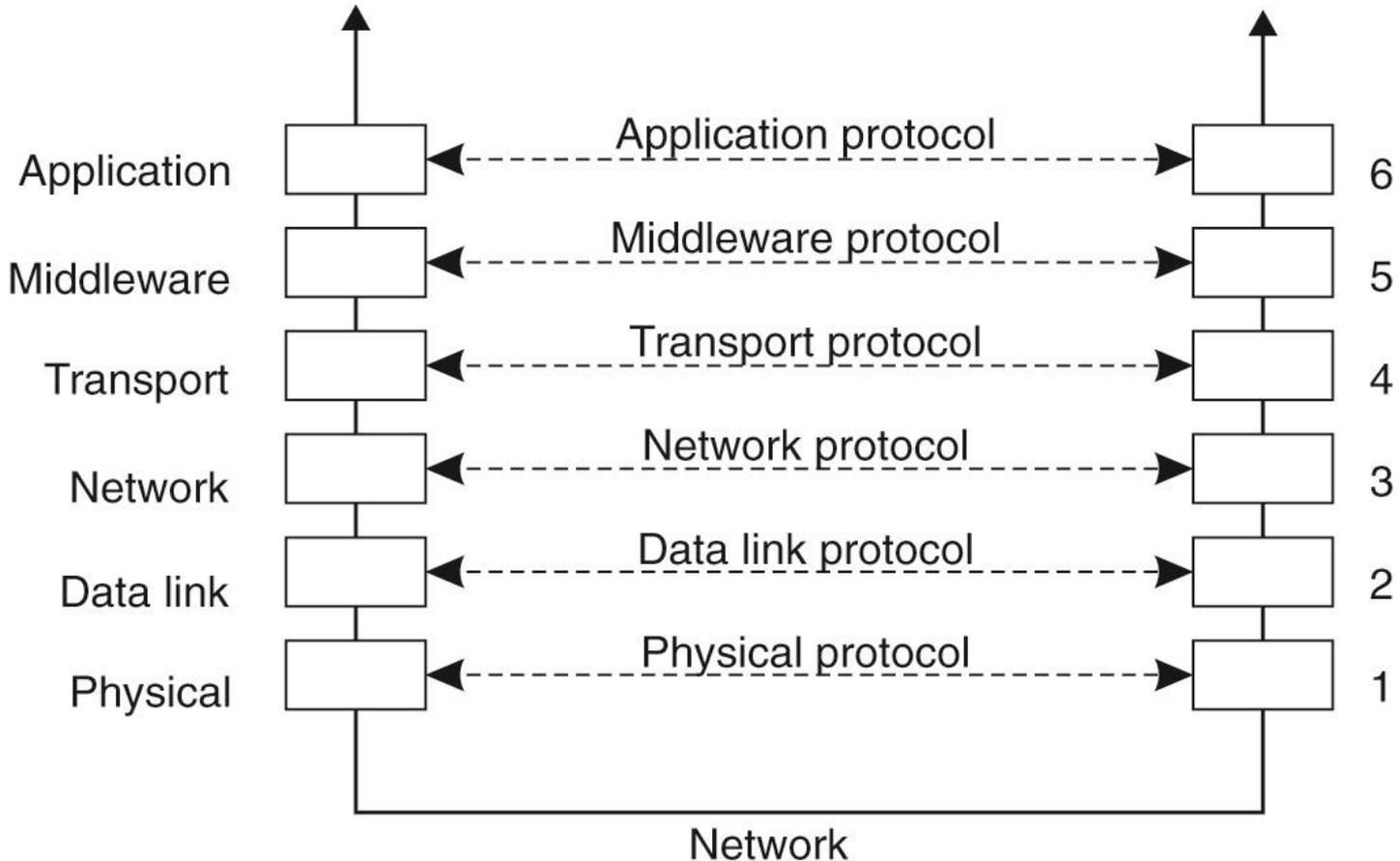
--[ TCP Data ]-----
HTTP/1.1 200 OK

```

# Communication protocols are not enough

- We need more than just communication protocols
  - E.g., resolving names, electing a communication coordinator, locking resources, and synchronizing time
- Middleware
  - Including a set of general-purpose but application-specific protocols, middleware communication protocols, and other specific services.

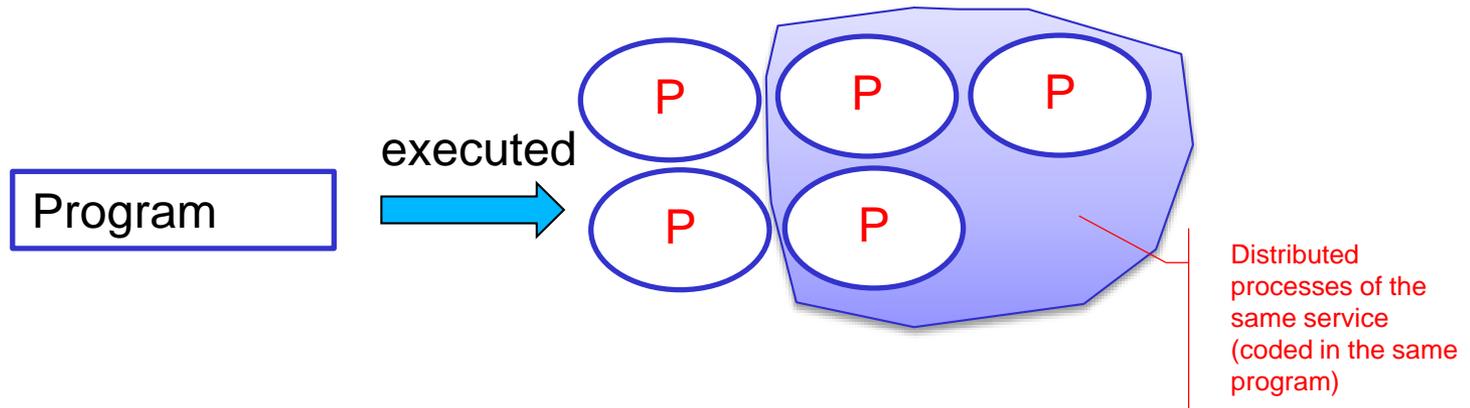
# Middleware Protocols



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

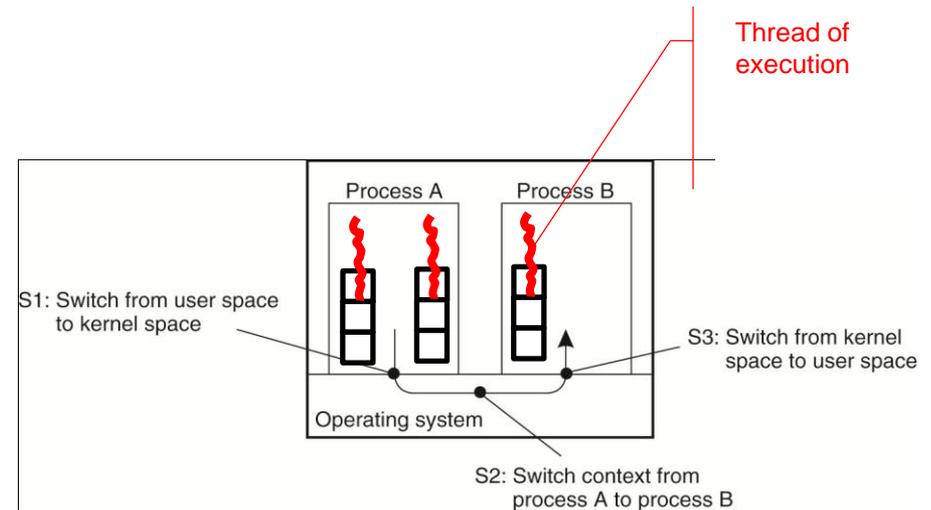
# HANDLING COMMUNICATION MESSAGES/REQUESTS

# Process versus thread



## Within a non distributed OS

- Process – the program being executed by the OS
- Threads within a process
- Switching thread context is much cheaper than that for the process context
- Blocking calls in a thread do not block the whole process



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Where communication tasks take place?

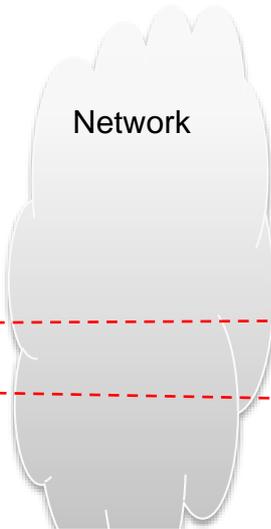
- Message passing – send/receive
  - Processes **send** and **receive** messages
    - Sending process versus receiving process
    - Communication is done by using **a set of functions for communication implementing protocols**
- Remote method/procedure calls
  - A process **calls/invokes a (remote) procedure** in another process
    - Local versus remote procedure call, but in the same manner
- Remote object calls
  - A process **calls/invokes a (remote) object** in another process

# Basic send/receive communication

```
# Echo client program
import socket
```

```
HOST = 'daring.cwi.nl' # The remote host
PORT = 50007           # The same port as
                        # used by the server
s = socket.socket(socket.AF_INET,
                  socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024) ←
s.close()
print 'Received', repr(data)
```

Network



```
# Echo server program
import socket
```

```
HOST = "              # Symbolic name meaning the
                        # local host
PORT = 50007          # Arbitrary non-privileged
                        # port
s = socket.socket(socket.AF_INET,
                  socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    → data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

Python source: <http://docs.python.org/release/2.5.2/lib/socket-example.html>

# Remote procedure calls

```

void
hello_prog_1(char *host)
{
    CLIENT *clnt;
    char **result_1;
    char *hello_1_arg;

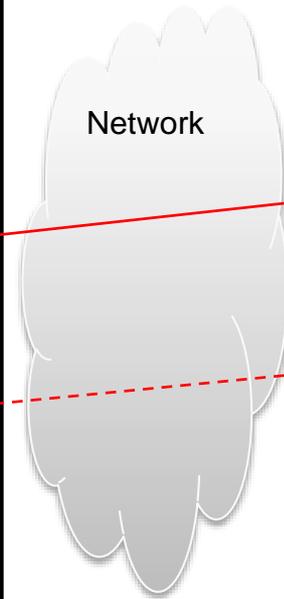
#ifdef DEBUG
    clnt = clnt_create (host, HELLO_PROG, HELLO_VERS, "udp");
    if (clnt == NULL) {
        clnt_pcreateerror (host);
        exit (1);
    }
#endif /* DEBUG */

    result_1 = hello_1((void*)&hello_1_arg, clnt);
    if (result_1 == (char **) NULL) {
        clnt_perror (clnt, "call failed");
    }
#ifdef DEBUG
    clnt_destroy (clnt);
#endif /* DEBUG */
    printf("result is: %s\n",(*result_1));
}

int
main (int argc, char *argv[])
{
    char *host;

    if (argc < 2) {
        printf ("usage: %s server_host\n", argv[0]);
        exit (1);
    }
    host = argv[1];
    hello_prog_1 (host);
    exit (0);
}

```



Procedure in a remote server

```

char **
hello_1_svc(void *argp, struct svc_req *rqstp)
{
    static char * result ="Hello";

    /*
     * insert server code here
     */

    return &result;
}

```

# Remote object calls

## Objects in a remote server

```

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}

```

```

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
....
public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
        }
    }
}

```

Java source:

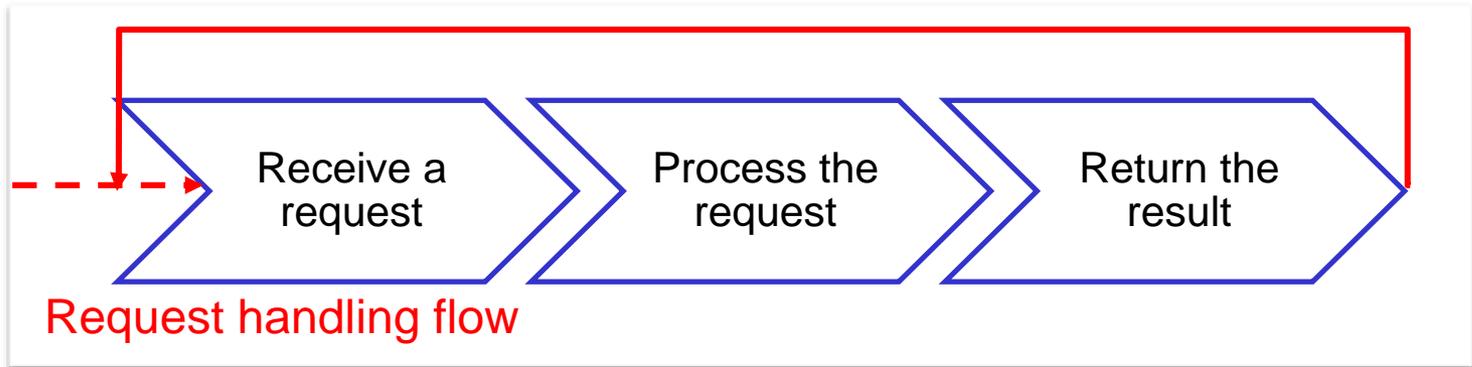
<http://docs.oracle.com/javase/tutorial/rmi/overview.html>

# Processing multiple requests

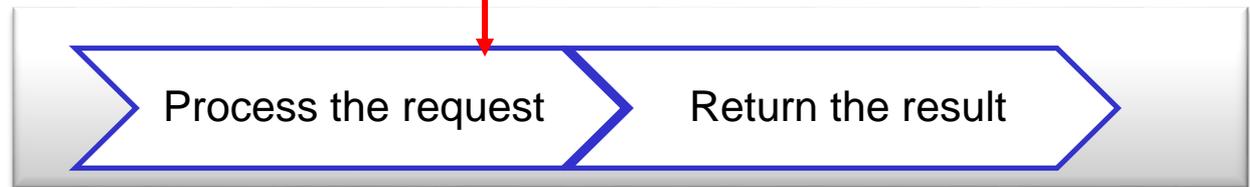
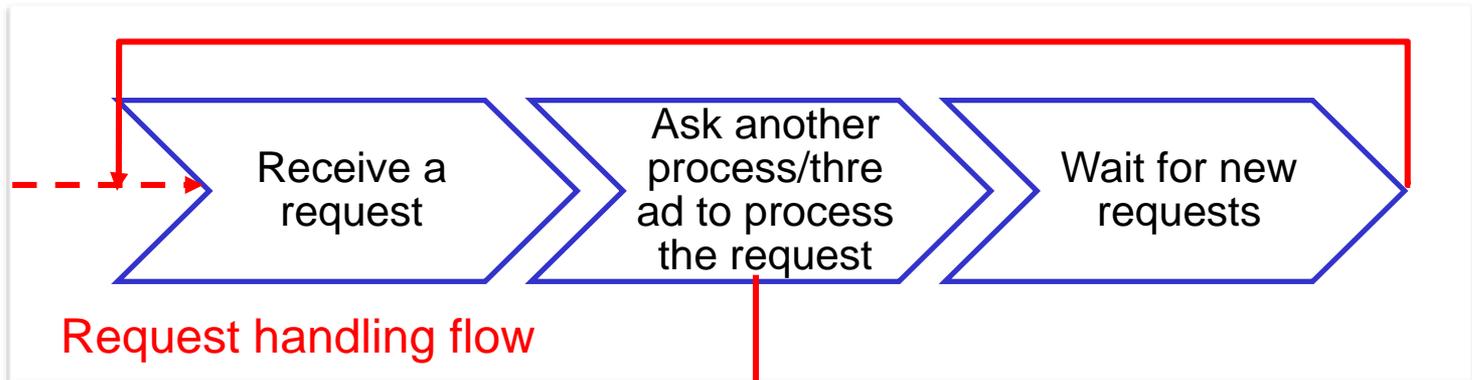
- How to deal with multiple, concurrent messages received?
- Problems:
  - Different roles: clients versus servers/services
    - A large **number of clients** interact with **a small number of servers/services**
    - A single process might receive a lot of messages at the same time
- Impacts
  - performance, reliability, cost, etc.

# Iterative versus concurrent processing

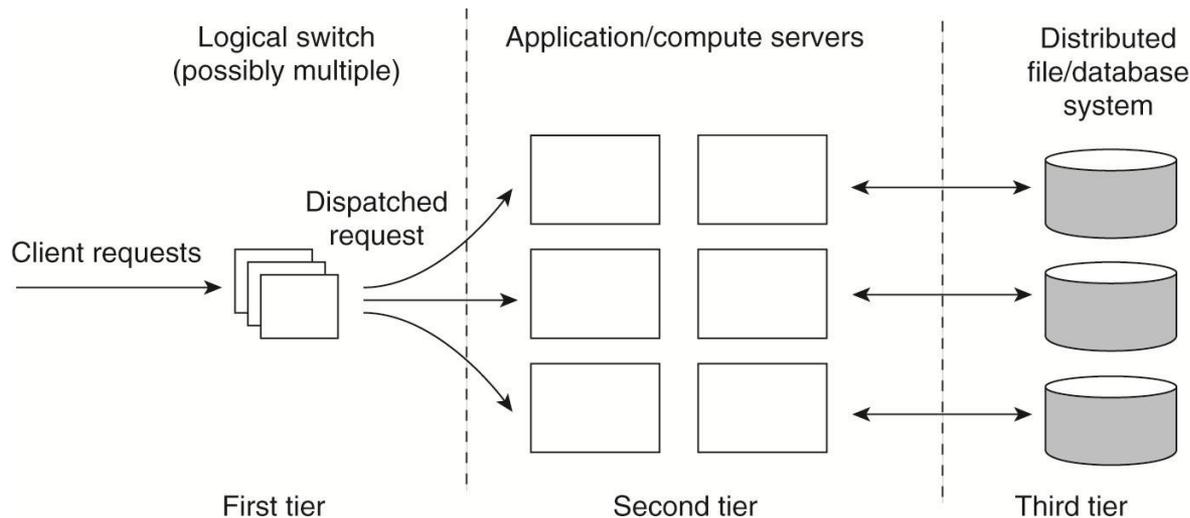
Iterative processing



Concurrent processing



# Using replicated processes

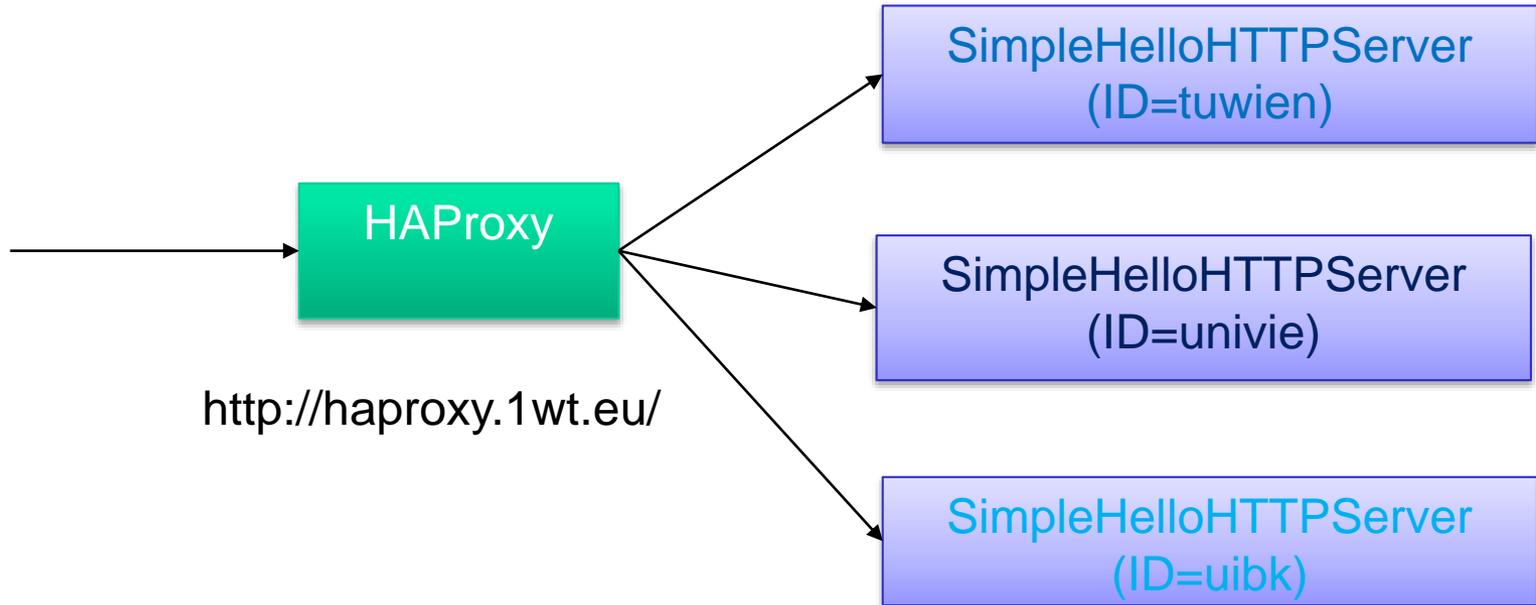


Often loadbalancing mechanisms are needed

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

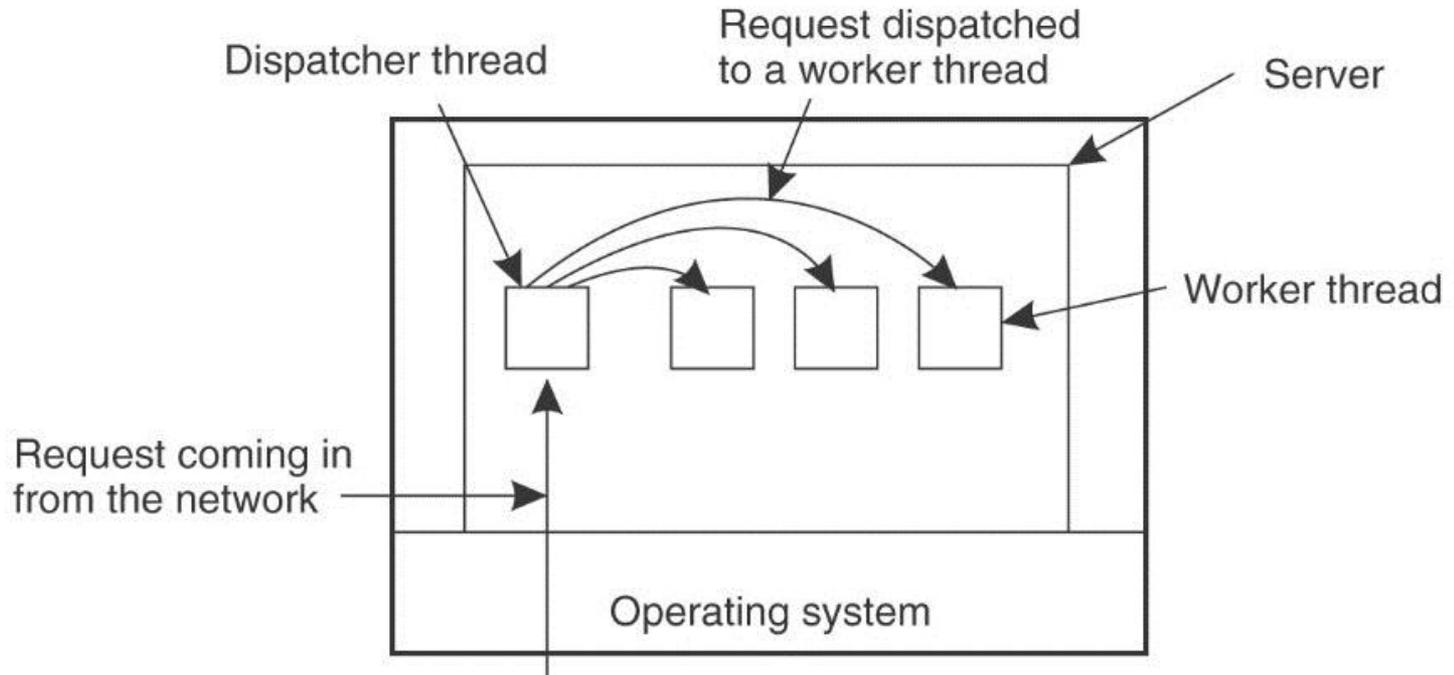
Q: How this model helps to improve performance and fault-tolerance? What would be a possible mechanism to reduce costs based on the number of client requests?

# Example



- Get a small test
  - Download haproxy, e.g.  
\$sudo apt-get install haproxy
  - Download SimpleHelloHTTPServer.java and haproxy configuration
    - <http://bit.ly/19xFDRC>
  - Run 1 haproxy instance and 3 http servers
    - Modify configuration and parameters if needed
  - Run a test client

# Using multiple threads



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Q: How this architectural model would be applied/similar to worker processes or the super-server model?



# Example

- Get a free instance of RabbitMQ from [cloudamqp.com](https://cloudamqp.com)
- Get code from: <https://github.com/cloudamqp/java-amqp-example>
- First run the test sender, then run the receiver



```
channel.queueDeclare(QueueName, false, false, false, null);
for (int i=0; i<100; i++) {
    String message = "Hello distributed systems guys: " + i;
    channel.basicPublish("", QueueName, null, message.getBytes());
    System.out.println(" [x] Sent " + message + "");
    new Thread().sleep(5000);
}
```

```
while (true) {
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();
    String message = new String(delivery.getBody());
    System.out.println(" [x] Received " + message + "");
}
```

**Note: i modified the code a bit**

# Summary

- Complex and diverse communication patterns, protocols and processing models
- Choices are based on communication requirements and underlying networks
  - Understand their pros/cons
  - Understand pros and cons of their technological implementations
- Dont forget to play some simple examples to understand existing concepts

# Thanks for your attention

Hong-Linh Truong  
Distributed Systems Group  
Vienna University of Technology  
[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
<http://dsg.tuwien.ac.at/staff/truong>

# Communication in Distributed Systems – Programming

Hong-Linh Truong  
Distributed Systems Group,  
Vienna University of Technology

[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
[dsg.tuwien.ac.at/staff/truong](http://dsg.tuwien.ac.at/staff/truong)

# Learning Materials

- Main reading:
  - Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall
    - Chapters 3 & 4
- Others
  - George Coulouris, Jean Dollimore, Tim Kindberg, „Distributed Systems – Concepts and Design“, 2nd Edition
    - Chapter 5.
  - Sukumar Ghosh, “Distributed Systems: An Algorithmic Approach”, Chapman and Hall/CRC, 2007
    - Chapter 15
  - Papers referred in the lecture
- Test the examples in the lecture

- Recall
- Message-oriented Transient Communication
- Message-oriented Persistent Communication
- Remote Procedure Call
- Streaming data programming
- Group communication
- Gossip-based Data Dissemination
- Summary

- One-to-one versus group communication
- Transient communication versus persistent communication
- Message transmission versus procedure call versus object method calls
- Physical versus overlay network

# MESSAGE-ORIENTED TRANSIENT COMMUNICATION

# Message-oriented Transient Communication at Transport Layer

- How an application uses the transport layer communication to send/receive messages?

Transport-level socket programming via socket interface

- Socket interface – Socket APIs
  - Very popular, supported in almost all programming languages and operating systems
  - Berkeley Sockets (BSD Sockets)
    - Java Socket, Windows Sockets API/WinSock, etc.



# Message-oriented Transient Communication at Transport Level (2)

**What is a socket:** a **communication end point** to/from which an application can send/receive data through the underlying network.

- Client
  - Connect, send and then receive data through sockets
- Server:
  - Bind, listen/accept, receive incoming data, process the data, and send back to the client the result

Q: Which types of information are used to describe the identifier of the “end point”?



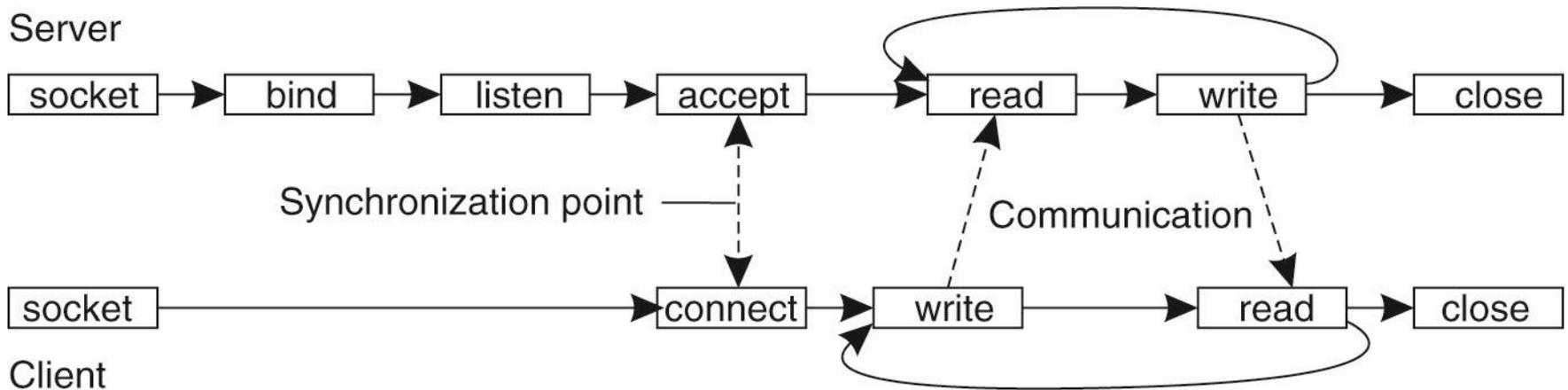
# Socket Primitives

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Client-server interactions

## Connection-oriented communication interaction



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

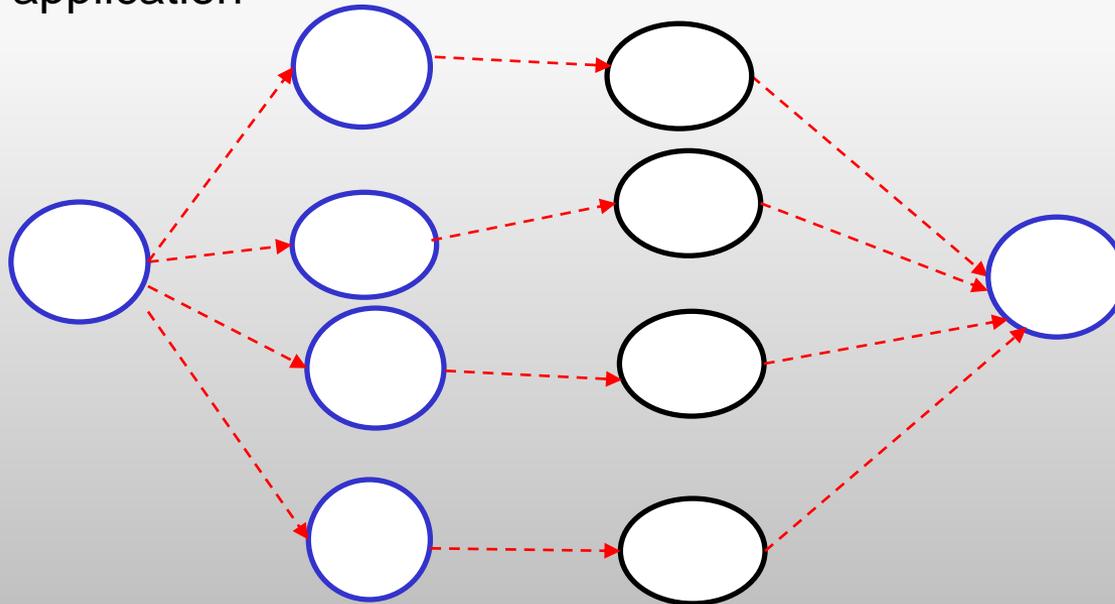
- Q1: How to implement a multi-threaded server?  
 Q2: What if connect() happens before listen()/accept()?

# Example

- Simple echo service
  - Client sends a message to a server
  - Server returns the message
- Source code:  
<http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/exs/socket-ex.tar.gz>

# Message-oriented Transient Communication at the Application level

Complex communication, large-scale number processes in the same application



Why transport level socket programming primitives are not good enough?

# Message-passing Interface (MPI)

- Designed for parallel processing: <http://www.mpi-forum.org/>
  - Well supported in clusters and high performance computing systems
  - One-to-one/group and synchronous/asynchronous communication
- Basic MPI concepts
    - **Communicators/groups** to determine a set of processes that can be communicated: MPI\_COMM\_WORLD represents all mpi processes
    - **Rank**: a unique identifier of a process
    - A set of functions to **manage the execution environment**
    - **Point-to-point communication functions**
    - **Collective communication functions**
    - **Functions handling data types**

# Message-passing Interface (MPI)

Function	Description
MPI_Init	Initialize the MPI execution environment
MPI_Comm_size	Determine the size of the group given a communicator
MPI_Comm_rank	Determine the rank of the calling process in group
MPI_Send()	Send a message, blocking mode
MPI_Recv()	Receive a message, blocking mode
...	
MPI_Bcast()	Broadcast a message from a process to others
MPI_Reduce()	Reduce all values from all processes to a single value
...	
MPI_Finalize()	Terminate the MPI execution environment

# Example

```

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if(myid == 0) {
    printf("I am %d: We have %d processors\n", myid,
        numprocs);
    sprintf(output, "This is a message sending from %d",
        i);
    for(i=1;i<numprocs;i++)
        MPI_Send(output, 80, MPI_CHAR, i, 0,
            MPI_COMM_WORLD);
}
else {
    MPI_Recv(output, 80, MPI_CHAR, i, 0,
        MPI_COMM_WORLD, &status);
    printf("I am %d and I receive: %s\n", myid, output);
}

```

```

source=0;
count=4;
if(myid == source){
    for(i=0;i<count;i++)
        buffer[i]=i;
}

    MPI_Bcast(buffer,count,MPI_INT,source,MPI_COM
M_WORLD);
for(i=0;i<count;i++) {
    printf("I am %d and I receive: %d \n",myid, buffer[i]);
}
printf("\n");
MPI_Finalize();

```

Code: <http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/exs/mpi-ex.tar.gz>

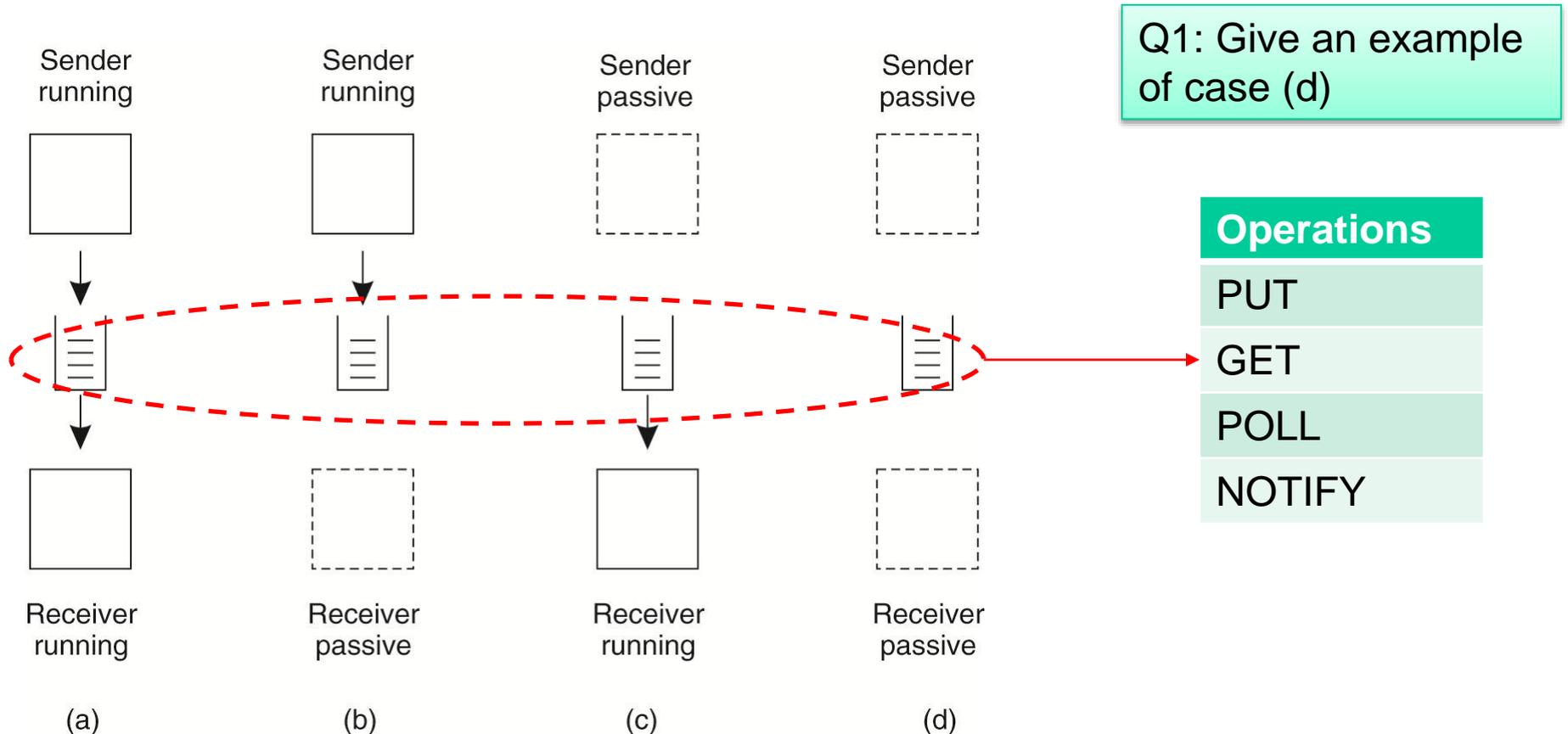
# MESSAGE-ORIENTED PERSISTENT COMMUNICATION

# Message-oriented Persistent Communication – Queuing Model

- Message-queuing systems or Message-Oriented Middleware (MOM)
- Well-supported in large-scale systems for
  - Persistent but asynchronous messages
  - Scalable message handling
- Several Implementations

# Message-oriented Persistent Communication – Queuing Model

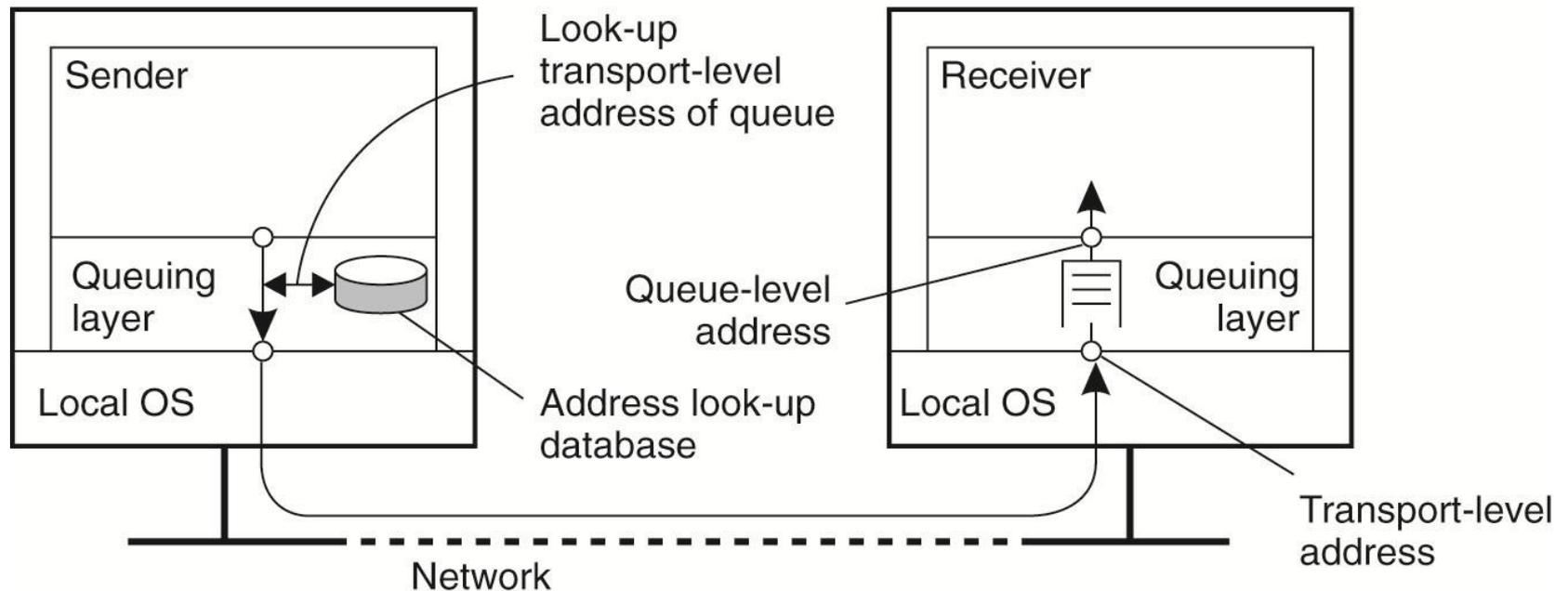
## Communication models



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall



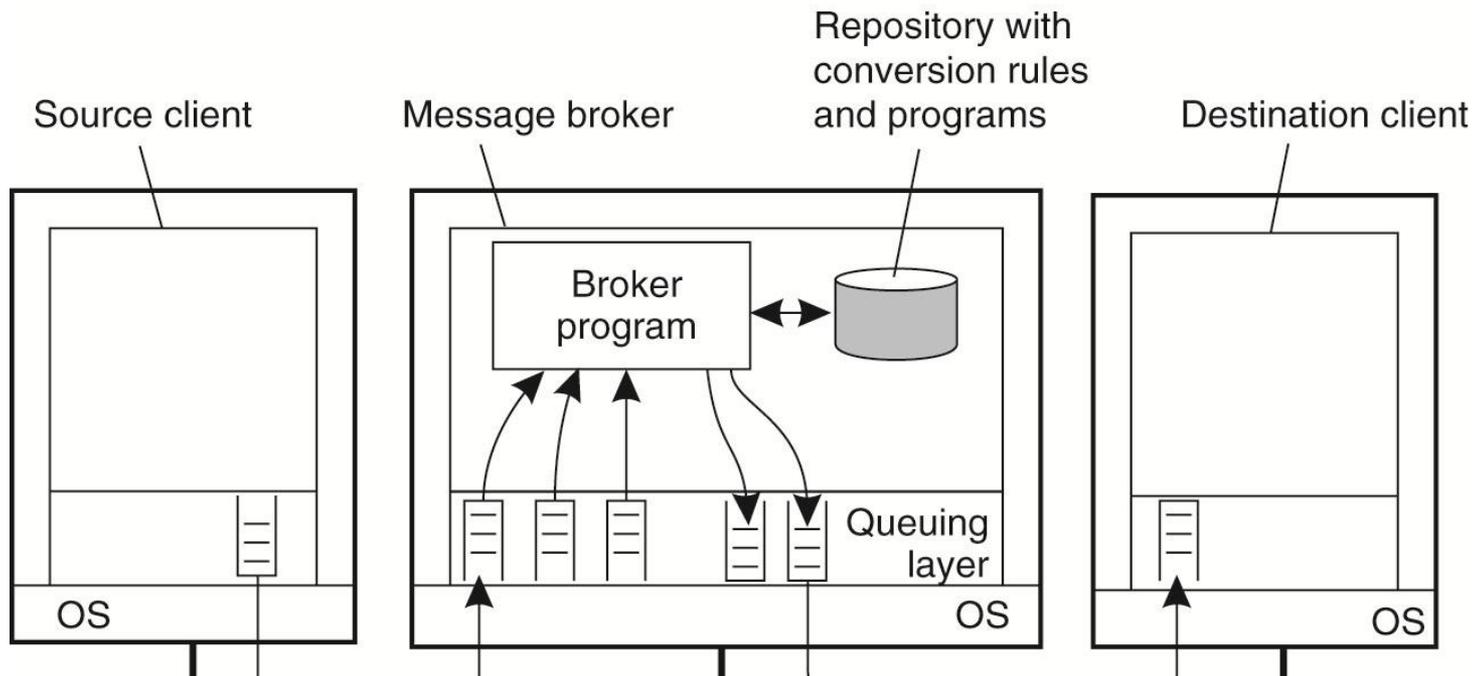
# Message-oriented Persistent Communication – Queuing Model



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Message Brokers

- **Publish/Subscribe**: messages are matched to applications
- **Transform**: messages are transformed from one format to another one suitable for specific applications



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Network

# Example – Advanced Message Queuing Protocol (AMQP)

- <http://www.amqp.org>



Apache Qpid™



# Example: AMQP

```

ConnectionFactory factory = new ConnectionFactory();
factory.setUri(uri);
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();

channel.queueDeclare(QUEUE_NAME, false, false, false, null);
for (int i=0; i<100; i++) {
    String message = "Hello distributed systems guys: " + i;
    channel.basicPublish("", QUEUE_NAME, null,
        message.getBytes());

    System.out.println(" [x] Sent " + message + "");
    new Thread().sleep(5000);
}

channel.close();
connection.close();

```

Source code:

<https://github.com/cloudamqp/java-amqp-example>, see also the demo in the lecture 2

```

ConnectionFactory factory = new ConnectionFactory();
factory.setUri(uri);
Connection connection = factory.newConnection();
Channel channel = connection.createChannel();

channel.queueDeclare(QUEUE_NAME, false, false,
    false, null);

System.out.println(" [*] Waiting for messages");

QueueingConsumer consumer = new
    QueueingConsumer(channel);

channel.basicConsume(QUEUE_NAME, true,
    consumer);

while (true) {
    QueueingConsumer.Delivery delivery =
        consumer.nextDelivery();

    String message = new String(delivery.getBody());
    System.out.println(" [x] Received " + message + "");
}

```

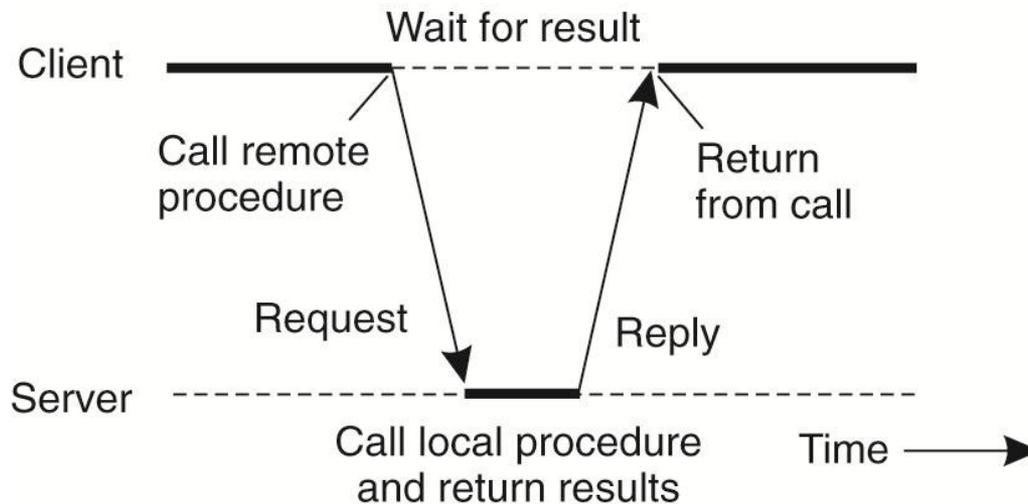


# REMOTE PROCEDURE CALL

# Remote Procedure Call

How can we call **a procedure in a remote process** in a similar way to a **local** procedure?

Remote Procedure Call (RPC): hide all complexity in calling remote procedures



- Well support in many systems and programming languages

Q1: Which types of applications are suitable for RPC?

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Message format and data structure description

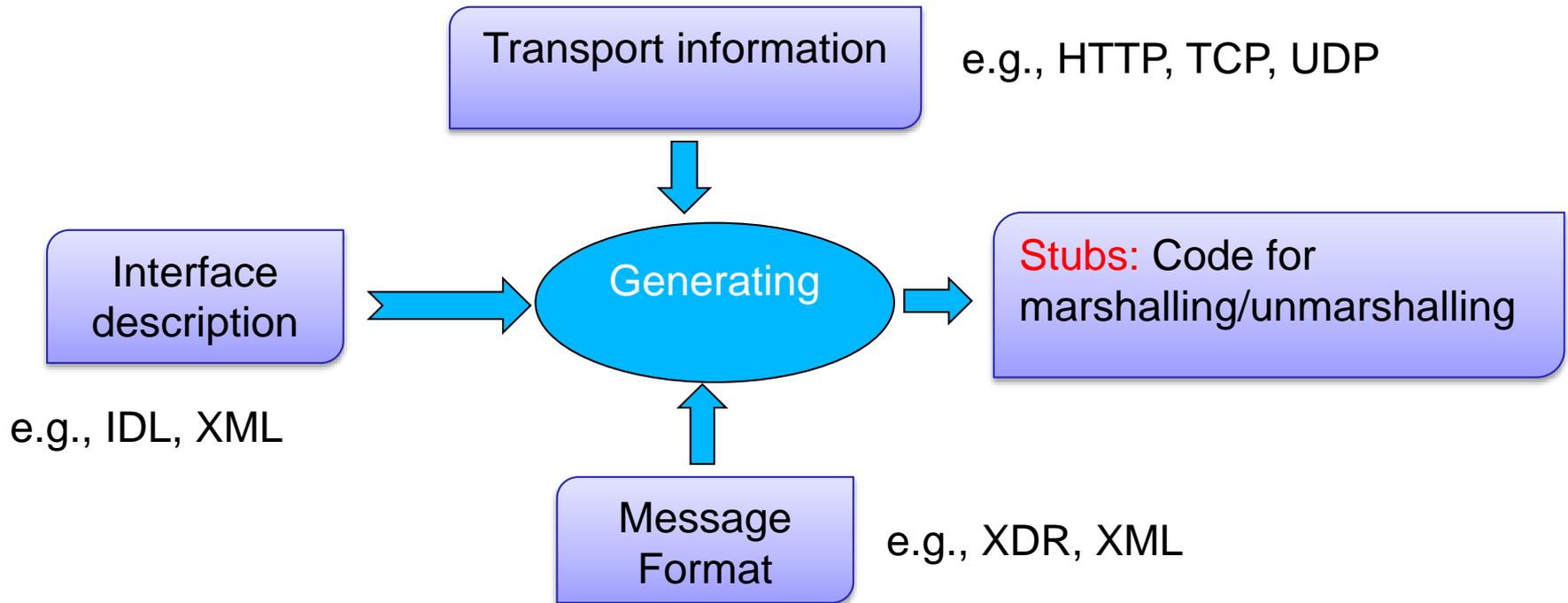
- Passing parameters and results needs **agreed message format** between a client and a server

**Marshaling/unmarshaling** describes the process packing/unpacking parameters into/from messages (note: **encoding/decoding** are also the terms used)

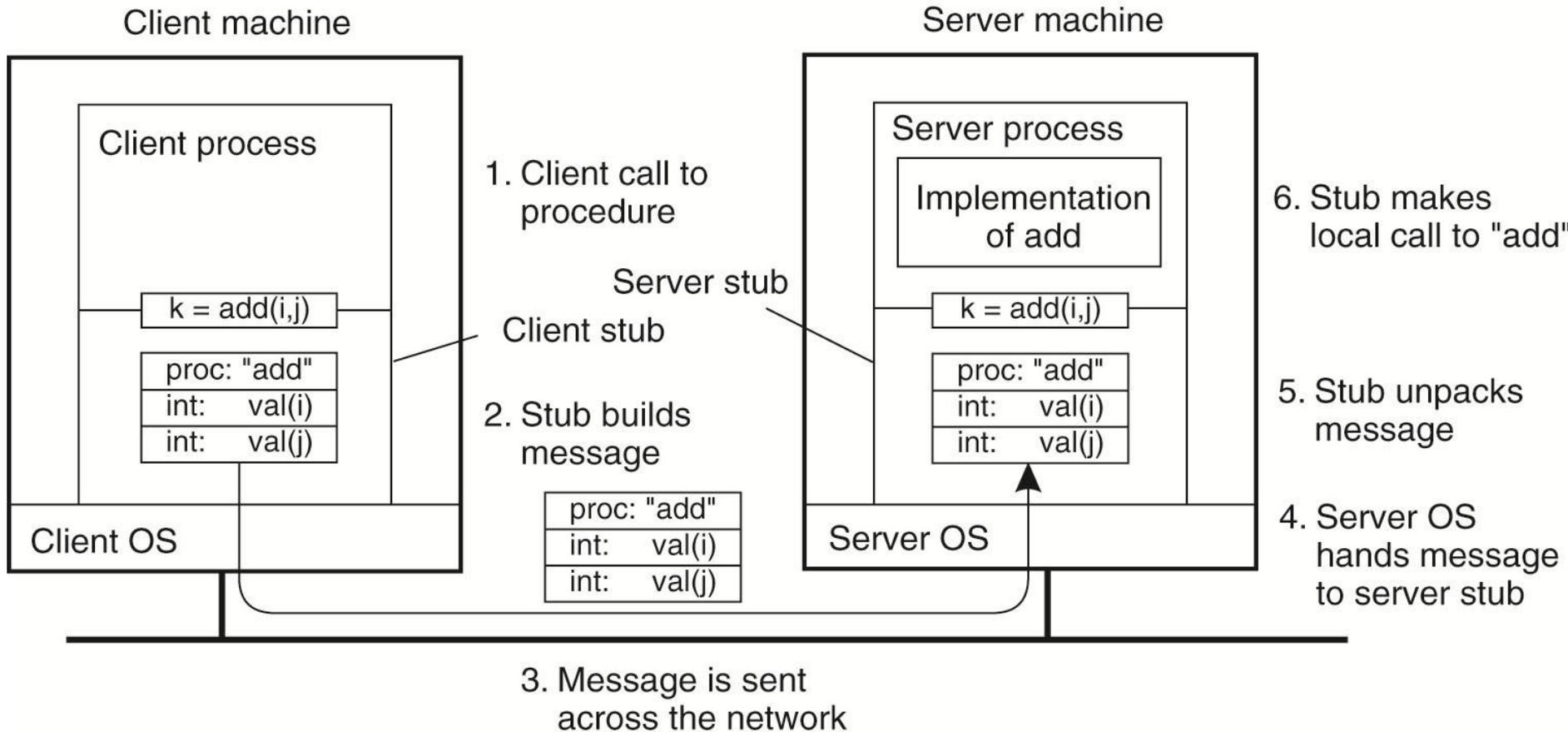
- Data types may have **different representations** due to different machine types (e.., SPARC versus Intel x86)

Interface languages can be used to describe the common interfaces between clients and server

# Generating stubs

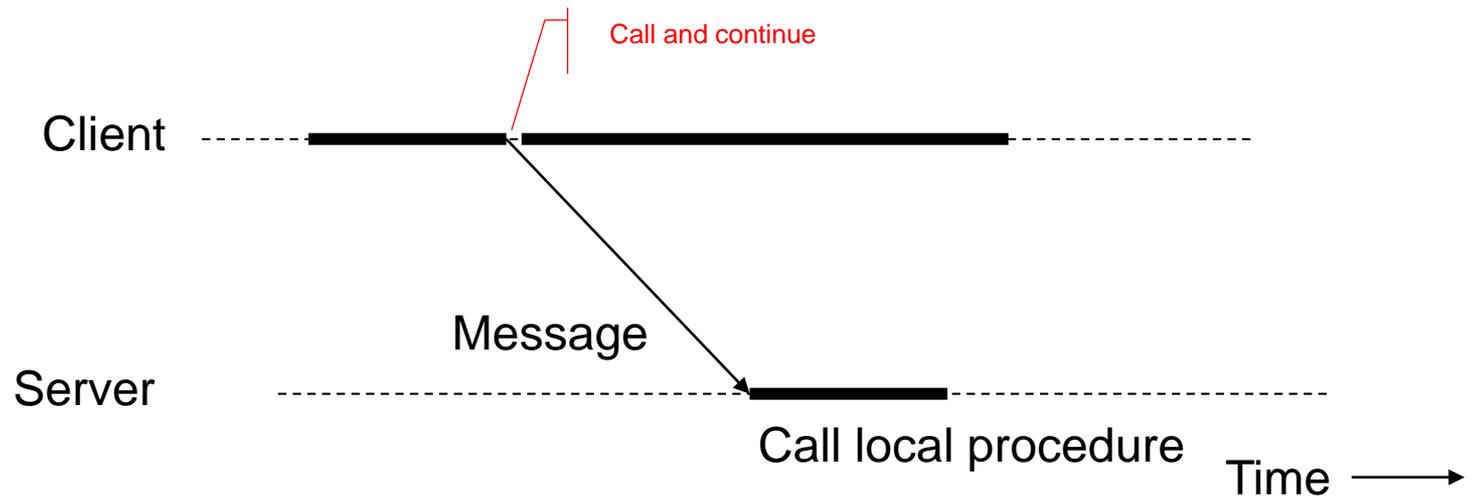


# Detailed Interactions



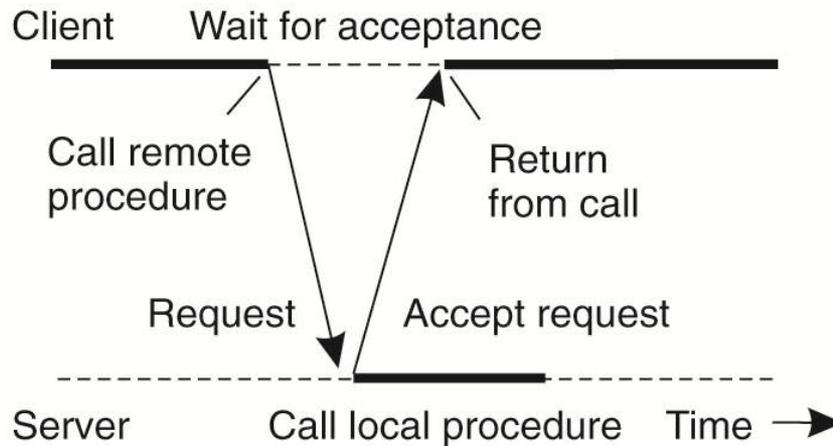
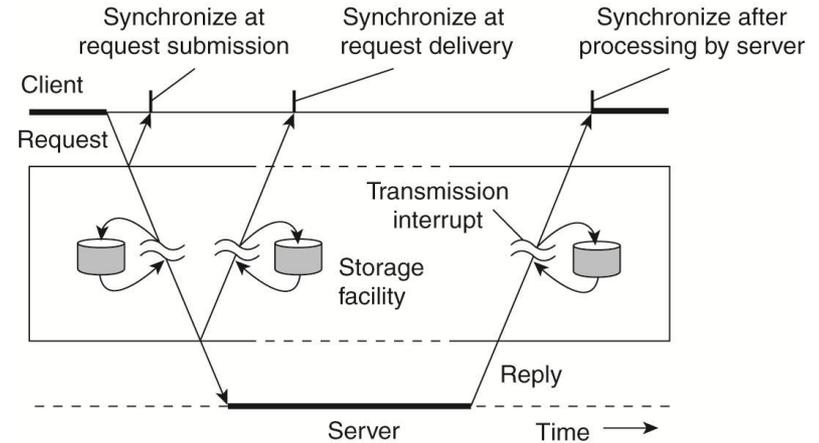
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# One-way RPC



# Asynchronous RPC

Recall: (A)synchronous communication  
 Q1: How can we implement asynchronous RPC

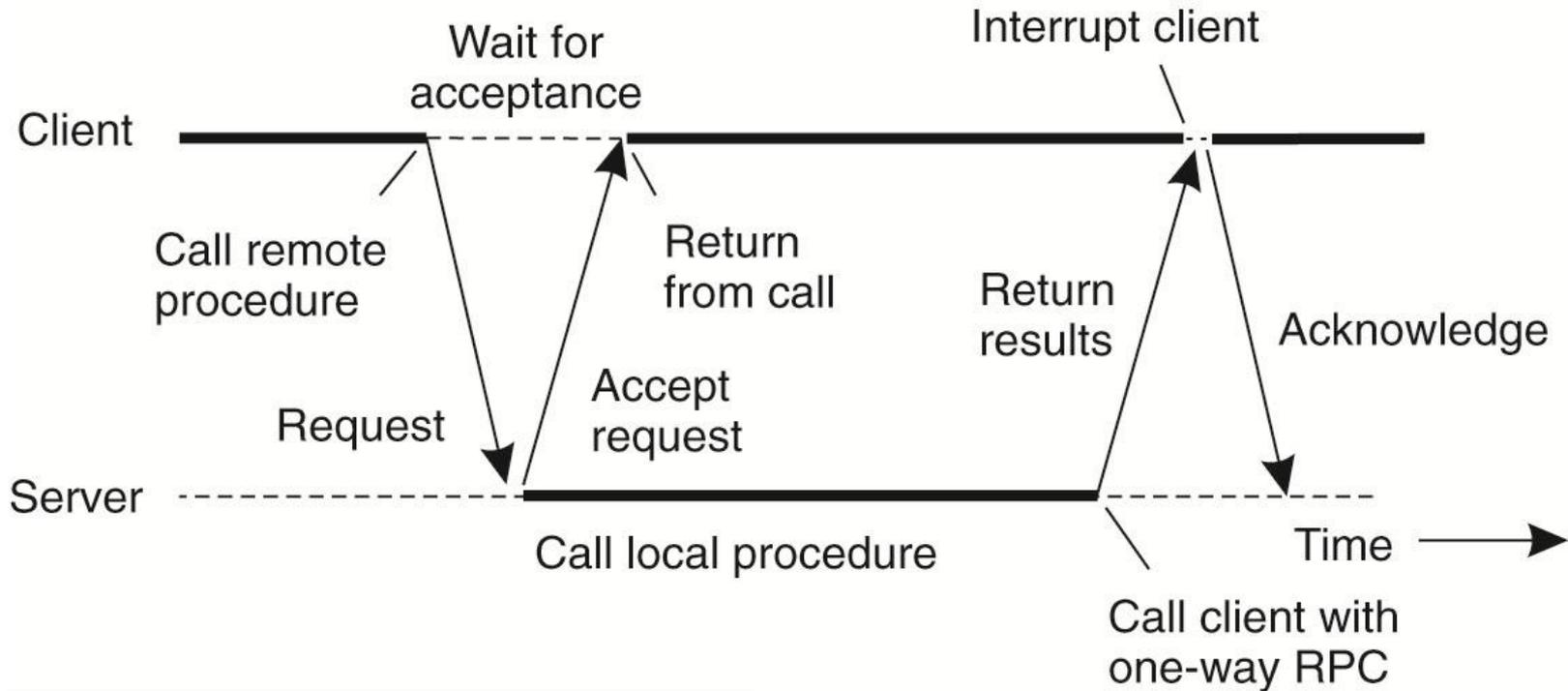


(b)

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Asynchronous RPC

Two asynchronous RPC/ Deferred synchronous RPC



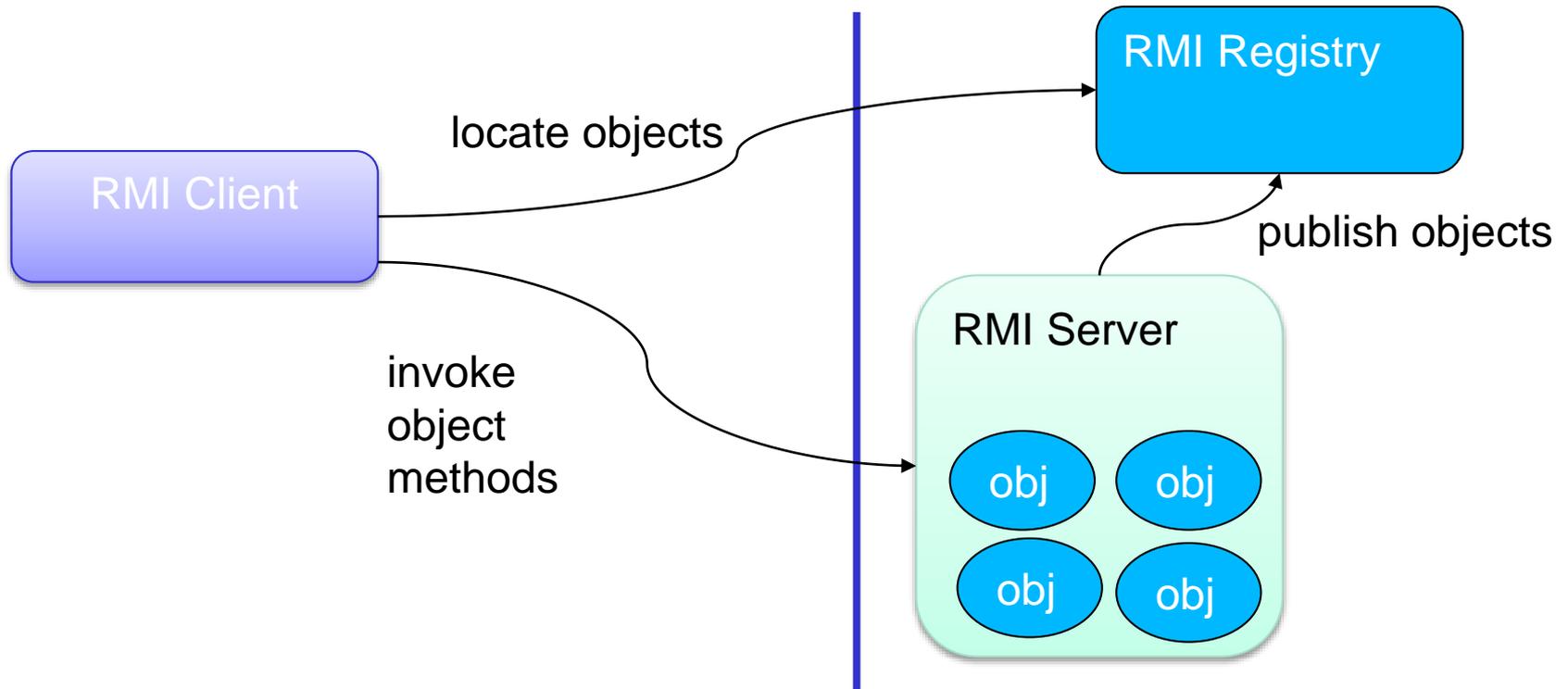
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Some RPC implementations

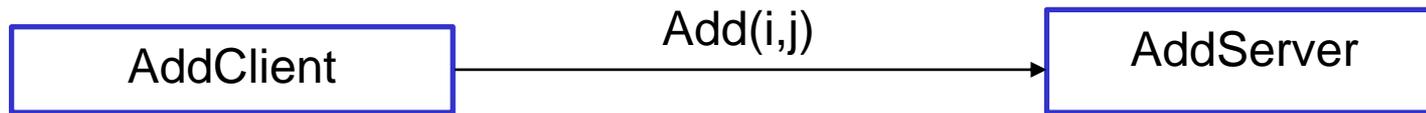
- rpcgen – SUN RPC
  - IDL for interface description
  - XDR for messages
  - TCP/UDP for transport
- XML-RPC
  - XML for messages
  - HTTP for transport
- JSON-RPC
  - JSON for messages
  - HTTP and/or TCP/IP for transport

# Remote Method Invocation/Remote Object Call

- RPC style in Java
  - Remote object method invocation/call



# Example of RPC



```

program ADD_PROG {
  version ADD_VERS {
    int add(int , int ) = 1;
  } = 1;
} = 0x23452345;
  
```

➔ `$rpcgen -N -a add.x` ➔

- add.h
- add\_xdr.c
- add\_client.c
- add\_clnt.c
- add\_server.c
- add\_svc.c

Code: <http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/exs/rpcadd-ex.tar.gz>

# STREAMING DATA PROGRAMMING

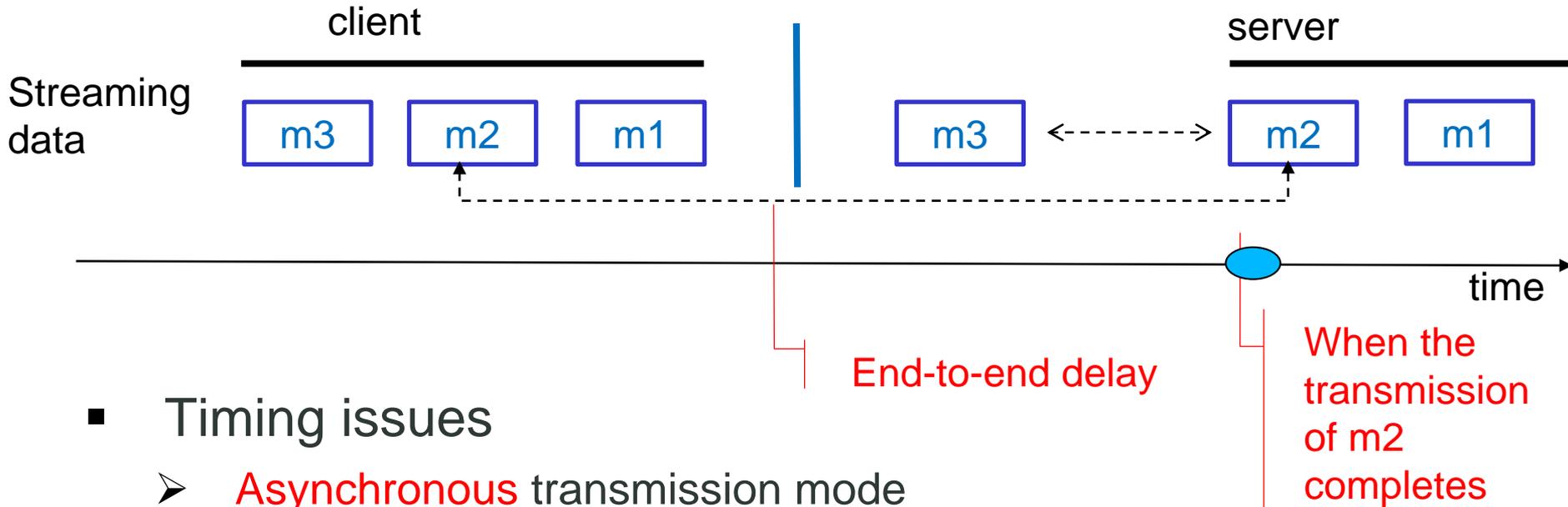
# Data stream programming

**Data stream:** a sequence of data units

e.g. reading bytes from a file and send bytes via a TCP socket

- Data streams can be used for
  - Continuous media (e.g., video)
  - Discrete media (e.g., stock market events/twitter events)

# Timing issues



- Timing issues

- **Asynchronous** transmission mode

- no constraints on when the transmission completes

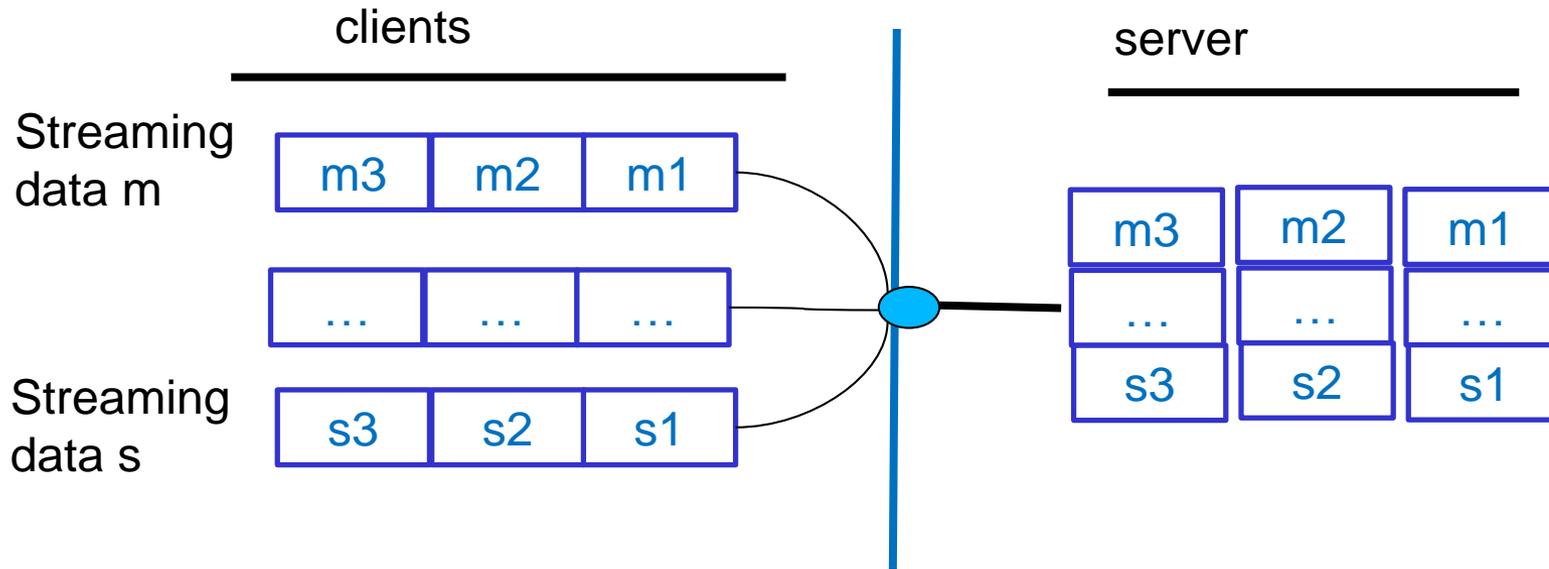
- **Synchronous** transmission mode:

- maximum end-to-end delay defined for each data unit

- **Isochronous** transmission

- maximum and minimum end-to-end delay defined

# Multiple streams

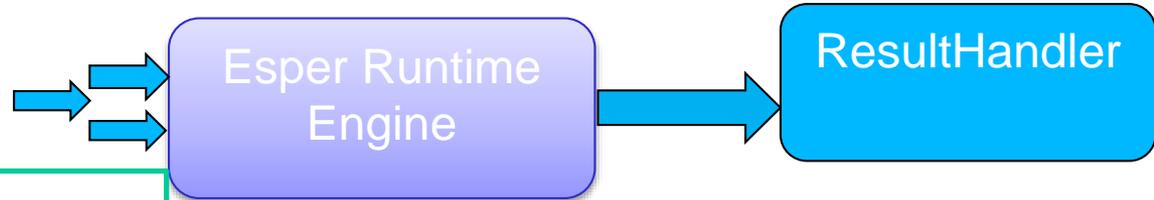


Complex stream/multiple streams data processing

# Example: Complex event processing with Esper

<http://esper.codehaus.org/esper>.

Streaming event data



```

public class InteractionEvent {
    public final static String REQUEST = "Request";
    public final static String RESPONSE = "Response";
    private String clientEndpoint=null;
    private String activityURI=null;

    private String serviceEndpoint=null;

    private String messageCorrelationID=null;

    private String messageType=null;
    ///....
}
  
```

EPL (Event Processing Language)

```

public class NumberCallHandler extends BaseResultHandler {

    @Override
    public void update(Map[] insertStream,
        Map[] removeStream) {
        ///....
    }
}
  
```

```

select clientEndpoint, serviceEndpoint
from InteractionEvent.win:length(100)
where messageType="Request"
  
```

# GROUP COMMUNICATION

# Group communication

- Group communication use multicast messages
  - E.g., IP multicast or application-level multicast

**Atomic Multicast:** Messages are received either by every member or by none of them

**Reliable multicast:** messages are delivered to all members in the best effort – but not guaranteed.

# Atomic Multicast

Q1: Give an example of atomic multicast

Example of implementing multicast using one-to-one communication

## Sender's program

```

i:=0;
do i ≠ n →
    send message to member[i];
    i:= i+1
od
  
```

## Receiver's program

```

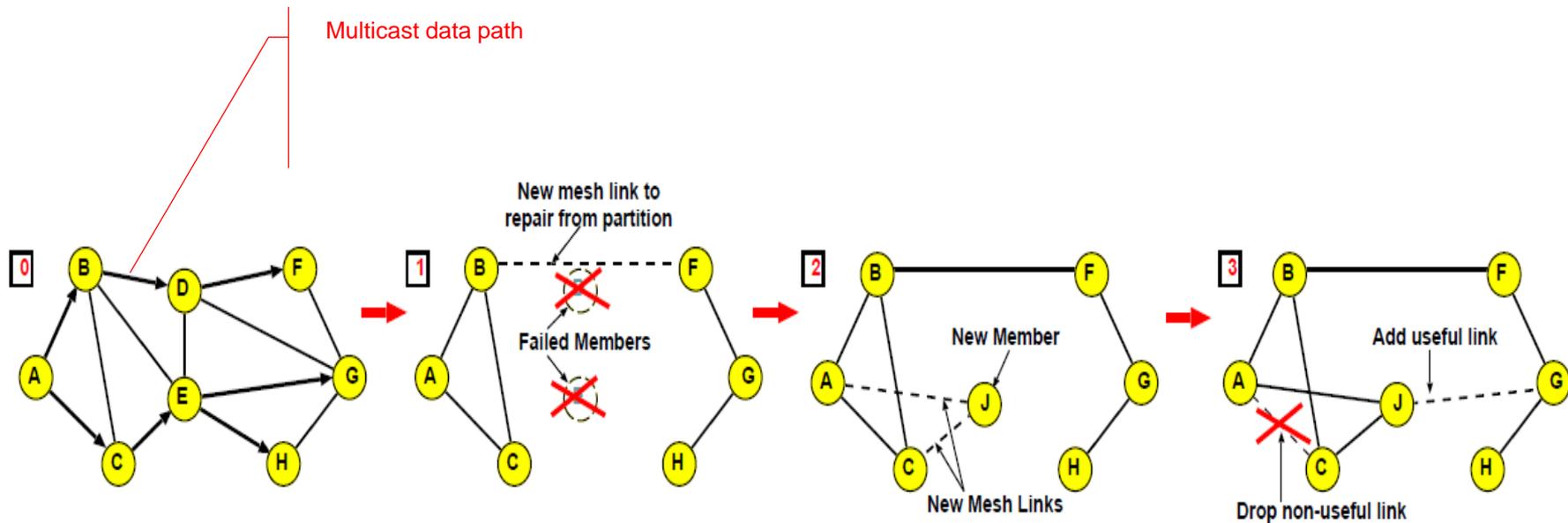
if m is new →
    accept it;
    multicast m;
[] m is duplicate → discard m
fi
  
```

Source: Sukumar Ghosh, Distributed Systems: An Algorithmic Approach, Chapman and Hall/CRC, 2007

Q2: How to know “**m is new**”?

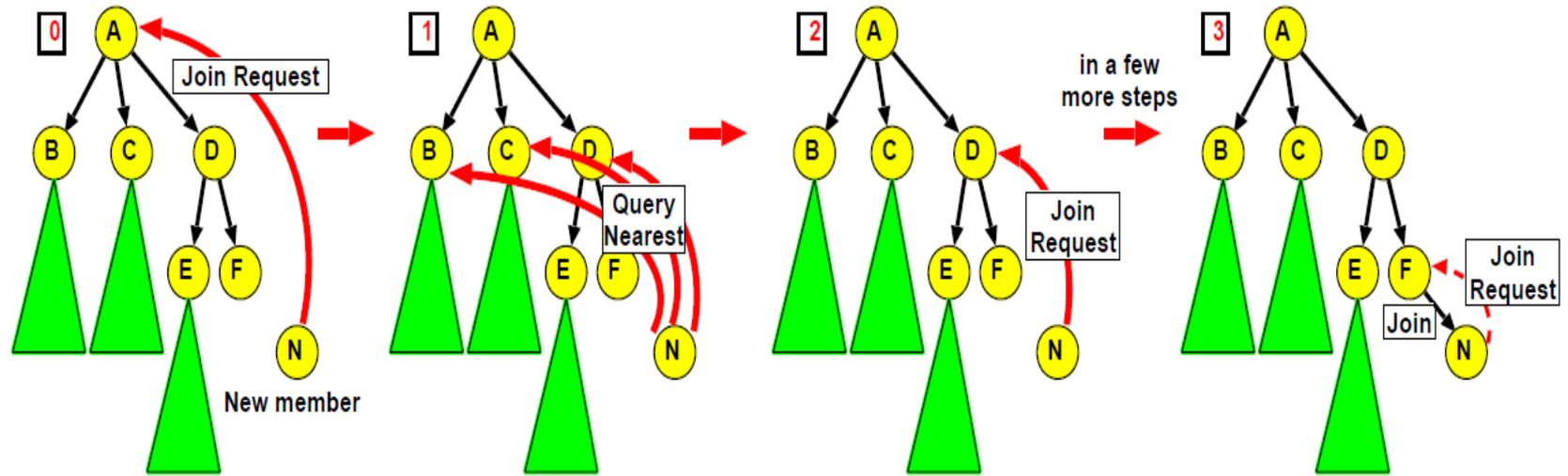
# Application-level Multicast Communication (1)

- Application processes are organized into an overlay network, typically in a tree or a mesh



Source: Suman Banerjee , Bobby Bhattacharjee , A Comparative Study of Application Layer Multicast Protocols (2001) , <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.2832>

# Application-level Multicast Communication (2)



Sources: Suman Banerjee , Bobby Bhattacharjee , A Comparative Study of Application Layer Multicast Protocols (2001) , <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.2832>

# Gossip-based Data Dissemination

## (1)

Why gossip?

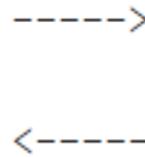
It can spread messages fast and reliable

Active thread (peer P):

```
(1) selectPeer(&Q);
(2) selectToSend(&bufs);
(3) sendTo(Q, bufs);
(4)
(5) receiveFrom(Q, &bufr);
(6) selectToKeep(cache, bufr);
(7) processData(cache);
```

Passive thread (peer Q):

```
(1)
(2)
(3) receiveFromAny(&P, &bufr);
(4) selectToSend(&bufs);
(5) sendTo(P, bufs);
(6) selectToKeep(cache, bufr);
(7) processData(cache)
```



Source: Anne-Marie Kermarrec and Maarten van Steen. 2007. Gossiping in distributed systems. SIGOPS Oper. Syst. Rev. 41, 5 (October 2007), 2-7. DOI=10.1145/1317379.1317381 <http://doi.acm.org/10.1145/1317379.1317381>

# Gossip-based Data Dissemination (2)

- Give a system of **N nodes** and there is the need to send some data items
- Every node has been updated for data item  $x$ 
  - Keep  $x$  in a buffer whose maximum capability is  **$b$**
  - Determine a number of times  **$t$**  that the data item  $x$  should be forwarded
  - **Randomly** contact  **$f$**  other nodes (the fan-out) and forward  $x$  to these nodes

Different configurations of  $(b,t,f)$  create different algorithms

Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, Laurent Massoulié, "Epidemic Information Dissemination in Distributed Systems," Computer, vol. 37, no. 5, pp. 60-67, May 2004, doi:10.1109/MC.2004.1297243

# Summary

- Various techniques for programming communication in distributed systems
  - Transport versus application level programming
  - Transient versus persistent
  - Procedure call versus messages
  - Streaming data
  - Multicast and gossip-based data dissemination
- Dont forget to play some simple examples to understand existing concepts

# Thanks for your attention

Hong-Linh Truong  
Distributed Systems Group  
Vienna University of Technology  
[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
<http://dsg.tuwien.ac.at/staff/truong>

# Naming in Distributed Systems

Hong-Linh Truong  
Distributed Systems Group,  
Vienna University of Technology

[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
[dsg.tuwien.ac.at/staff/truong](http://dsg.tuwien.ac.at/staff/truong)

# Learning Materials

- Main reading:
  - Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall
    - Chapter 5
- Others
  - George Coulouris, Jean Dollimore, Tim Kindberg, „Distributed Systems – Concepts and Design“, 2nd Edition
    - Chapter 9.
- Test the examples in the lecture

# Outline

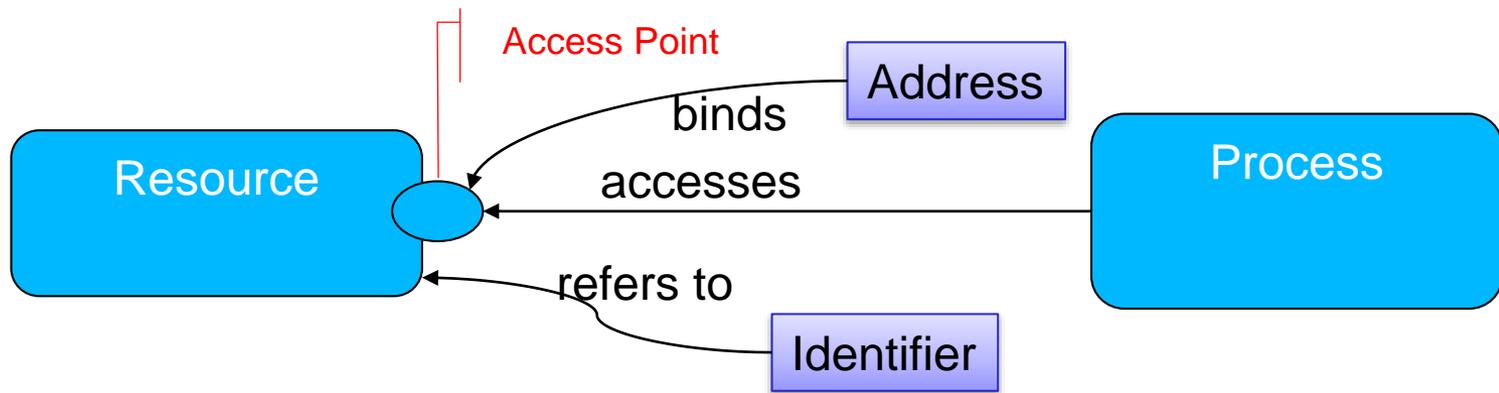
- Basic concepts and design principles
- Flat naming
- Structured naming
- Attribute-based naming
- Some naming systems in the Web
- Summary

# BASIC CONCEPTS AND DESIGN PRINCIPLES

# Why naming systems are important?

- **Entity**: any kind of objects we see in distributed systems: process, file, printer, host, communication endpoint, etc
- **Diverse types** of and **complex dependencies** among entities at different levels
  - E.g, printing service → the network level communication end points → the data link level communication end points
- But there are just so many entities, how do we **create and manage** names and **identify** an entity?

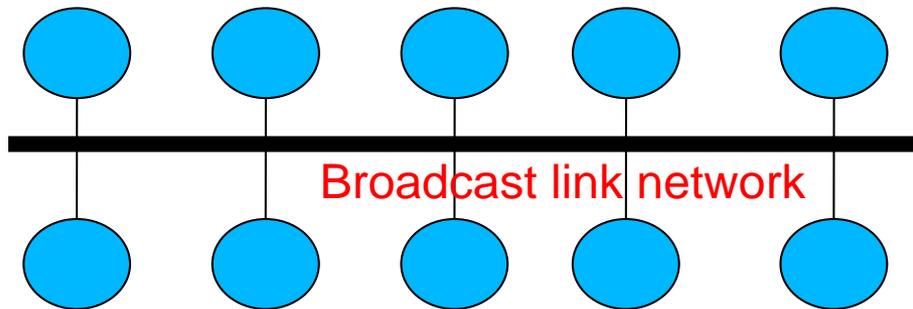
# Names, identifiers, and addresses



- Name: set of bits/characters used to identify/refer to an entity, a collective of entities, etc. in a specific context or uniquely
  - Simply comparing two names, we might not be able to know if they refer to the same entity
- Identifier: **a name that uniquely identifies an entity**
  - the identifier is unique and refers to only one entity
- Address: the name of an access point

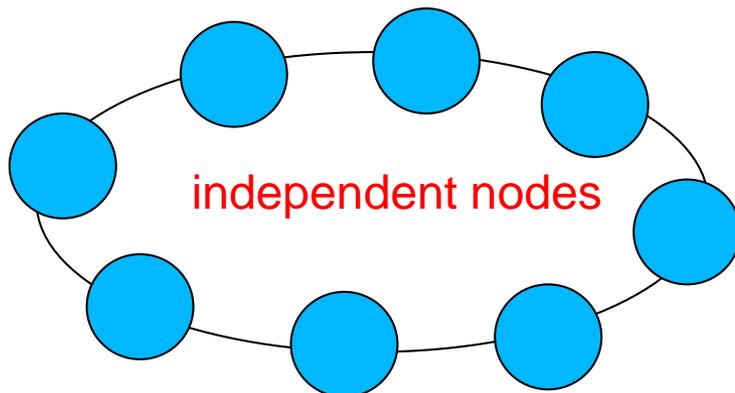
# Naming design principles

- Naming design is based on specific system organizations and characteristics



## Examples

- Network/Ethernet
- Identifier: IP and MAC address
- Name resolution: the network address to the data link address



- P2P systems
- Identifier: m-bit key
- Name resolution: distributed hash tables

# Naming design principles

- Structures and characteristics of names are based on different purposes
- Data structure:
  - Can be simple, no structure at all, e.g., a set of bits:  
**\$ uuid**  
**bcff7102-3632-11e3-8d4a-0050b6590a3a**
  - Can be complex
    - Include several data items to reflect different aspects on a single entity
  - Names can include location information/reference or not, e.g., GLN (Global Location Number) in logistics
  - Readability:
    - Human-readable or machine-processable formats

# Naming design principles

- **Diverse name-to-address binding mechanisms**
  - How a name is associated with an address or how an identifier is associated with an entity
  - Names can be changed over the time and names are valid in specific contexts
    - Dynamic or static binding?
- **Distributed or centralized management**
  - Naming data is distributed over many places or not
- **Discovery/Resolution protocol**
  - Names are managed by distributed services
  - Noone/single system can have a complete view of all names

# FLAT NAMING

# Flat naming

Unstructured/flat names: identifiers have no structured description, e.g., just a set of bits

- Simple way to represent identifiers
- Do not contain information for locating the access point of the entity
- Examples
  - Internet Address at the Network layer
  - m-bit numbers in Distributed Hash Tables

Q1: Flat naming are suitable for which types of systems

# Broadcast based Name Resolution

- Principles
  - Assume that we want find the access point of the entity **en**
  - Broadcast the identifier of **en**, e.g., **broadcast(ID(en))**
  - Only **en** will return the access point, when the broadcast message reaches nodes
- Examples
  - ARP: from IP address to MAC address (the datalink access point)

mail.infosys.tuwien.ac.at (128.131.172.240) at 00:19:b9:f2:07:55 [ether] on eth0  
sw-ea-1.kom.tuwien.ac.at (128.131.172.1) at 00:08:e3:ff:fc:c8 [ether] on eth0

# Dynamic systems

- Nodes form the system, no centralized coordination
- Nodes can join/leave/fail anytime
- A large number of nodes but a node knows only a subset of nodes
- Examples
  - Large-scale p2p systems, e.g., Chord, CAN (Content Addressable Network), and Pastry

Q1: How to define identifiers for such a system?

# Distributed Hash Tables

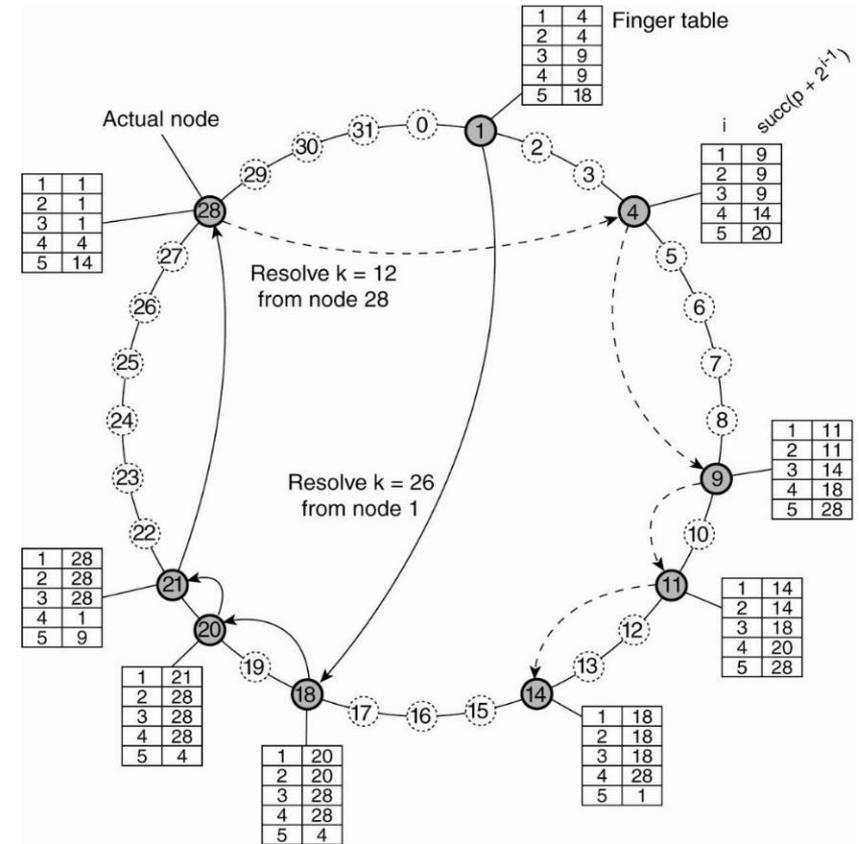
- Main concepts
  - $m$ -bit is used for **the keyspace** for identifiers
  - (Processing) Node identifier **nodeID** is one **key** in the keyspace
  - An entity **en** is identified by a hash function  **$k = \text{hash}(en)$**
  - A node **p** is responsible for managing entities associated with **a range of keys**
    - If  **$(k = \text{hash}(en) \in \text{range}(p))$** , then **put (k, en)** will store **en** in **p**
  - Nodes will relay messages (including entities/name resolution requests) till the messages reach the right destination

Q: Why DHT is useful for P2P systems? Is the nodeID fixed?



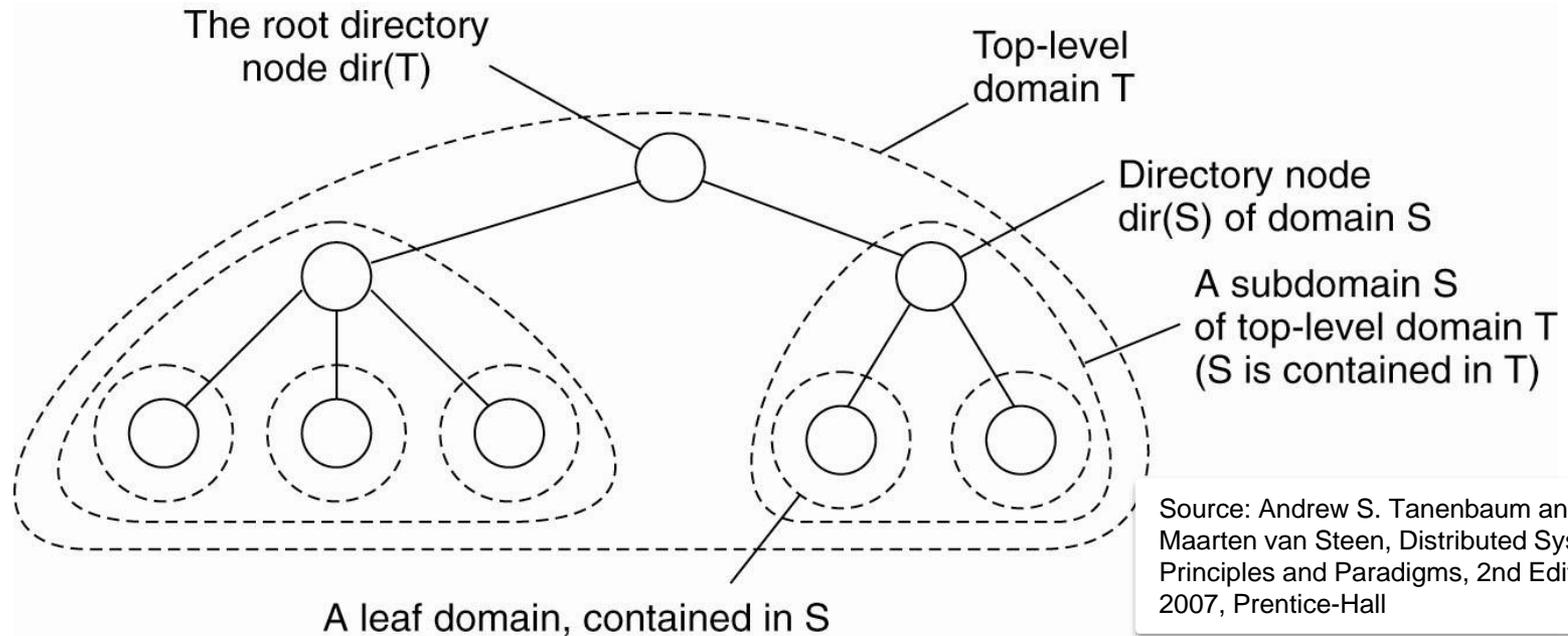
# Example - Chord

- Resolving at  $p$ 
  - Keep  $m$  entries in a finger table  $FT$   
 $FT_p[i]$   
 $= (\text{successor}(p$



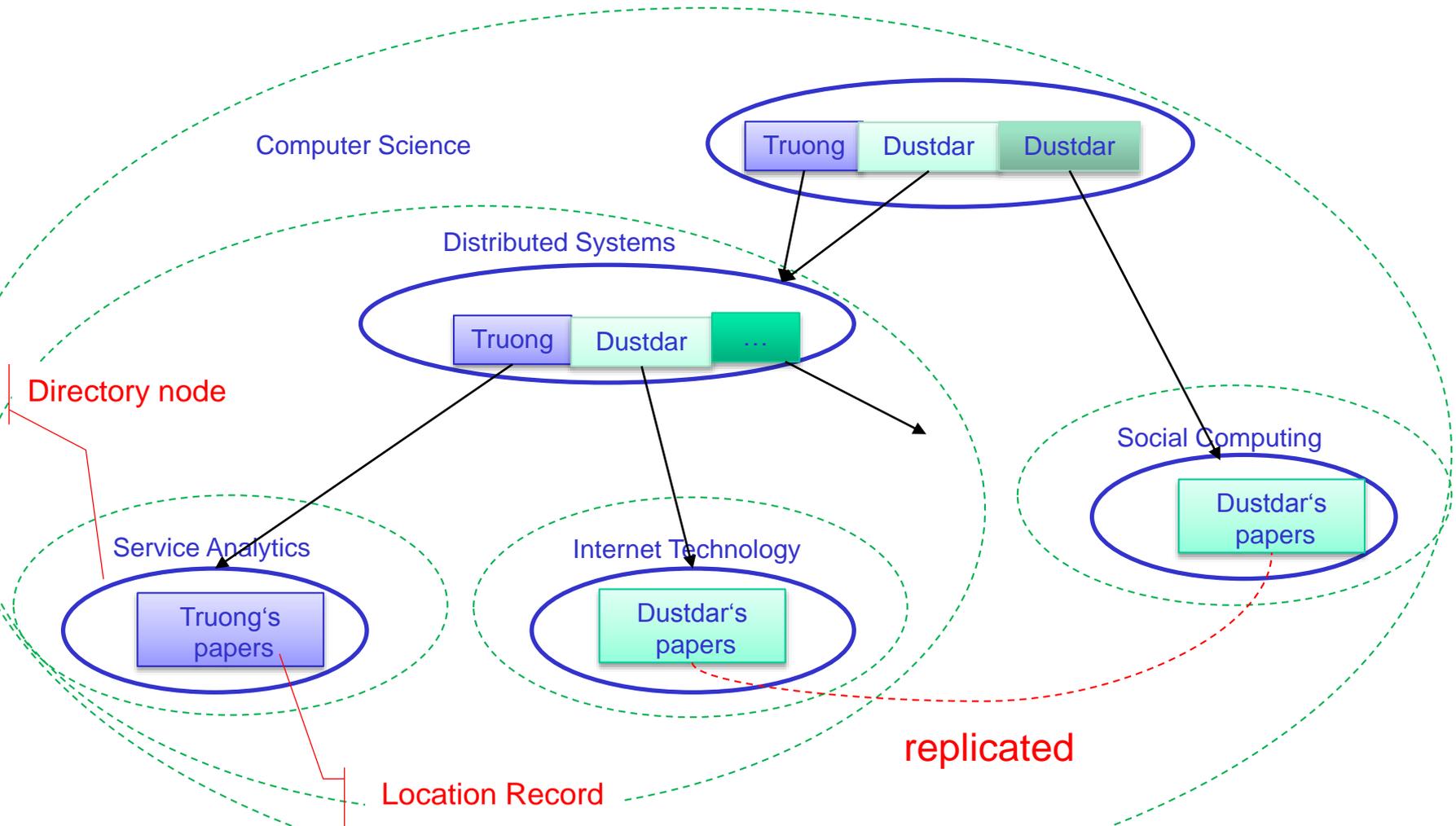
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Name management and resolution - - Hierarchical approach



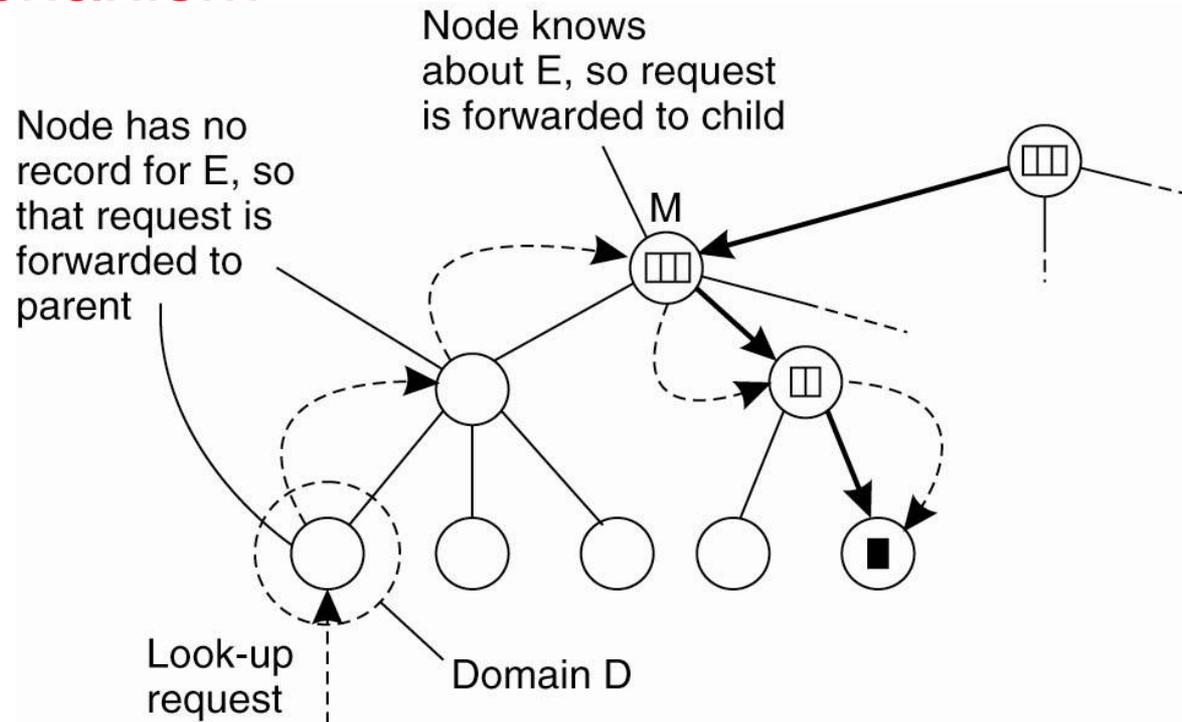
- The directory node has several location records.
- **A location record** is used to keep information about an entity in a domain  $D$ .
- The directory nodes contains both location records and pointers

# Name management and resolution - - Hierarchical approach



# Name management and resolution - - Hierarchical approach

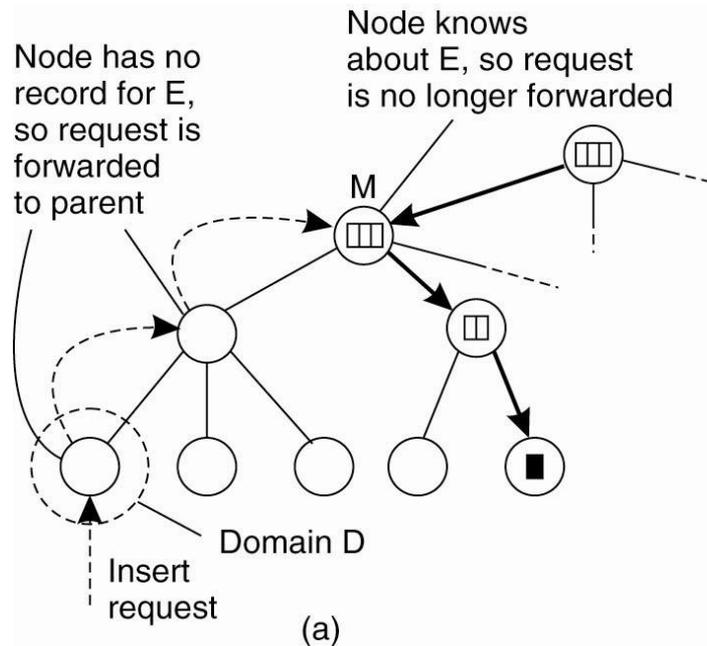
## Lookup mechanism



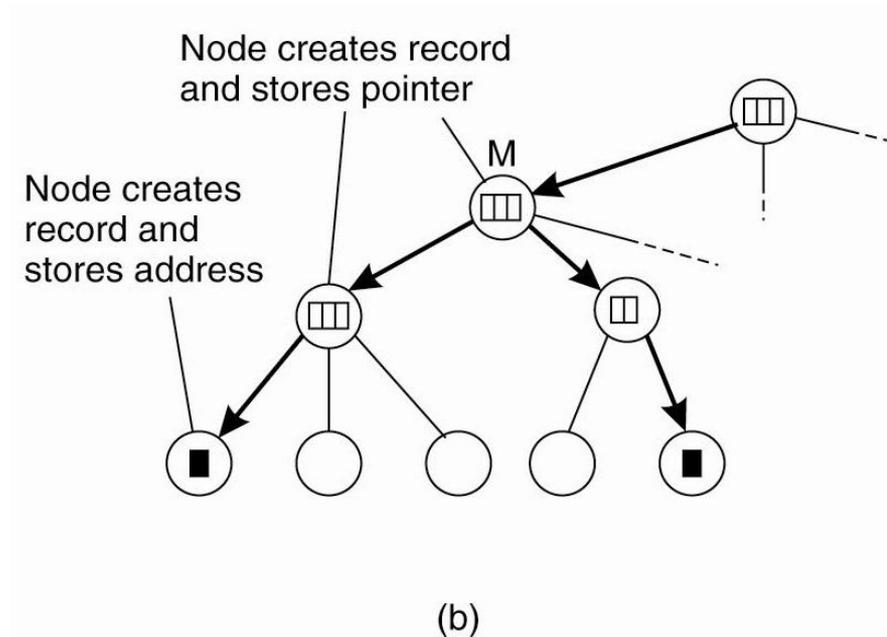
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Name management and resolution - - Hierarchical approach

## Insert/update mechanism



Insert request chain



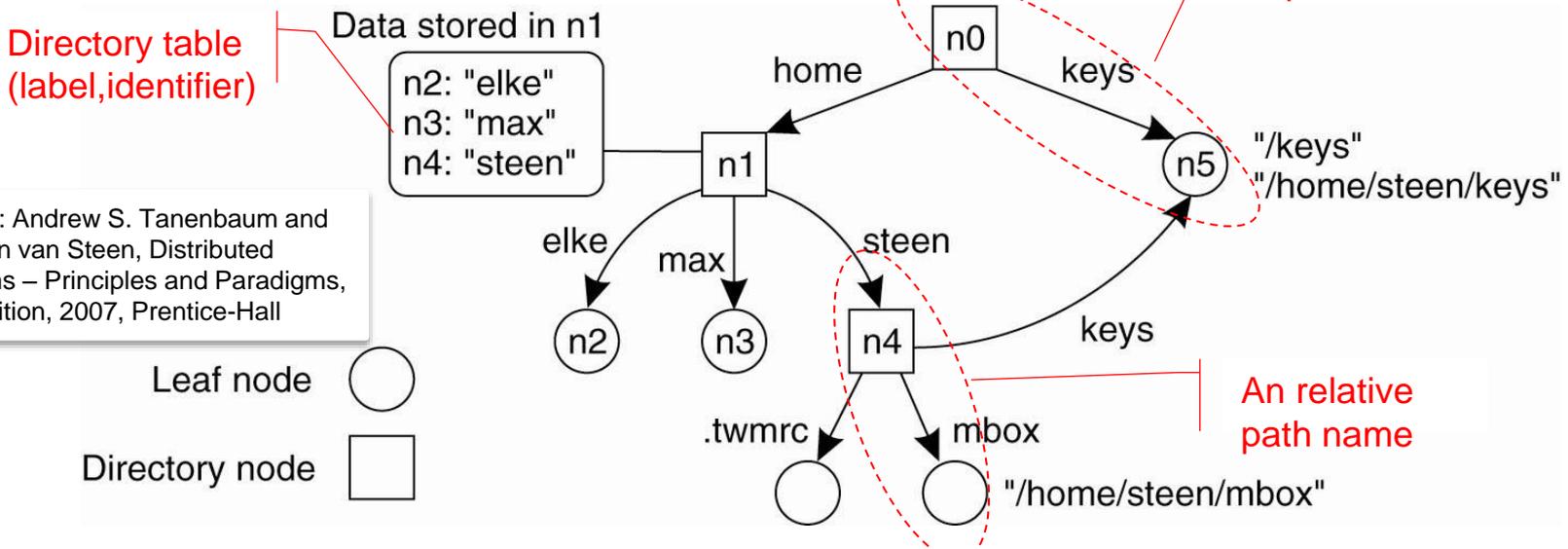
Create forwarding pointers chain

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# STRUCTURED NAMING

# Name spaces

- Names are organized into a name space
- A name space can be modeled as a graph:
  - Leaf node versus directory node
- Each node represents an entity**



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Q: How this differs from the flat naming with hierarchical approach?

# Name resolution – Closure Mechanism

- Name resolution:

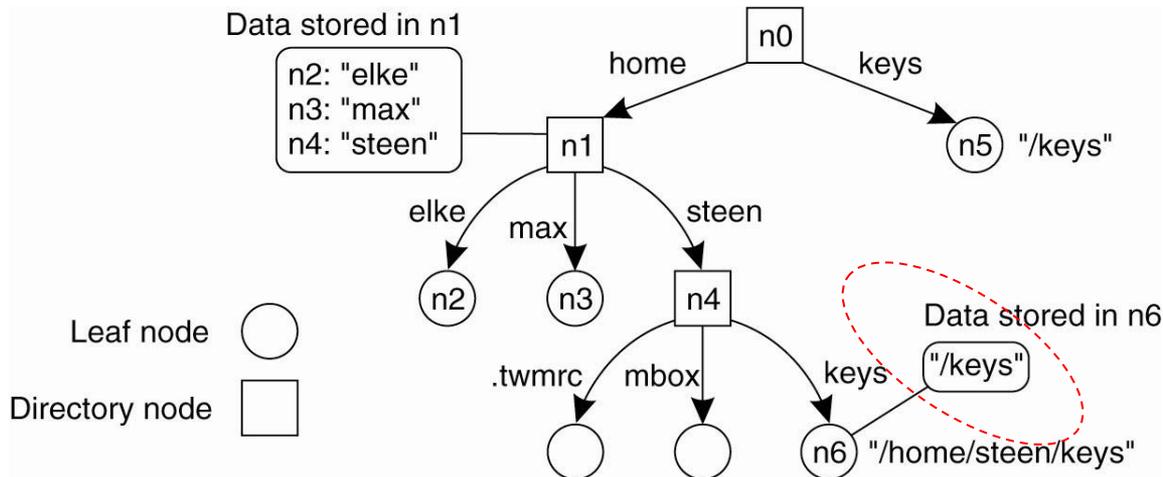
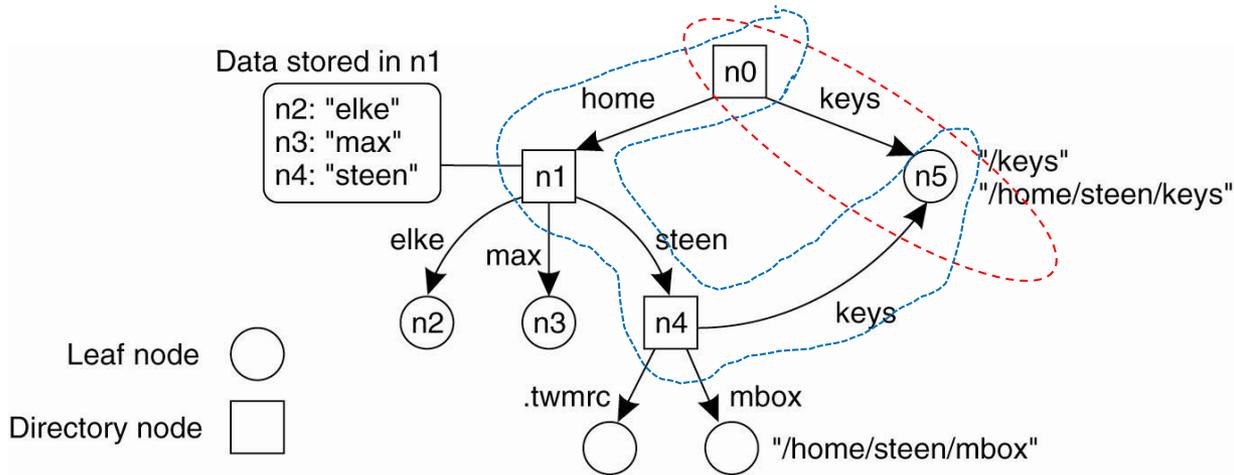
N:<label1,label2,label3,...labeln>

- Start from node **N**
- Lookup (**label1,identifier1**) in **N's** directory table
- Lookup (**label2, identifier2**) in **identifier1's** directory table
- and so on

**Closure Mechanism:** determine where and how name resolution would be started

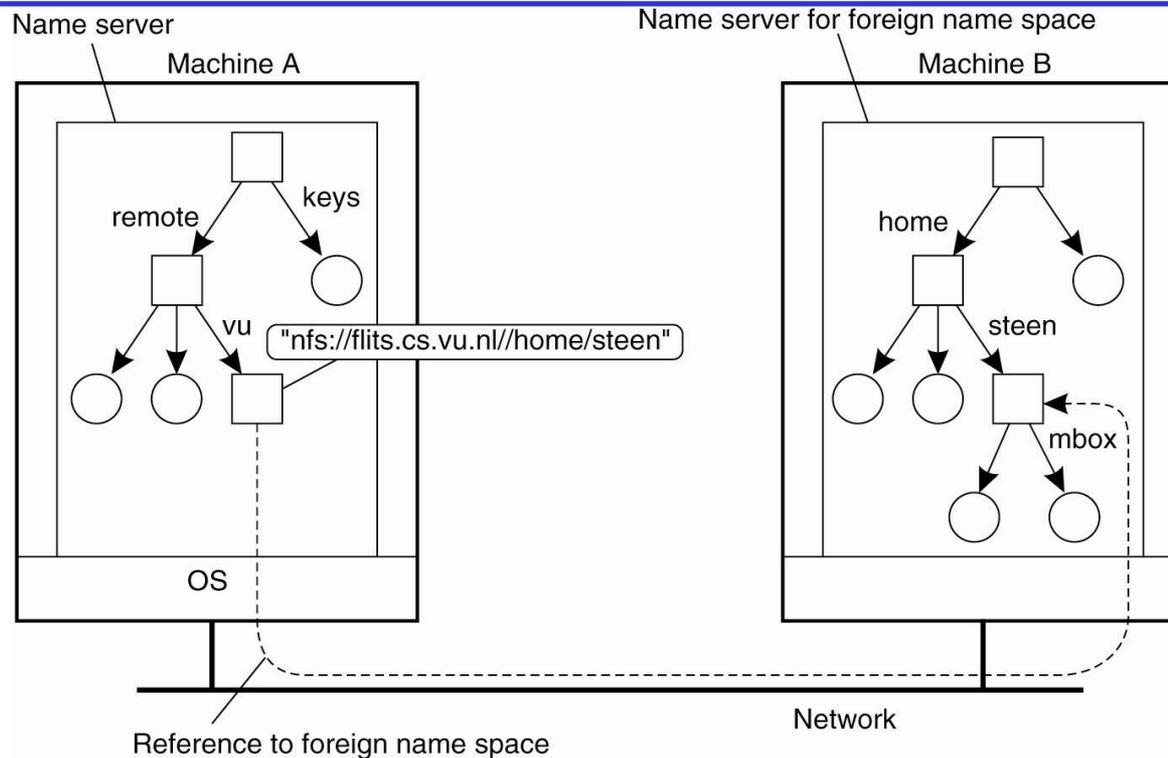
- E.g., name resolution for [/home/truong/ds.txt](#) ?
- Or for <https://me.yahoo.com/a/.....>

# Enabling Alias Using Links



# Name resolution - Mounting

- A directory node (mounting point) in a remote server can be mounted into a local node (mount point)



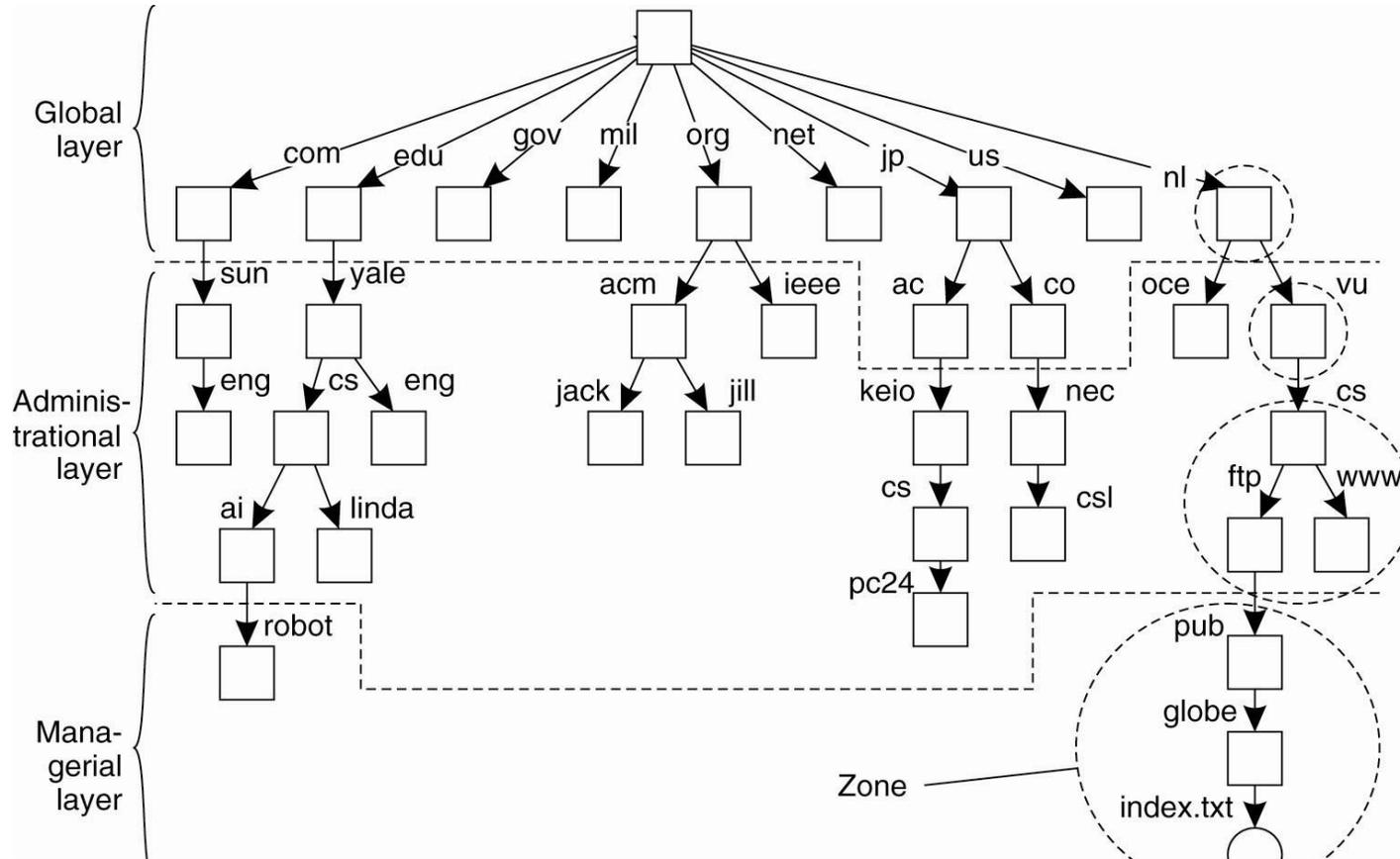
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall



# Name space implementation

- **Distributed name management**
  - Several servers are used for managing names
- **Many distribution layers**
  - **Global layer:** the root node and its close nodes
  - **Administrational layer:** directory nodes managed within a single organization
  - **Managerial layer:** nodes typically change regularly.

# Example in Domain Name System



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

aaa

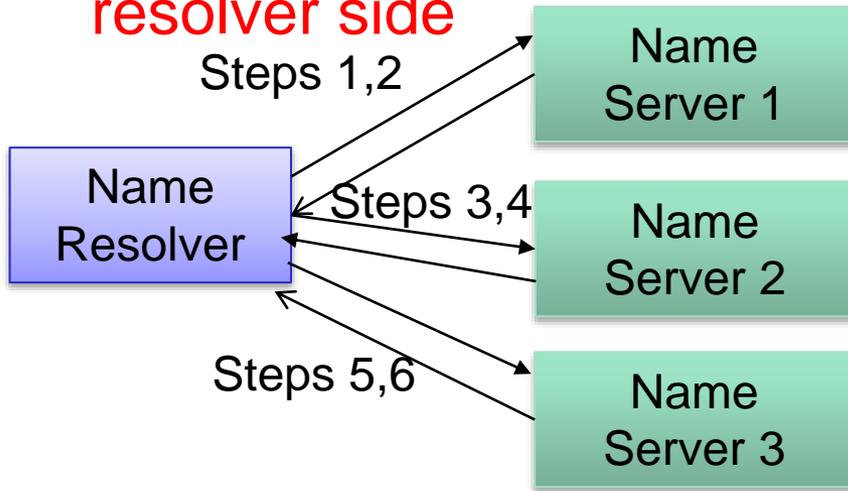
# Characteristics of distribution layers

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

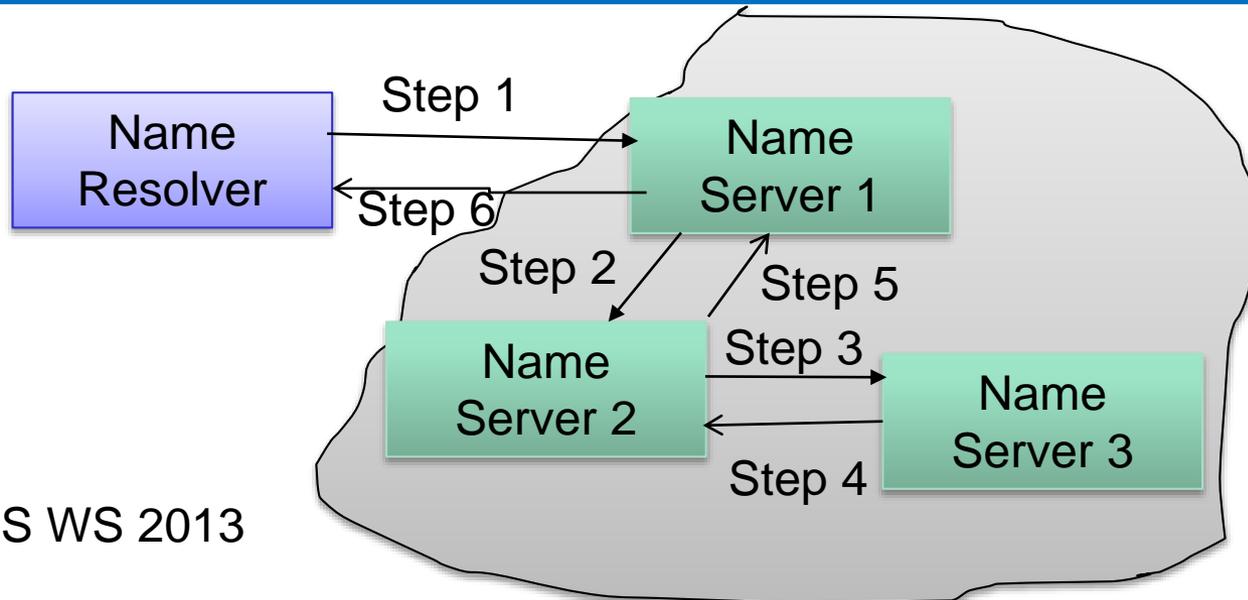
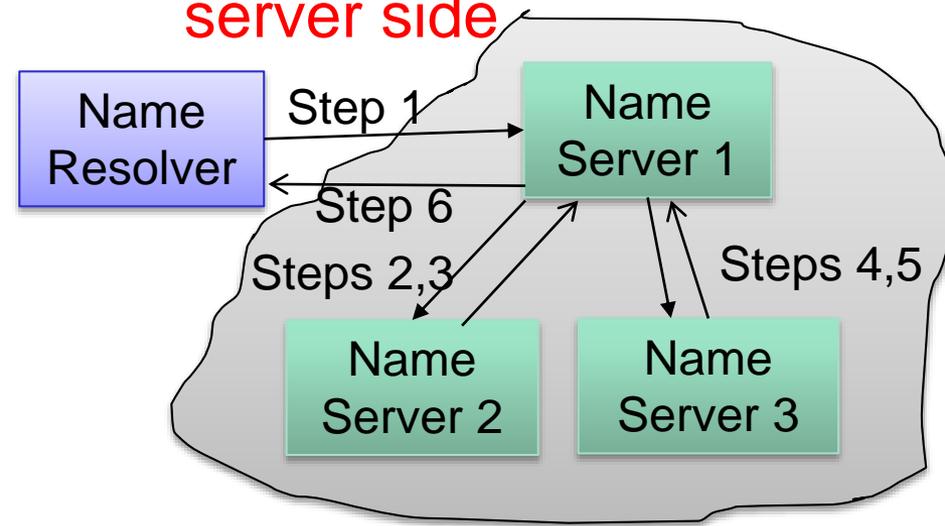
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Name Resolution

Iterative name resolution at  
resolver side

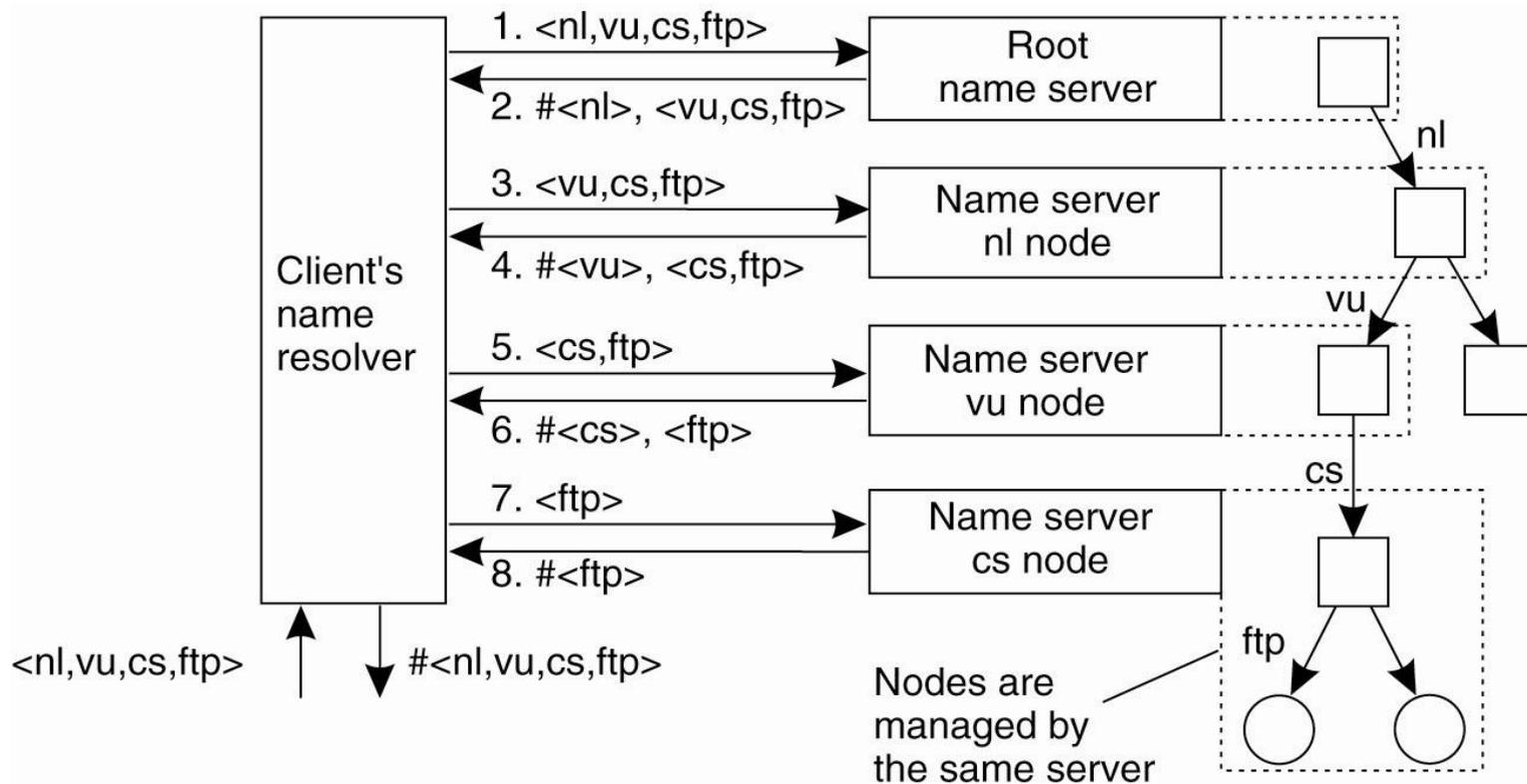


Iterative name resolution at  
server side



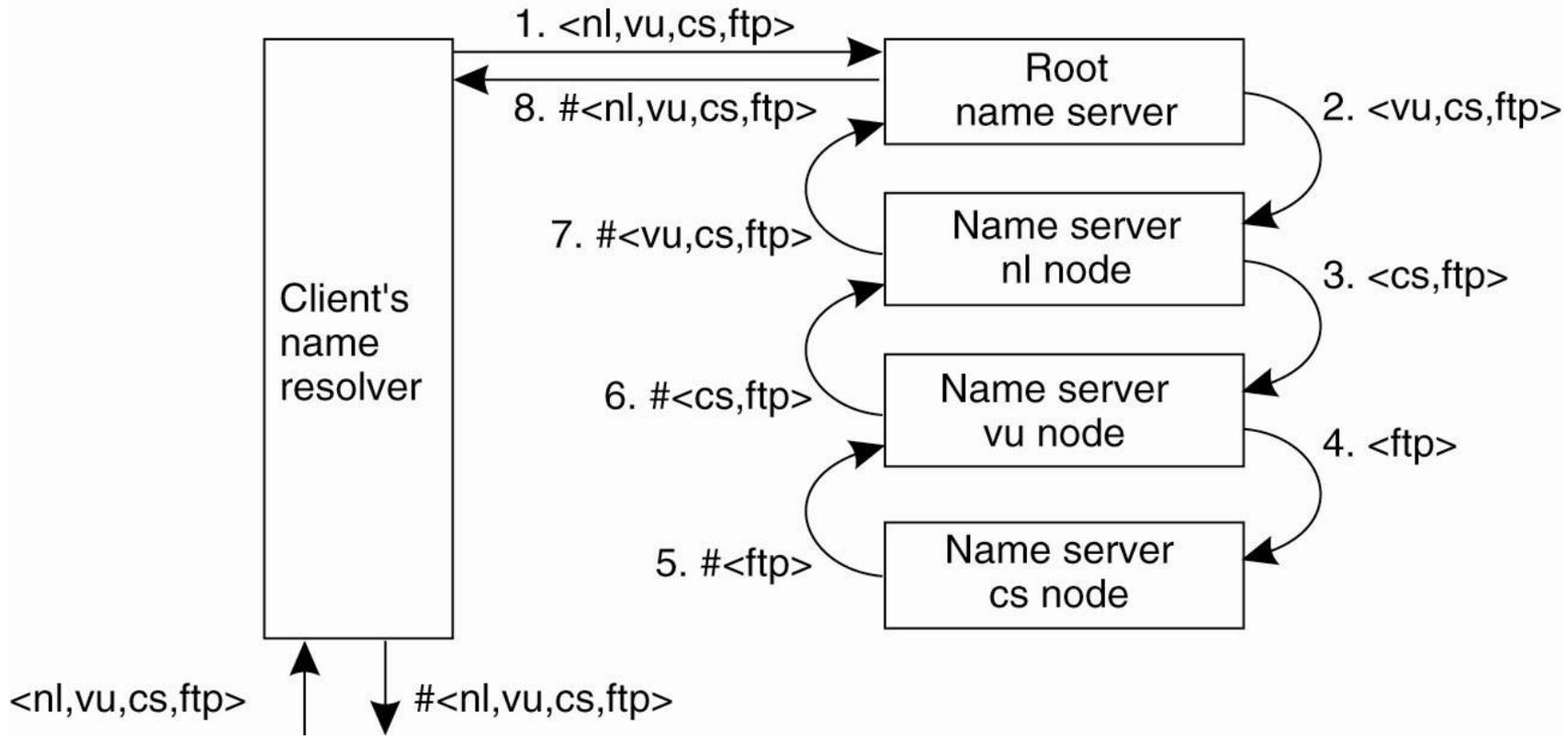
Recursive name  
resolution

# Example -- Iterative name resolution



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Example -- Recursive name resolution



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Q: Pros and cons of recursive name resolution

# Domain Name System in Internet

- We use to remember „human-readable“ machine name  
→ we have the name hierarchy
  - E.g., [www.facebook.com](http://www.facebook.com)
- But machines in Internet use IP address
  - E.g., 31.13.84.33
  - Application communication use IP addresses and ports
- DNS
  - Mapping from the domain name hierarchy to IP addresses

www.facebook.com    canonical name = star.c10r.facebook.com.  
Name: star.c10r.facebook.com  
Address: 31.13.84.33

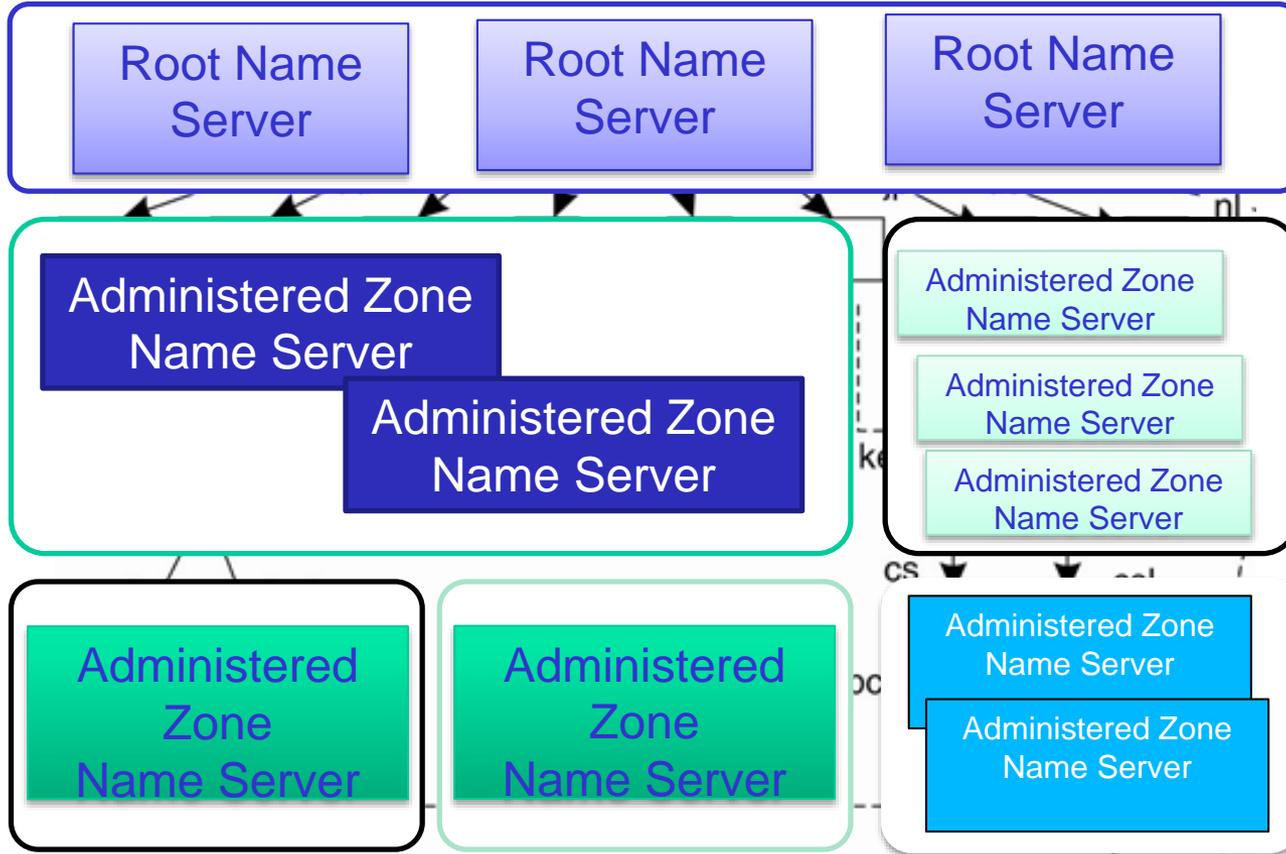
# Domain Name System

## Information in records of DNS namespace

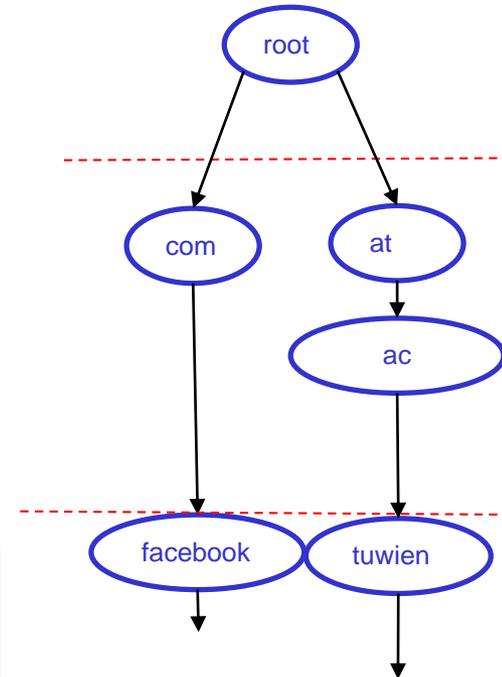
Type of record	Associated entity	Description
SOA	Zone	Holds information on the represented zone
A	Host	Contains an IP address of the host this node represents
MX	Domain	Refers to a mail server to handle mail addressed to this node
SRV	Domain	Refers to a server handling a specific service
NS	Zone	Refers to a name server that implements the represented zone
CNAME	Node	Symbolic link with the primary name of the represented node
PTR	Host	Contains the canonical name of a host
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any entity-specific information considered useful

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# DNS Name Servers



## Example



- **Authoritative name server:** answer requests for a zone
- **Primary and secondary servers:** the main server and the replicated server (maintained copied data from the main server)
- **Caching server**



# DNS Queries

- **Simple host name resolution**
  - Which is the IP of [www.tuwien.ac.at](http://www.tuwien.ac.at)?
- **Email server name resolution**
  - Which is the email server for [truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at) ?
- **Reverse resolution**
  - From IP to hostname
- **Host information**
- **Other service**

# Examples

- Iterative hostname resolution:  
<http://www.simplifiedns.com/lookup-dg.aspx>
- Mail server resolution:  
<https://www.mailive.com/mxlookup/>

# ATTRIBUTE-BASED NAMING

# Attributes/Values

- A tuple (**attribute,value**) can be used to describe a property
  - E.g., („country“,“Austria“), („language“, „German“),
- A set of tuples (attribute, value) can be used to describe an entity

AustriaInfo

Attribute	Value
CountryName	Austria
Language	German
MemberofEU	Yes
Capital	Vienna

# Attribute-based naming systems

- Employ (attribute,value) tuples for describing entities
  - Why flat and structured naming are not enough?
- Also called **directory services**
- Naming resolution
  - Usually based on querying mechanism
  - Querying usually deal with the whole space
- Implementations
  - LDAP
  - RDF (Resource Description Framework)

# LDAP data model

- **Object class**: describe information about objects/entities using **tuple(attribute,value)**
  - Hierarchical object class
- **Directory entry**: object entry for a particular object, alias entry for alternative naming and subentry for other information
- **Directory Information Base (DIB)**: collection of all directory entries
  - Each entry is identified by a **distinguished name (DN)**
- **Directory Information Tree (DIT)**: the tree structure for entries in DIB

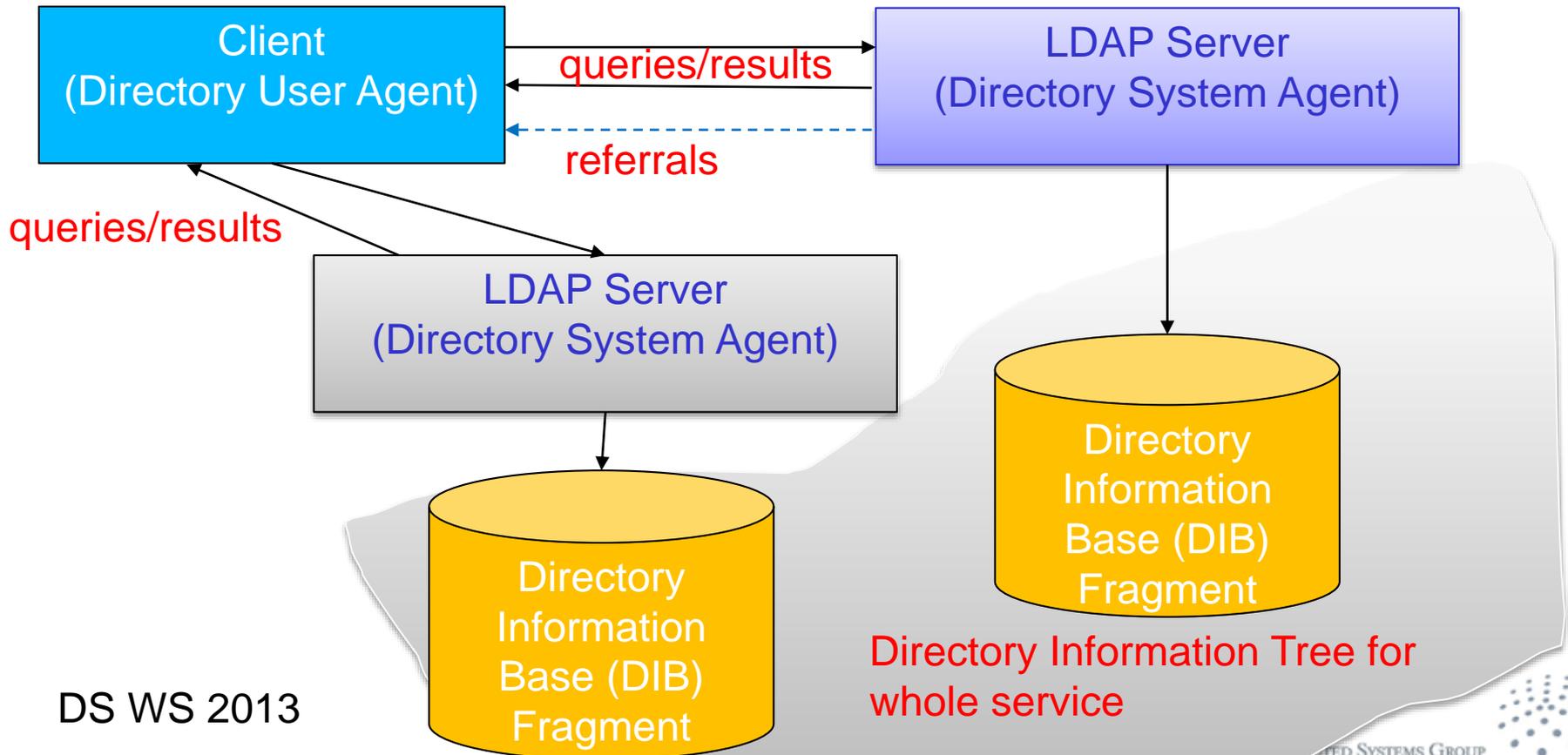
# LDAP – Lightweight Directory Access Protocol

- <http://tools.ietf.org/html/rfc4510>
- Example of attributes/values

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	130.37.20.20

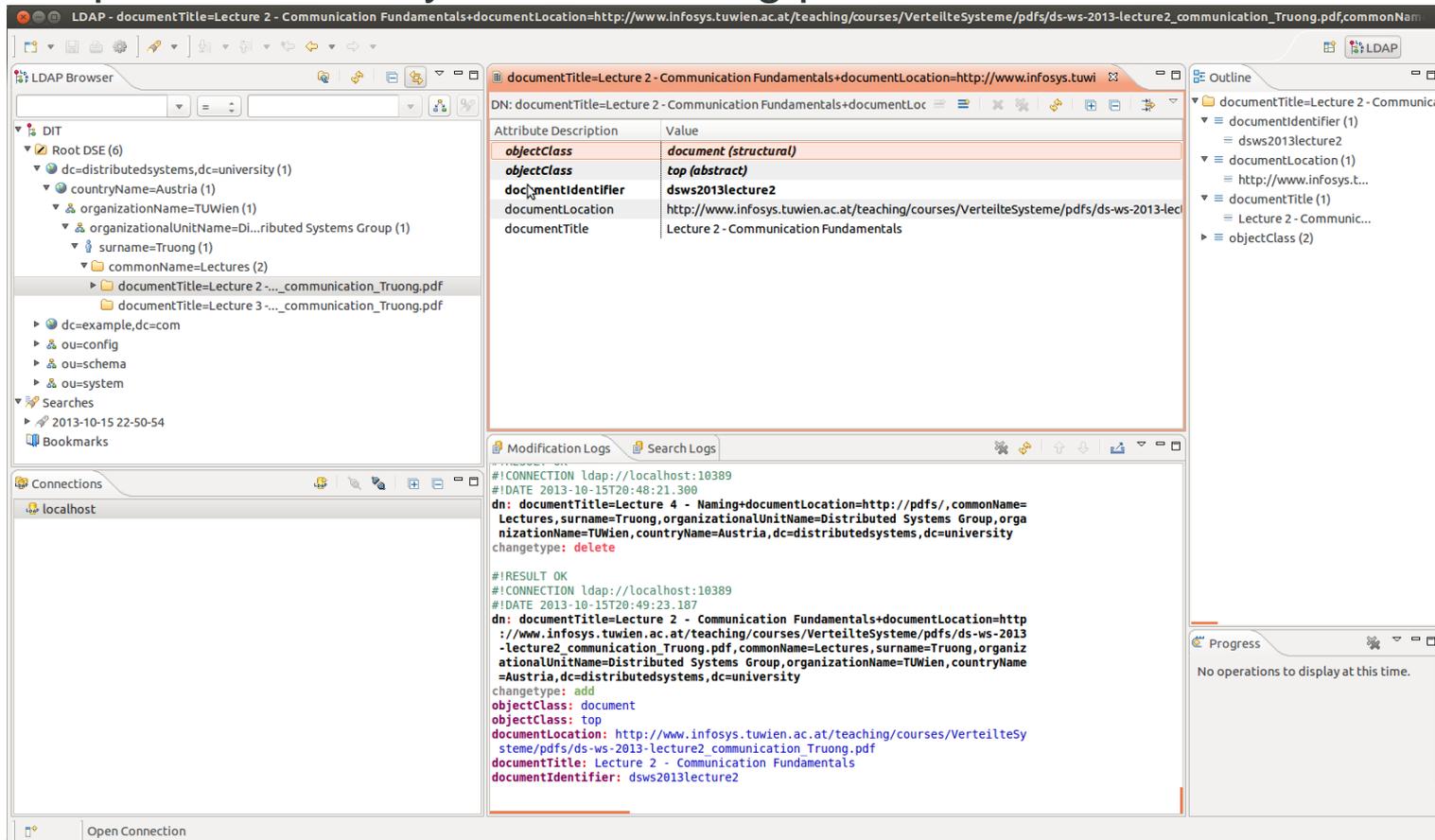
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

## Client-server protocol



# Example with Apache DS/DS Studio

- <http://directory.apache.org/>
- Apache DS: a directory service supporting LDAP and others
- Apache Directory Studio: tooling platform for LDAP



The screenshot displays the Apache Directory Studio (DS Studio) interface. The main window is titled "LDAP - documentTitle=Lecture 2 - Communication Fundamentals+documentLocation=http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/pdfs/ds-ws-2013-lecture2\_communication\_Truong.pdf,commonName=...".

The interface is divided into several panes:

- LDAP Browser:** Shows a tree view of the directory structure. The selected entry is "documentTitle=Lecture 2 - Communication Fundamentals+documentLocation=http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/pdfs/ds-ws-2013-lecture2\_communication\_Truong.pdf".
- Attribute Table:** Displays the attributes of the selected entry:
 

Attribute Description	Value
objectClass	document (structural)
objectClass	top (abstract)
documentIdentifier	dsws2013lecture2
documentLocation	http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/pdfs/ds-ws-2013-lecture2_communication_Truong.pdf
documentTitle	Lecture 2 - Communication Fundamentals
- Modification Logs:** Shows the log of the current operation:
 

```

      #!CONNECTION ldap://localhost:10389
      #!DATE 2013-10-15T20:48:21.300
      dn: documentTitle=Lecture 4 - Naming+documentLocation=http://pdfs/,commonName=Lectures,surname=Truong,organizationalUnitName=Distributed Systems Group,organizationalUnitName=TUWien,countryName=Austria,dc=distributedsystems,dc=university
      changetype: delete

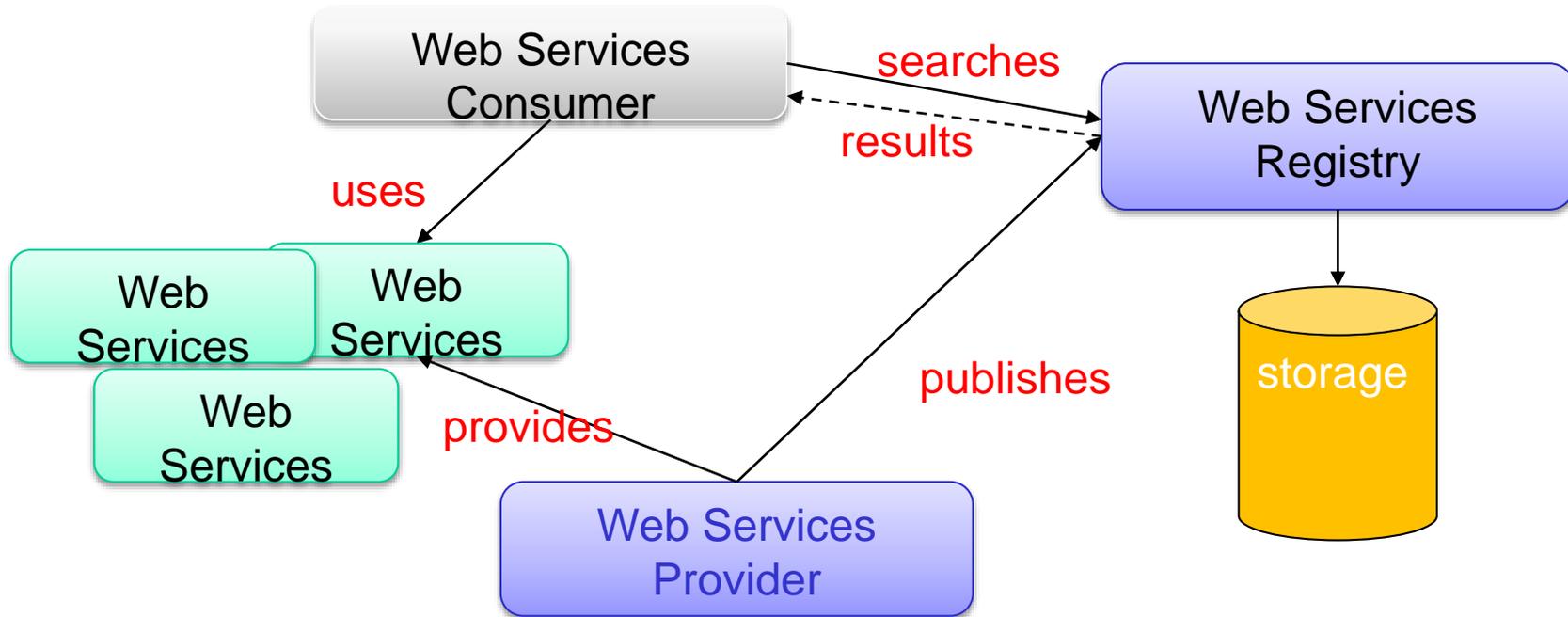
      #!RESULT OK
      #!CONNECTION ldap://localhost:10389
      #!DATE 2013-10-15T20:49:23.187
      dn: documentTitle=Lecture 2 - Communication Fundamentals+documentLocation=http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/pdfs/ds-ws-2013-lecture2_communication_Truong.pdf,commonName=Lectures,surname=Truong,organizationalUnitName=Distributed Systems Group,organizationalUnitName=TUWien,countryName=Austria,dc=distributedsystems,dc=university
      changetype: add
      objectClass: document
      objectClass: top
      documentLocation: http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/pdfs/ds-ws-2013-lecture2_communication_Truong.pdf
      documentTitle: Lecture 2 - Communication Fundamentals
      documentIdentifier: dsws2013lecture2
      
```
- Outline:** Shows a hierarchical view of the directory structure, including "documentTitle=Lecture 2 - Communication Fundamentals" and its sub-entries.
- Progress:** Shows "No operations to display at this time."

# SOME NAMING SERVICES IN THE WEB

# Web services – service identifier

- **Web service:** basically an entity which offers software function via well-defined, interoperable interfaces that can be accessed through the network
  - E.g.,  
<http://www.webservices.net/globalweather.asmx>
- **Web services identifier:**
  - A web service can be described via WSDL
  - Inside WSDL, there are several „addresses“ that identify where and how to call the service access points

# Web services -- discovery



- Registry implementations
  - WSO2 Governance Registry - <http://wso2.com/products/governance-registry/>
  - java UDDI (jUDDI) - <http://juddi.apache.org/>

# OpenID – people identifier in the Web

- Several services offering individual identifiers
  - Your google ID, Your yahoo ID, etc.
- But there will be no single provider for all people

We need mechanisms to accept identifiers from different providers

- OpenID standard enables identifiers for people that can be accepted by several service provider
- An OpenID identifier is described as a URL
  - E.g., <https://me.yahoo.com/a/.....>

Q: Why OpenID identifier can be considered unique?

## Using OpenID to login to some services



The screenshot shows the LiveJournal website header with navigation links: Home, Create an account, Explore, Shop, and LJ Extras. On the right, there are input fields for Username and Password, a Remember checkbox, and language options for English and Spanish.

### Log in with OpenID, Facebook or Twitter

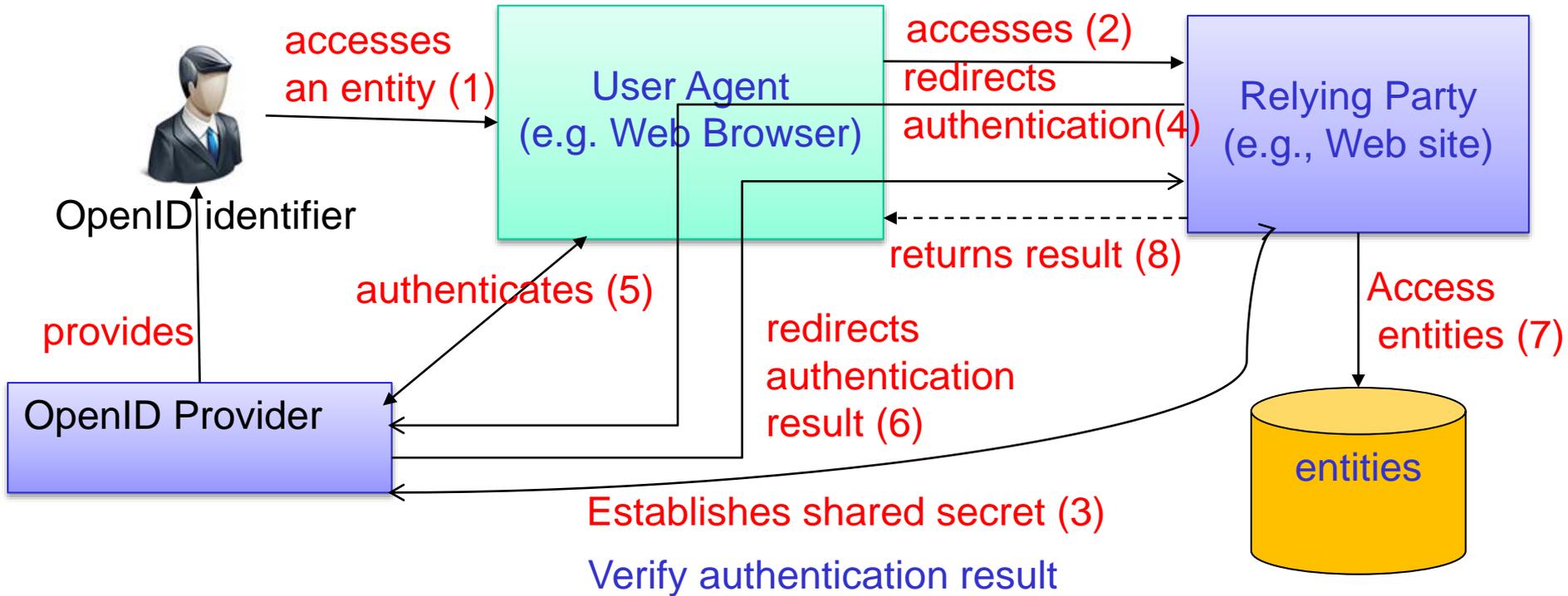
OpenID Facebook Twitter Google Mail.ru VKontakte

LiveJournal.com supports the OpenID distributed identity system, letting you bring your LiveJournal.com identity to other sites, and letting non-LiveJournal.com users bring their identity here.

Your OpenID URL:  Login

ex. <http://myblog.domain.com>

# OpenID interactions



# Summary

- Naming is a complex issue
  - Fundamental for other topics, e.g., communication and access control
- Different models
  - Flat, structured and attributed-based naming
- Different techniques to manage names
  - Centralized versus distributed
- Different protocols for naming resolution
- Dont forget to play some simple examples to understand existing concepts

# Thanks for your attention

Hong-Linh Truong  
Distributed Systems Group  
Vienna University of Technology  
[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
<http://dsg.tuwien.ac.at/staff/truong>

# Time and Synchronization in Distributed Systems

Hong-Linh Truong  
Distributed Systems Group,  
Vienna University of Technology

[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
[dsg.tuwien.ac.at/staff/truong](http://dsg.tuwien.ac.at/staff/truong)

# Learning Materials

- Main reading:
  - Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall
    - Chapter 6
  - Roberto Baldoni, Michel Raynal: Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. IEEE Distributed Systems Online 3(2) (2002)  
<http://www.dis.uniroma1.it/~baldoni/baldoni-112865.pdf>
- Others
  - George Coulouris, Jean Dollimore, Tim Kindberg, „Distributed Systems – Concepts and Design“, 2nd Edition
    - Chapter 10
  - Sukumar Ghosh, Distributed Systems: An Algorithmic Approach, Chapman and Hall/CRC, 2007, Chapters 6, 7, 11

- Clock synchronization
  - Physical clock
  - Logical clock
  - Vector Clock
- Distributed coordination
  - Mutual exclusion
  - Leader election
- Summary

# PHYSICAL CLOCK SYNCHRONIZATION

# Why do we need clock/time synchronization?

Documentation\Installation\Regulatory Compliance\NYSE



## The New York Stock Exchange

The New York Stock Exchange has various regulations regarding the synchronization of clocks used for timestamping, particularly in regards to use of the Front End Systemic Capture (FESC) system.

NYSE Rules 123 and, in particular, 132A detail these requirements. NYSE Information Memo 03-26, June 10, 2003 specifies:

"New Rule 132A requires members to synchronize the business clocks used to record the date and time of any event that the Exchange requires to be recorded. The Exchange will require that the date and time of orders in Exchange-listed securities to be recorded. The Rule also requires that members maintain the synchronization of this equipment in conformity with procedures prescribed by the Exchange."

### Specific NYSE Time Synchronization Requirements

Rule 132A contains two specific requirements:

- **Clocks Synchronized to Commonly Used Time Standard**

All computer clocks and mechanical timestamping devices must be synchronized to a commonly used time standard, either the National Institute of Standards and Technology (NIST) or United States Naval Observatory (USNO) atomic clocks.

- **Synchronization must be maintained**

Rule 132A also indicates that the member must ensure that their systems remain synchronized.

<https://www.greyscale.com/software/domain/time/instructions/quickstart/regulatory-nyse.asp>

### How to Use Domain Time II to comply with the NYSE Rule 132A Requirements

- Some reasons
  - **Accountability** of processes
  - **Consistency** in processing messages
  - **Validity** of important messages
  - **Fairness** in processing requests

# Real clock synchronization

Challenging issue: it is impossible to guarantee timers/clocks in different computers due to the clock drift problem

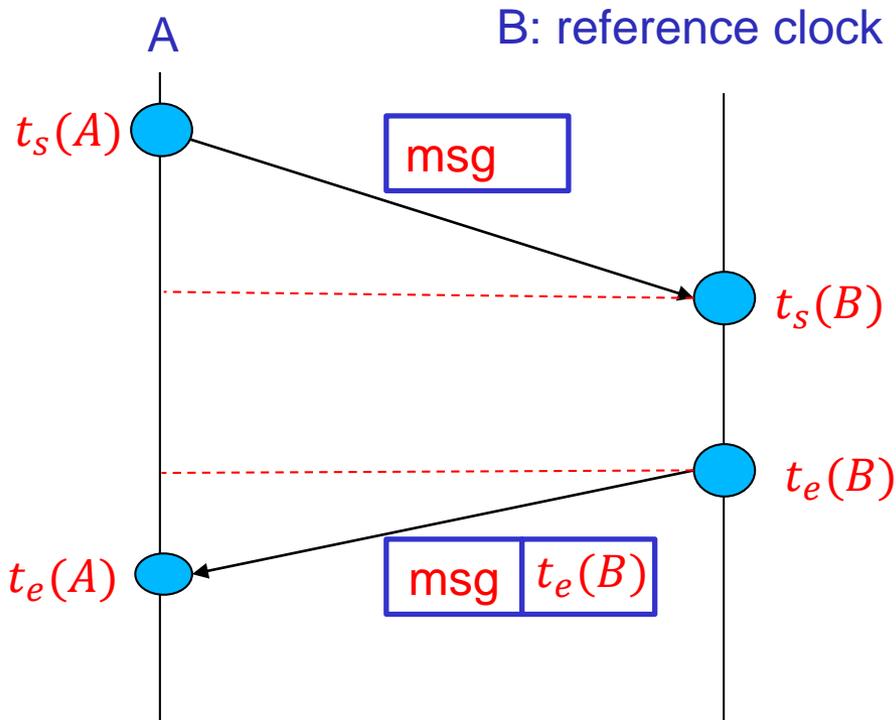
- **Establish/Decide** reference physical clocks → to provide an accurate timing system
  - Coordinated Universal Time (UTC)
    - Based on atomic time produced by the most accurate physical clocks using atomic oscillators
- **Operate/Utilize** accuracy physical clocks providing UTC time
- **Synchronize** other physical clocks using time synchronization algorithms



# Time provided by real physical clocks

- Computer clocks/timers
  - Every computer has a clock/timer
- Radio clocks receiving time codes via radio wave
  - Radio transmitter connects to an accuracy time source based on UTC time standard
- GPS (Global Positioning System) - a system of satellites, each broadcasts
  - its positions and the time stamps, based on its local time

# Cristian's Algorithm



$$RTT = (t_e(A) - t_s(A)) - (t_e(B) - t_s(B))$$

The most simple case: Assume that times spent in sending messages are the same and that the processing time at B is 0 then

$$RTT = (t_e(A) - t_s(A))$$

Based on B's clock the message should arrive at A at

$$t'_e(A) = (t_e(B) + RTT/2)$$

A's clock:

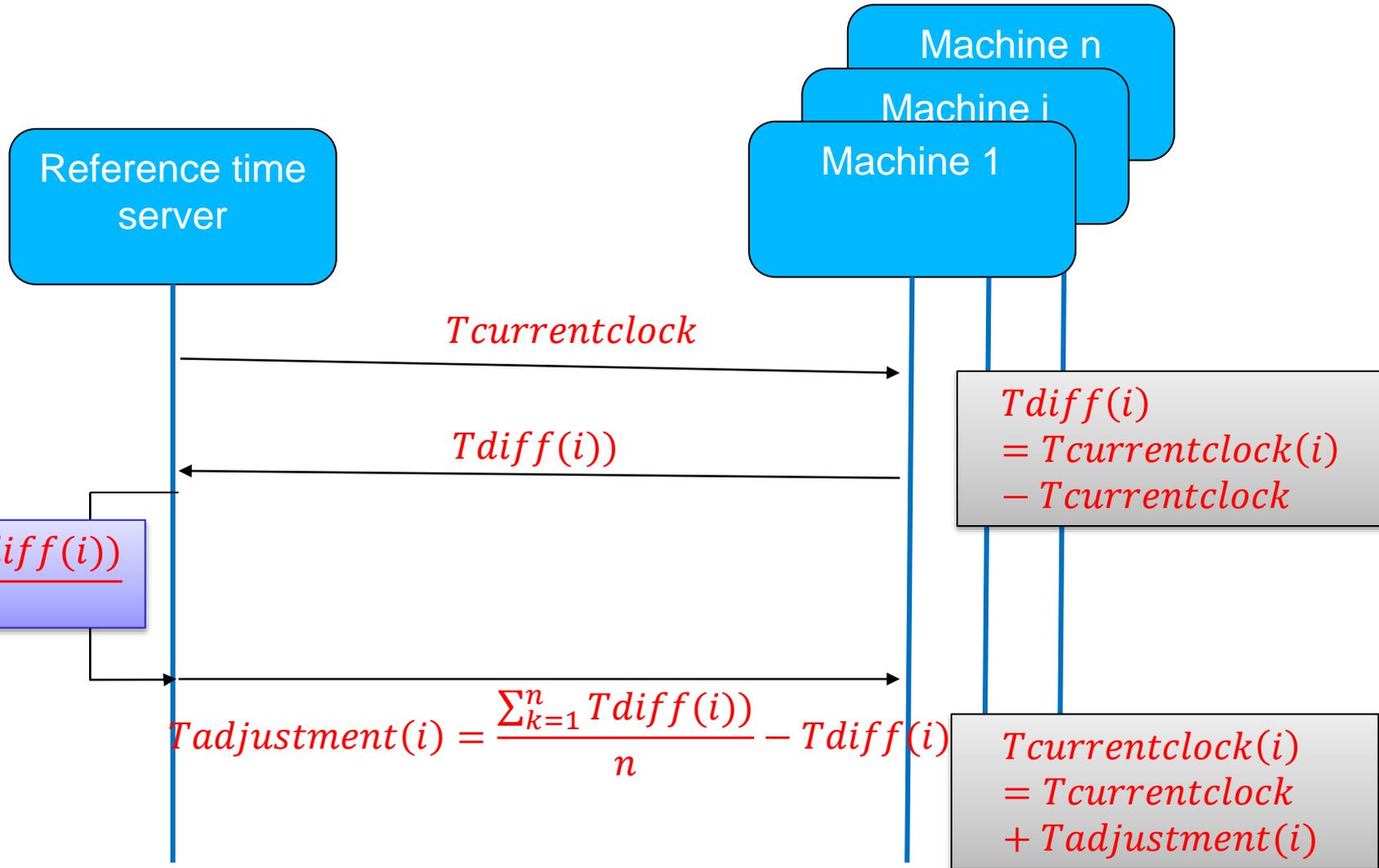
$$\max((t_e(A), (t_e(B) + \frac{RTT}{2})))$$

Q1: RTT is varying, how to improve the accuracy?

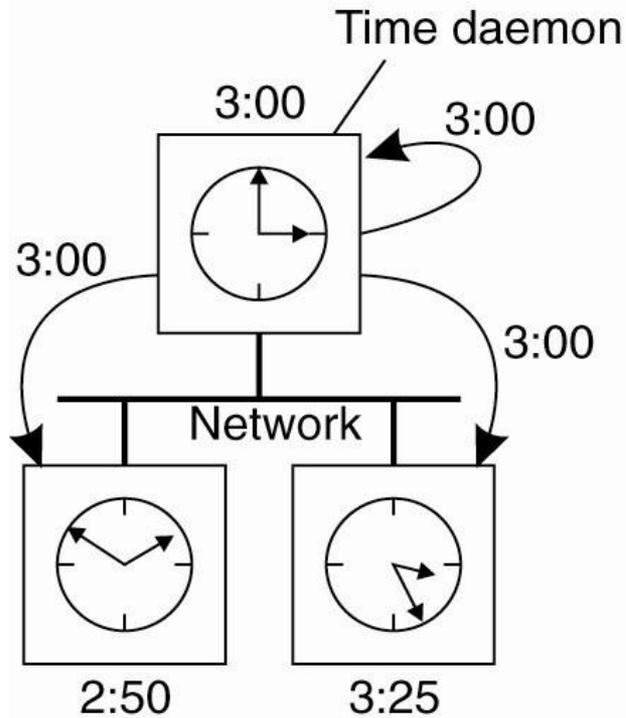
Q2: Drawbacks of this algorithm?

Q3: Assume we know the minimum time required for sending a message, Can you estimate the accuracy?

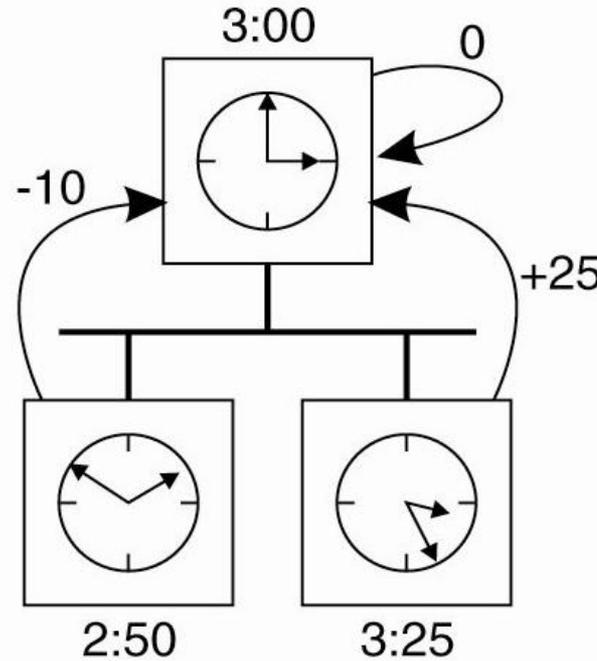
# Berkeley Algorithm



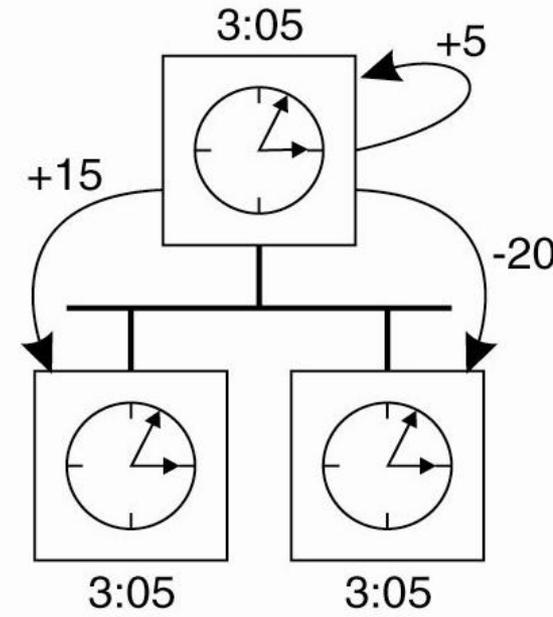
# Berkeley Algorithms



(a)



(b)



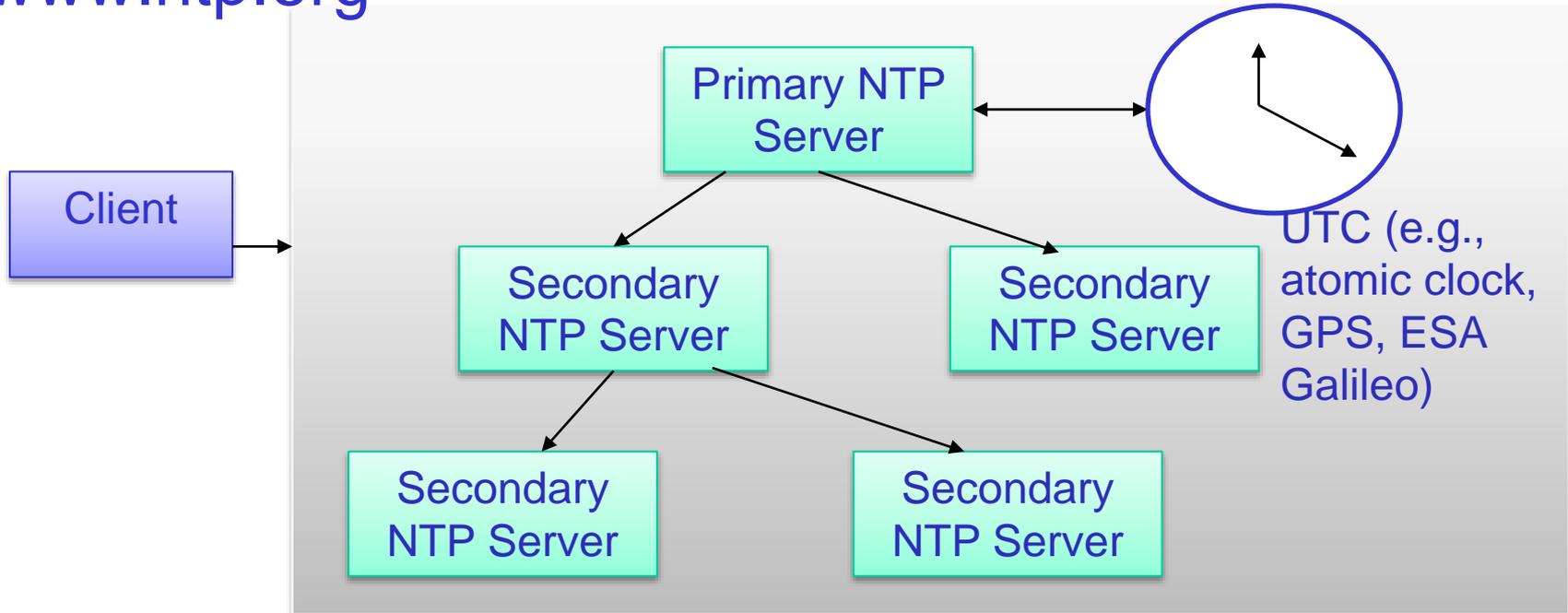
(c)

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Q: Why it is not good to use it outside LAN?

# Example: Network Time Protocol (NTP)

[www.ntp.org](http://www.ntp.org)



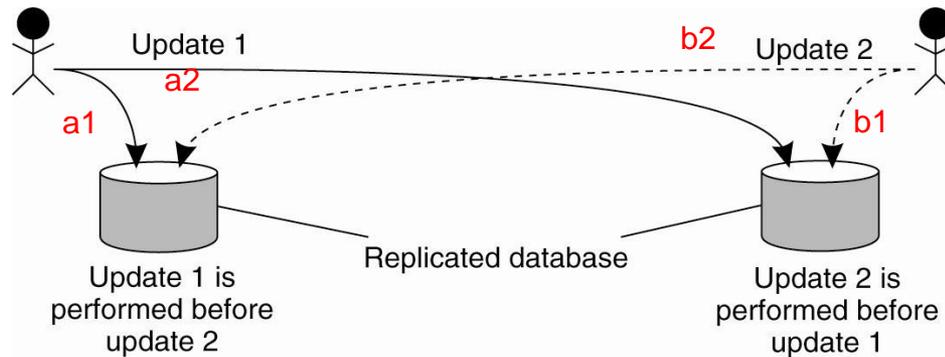
Protocol variants using unreliable communication (UDP):

- **Multicast** (servers send the time), **client/server** (similar to Cristina's algorithm), **symmetric** (between high and lower level server)

# LOGICAL CLOCKS

# Logical clocks

- In many cases: we do not need an exact physical timing, as long as we are able to maintain the physical causality



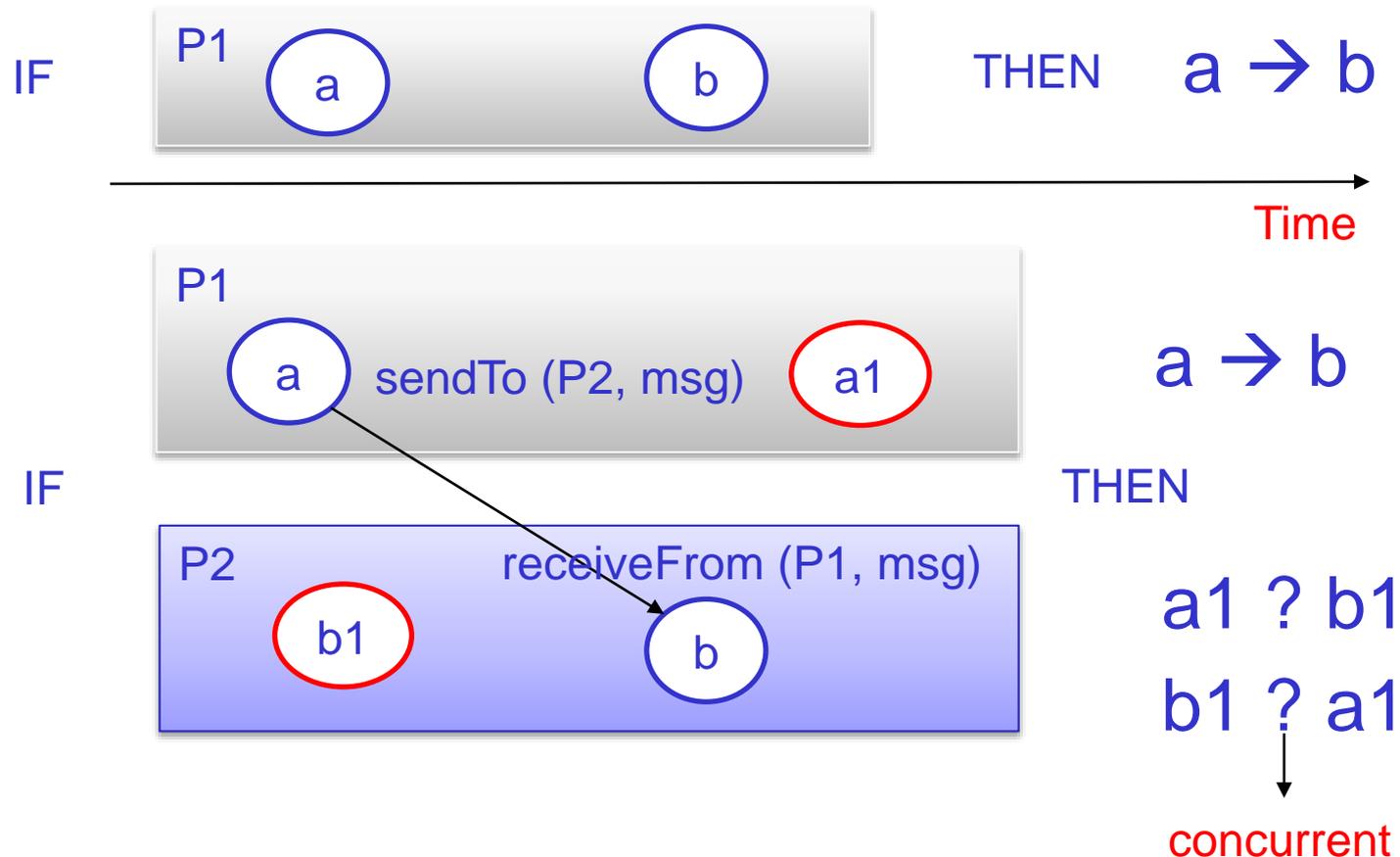
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Intention:  
We just need  
(a1,a2) being executed  
before (b1,b2) or  
another way around

Logical clock: using physical causality model for ordering events among distributed processes

# Happen-before relation

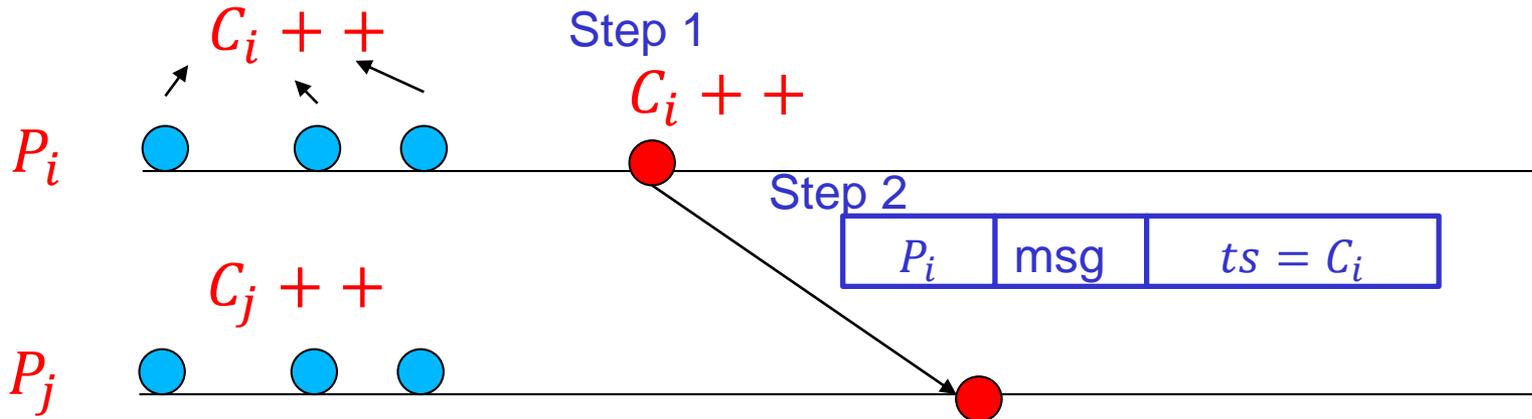
**Happen-before** ( $\rightarrow$ ) relation between **a and b** indicates that event **a** occurs before **b** **logically**. It is **possible that a affects b**



# Lamport's logical clock

- Used to synchronize a logical clock  $C_i$  of process  $P_i$

Increase the clock before executing an event



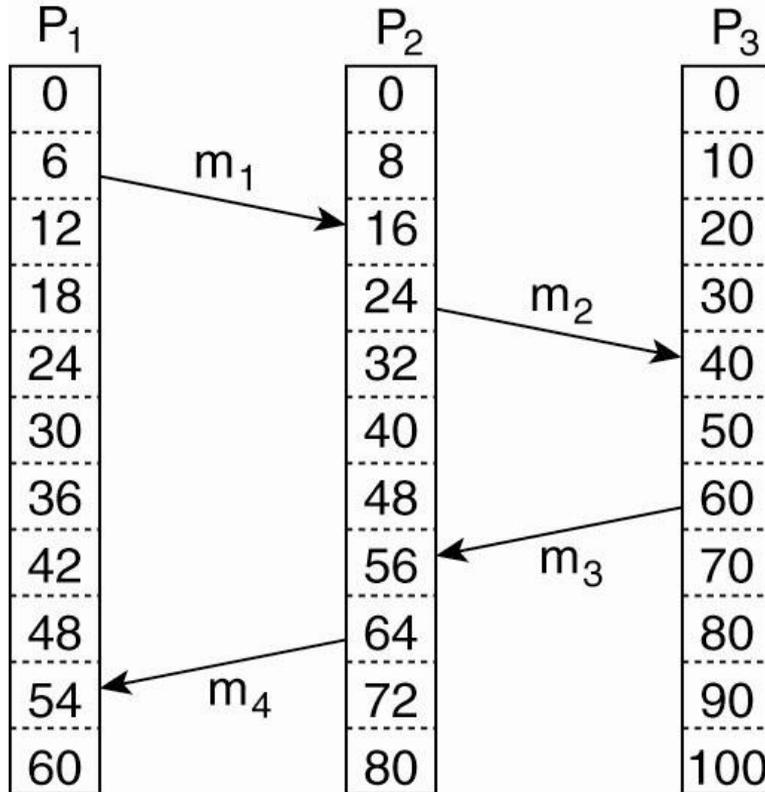
Step 1  $C_j = \max(C_j, ts)$

Step 2  $C_j ++$

Step 3: process the message

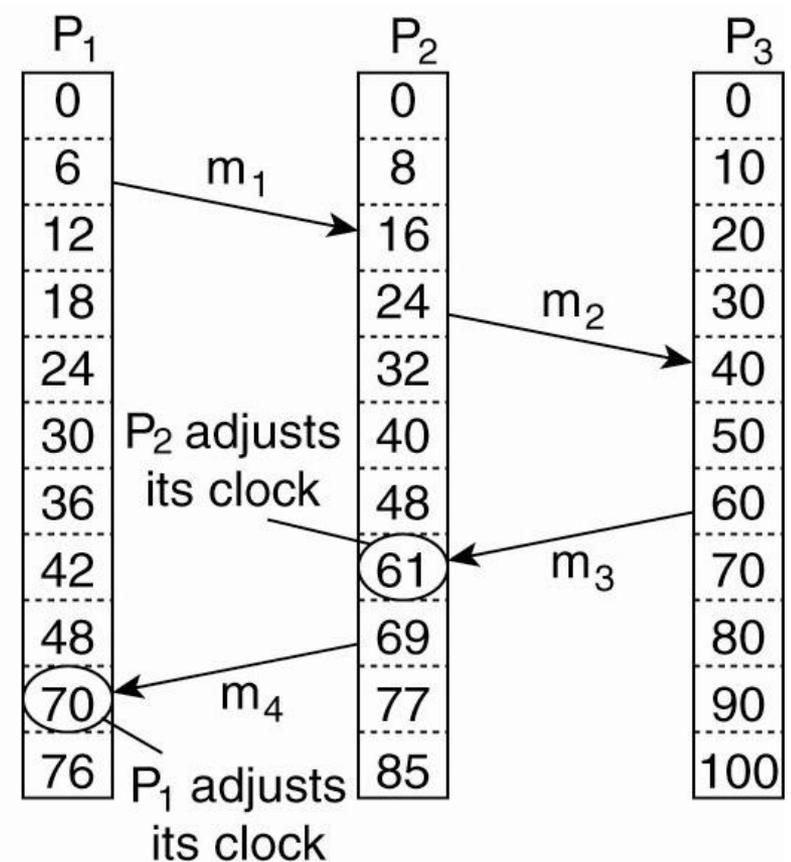
# Example of Lamport's logical clock

Without Lamport's logical clock



(a)

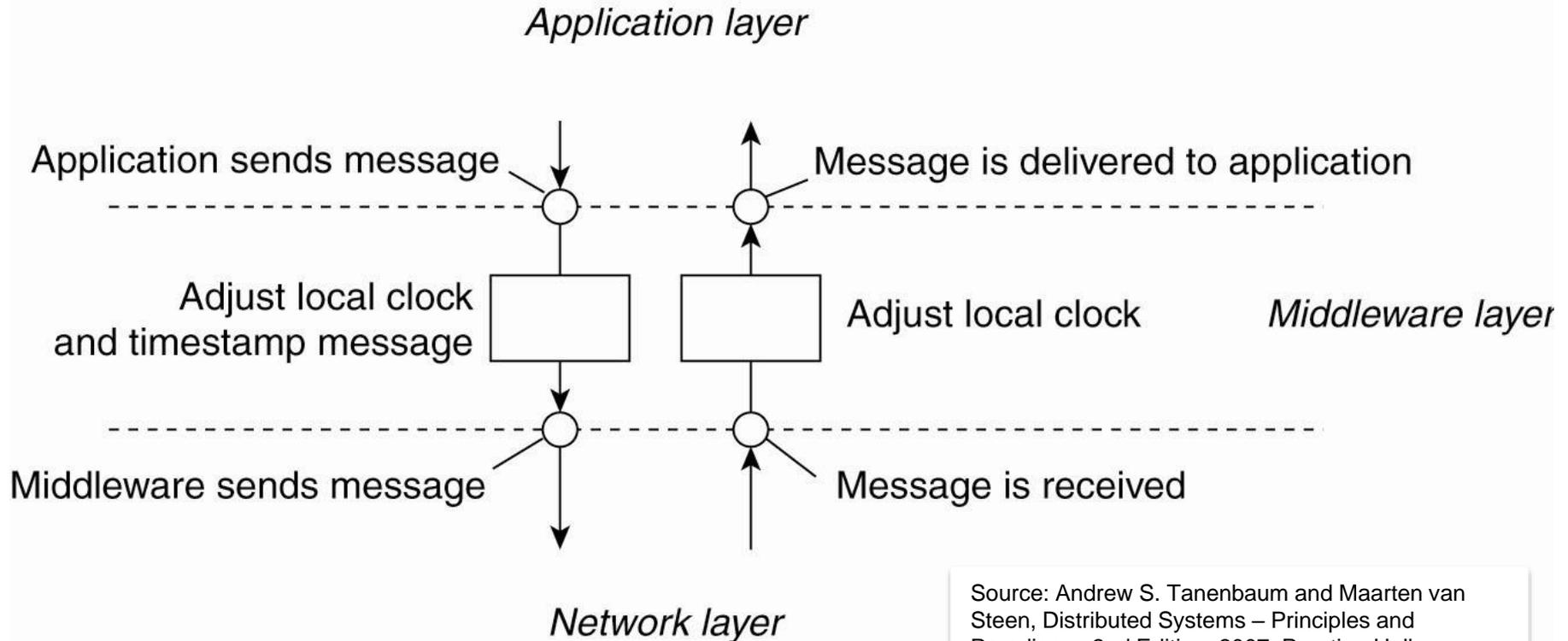
With Lamport's logical clock



(b)

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

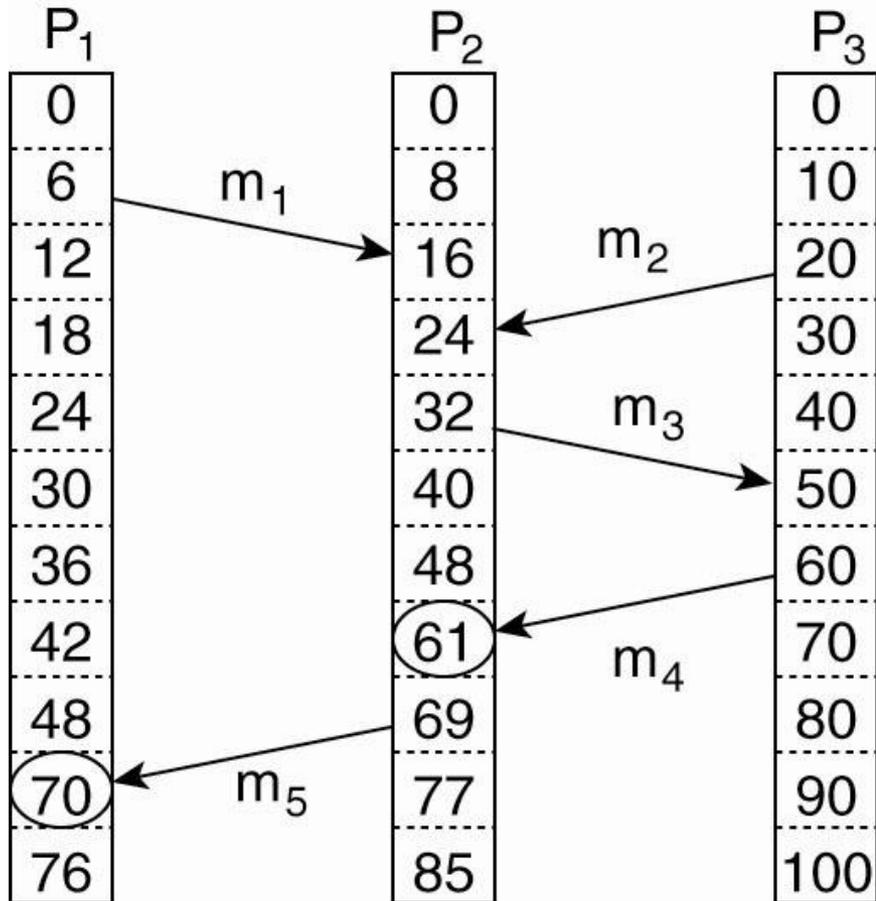
# Message interception and logical clock adjustment implementation



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

Home work: work out on in detail how Lamport's logical clock could be used for the update problem with replicated database

# Limitation of Lamport's logical clock



$recv(m_4) < send(m_5)$ :

Maybe m5 is dependent on m4 (causality)

$Recv(m_1) < send(m_2)$ :

We do not know their relationship

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

$C(a) < C(b) \neq a \rightarrow b$ , We miss causality information

# Vector Clocks

Goal: a vector clock (VC) allows us to interpret if  $VC(a) < VC(b)$  then a causally precedes b

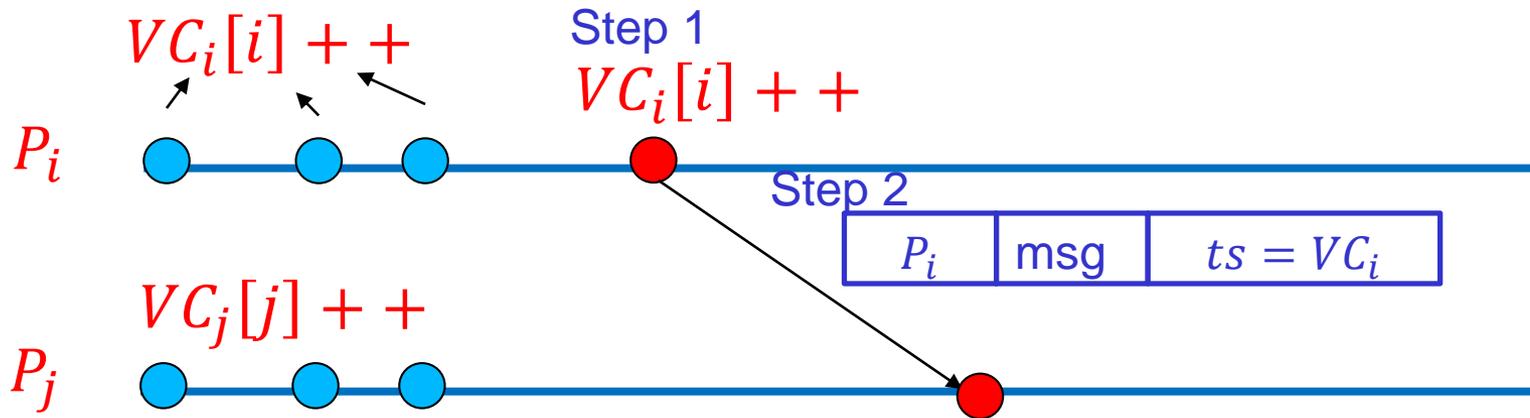
A process  $P_i$  maintains a vector clock  $VC_i$  where

- $VC_i[i]$  is the number of events happened in  $P_i$
- $VC_i[j] = k$  means that  $P_i$  knows there were **k events** occurred in  $P_j$  that have causal relation with  $P_i$

Implementation

- Each message is associated with a VC
- For event **a** and **b**, it is **possible** that **a affects b**, then  **$a.VC < b.VC$**

# Vector Clocks

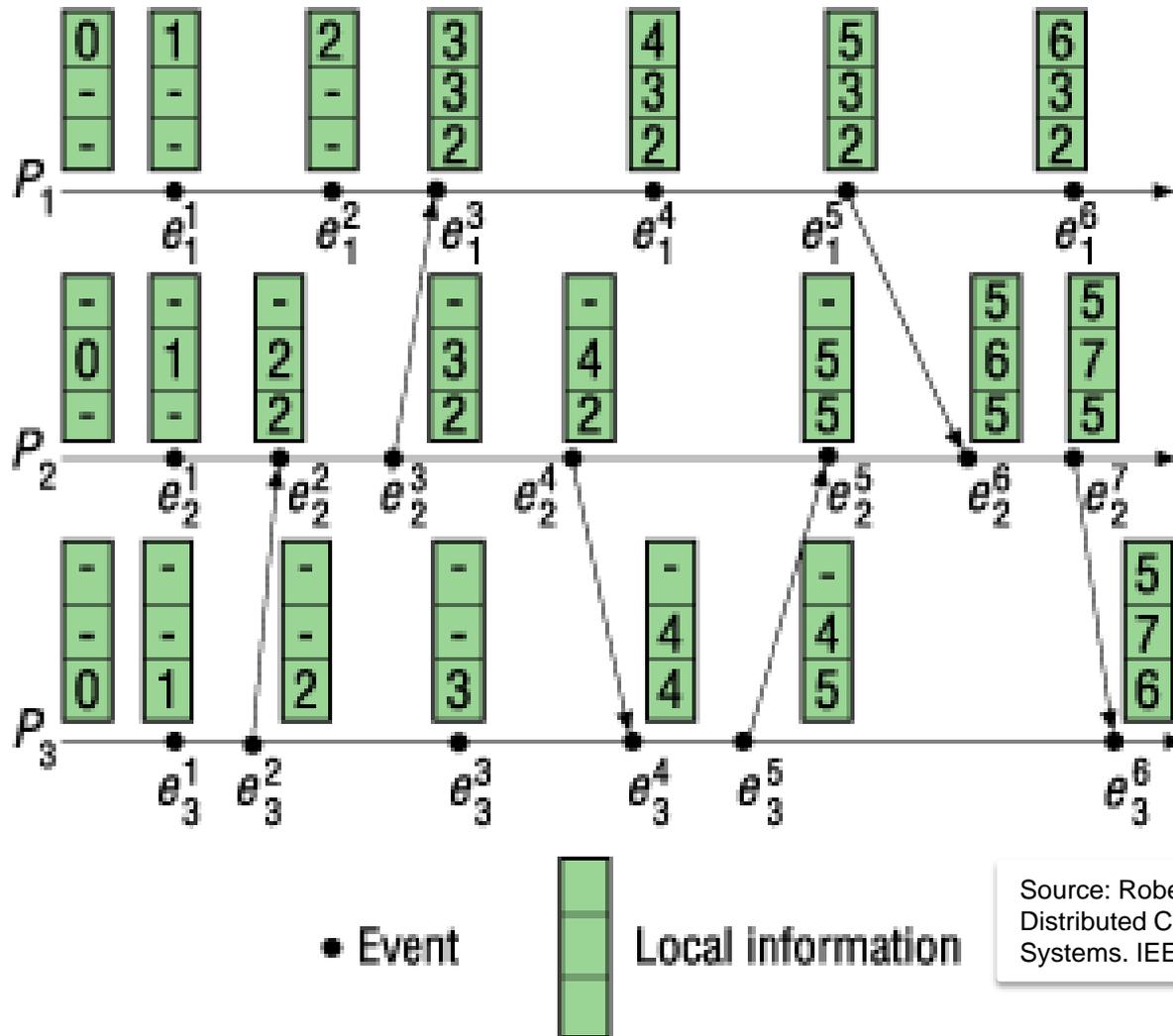


Step 1  $VC_j[k] = \max(VC_j[k], ts[k])$

Step 2  $VC_j[j] ++$

Step 3: process the message

# Example of vector clocks

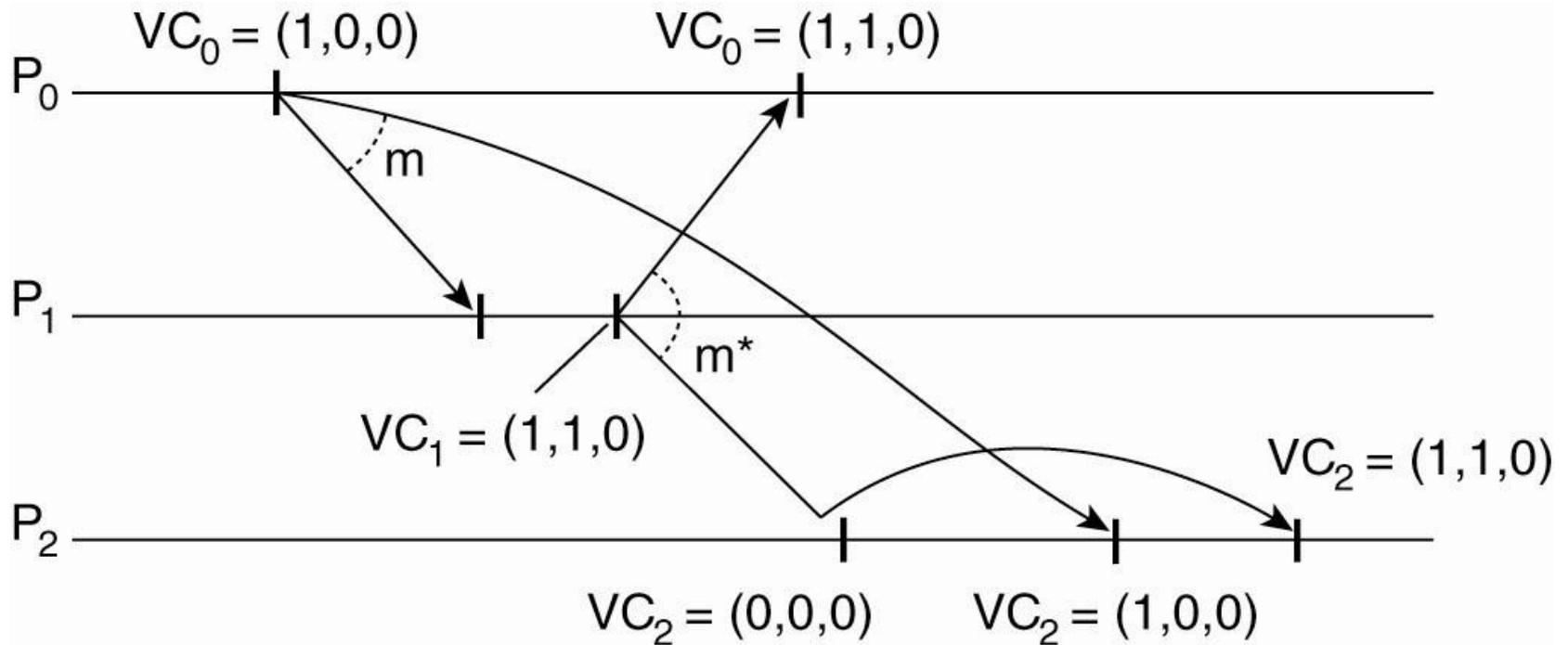


Source: Roberto Baldoni, Michel Raynal: Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. IEEE Distributed Systems Online 3(2) (2002)

# Applications of logical/vector clocks

- Replication by using totally order multicast
  - atomic multicast in which all members accept messages in the same order
- Multimedia real-time applications, teleconferencing using causal multicast
  - If  $\text{multicast}(m1) \rightarrow \text{multicast}(m2)$ , then  $(m1)$  must be delivered before  $m2$  for all processes

# Causal broadcast example



## Note

Upon sending a message  $P_i$  only increases  $VC_i[i]$  by 1

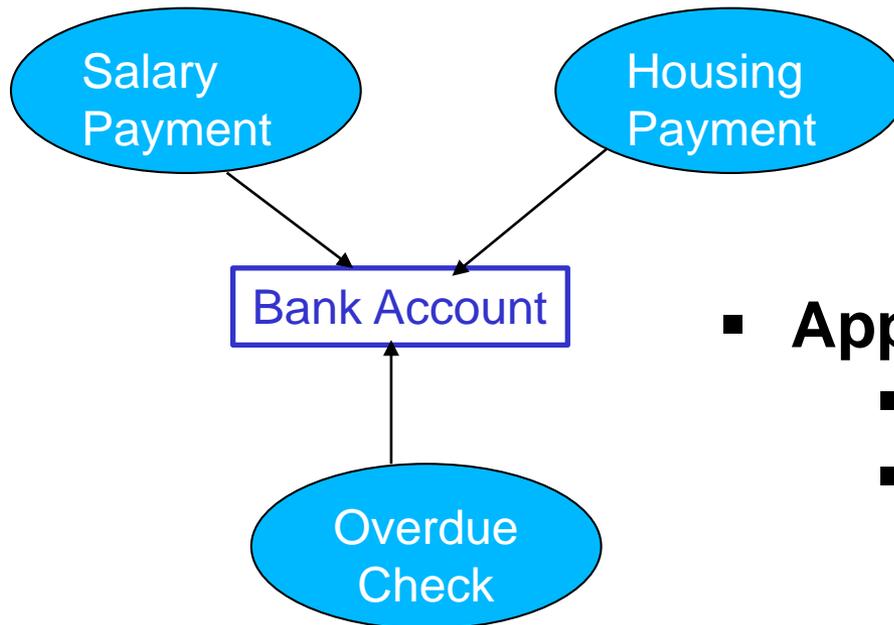
When receiving a message only adjust  $VC_j[k]$  to  $\max(VC_j[k], ts[k])$

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# MUTUAL EXCLUSION

# Mutual exclusion in distributed systems

- Multiple processes might access the same resource
- Mutual exclusion: prevent them to use the resource at the same time to avoid making resource inconsistent/corrupted



- **Approaches:**
  - Token-based
  - Permission-based

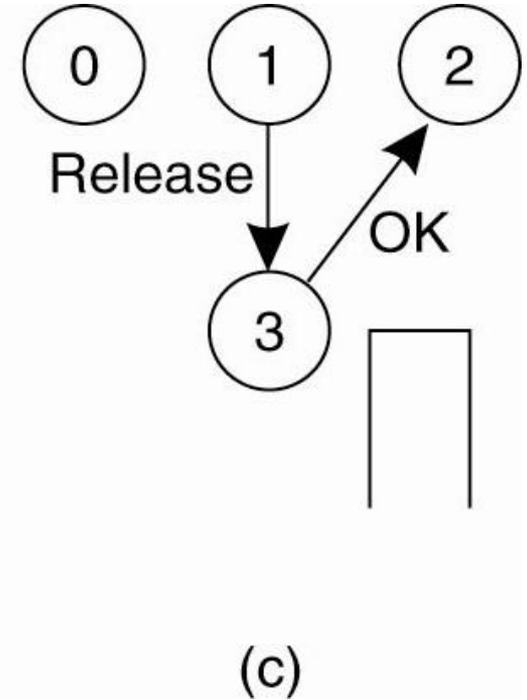
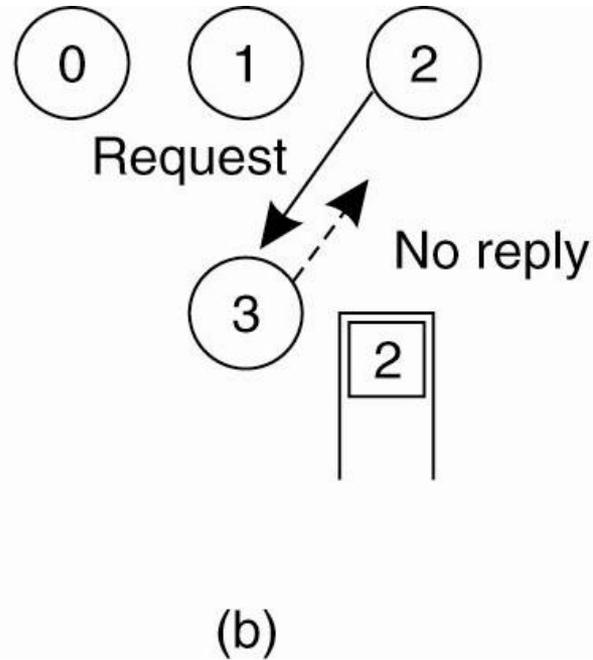
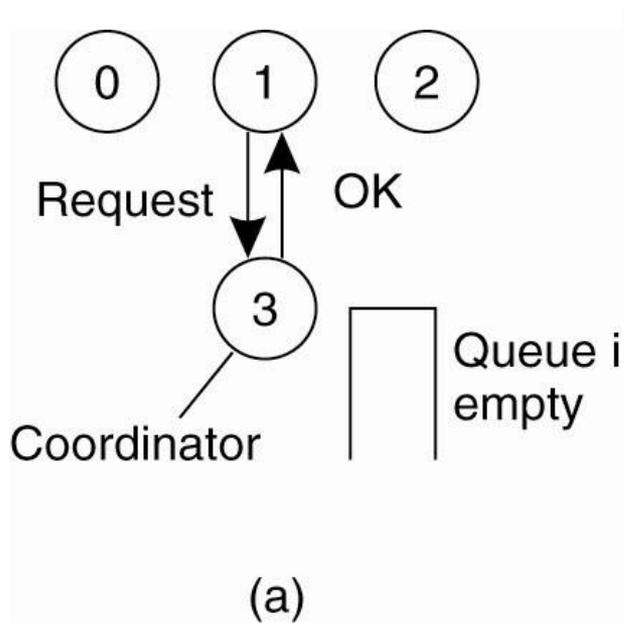
# Centralized Model

Permission-based approach: a deciated server gives permission, emulating the execution of critical section

```
private static Lock lock =new ReentrantLock();
public void criticalSection(){
    System.out.println("This is a critical section: access only with permission");
    System.out.println("==== I am "+id+" Waiting for the lock====");
    lock.lock();
    System.out.println("I am "+id+" I got the lock now");
    System.out.println(id + " doing some work ");
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
    }
    System.out.println("==== I am "+id+" releasing the lock====");
    lock.unlock();
}
```

<http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/exs/CriticalSectionExample.java>

# Centralized Model



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Example

- A very simple code
  - for a single resource using TCP communication
  - <http://www.infosys.tuwien.ac.at/teaching/courses/VerteilteSysteme/exs/CentralizedMutualExclusion.java>

```
java
at.ac.tuwien.dsg.dsexamples.Centr
alizedMutualExclusion localhost
4001 no tuwien
```



```
java
at.ac.tuwien.dsg.dsexamples.
CentralizedMutualExclusion
localhost 4001 yes null
```

Q1: What are main problems with this centralized model?

# Distributed algorithm (Ricart, Agrawala, Lamport)

Given a set of processes  $\{P_1, P_2, \dots, P_n\}$

If  $P_i$  wants to access a resource  $R$ ,  $P_i$  broadcast a message **msg(R,  $P_i$ , ts)**

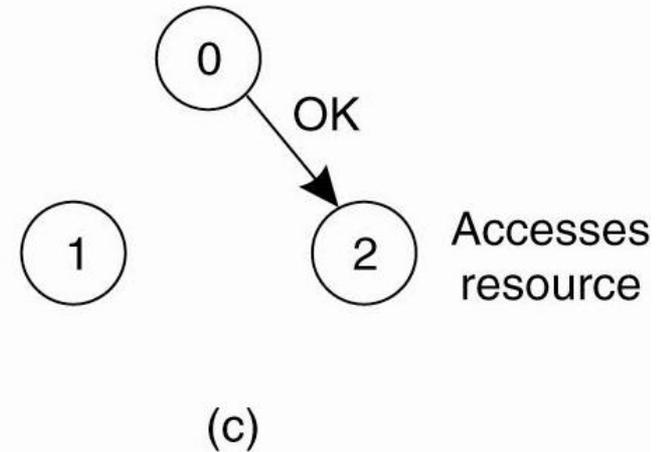
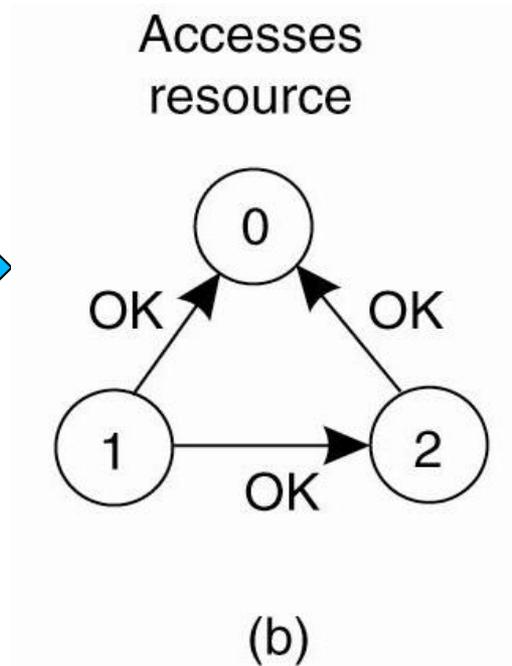
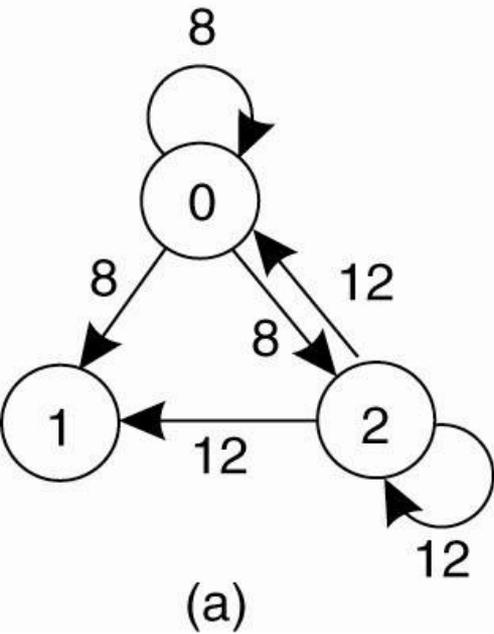
If  $P_j$  receives **msg(R,  $P_i$ , ts)** then

- No interest, no access  $\rightarrow$  return „OK“
- **Already access R** then does not reply by putting the msg into the queue
- If already sent **msg(R,  $P_j$ , tsj)** but **has not accessed R**:
  - If **ts < tsj** then returns „OK“, otherwise put it in queue

If  $P_i$  **gets all OK** then it can access the resource after that it sends an OK to all

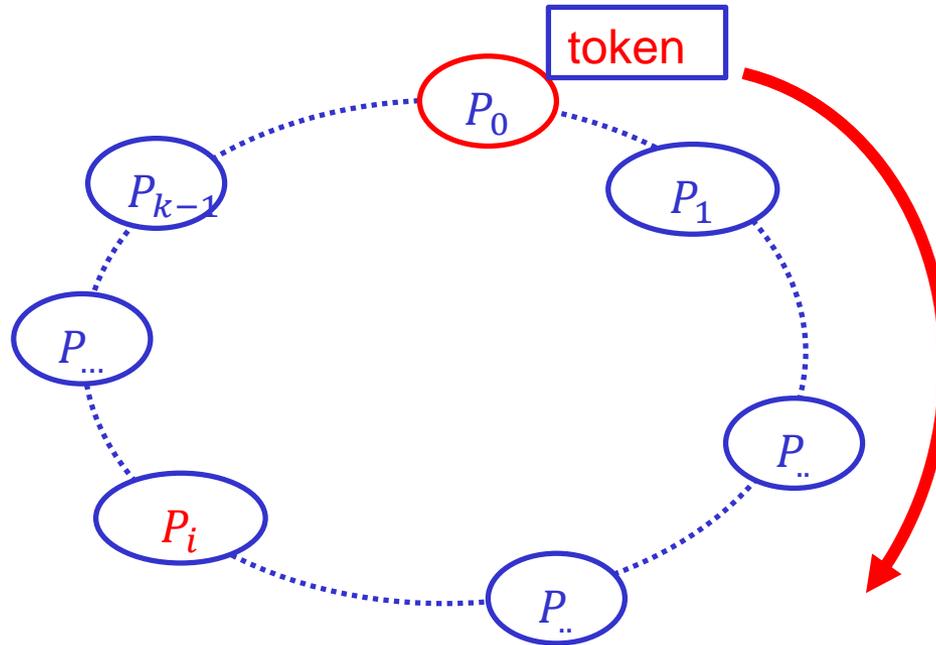
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Example



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Ring algorithm



When  $P_i$  receives the token:

1. Access the resource and release resource and pass the token
2. Otherwise just pass the token

# ELECTION ALGORITHMS

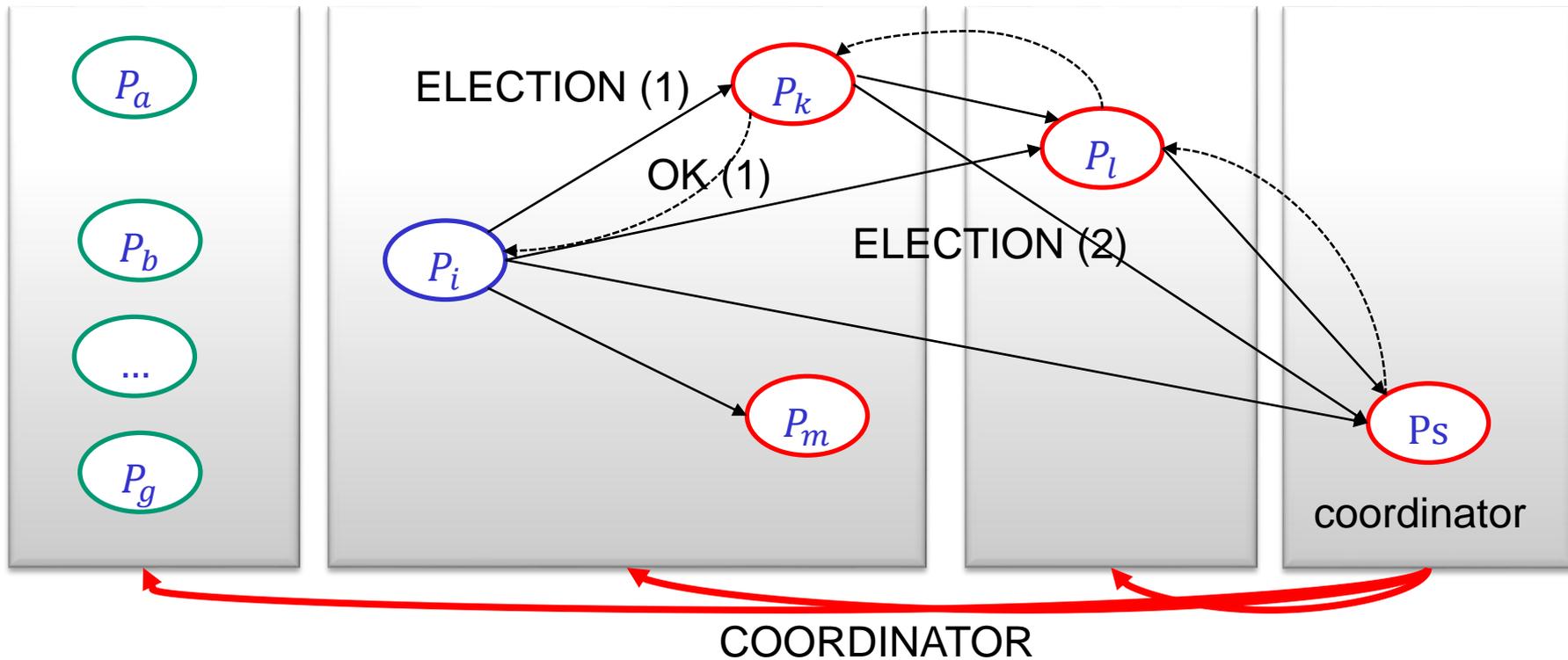
# Leader election

- In many situations we need a **coordinator**
  - **The coordinator is selected from a set of processes**
- Why it is challenging to elect a coordinator?
  - Distributed, multiple processes involvement
- Election algorithms
  - Designed for electing leaders
  - Processes are uniquely identified, e.g., using process id
  - Election process occurs when
    - Initiating the systems, existing coordinator failed, etc.

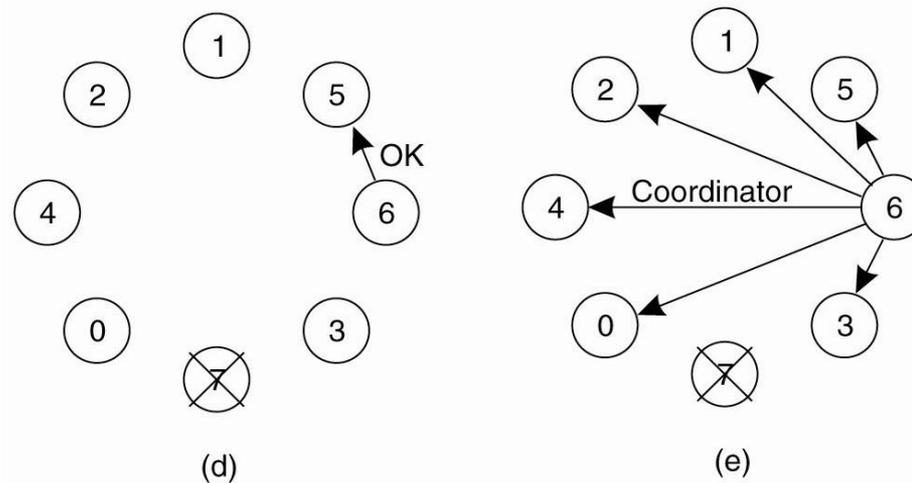
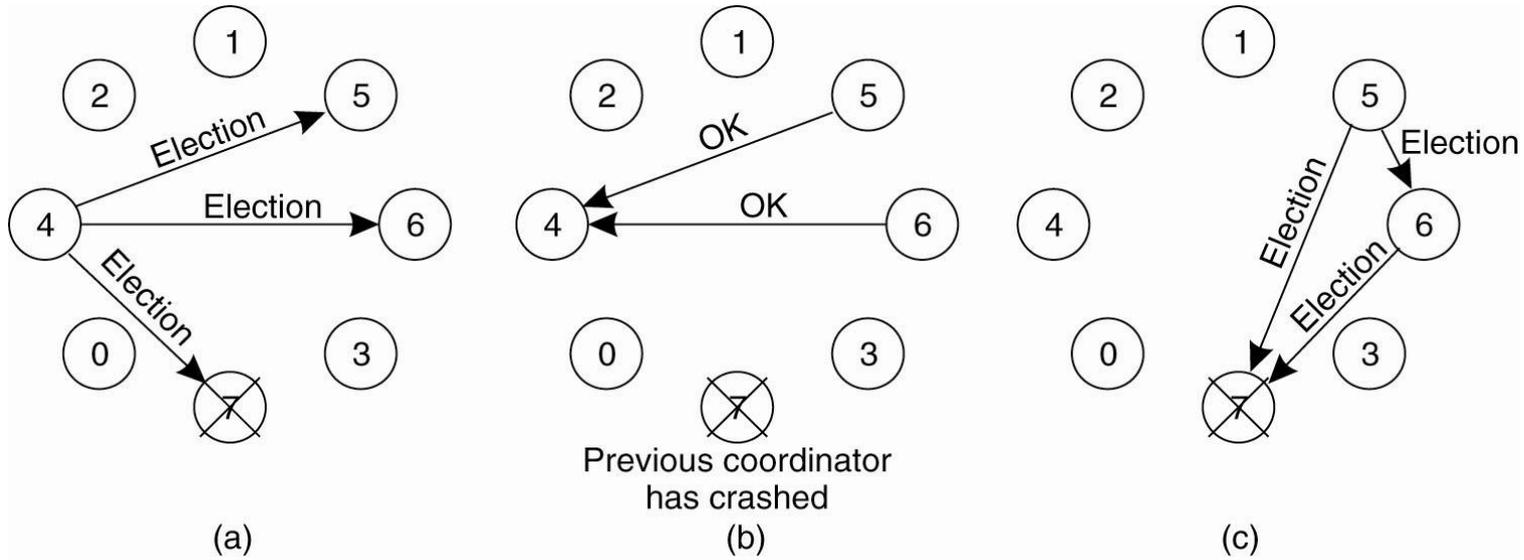
# Bully algorithm

Lower rank processes

Higher rank processes



# Example



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Ring algorithm

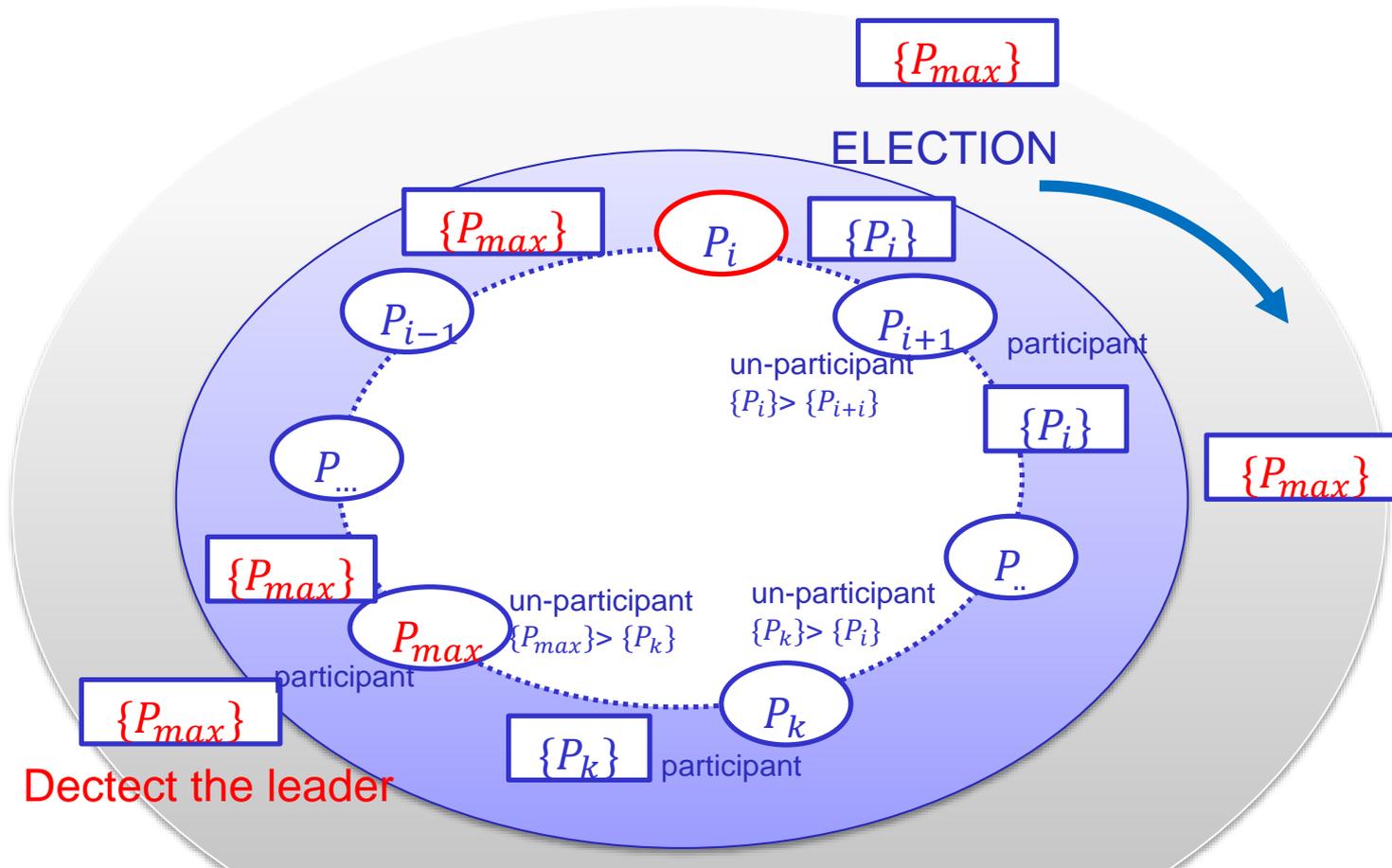
- From Le Lann, Chang and Roberts
- Processes are organized into a ring, initially „non-participant“ in the election
- Election message (ELECTION) and elected message (COORDINATION)
- Messages are forwarded or created and sent clockwise

Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

George Coulouris, Jean Dollimore, Tim Kindberg, „Distributed Systems – Concepts and Design“, 2nd Edition, Chapter 10

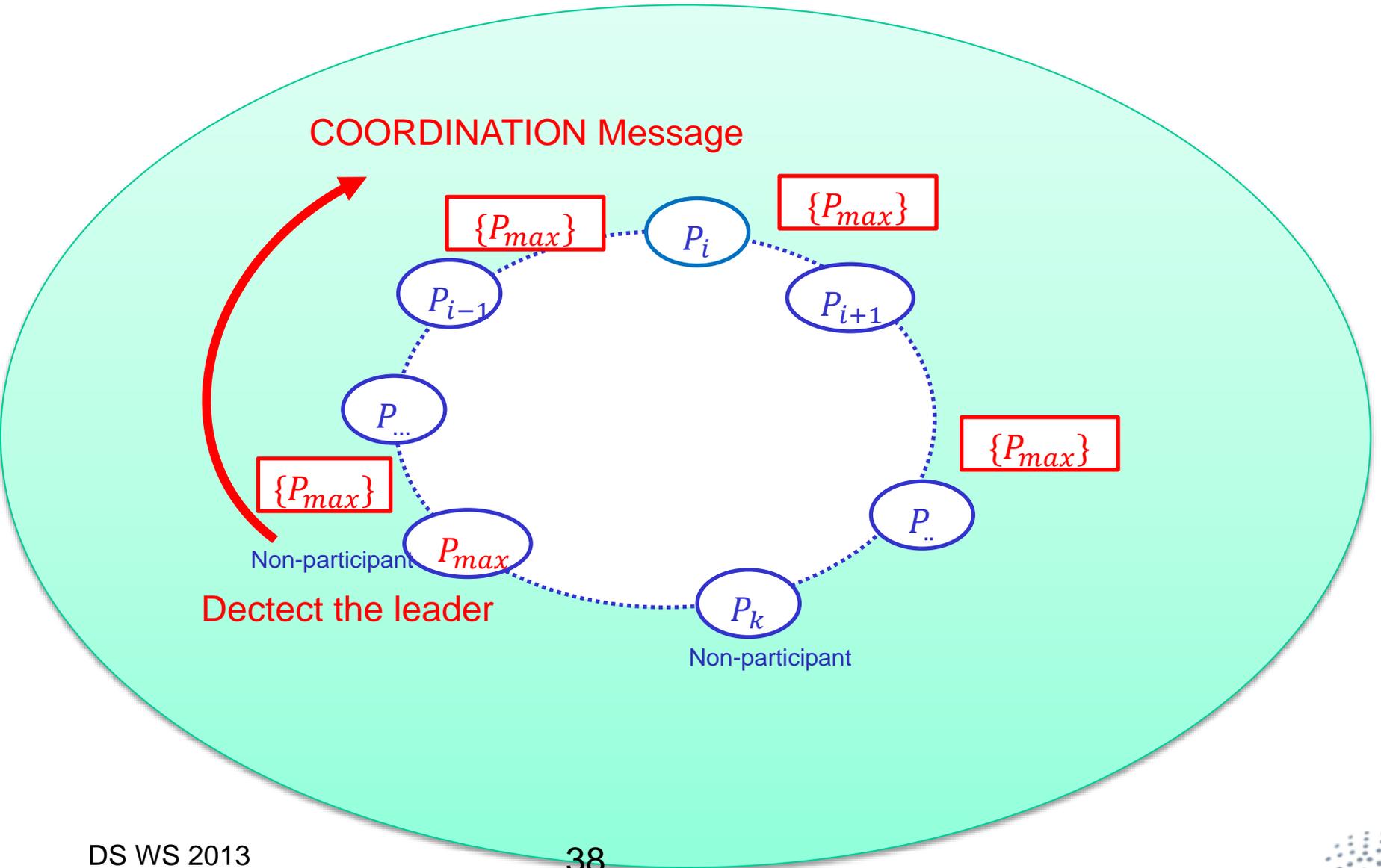
Nancy A Lynch, Distributed Algorithms, 1996, Chapter 3.

# Ring algorithm



Q1: if  $P_k$  receives another ELECTION message with a smaller identifier after becoming participant, what should it do?

# Ring algorithm



# Simple Flooding Algorithm

**Assumption:** processes are structured into a directed graph

## Steps

- **P** maintains the maximum unique process identifier (UID) it knows
- At a round, each **P** sends this UID to all **nodes in its outgoing edges**
- After **n** rounds, if a process **P** sees **its ID equal to the maximum UID**, then the process becomes the leader

Source: Nancy A Lynch, Distributed Algorithms, 1996, Chapter 4.

# Summary

- Time synchronization is important in real-world
  - But complex problem in distributed systems
  - Different algorithms with different pros and cons
- Logical clocks are useful in many situations
  - Happen-before or physical causality is the main principle
- Distributed coordination needs both mutual exclusion and election mechanism
- Dont forget to analyze algorithms to understand their pros and cons

# Thanks for your attention

Hong-Linh Truong  
Distributed Systems Group  
Vienna University of Technology  
[truong@dsg.tuwien.ac.at](mailto:truong@dsg.tuwien.ac.at)  
<http://dsg.tuwien.ac.at/staff/truong>

# Distributed Systems Principles and Paradigms

Christoph Dorn

Distributed Systems Group,  
Vienna University of Technology

`c.dorn@infosys.tuwien.ac.at`

`http://www.infosys.tuwien.ac.at/staff/dorn`

Slides adapted from Maarten van Steen, VU Amsterdam, `steen@cs.vu.nl`

## Chapter 07: Consistency & Replication



<b>Chapter</b>
01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
<b>07: Consistency &amp; Replication</b>
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems



# Consistency & replication

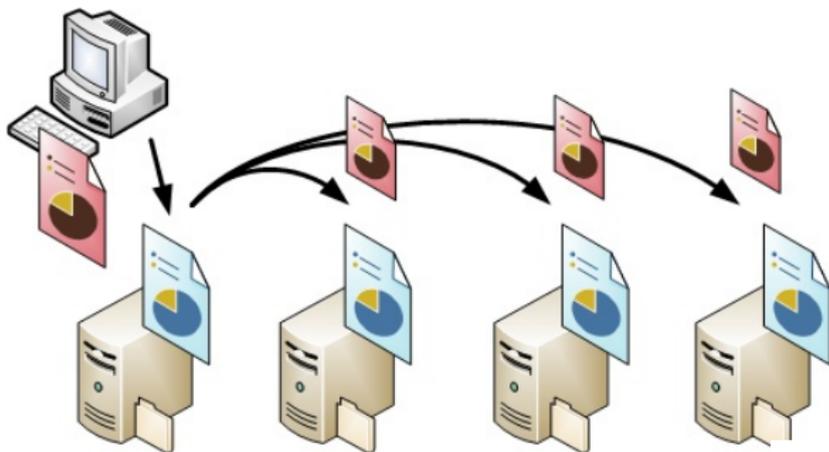
- Introduction (what's it all about)
- Data-centric consistency
- Client-centric consistency
- Replica management
- Consistency protocols



# What is Consistency and Replication?

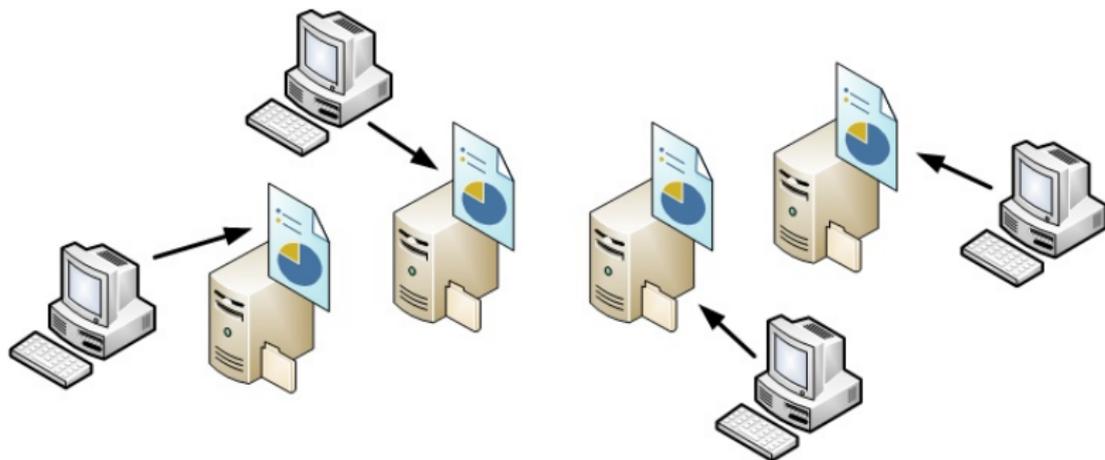
## Some good enough definitions

- Replication is the process of maintaining several copies of an data item at different locations.
- Consistency is the process of keeping data item copies the same when changes occur.



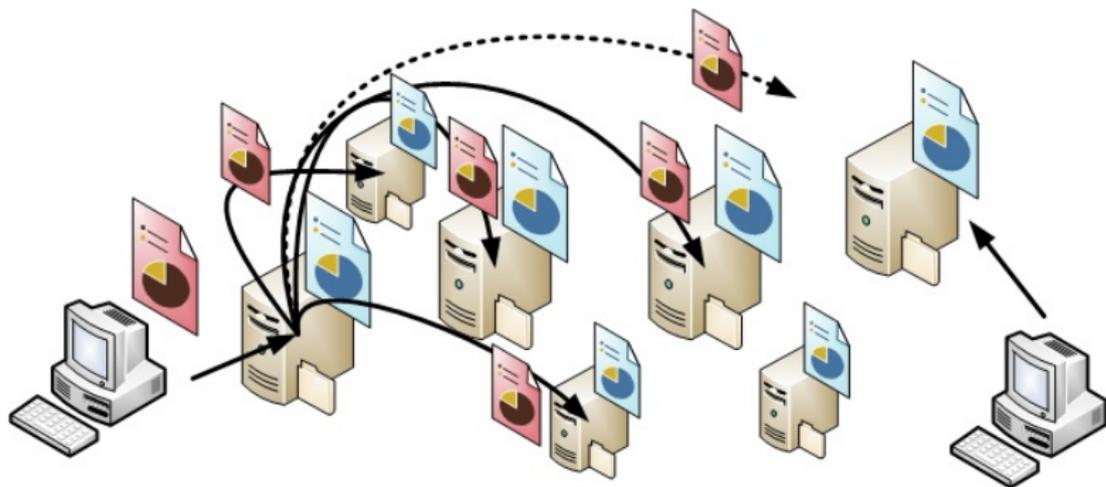
## Benefits

- More replicas can serve more client requests.
- Replicas close to the client improves response time/reduces bandwidth.



## Drawbacks

- Keeping replicas up to date consumes bandwidth
- Updates to replicas are not immediately propagated (stale data)



# A cure potentially worse than the disease

## Replication - pick any two:

- Performance: low response time (for reading and writing)
- Scalability: support a lot of clients
- Consistency: any update should be reflected at all replicas else before any subsequent operation takes place

## Synchronous Replication issue

Updates performed as a single atomic operation (transaction) requires agreement of all replicas when to perform the update. Becomes extremely costly very quickly.

## Mitigation

Avoid (instantaneous) global synchronization



# Maintaining Performance and Scalability

## Main issue

To keep replicas consistent, we generally need to ensure that all **conflicting** operations are done in the the same order everywhere

## Conflicting operations

From the world of transactions:

- **Read–write conflict**: a read operation and a write operation act concurrently
- **Write–write conflict**: two concurrent write operations

## Issue

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability

**Solution**: weaken consistency requirements so that hopefully global synchronization can be avoided



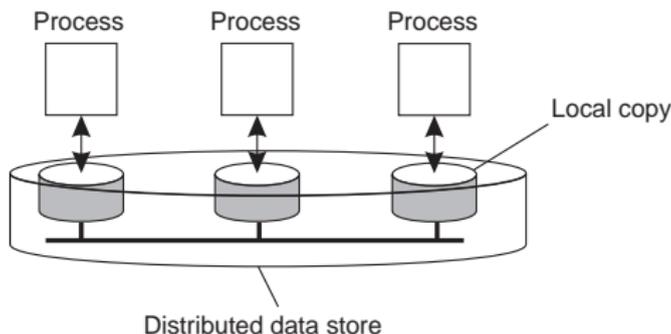
# Data-centric consistency models

## Consistency model

A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

## Essential

A data store is a distributed collection of storages:



## Observation

We can actually talk about a **degree of consistency**:

- replicas may differ in their **numerical value**
- replicas may differ in their relative **staleness**
- there may be differences with respect to (number and order) of **performed update operations**

## Conit

Consistency unit  $\Rightarrow$  specifies the **data unit** over which consistency is to be measured.

## Conit examples

webpage, table entry, entire table in DB, ...



# Continuous Consistency Example

## Conit Example

Consistency unit in our example is the price of a particular stock.

## Example constraints

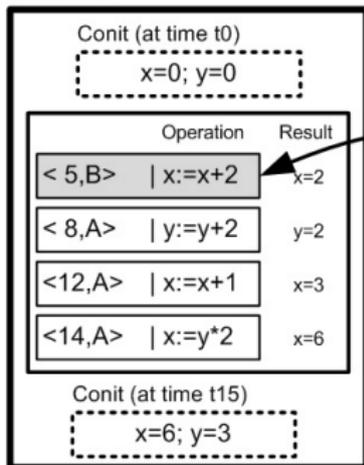
Specify **degree of consistency** for stock price:

- Local value may differ in **numerical value** from other replica by 10 cents
- Local value needs to be checked for **staleness** at least every 10 seconds
- There may be no more than 3 unseen **performed update operations**

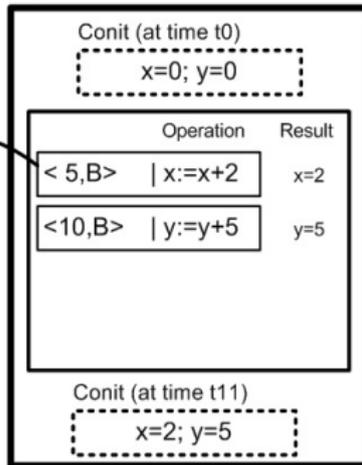


# Example: Conit

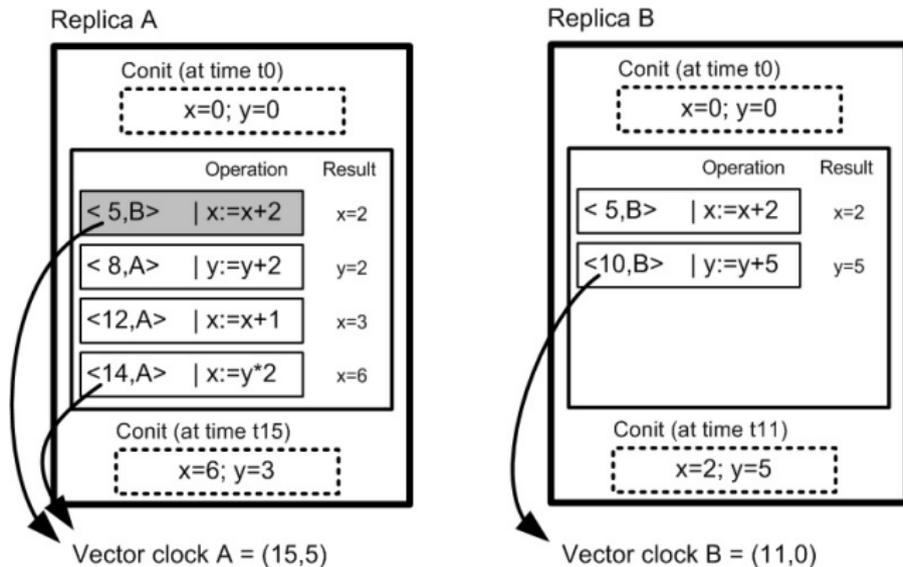
Replica A



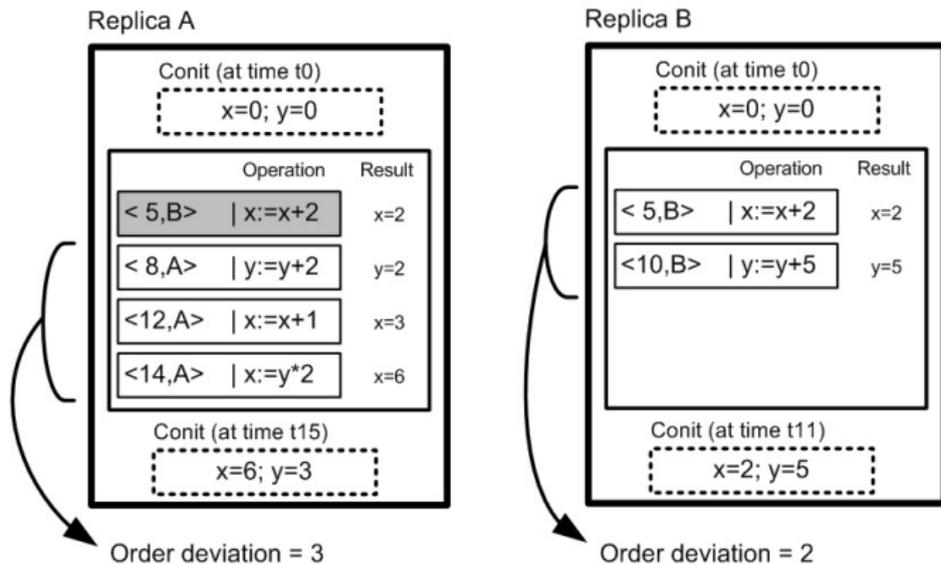
Replica B



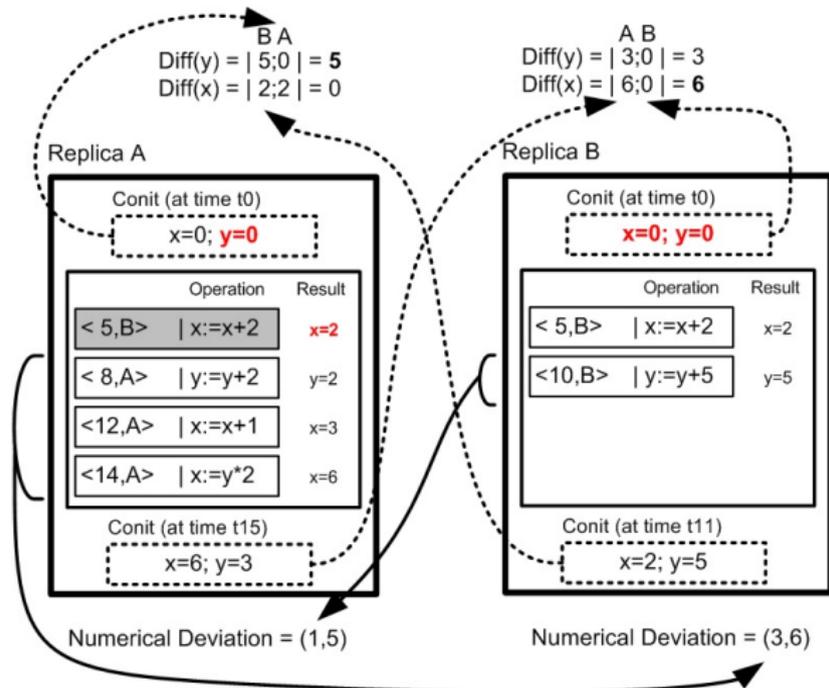
# Example: Conit



# Example: Conit



# Example: Conit



## Desired Behavior

read returns result of most recent write

## No global clock!

What is the **most recent** (last) write?

## Relax timing

- consider intervals of R/W operations
- define precisely what are acceptable behavior for conflicting operations
- replicas need to agree on consistent global ordering of updates



## Definition

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

P1: W(x)a			
P2: W(x)b			
P3: R(x)b	R(x)a		
P4: R(x)b	R(x)a		

(a)

P1: W(x)a			
P2: W(x)b			
P3: R(x)b	R(x)a		
P4: R(x)a	R(x)b		

(b)

(a) sequentially consistent, (b) not consistent

## Example

- Assume  $x$  is a shared social network timeline
- Peter posts: I'm going skiing, who's in?
- Paul posts: I'm going hiking, who's in?
- Petra reads: first Paul's, then Peter's post
- Pam reads: first Paul's, then Peter's post

## Beware

To be sequentially consistent: every reader of the timeline needs to receive the updates in exactly the same order.



## Example

- Assume  $x$  is a shared social network timeline
- Peter posts: I'm going skiing, who's in?
- Paul posts: I'm going hiking, who's in?
- Petra reads: first Paul's, then Peter's post
- Pam reads: first Paul's, then Peter's post

## Beware

To be sequentially consistent: every reader of the timeline needs to receive the updates in exactly the same order.

## Definition

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order by different processes.

P1: W(x)a	W(x)c
P2: R(x)a	W(x)b
P3: R(x)a	R(x)c R(x)b
P4: R(x)a	R(x)b R(x)c

(a) causally consistent

P1: W(x)a	W(x)c
P2: R(x)a	W(x)b
P3: R(x)a	R(x)c R(x)b
P4: R(x)a	R(x)b R(x)c

(b) causally consistent

# Causal consistency - more examples

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a				
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

(a) causally inconsistent, (b) consistent

## Example

- Again, assume  $x$  is a shared social network timeline
- Peter posts: I had a car accident!
- Peter posts: But I'm ok!
- Paul read this and posts: Happy for you!
- Petra reads: Peter's first post, the Paul's, the Peter's second
- Pam reads: both Peter's, then Paul's post

## Beware

How to determine causally related writes?



## Example

- Again, assume  $x$  is a shared social network timeline
- Peter posts: I had a car accident!
- Peter posts: But I'm ok!
- Paul read this and posts: Happy for you!
- Petra reads: Peter's first post, the Paul's, the Peter's second
- Pam reads: both Peter's, then Paul's post

## Beware

How to determine causally related writes?





## Implications

- Easy to implement: Writes from different processes are always assumed "concurrent".
- Different processes may see the statements executed in different order
- Some results may be counterintuitive

## Example

- Process P1:  $x := 1$ ; if  $(y == 0)$  kill (P2);
- Process P2:  $y := 1$ ; if  $(x == 0)$  kill (P1);

## Effect

Two concurrent processes, **both may be killed** with FIFO (but not with sequential consistency).



## Get ready!

From time to time I will do **Simultaneous Voting** to check whether the presented concepts are clear to everyone.

## Will you attend the lecture next Monday?

Your choices are:

- Sure, I just love Distributed Systems! (head)
- Not sure yet, do I really need to? (ear)
- No way, Garfield is my second name! (nose)



# Know your Consistency Models

## Question to the audience

Observe following read/write events. The sequence is only valid for one of the following consistency models, which one?

## Your choices are:

- Sequential Consistency (head)
- Causal Consistency (ear)
- FIFO Consistency (nose)

P1: W(x)a		W(x)c			
P2: R(x)a	W(x)b				
P3:		R(x)a		R(x)c	R(x)b
P4:		R(x)c	R(x)a		R(x)b

Answer: causal consistency



# Know your Consistency Models

## Question to the audience

Observe following read/write events. The sequence is only valid for one of the following consistency models, which one?

## Your choices are:

- Sequential Consistency (head)
- Causal Consistency (ear)
- FIFO Consistency (nose)

P1: W(x)a						W(x)c
P2:	R(x)a	W(x)b				
P3:			R(x)a		R(x)c	R(x)b
P4:			R(x)c	R(x)a		R(x)b

Answer: causal consistency



## Definition

- Accesses to **synchronization variables** are sequentially consistent.
- No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.



## Definition

- Everyone has exactly the same view on a lock (a **synchronization variable**)
- Have a lock: Cannot unlock until data value is synchronized everywhere
- Grab a lock: then only allowed to proceed when everyone has the same view on the lock

## Basic idea

You don't care that reads and writes of a **series** of operations are immediately known to other processes. You just want the **effect** of the series itself to be known.



## Definition

- Everyone has exactly the same view on a lock (a **synchronization variable**)
- Have a lock: Cannot unlock until data value is synchronized everywhere
- Grab a lock: then only allowed to proceed when everyone has the same view on the lock

## Basic idea

You don't care that reads and writes of a **series** of operations are immediately known to other processes. You just want the **effect** of the series itself to be known.



# Grouping operations

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)		
P2:	Acq(Lx) R(x)a	R(y) NIL
P3:	Acq(Ly) R(y)b	

## Observation

Weak consistency implies that we need to lock and unlock data (implicitly or not).

## Question

Why do we need a lock here?

The underlying distributed system might decide to push updates to all replicas after lock release OR not until a new lock is acquired.

# Grouping operations

P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)		
P2:	Acq(Lx) R(x)a	R(y) NIL
P3:	Acq(Ly) R(y)b	

## Observation

Weak consistency implies that we need to lock and unlock data (implicitly or not).

## Question

Why do we need a lock here?

The underlying distributed system might decide to push updates to all replicas after lock release OR not until a new lock is acquired.



## Concurrent processes

- So far required simultaneous updates of shared data
- Consistency and Isolation have to be maintained
- Synchronization required

## Lack of concurrent processes

- Lack of simultaneous updates (or distinct update regions)
- Easy resolved or acceptable inconsistencies
- Focus on guarantees for a **single (mobile)** client (but not for concurrent access)!



## Overview

- System model
- Monotonic reads
- Monotonic writes
- Read-your-writes
- Write-follows-reads

## Goal

Show how we can perhaps avoid systemwide consistency, by concentrating on what specific **clients** want, instead of what should be maintained by servers.



## Example

Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.

- At location *A* you access the database doing reads and updates.
- At location *B* you continue your work, but unless you access the same server as the one at location *A*, you may detect inconsistencies:
  - your updates at *A* may not have yet been propagated to *B*
  - you may be reading newer entries than the ones available at *A*
  - your updates at *B* may eventually conflict with those at *A*



# Consistency for mobile users

## Note

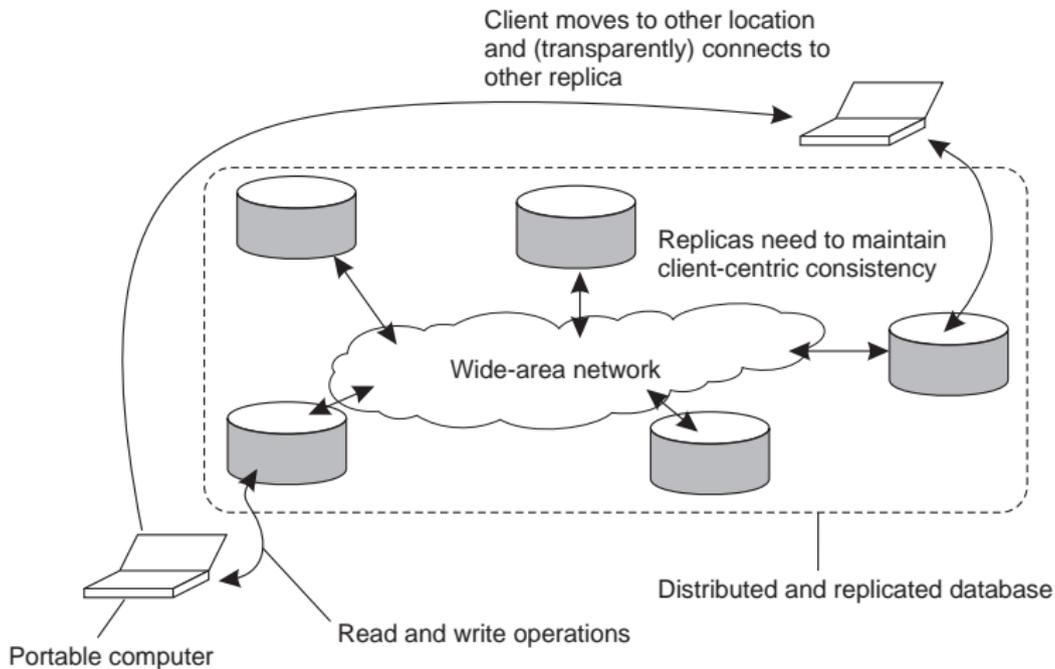
The only thing you really want is that the entries you updated and/or read at  $A$ , are in  $B$  the way you left them in  $A$ . In that case, the database will appear to be consistent to you.

## Eventual Consistency

- Update is performed at one replica (at a time)
- Propagation to other replicas is performed in a lazy fashion
- Eventually, all replicas will be updated
- I.e., replicas gradually become consistent if no update takes place for a while

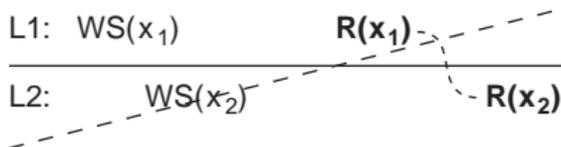
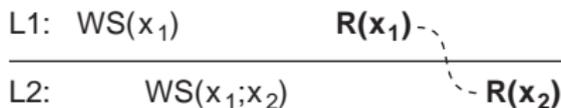


# Basic architecture



## Definition

If a process reads the value of a data item  $x$ , any successive read operation on  $x$  by that process will always return that same or a more recent value.



## Notation

- $WS(x_i[t])$  is the set of write operations (at  $L_i$ ) that lead to version  $x_i$  of  $x$  (at time  $t$ )
- $WS(x_i[t_1]; x_j[t_2])$  indicates that it is known that  $WS(x_i[t_1])$  is part of  $WS(x_j[t_2])$ .
- **Note:** Parameter  $t$  is omitted from figures.



## Example

Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

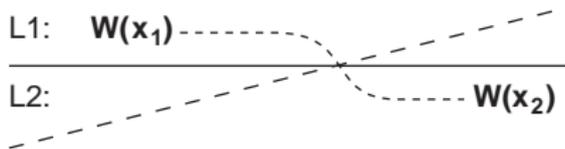
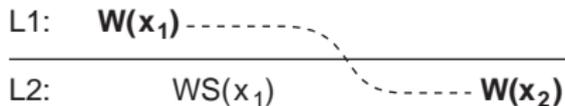
## Example

Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.



## Definition

A write operation by a process on a data item  $x$  is completed before any successive write operation on  $x$  by the same process.



### Example

Updating a program at server  $S_2$ , and ensuring that all components on which compilation and linking depends, are also placed at  $S_2$ .

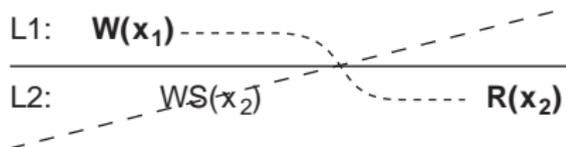
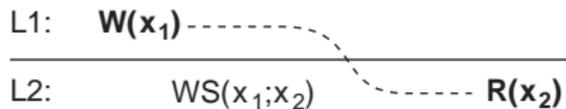
### Example

Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).



## Definition

The effect of a write operation by a process on data item  $x$ , will always be seen by a successive read operation on  $x$  by the same process.



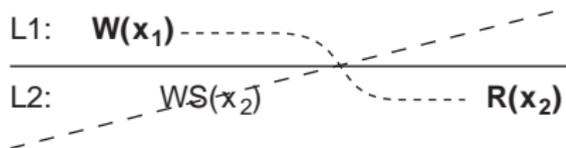
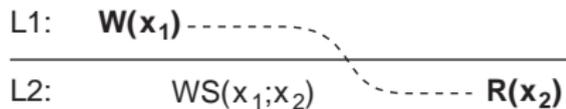
## Example

Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.



## Definition

The effect of a write operation by a process on data item  $x$ , will always be seen by a successive read operation on  $x$  by the same process.



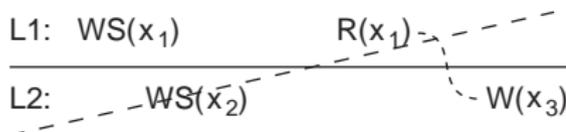
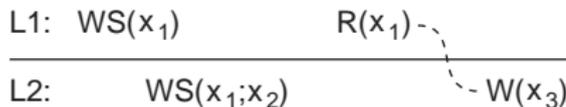
## Example

Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.



## Definition

A write operation by a process on a data item  $x$  following a previous read operation on  $x$  by the same process, is guaranteed to take place on the same or a more recent value of  $x$  that was read.



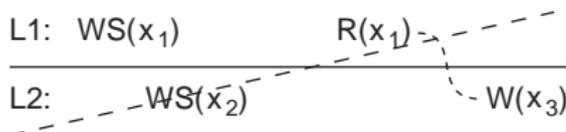
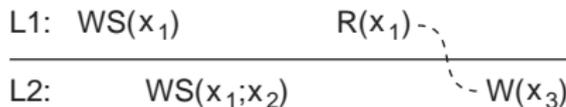
## Example

See reactions to posted articles only if you have the original posting (a read “pulls in” the corresponding write operation).



## Definition

A write operation by a process on a data item  $x$  following a previous read operation on  $x$  by the same process, is guaranteed to take place on the same or a more recent value of  $x$  that was read.



## Example

See reactions to posted articles only if you have the original posting (a read “pulls in” the corresponding write operation).







## From consistency models to management

- Replicas need to be kept consistent according to some model
- No update  $\rightarrow$  no problem
- If **access-to-update ratio** is high, replication will help
- If **updates-to-access ratio** is high, updates will not be consumed
- Ideally, update only replicas that are going to be accessed
- In general, try to keep replicas in proximity to clients



## Challenges

- Replica server placement
  - often a management or commercial issue
- Content replication and placement
- Content distribution
  - state vs. operation
  - push vs. pull vs. lease
  - blocking vs. non-blocking (eager vs lazy)
  - unicast vs multicast (group communication)



## Essence

Figure out what the best  $K$  places are out of  $N$  possible locations.

- Select best location out of  $N - K$  for which the **average distance to clients is minimal**. Then choose the next best server. (**Note**: The first chosen location minimizes the average distance to all clients.) **Computationally expensive**.
- Select the  $K$ -th largest **autonomous system** and place a server at the best-connected host. **Computationally expensive**.
- Position nodes in a  $d$ -dimensional geometric space, where distance reflects latency. Identify the  $K$  regions with highest density and place a server in every one. **Computationally cheap**.



## Essence

Figure out what the best  $K$  places are out of  $N$  possible locations.

- Select best location out of  $N - K$  for which the **average distance to clients is minimal**. Then choose the next best server. (**Note:** The first chosen location minimizes the average distance to all clients.) **Computationally expensive.**
- Select the  $K$ -th largest **autonomous system** and place a server at the best-connected host. **Computationally expensive.**
- Position nodes in a  $d$ -dimensional geometric space, where distance reflects latency. Identify the  $K$  regions with highest density and place a server in every one. **Computationally cheap.**



## Essence

Figure out what the best  $K$  places are out of  $N$  possible locations.

- Select best location out of  $N - K$  for which the **average distance to clients is minimal**. Then choose the next best server. (**Note:** The first chosen location minimizes the average distance to all clients.) **Computationally expensive**.
- Select the  $K$ -th largest **autonomous system** and place a server at the best-connected host. **Computationally expensive**.
- Position nodes in a  $d$ -dimensional geometric space, where distance reflects latency. Identify the  $K$  regions with highest density and place a server in every one. **Computationally cheap**.



## Essence

Figure out what the best  $K$  places are out of  $N$  possible locations.

- Select best location out of  $N - K$  for which the **average distance to clients is minimal**. Then choose the next best server. (**Note:** The first chosen location minimizes the average distance to all clients.) **Computationally expensive**.
- Select the  $K$ -th largest **autonomous system** and place a server at the best-connected host. **Computationally expensive**.
- Position nodes in a  $d$ -dimensional geometric space, where distance reflects latency. Identify the  $K$  regions with highest density and place a server in every one. **Computationally cheap**.



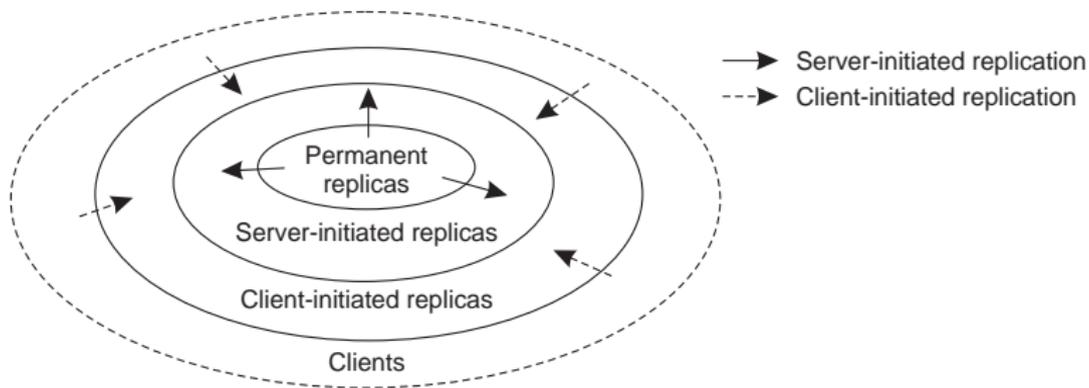
## Distinguish different processes

A process is capable of hosting a replica of an object or data:

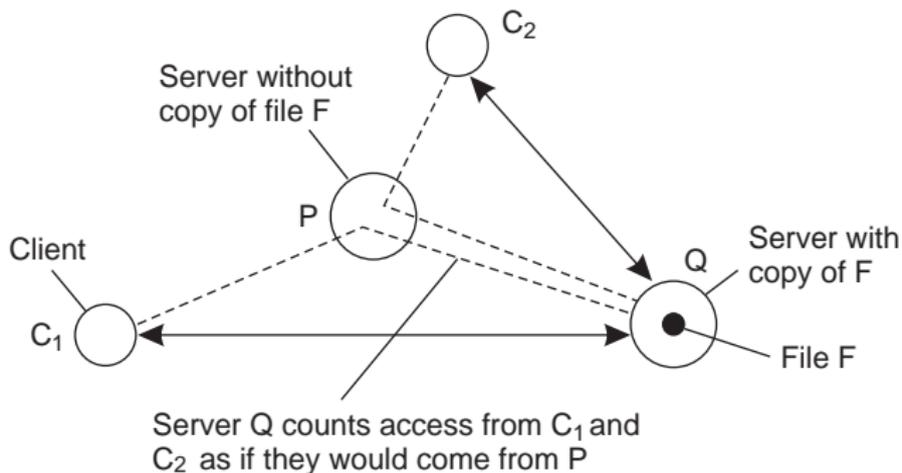
- **Permanent replicas:** Process/machine always having a replica (i.e. origin server)
  - initial set (small)
  - LAN; e.g., Web server **cluster** or database cluster
  - geographically; e.g., Web **mirror** or **federated database**
- **Server-initiated replica:** Process that can dynamically host a replica on request of another server in the data store
  - performance, e.g., **push cache** or **Web hosting service**
  - reduce **server load** and replicate to server placed in the **proximity** of requesting clients
- **Client-initiated replica:** Process that can dynamically host a replica on request of a client (**client cache**)



# Content replication



# Server-initiated replicas



- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold  $D \Rightarrow$  drop file
- Number of accesses exceeds threshold  $R \Rightarrow$  replicate file
- Number of access between  $D$  and  $R \Rightarrow$  migrate file



## Model

Consider only a client-server combination:

- Propagate only **notification/invalidation** of update (often used for caches)
- Transfer **data** from one copy to another (distributed databases): **passive replication**
- Propagate the update **operation** to other copies: **active replication**

## Note

No single approach is the best, but depends highly on available bandwidth and read-to-write ratio at replicas.



## Push (server)-based protocols

- Updates are propagated to other replicas without those replicas asking for updates
- Used by permanent and server-initiated replicas, but also by some client caches
- High degree of consistency (consistent data can be made available faster)
- If server keeps track of clients that have cached the data, we have a **stateful** server: limited scalability and less fault tolerant
- Often, multicasting is more efficient



## Pull (client)-based protocols

- A replica requests another replica to send it any updates it has at the moment
- Often used by client caches
- I.e. client polls server if updates are available
- E.g. Web modified since
- Response time increases in case of a cache miss
- Unicasting instead of multicasting



# Content distribution: client/server system

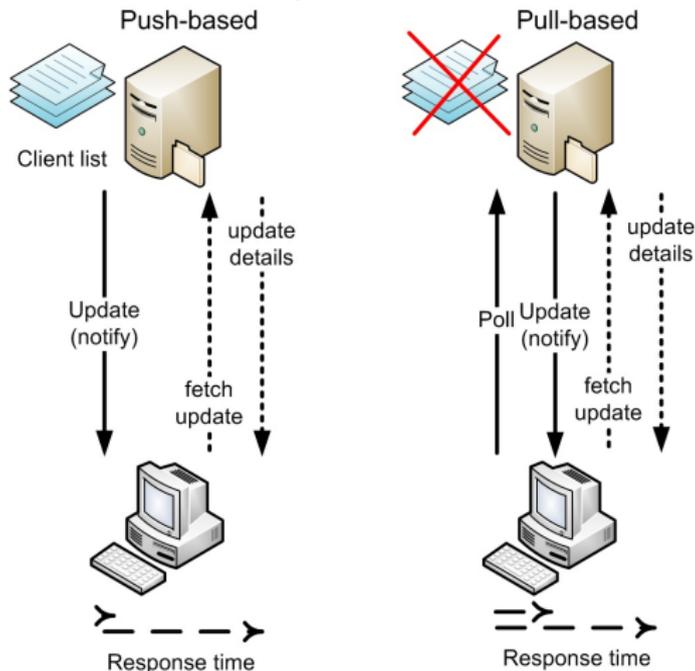
- **Pushing updates:** **server-initiated approach**, in which update is propagated regardless whether target asked for it.
- **Pulling updates:** **client-initiated approach**, in which client requests to be updated.

Issue	Push-based	Pull-based
1:	List of client caches	None
2:	Update (and possibly fetch update)	Poll and update
3:	Immediate (or fetch-update time)	Fetch-update time
<p><i>1: State at server</i></p> <p><i>2: Messages to be exchanged</i></p> <p><i>3: Response time at the client</i></p>		



# Content distribution: client/server system

## Push-based vs Pull-based Updates



## Observation

We can dynamically switch between pulling and pushing using **leases**: A contract in which the server promises to push updates to the client until the lease expires.



## Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become

## Question

Why are we doing all this?

Trying to reduce the server's state as much as possible while providing strong consistency.



## Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become

## Question

Why are we doing all this?

Trying to reduce the server's state as much as possible while providing strong consistency.



## Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become

## Question

Why are we doing all this?

Trying to reduce the server's state as much as possible while providing strong consistency.



## Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become

## Question

Why are we doing all this?

Trying to reduce the server's state as much as possible while providing strong consistency.



## Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become

## Question

Why are we doing all this?

Trying to reduce the server's state as much as possible while providing strong consistency.



## Issue

Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become

## Question

Why are we doing all this?

Trying to reduce the server's state as much as possible while providing strong consistency.



## When are (push) updates propagated?

- Synchronous (**blocking, eager**):  
All replicas are updated immediately, then reply to client (that issued the update)
- Asynchronous (**non-blocking, lazy**):  
Update is applied to one copy, then reply to client, propagation to other replicas afterwards



## Consistency protocol

Describes the implementation of a specific consistency model.

- Continuous consistency
- Primary-based protocols
- Replicated-write protocols



# Continuous consistency: Numerical errors

## Principal operation

- Every server  $S_i$  has a log, denoted as  $\log(S_i)$ .
- Consider a data item  $x$  and let  $weight(W)$  denote the numerical change in its value after a write operation  $W$ . Assume that

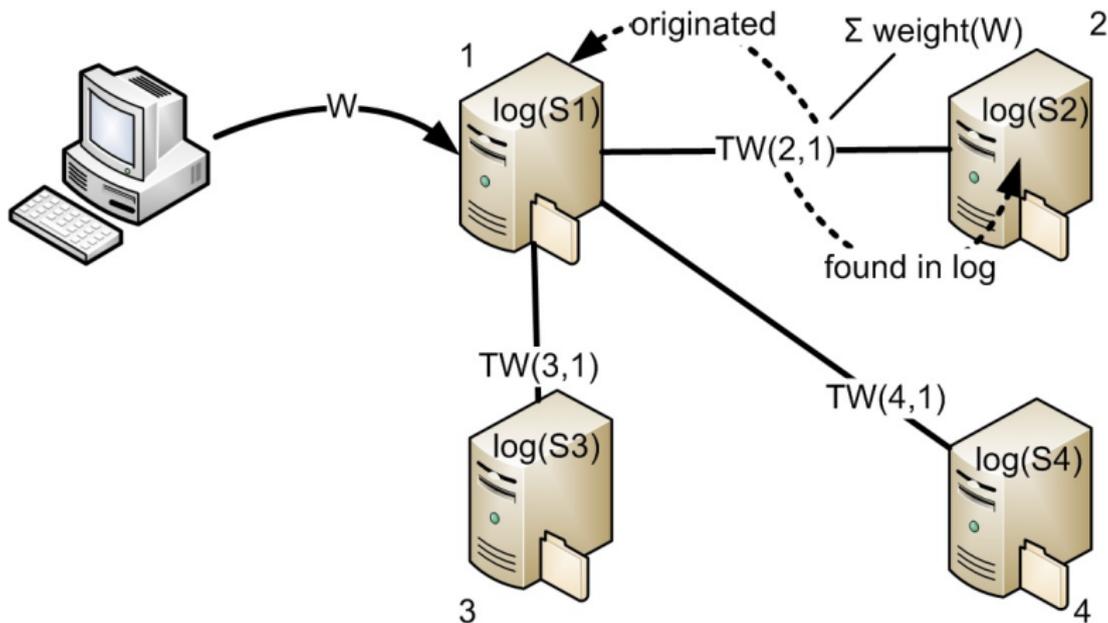
$$\forall W : weight(W) > 0$$

- $W$  is initially forwarded to one of the  $N$  replicas, denoted as  $origin(W)$ .  $TW[i, j]$  are the writes executed by server  $S_i$  that originated from  $S_j$ :

$$TW[i, j] = \sum \{ weight(W) \mid origin(W) = S_j \ \& \ W \in \log(S_i) \}$$



# Continuous consistency: Numerical errors



# Continuous consistency: Numerical errors

## Note

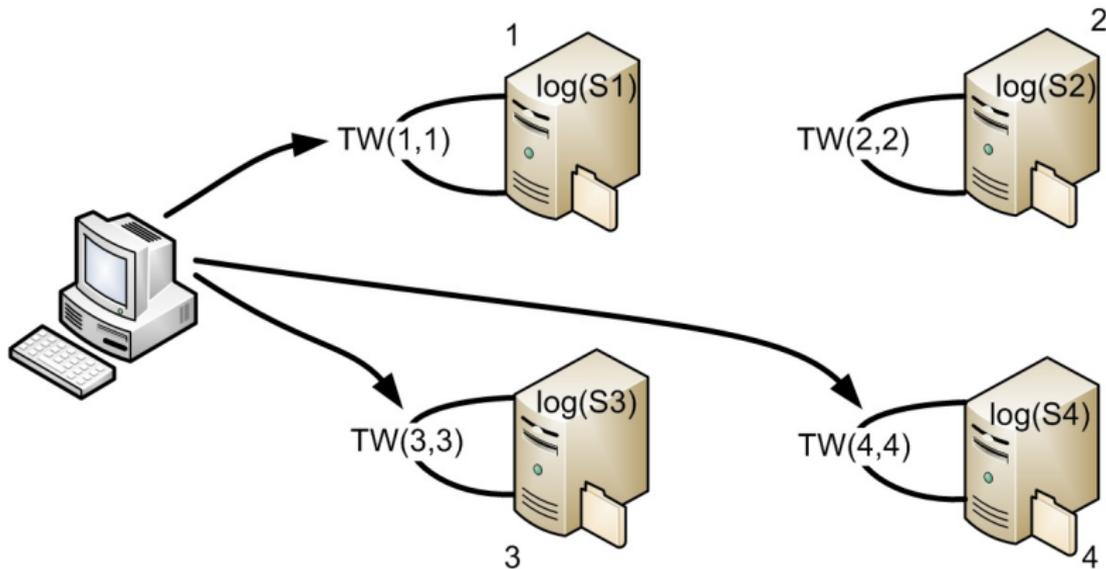
Actual value  $v(t)$  of  $x$ :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

value  $v_i$  of  $x$  at replica  $i$ :

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

# Continuous consistency: Numerical errors



# Continuous consistency: Numerical errors

## Note

Actual value  $v(t)$  of  $x$ :

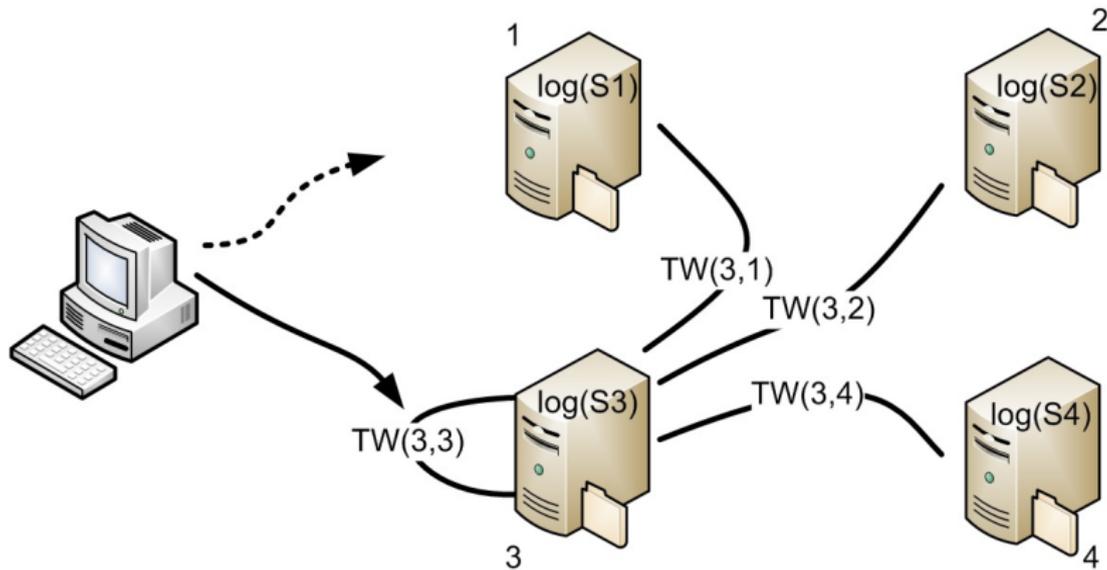
$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

value  $v_i$  of  $x$  at replica  $i$ :

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$



# Continuous consistency: Numerical errors



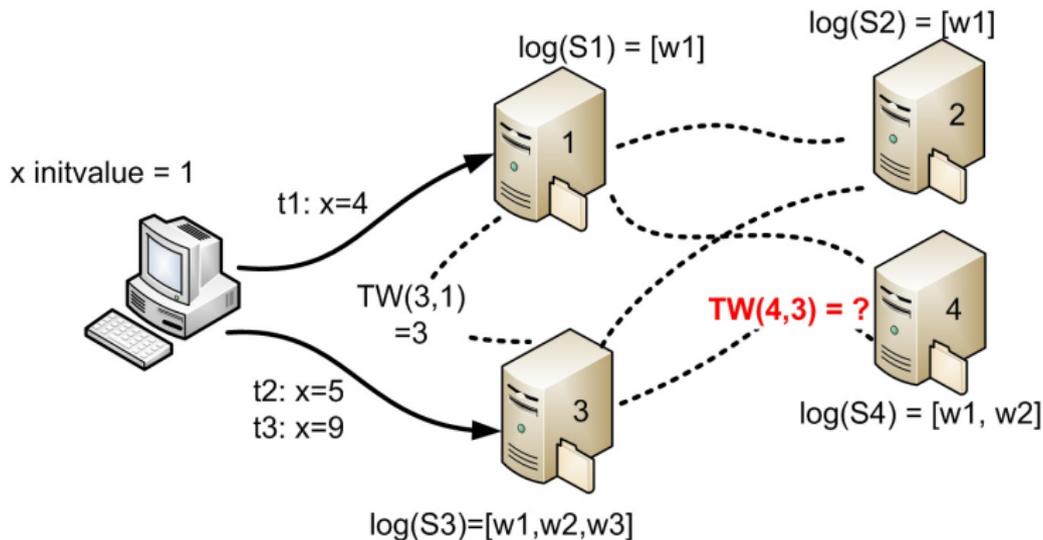
# Know your Consistency Models

## Question to the audience

What is the value of  $TW(4,3)$ ?

## Your choices are:

1 (head) ; 4 (ear); 8 (nose); Answer = 1



# Continuous consistency: Numerical errors

## Problem

We need to ensure that  $v(t) - v_i < \delta_i$  for every server  $S_i$ .

## Approach

Let every server  $S_k$  maintain a **view**  $TW_k[i, j]$  of what it believes is the value of  $TW[i, j]$ . This information can be **gossiped** when an update is propagated.

## Note

$$0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$$



# Continuous consistency: Numerical errors

## Problem

We need to ensure that  $v(t) - v_i < \delta_i$  for every server  $S_i$ .

## Approach

Let every server  $S_k$  maintain a **view**  $TW_k[i, j]$  of what it believes is the value of  $TW[i, j]$ . This information can be **gossiped** when an update is propagated.

## Note

$$0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$$



# Continuous consistency: Numerical errors

## Problem

We need to ensure that  $v(t) - v_i < \delta_i$  for every server  $S_i$ .

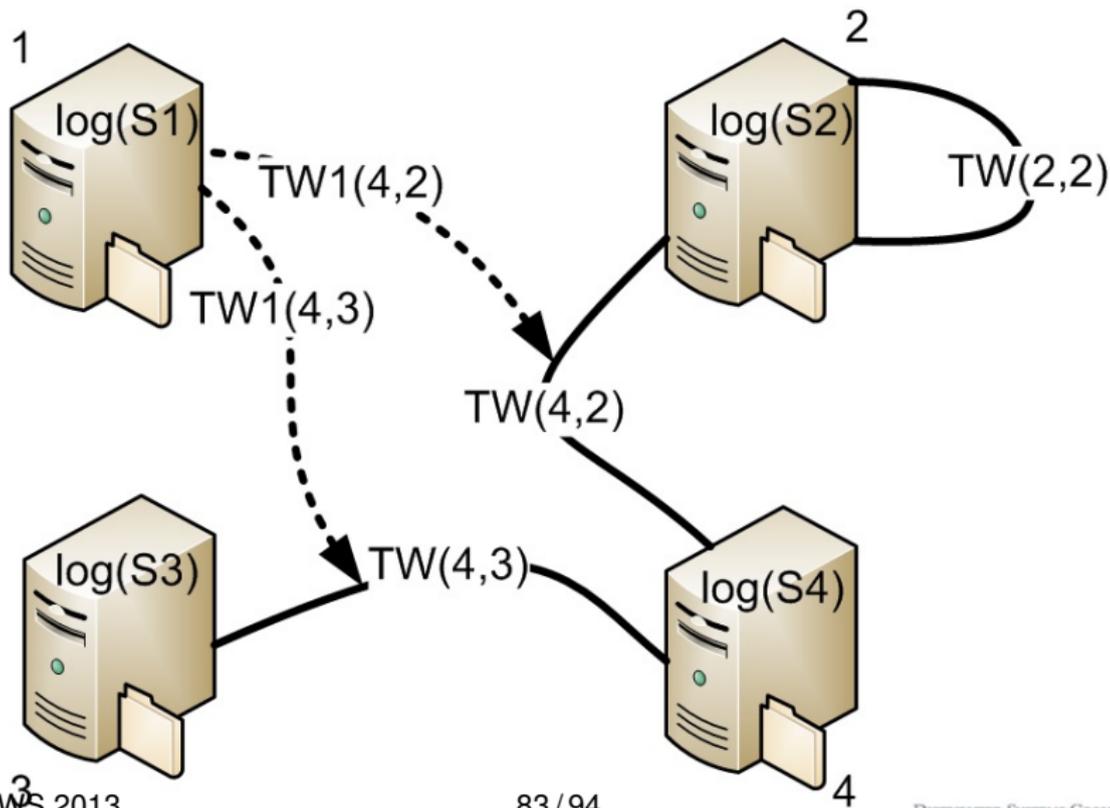
## Approach

Let every server  $S_k$  maintain a **view**  $TW_k[i, j]$  of what it believes is the value of  $TW[i, j]$ . This information can be **gossiped** when an update is propagated.

## Note

$$0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$$

# Continuous consistency: Numerical errors



# Continuous consistency: Numerical errors

## Solution

$S_k$  sends operations from its log to  $S_i$  when it sees that  $TW_k[i, k]$  is getting too far from  $TW[k, k]$ , in particular, when

$$TW[k, k] - TW_k[i, k] > \delta_i / (N - 1)$$

## Question

To what extent are we being pessimistic here: where does  $\delta_i / (N - 1)$  come from?

## Note

Staleness can be done analogously, by essentially keeping track of what has been seen last from  $S_i$  (see book)



# Continuous consistency: Numerical errors

## Solution

$S_k$  sends operations from its log to  $S_i$  when it sees that  $TW_k[i, k]$  is getting too far from  $TW[k, k]$ , in particular, when

$$TW[k, k] - TW_k[i, k] > \delta_i / (N - 1)$$

## Question

To what extent are we being **pessimistic** here: where does  $\delta_i / (N - 1)$  come from?

## Note

Staleness can be done analogously, by essentially keeping track of what has been seen last from  $S_i$  (see book)



# Continuous consistency: Numerical errors

## Solution

$S_k$  sends operations from its log to  $S_i$  when it sees that  $TW_k[i, k]$  is getting too far from  $TW[k, k]$ , in particular, when

$$TW[k, k] - TW_k[i, k] > \delta_i / (N - 1)$$

## Question

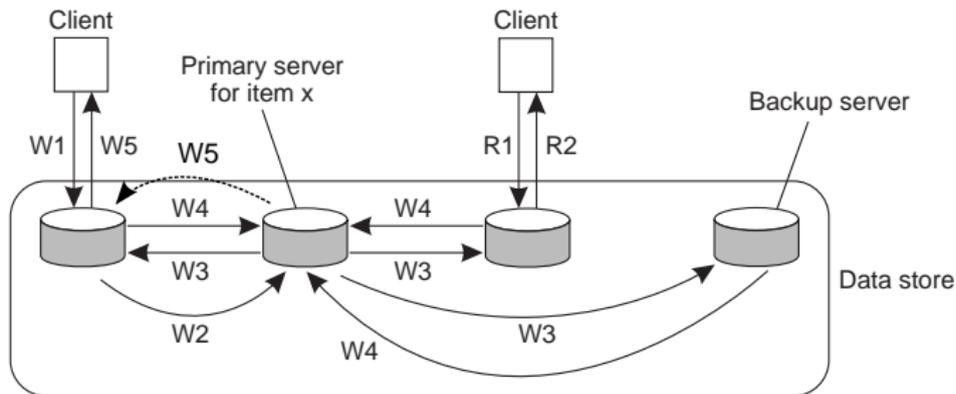
To what extent are we being pessimistic here: where does  $\delta_i / (N - 1)$  come from?

## Note

Staleness can be done analogously, by essentially keeping track of what has been seen last from  $S_i$  (see book)



## Primary-backup protocol



W1. Write request  
 W2. Forward request to primary  
 W3. Tell backups to update  
 W4. Acknowledge update  
 W5. Acknowledge write completed

R1. Read request  
 R2. Response to read

# Primary-based Synchronous Replication

## Advantages

- No inconsistencies (identical copies)
- Reading the local copy yields the most up-to-date value
- Changes are atomic

## Disadvantages

A write operation has to update all sites

- slow
- not resilient against network or node failure



# Primary-based Asynchronous Replication

## Advantages

- Fast, since only primary replica is updated immediately
- Resilient against node and link failure

## Disadvantages

- Data **inconsistencies** can occur
- a local read does not always return the most up-to-date value

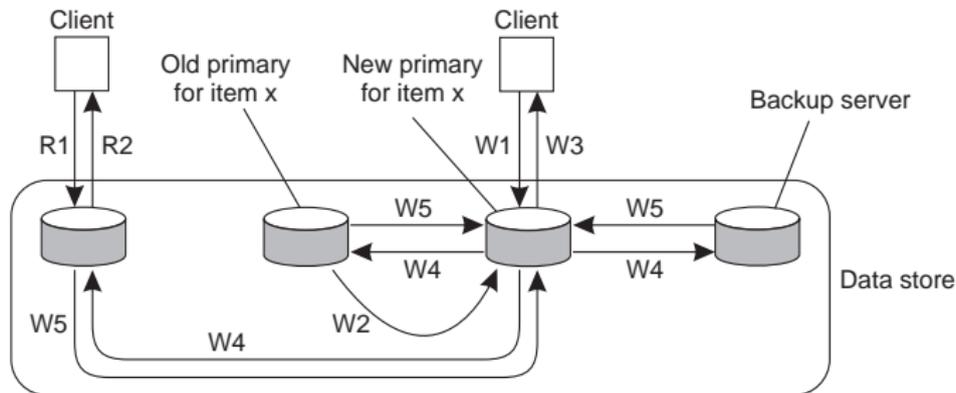


## Example primary-backup protocol

Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on same LAN.



## Primary-backup protocol with local writes



W1. Write request  
 W2. Move item x to new primary  
 W3. Acknowledge write completed  
 W4. Tell backups to update  
 W5. Acknowledge update

R1. Read request  
 R2. Response to read

## Advantages

- At least one node exists which has all updates
- ordering guarantees are relatively easy to achieve (no inter-site synchronization necessary)

## Disadvantages

- Primary is bottleneck and single point of failure
- High reconfiguration costs when primary fails



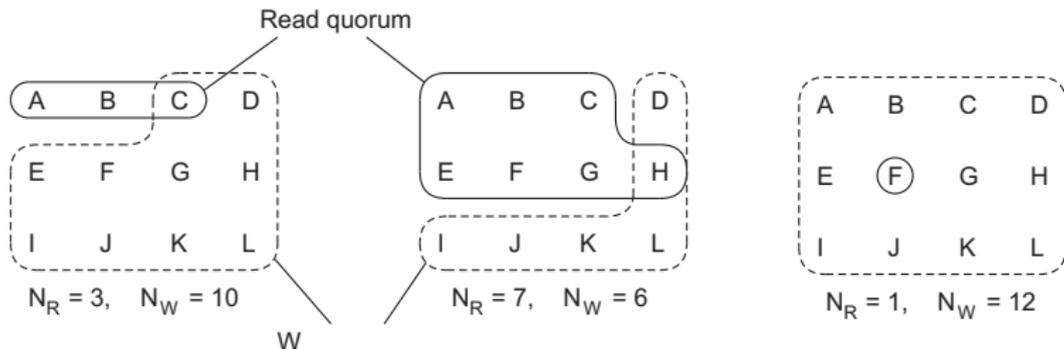
## Example primary-backup protocol with local writes

Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).



## Quorum-based protocols

Ensure that each operation is carried out in such a way that a majority vote is established: distinguish **read quorum** and **write quorum**:



**required:**  $N_R + N_W > N$  and  $N_W > N/2$

# Distributed Systems Principles and Paradigms

Christoph Dorn

Distributed Systems Group,  
Vienna University of Technology

`c.dorn@infosys.tuwien.ac.at`

`http://www.infosys.tuwien.ac.at/staff/dorn`

Slides adapted from Maarten van Steen, VU Amsterdam, `steen@cs.vu.nl`

## Chapter 11: Distributed File Systems

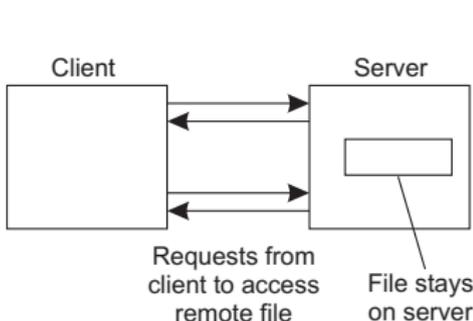


<b>Chapter</b>
01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
<b>11: Distributed File Systems</b>
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems

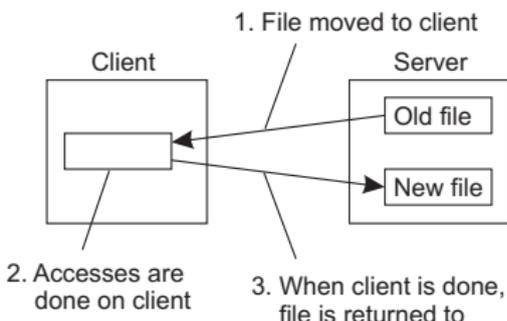


## General goal

Try to make a file system transparently available to remote clients.



Remote access model



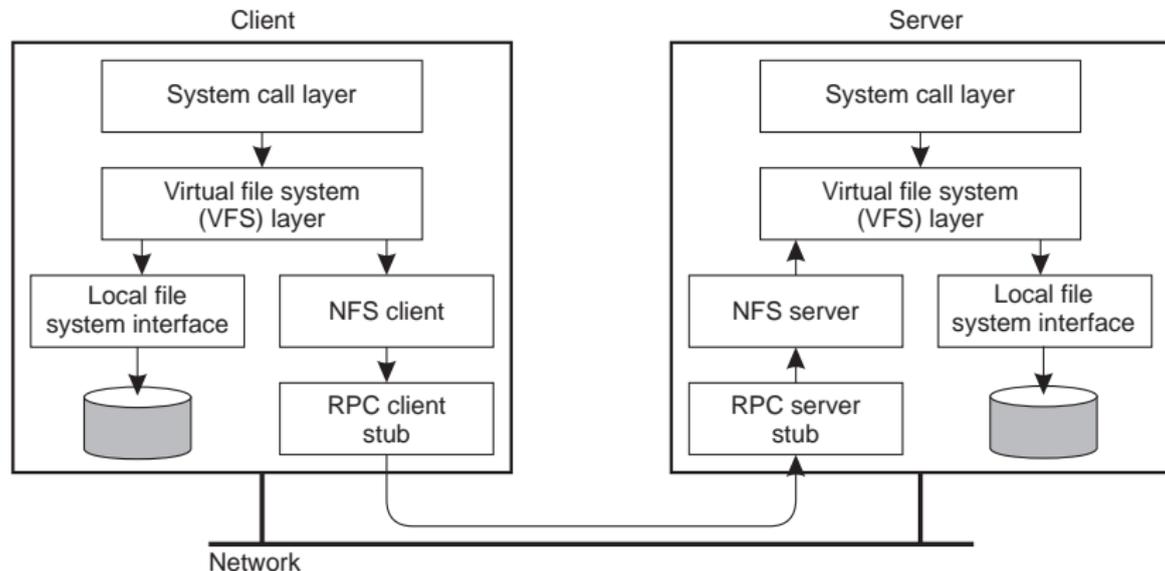
Upload/download model



# Example: NFS Architecture

## NFS

NFS is implemented using the **Virtual File System** abstraction, which is now used for lots of different operating systems.



# Example: NFS Architecture

## Essence

VFS provides standard file system interface, and allows to hide difference between accessing local or remote file system.

## Question

Is NFS actually a file system?



# NFS File Operations

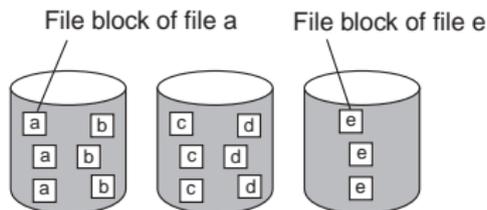
Oper.	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name in a symbolic link
Getattr	Yes	Yes	Get the attribute values for a file
Setattr	Yes	Yes	Set one or more file-attribute values
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file



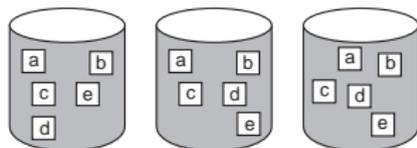
# Cluster-Based File Systems

## Observation

With very large data collections, following a simple client-server approach is not going to work  $\Rightarrow$  for **speeding up file accesses**, apply **striping** techniques by which files can be fetched in parallel.

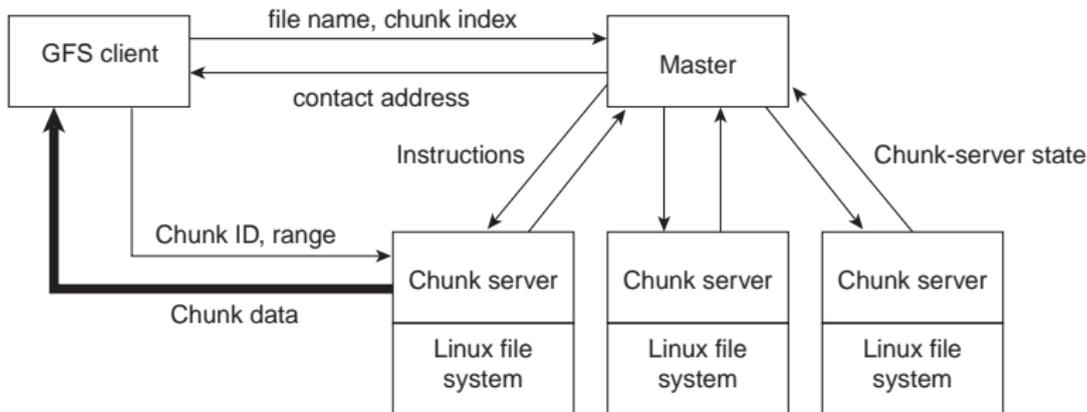


Whole-file distribution



File-striped system

# Example: Google File System



## The Google solution

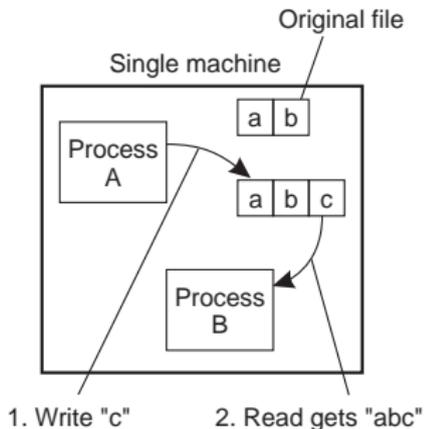
Divide files in large 64 MB chunks, and distribute/replicate chunks across many servers:

- The master maintains only a (file name, chunk server) table in **main memory**  $\Rightarrow$  minimal I/O
- Files are replicated using a **primary-backup** scheme; the master is kept **out of the loop**

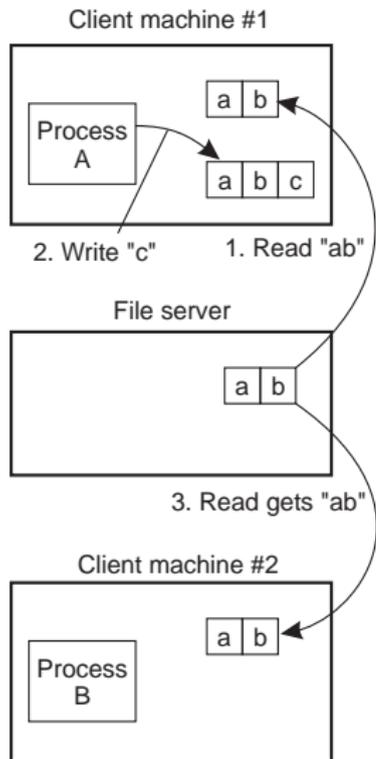


## Problem

When dealing with distributed file systems, we need to take into account the ordering of concurrent read/write operations and expected semantics (i.e., consistency).



(a)



(b)



## Semantics

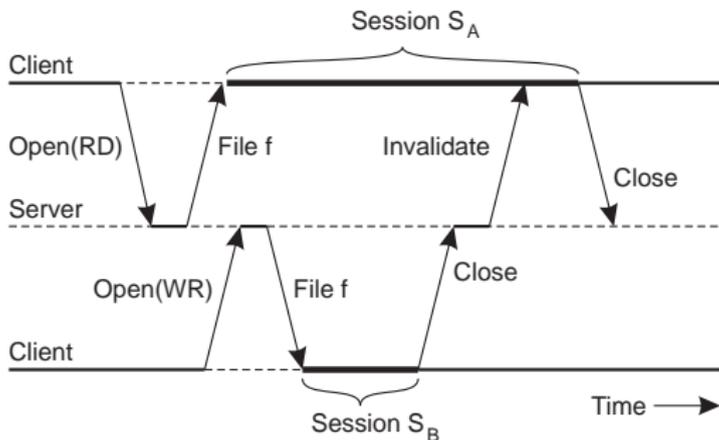
- **UNIX semantics:** a *read* operation returns the effect of the last *write* operation  $\Rightarrow$  can only be implemented for remote access models in which there is only a single copy of the file
- **Transaction semantics:** the file system supports transactions on a *single* file  $\Rightarrow$  issue is how to allow concurrent access to a physically distributed file
- **Session semantics:** the effects of *read* and *write* operations are seen only by the client that has opened (a local copy) of the file  $\Rightarrow$  what happens when a file is closed (only one client may actually win)



# Example: File sharing in Coda

## Essence

Coda assumes transactional semantics, but without the full-fledged capabilities of real transactions. **Note:** Transactional issues reappear in the form of “this ordering could have taken place.”

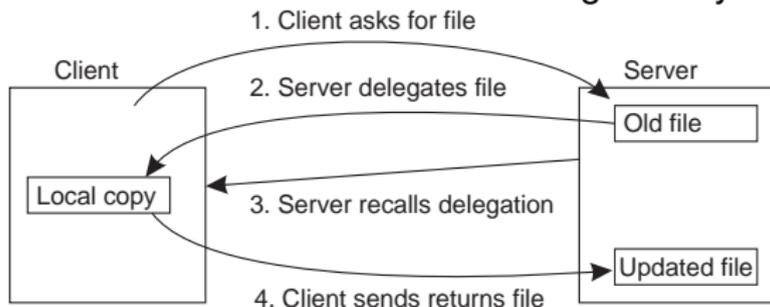


## Observation

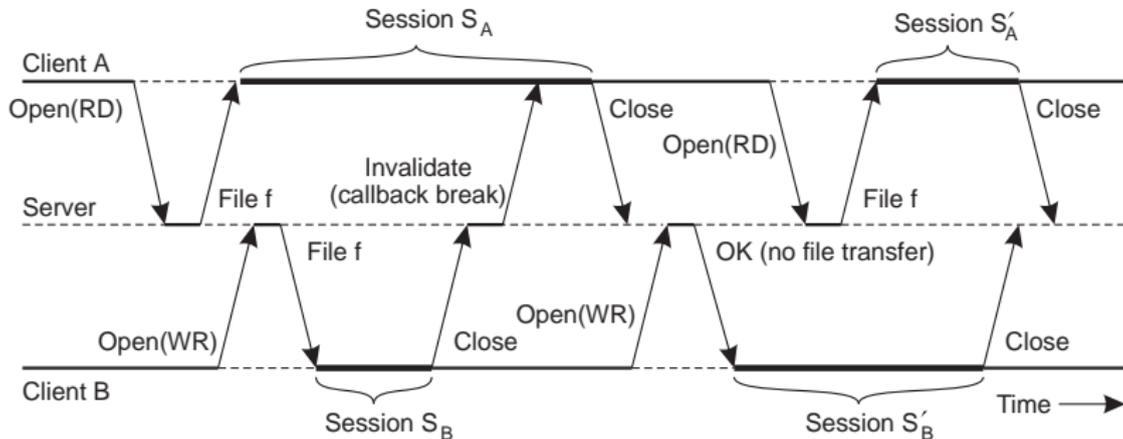
In modern distributed file systems, **client-side caching** is the preferred technique for attaining performance; **server-side replication** is done for fault tolerance.

## Observation

Clients are allowed to keep (large parts of) a file, and will be **notified** when control is withdrawn  $\Rightarrow$  servers are now generally **stateful**



# Example: Client-side caching in Coda

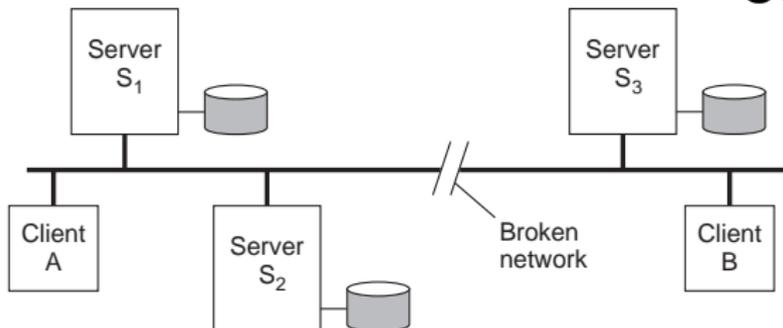


## Note

By making use of **transactional semantics**, it becomes possible to further improve performance.



# Example: Server-side replication in Coda



## Main issue

Ensure that concurrent updates are detected:

- Each client has an **Accessible Volume Storage Group (AVSG)**: is a subset of the actual VSG.
- **Version vector**  $CVV_i(f)[j] = k \Rightarrow S_i$  knows that  $S_j$  has seen version  $k$  of  $f$ .
- Example:  $A$  updates  $f \Rightarrow S_1 = S_2 = [+1, +1, +0]$ ;  $B$  updates  $f \Rightarrow S_3 = [+0, +0, +1]$ .



# Distributed Systems

Dr. Philipp Leitner  
Distributed Systems Group  
Vienna University of Technology

[leitner@infosys.tuwien.ac.at](mailto:leitner@infosys.tuwien.ac.at)

1. Introduction
2. Authentication
3. Integrity
4. Access Control
5. Some State-of-the-Art Attacks
  
6. Advanced Security Lectures

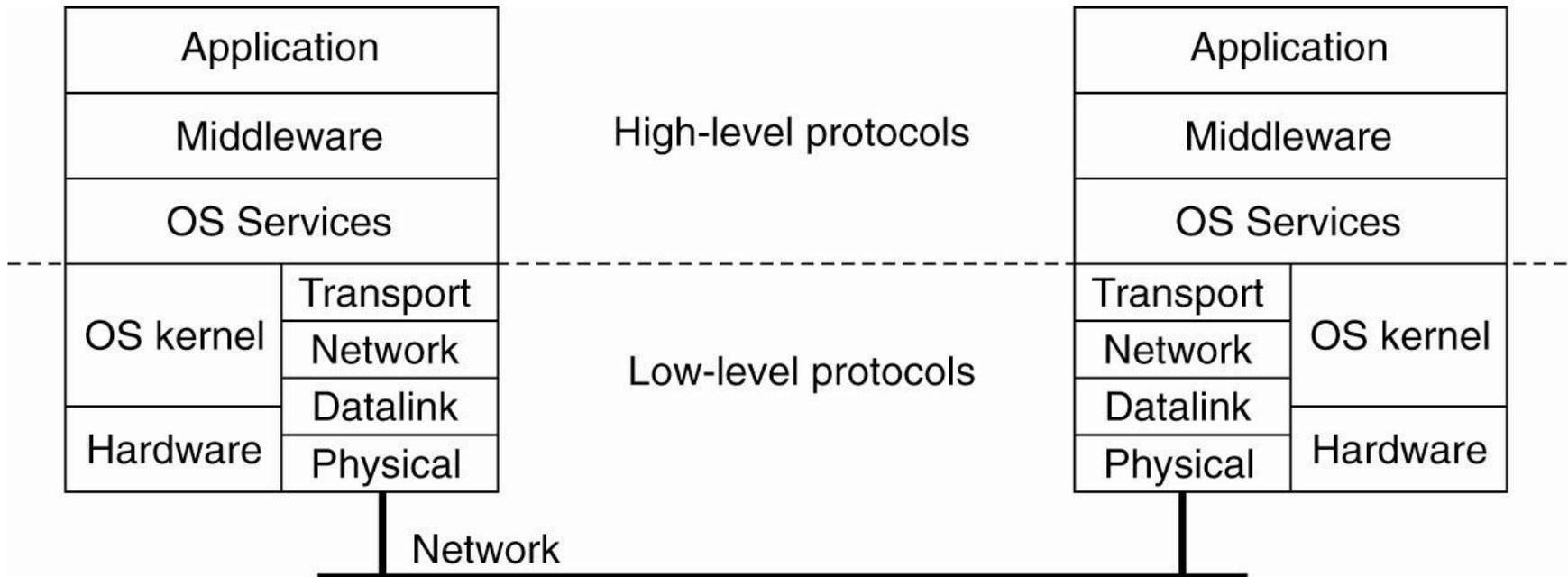
# INTRODUCTION

# General Concepts of Security in Distributed Systems

- Two main areas:
  - How to establish a **secure channel** between users / processes across process and machine borders?
  - How to **authorize** users and processes?
  
- Threats:
  - Interception
  - Interruption
  - Modification
  - Fabrication

# Layering of Security Mechanisms

## (1)

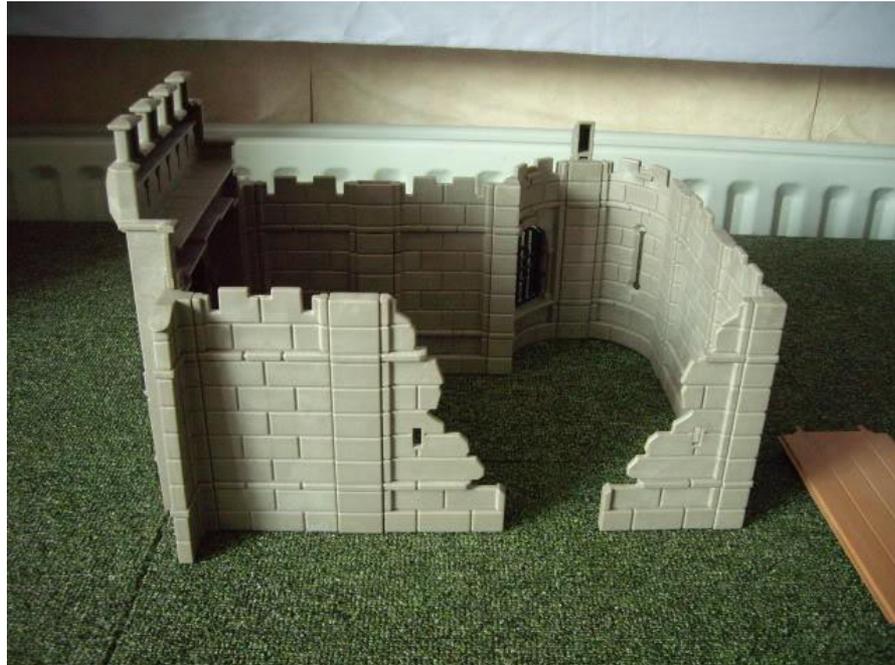


Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

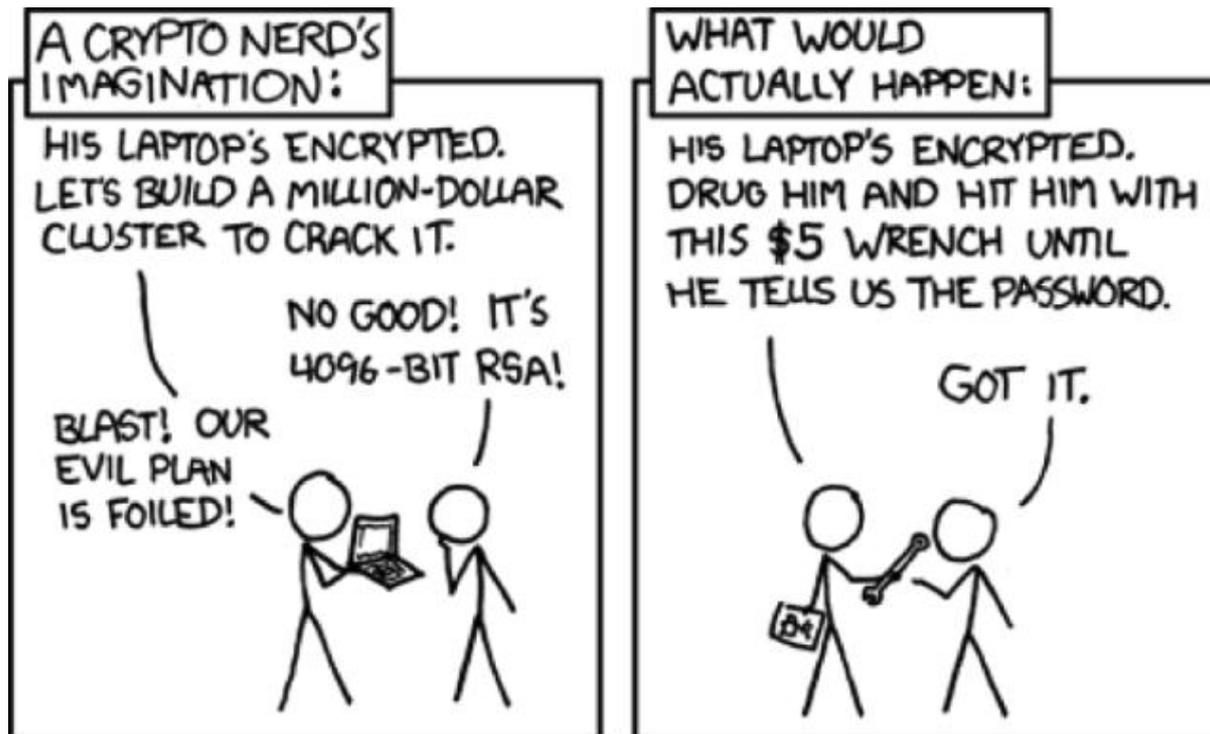
# Layering of Security Mechanisms (2)

- Considerations:
  - Security on a lower layer is often more convenient
  - Security on a higher layer may allow secure communication over an otherwise insecure channel
    - E.g., SSL / TLS (secure communication over insecure TCP via an Transport Layer protocol)

- The security of any distributed system is **exactly** as good as its weakest component.



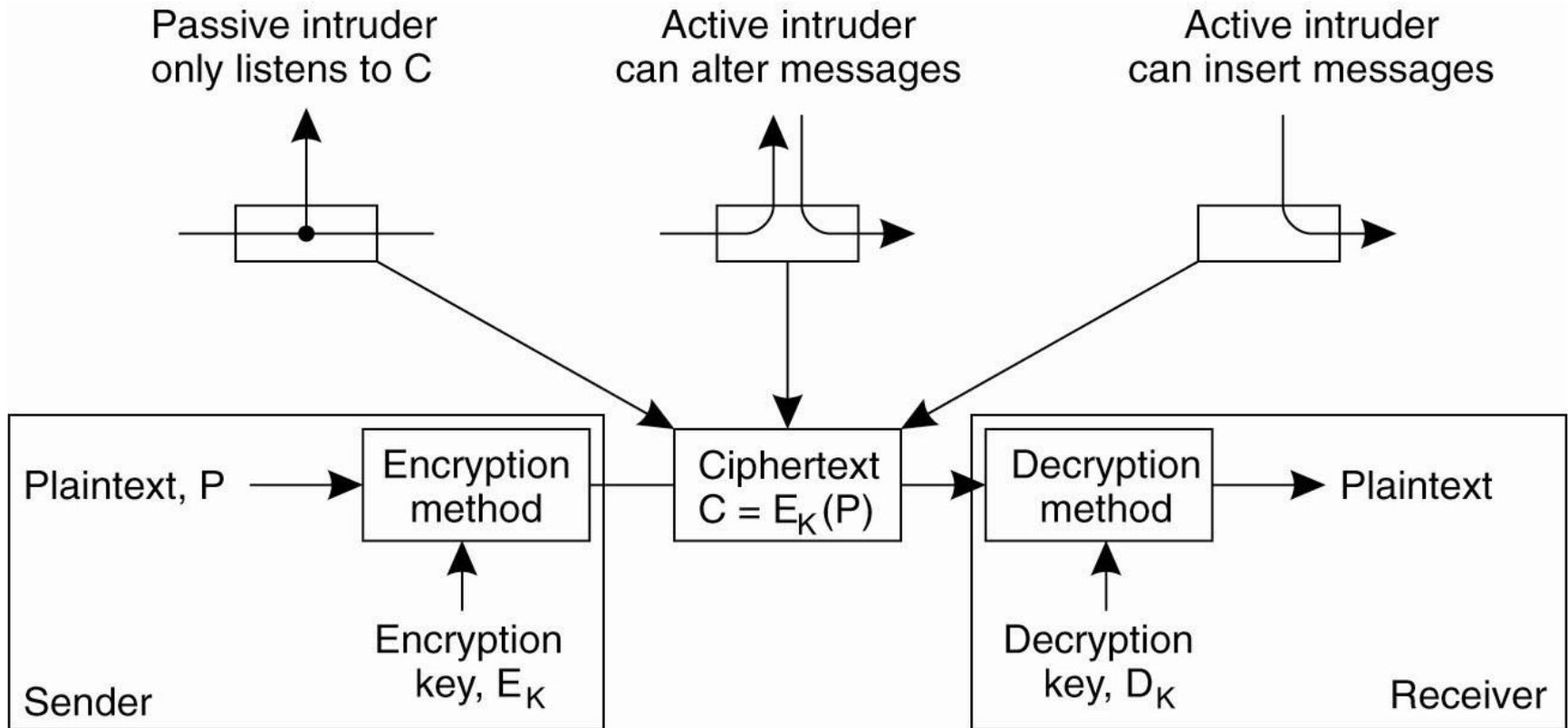
- This weakest component is typically the **human** in the loop.



- The security of your system needs to depend on technical and mathematical facts, and never on **hidden information**.

SECURITY BY OBSCURITY 101!



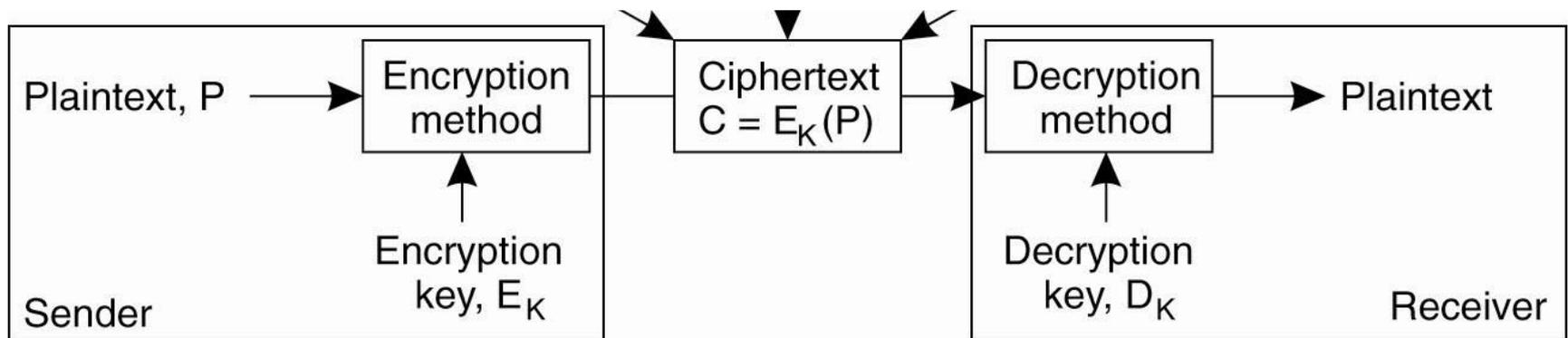


Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

P = 'Hello World'

C = 'xYYnuYY542b'

P = 'Hello World'





# Types of Cryptographic Methods (1)

- **Symmetric** cryptosystems
  - The same key is used as encryption key  $E_K$  and decryption key  $D_K$
  - E.g., DES, AES
- **Asymmetric** cryptosystems
  - $E_K$  and  $D_K$  differ, but form a pair
  - Other common name: **public-key** cryptosystem
  - E.g., RSA

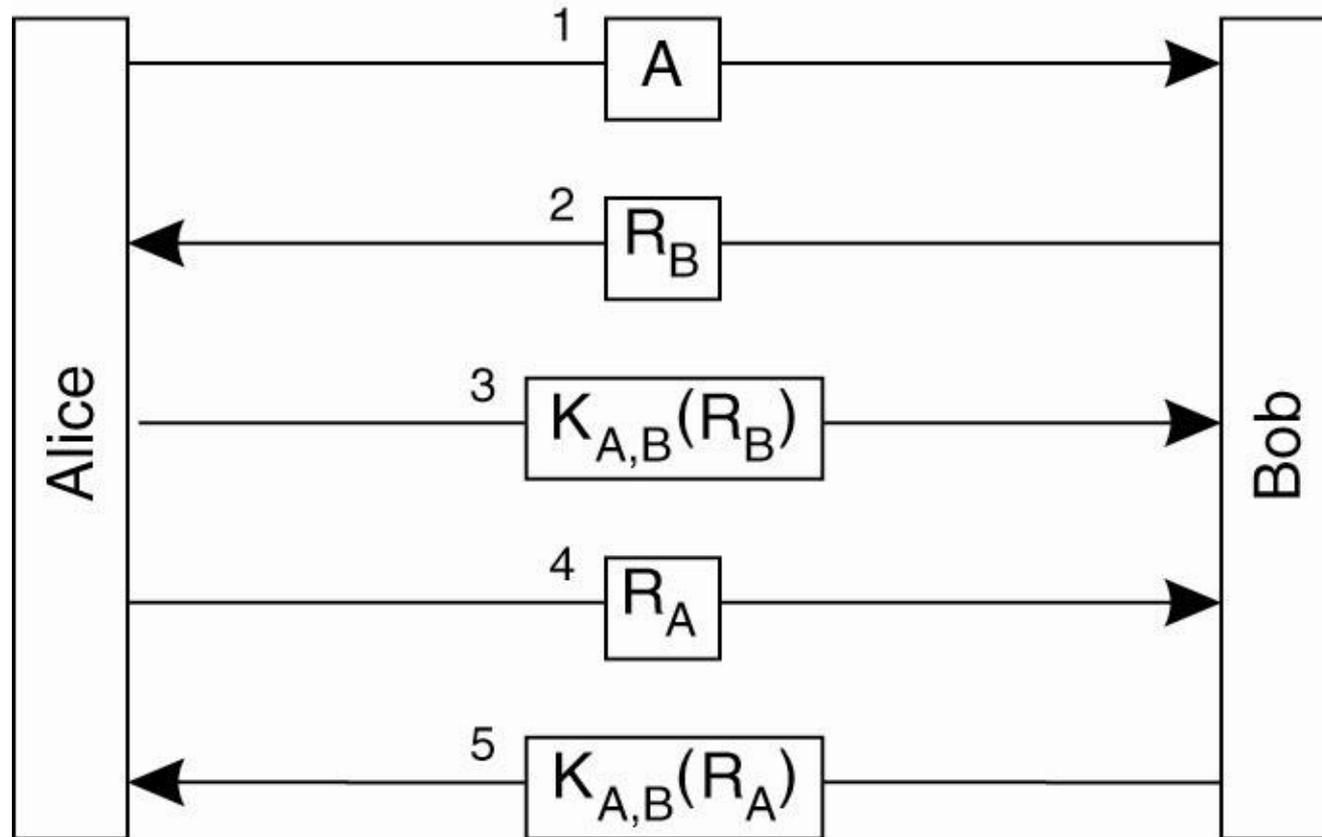
# TU WIEN Types of Cryptographic Methods (2)

- **Hashing cryptosystems**
  - Basically encryption method where no decryption key  $D_K$  exists
  - E.g., MD5, SHA-1
- **Properties:**
  - **One-way functions**
    - Given a hash value, it is infeasible to find the original value
  - **Weak collision resistance**
    - Given a hash and an original value, it is infeasible to find another original value that leads to the same hash
  - **Strong collision resistance**
    - It is infeasible to find two original values that lead to the same hash

- Symmetric cryptosystems
  - Encryption (prevention of **interception**)
- Asymmetric cryptosystems
  - Authentication (prevention of **fabrication**)
- Hashing cryptosystems
  - Integrity (prevention of **modification**)

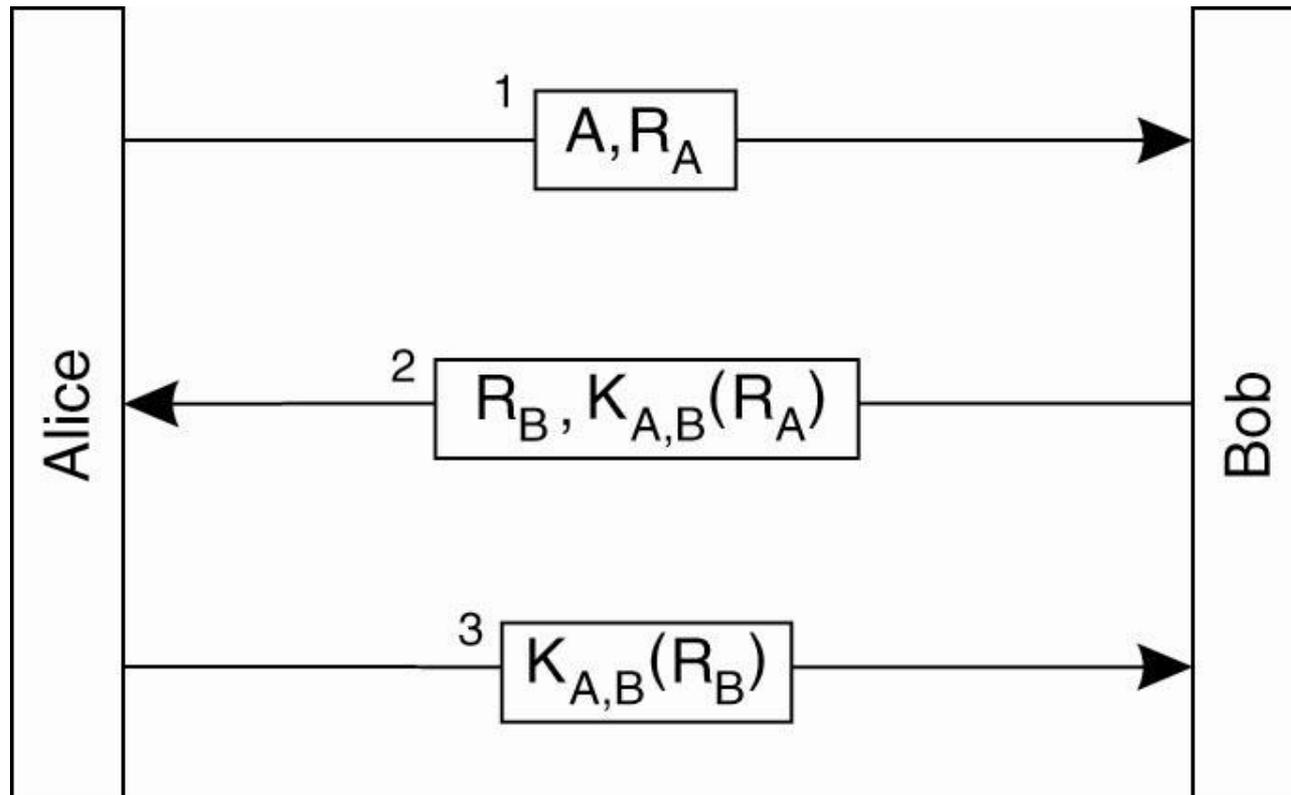
# AUTHENTICATION

# Authentication Based on a Shared Secret Key



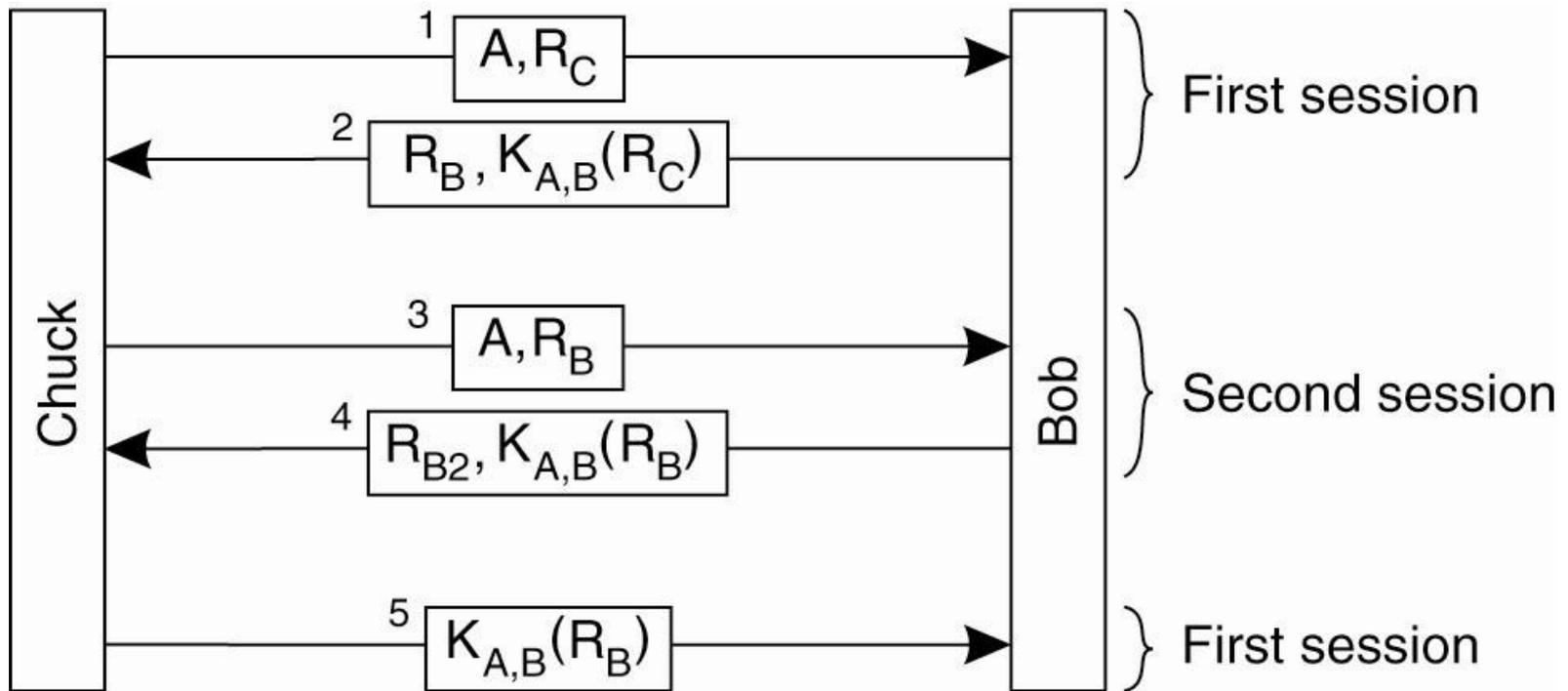
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# An “Optimized” Version of This Protocol?



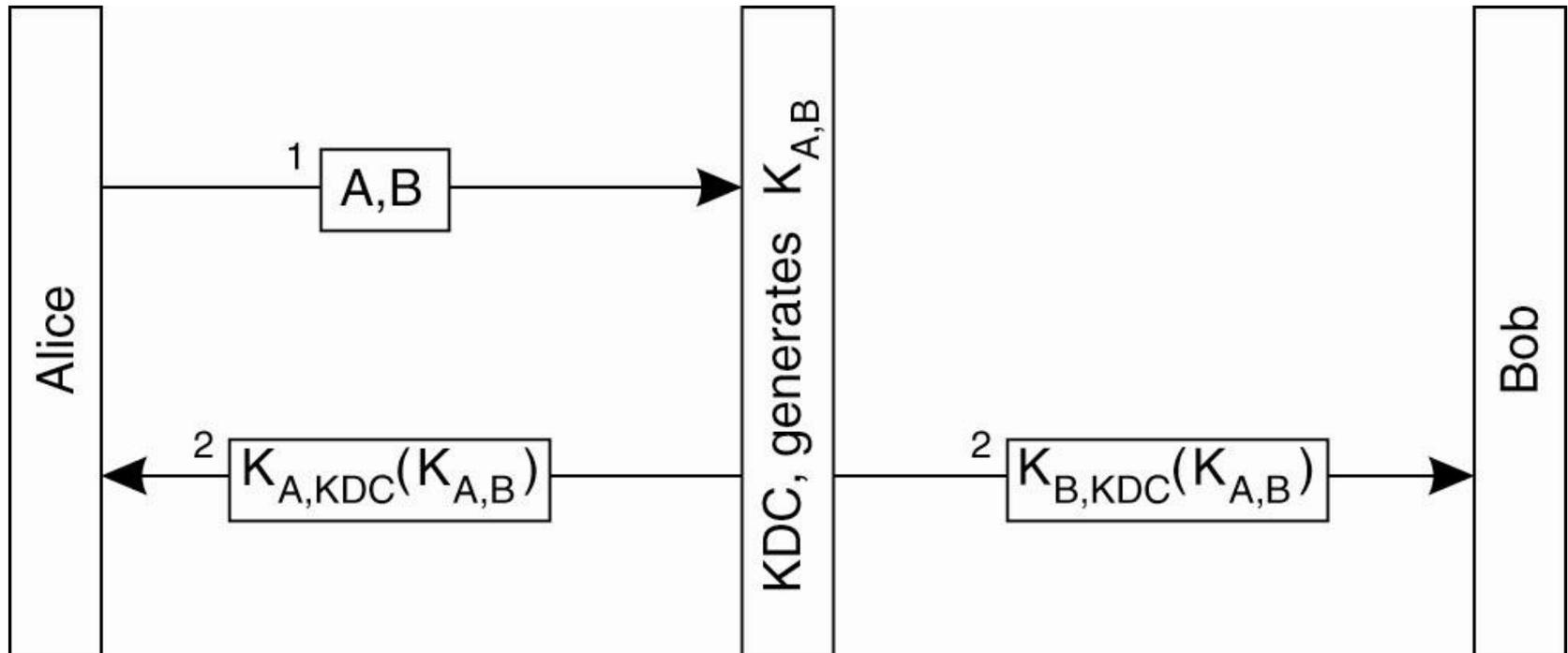
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Reflection Attack



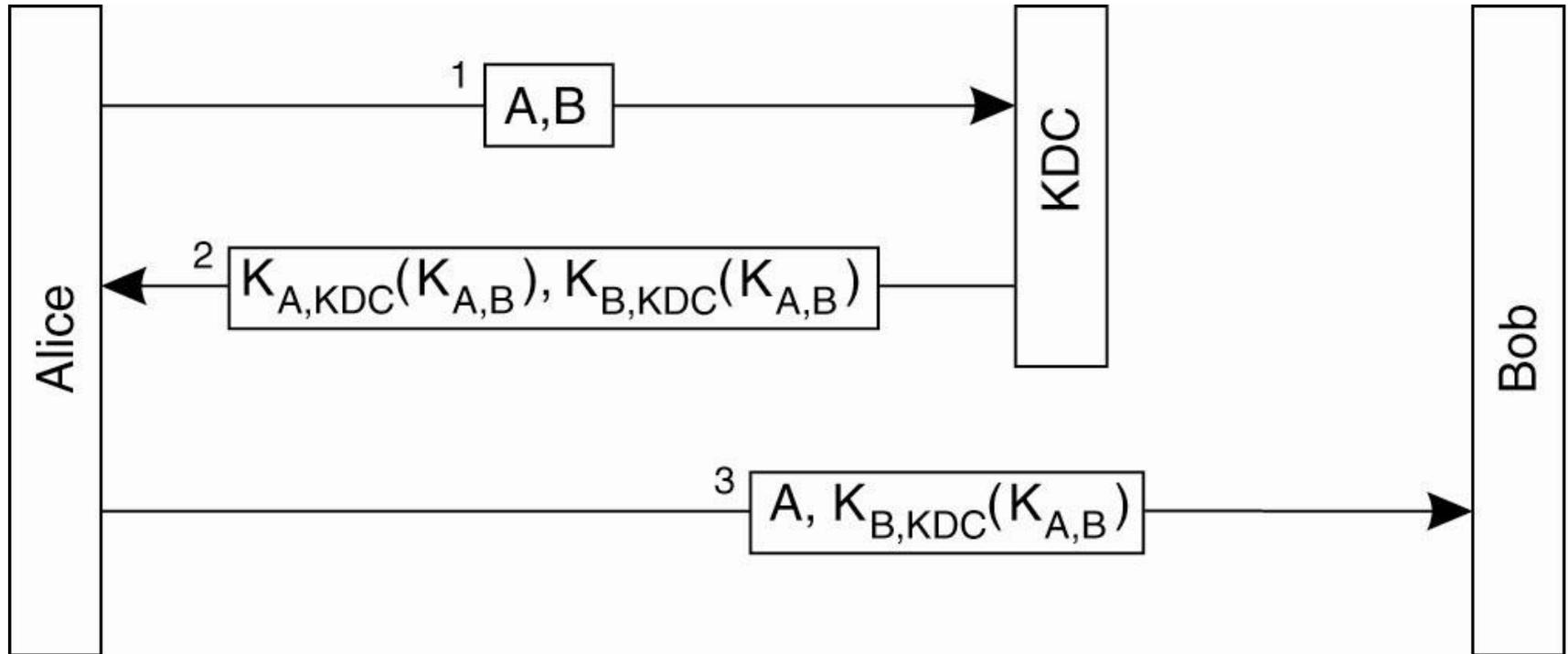
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Key Distribution Center



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

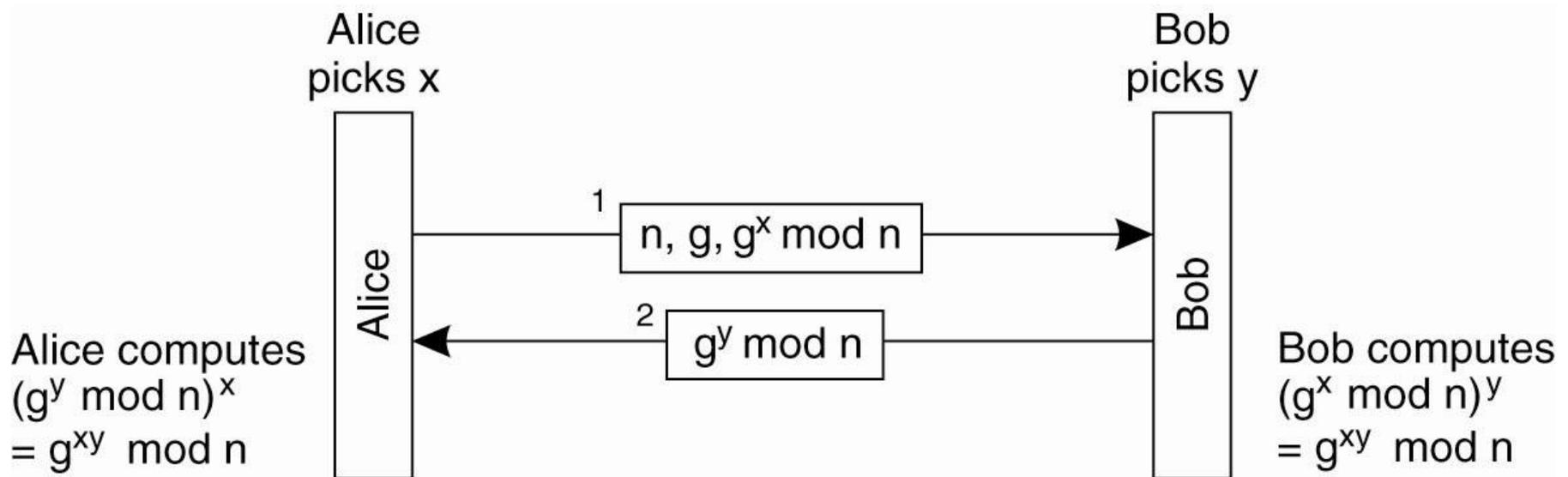
# Authentication using a Key Distribution Center



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

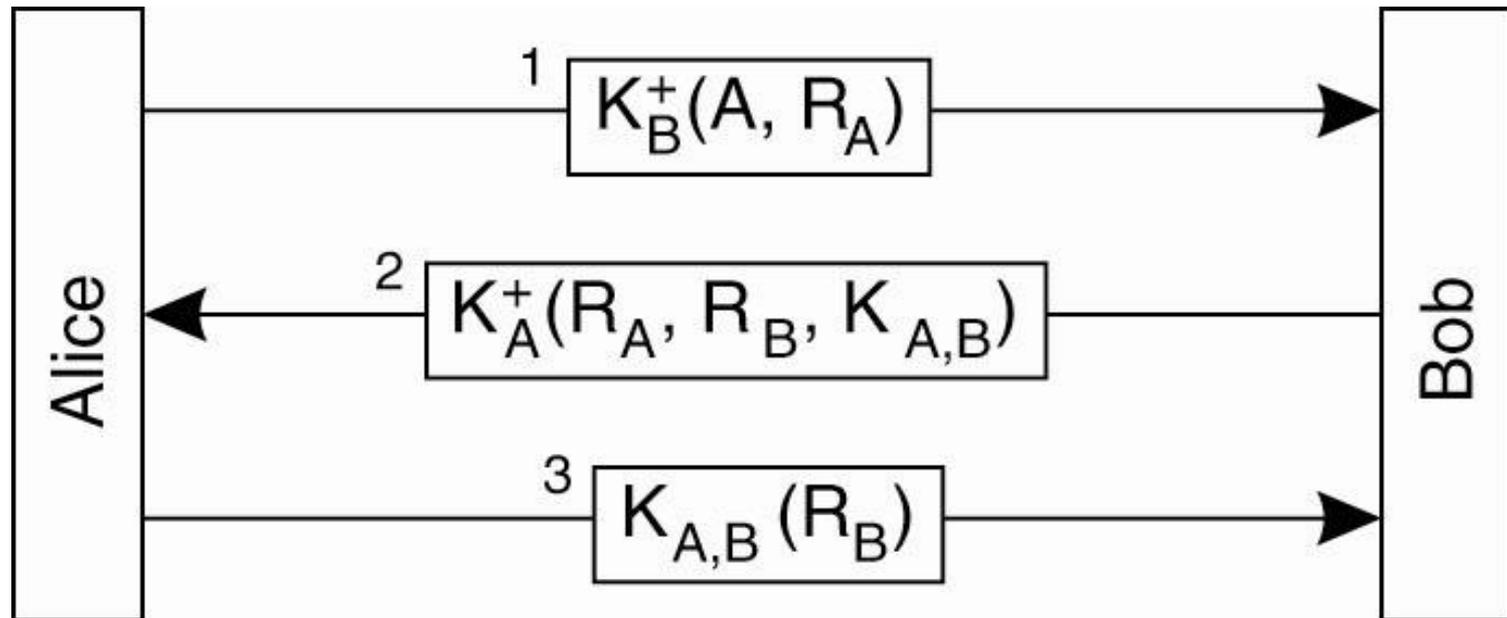
- Asymmetric cryptosystems are **computationally expensive**
  - → generally infeasible to encrypt everything using public keys
- Hence standard approach
  - Authenticate participants via public keys
  - Negotiate a shared one-time **session key**
  - Communicate using symmetric encryption using the session key
    - E.g., used in SSL/TLS

# Diffie-Hellman



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Authentication using Asymmetric Cryptosystems (1)



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Authentication using Asymmetric Cryptosystems (2)

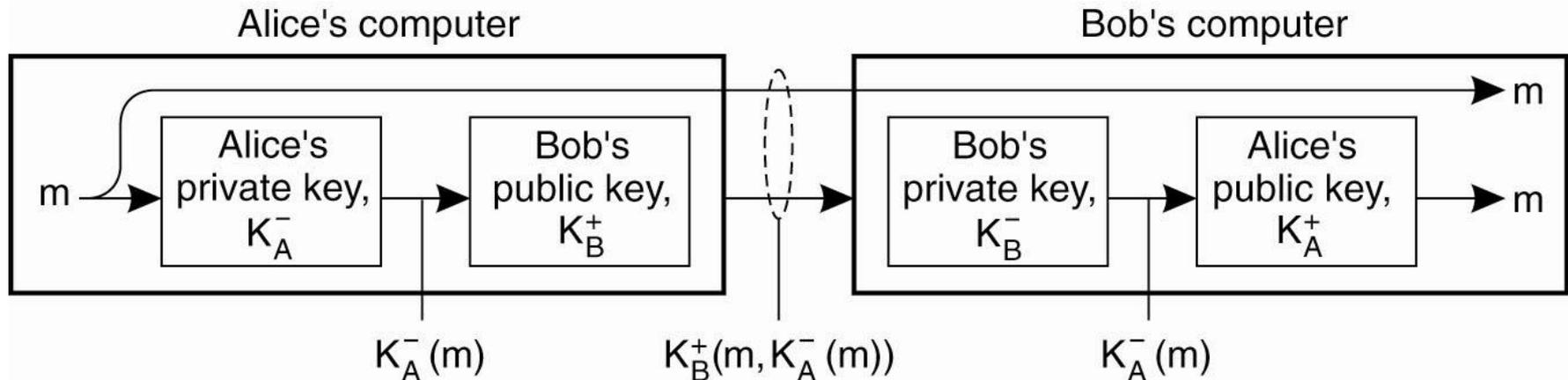
- How do participants learn about  $K_{[A|B]}^+$  ?
- Typically:
  - **Certification Authorities**
  - Trusted entities that generate public-private keypairs and validate the public key – participant mapping



- Keys in a cryptosystem exhibit some **wear-and-tear**
  - Risk of compromisation increases with duration of usage
    - Key expiration
  - In the real world, some keys get compromised anyway (loss, security breaches, ...)
    - Key revocation (e.g., **Certificate Revocation Lists**, CRLs)

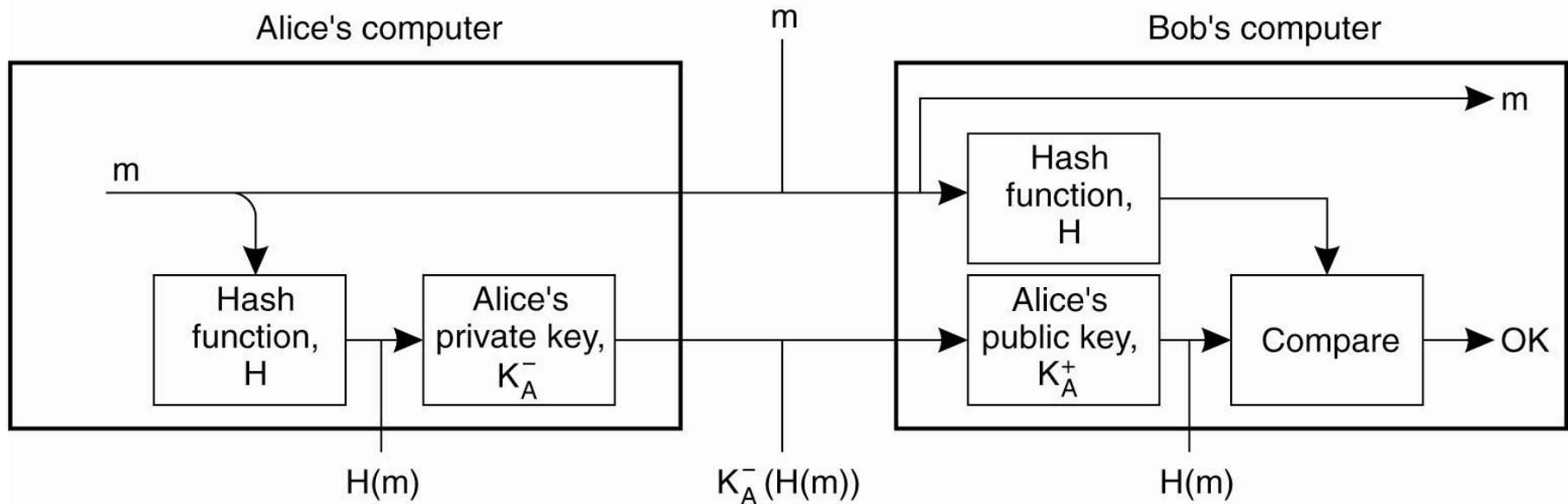
# INTEGRITY

# Signing a Message using Asymmetric Cryptosystems



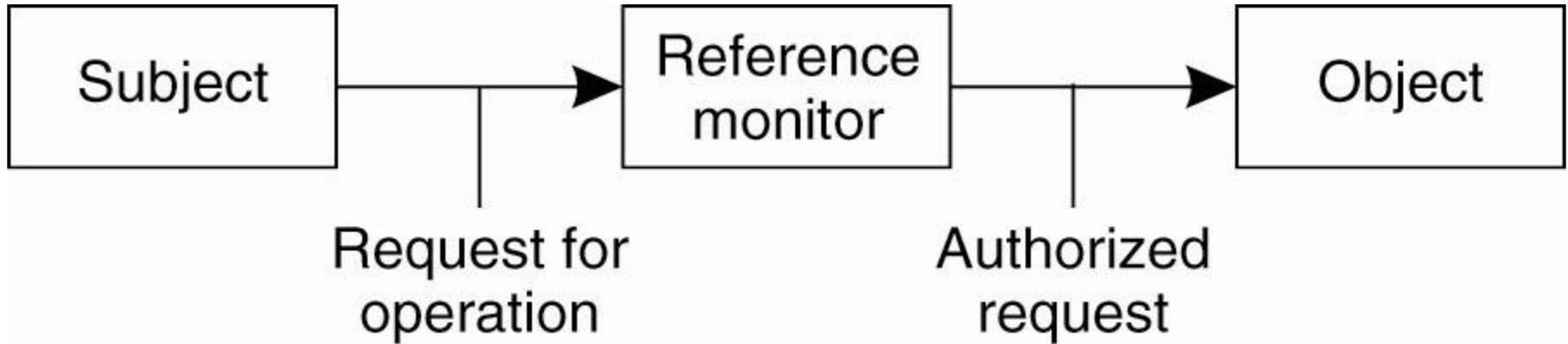
Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Signing a Message using Digital Signatures



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# ACCESS CONTROL



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

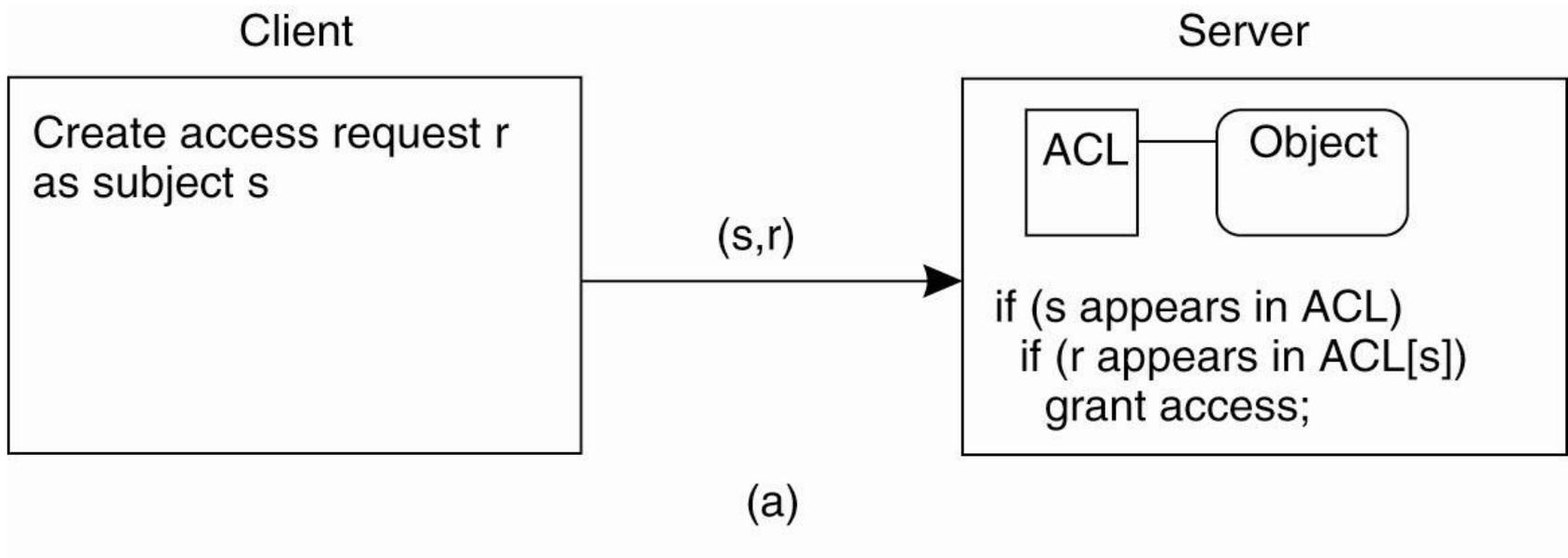
- Approaches
  - Access Control Matrices
  - Access Control Lists
  - Protection Domains

	Object X	Object Y	Object Z
Alice	rw+	rw	r
Bob	r	r	rw+
Chuck	-	-	-

- Access levels (typically):
  - RW+ (read / write / administer rights)
  - RW (read / write)
  - R (read)
  - - (none)

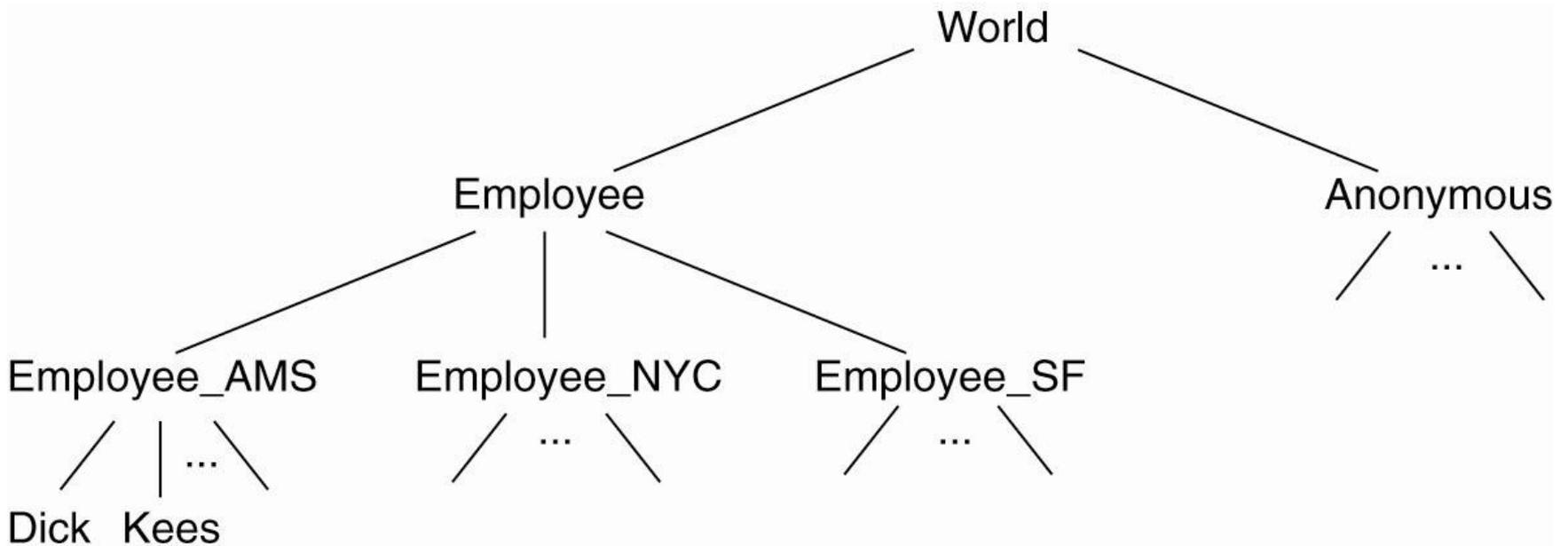
# Access Control Lists (ACLs)

Object X:	Alice: rw+	Bob: r	Chuck: -
Object Y:	Alice: rw	Bob: r	Chuck: -
Object Z:	Alice: r	Bob: rw+	Chuck: -



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# Protection Domains



Source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall

# SOME COMMON ATTACK SCENARIOS

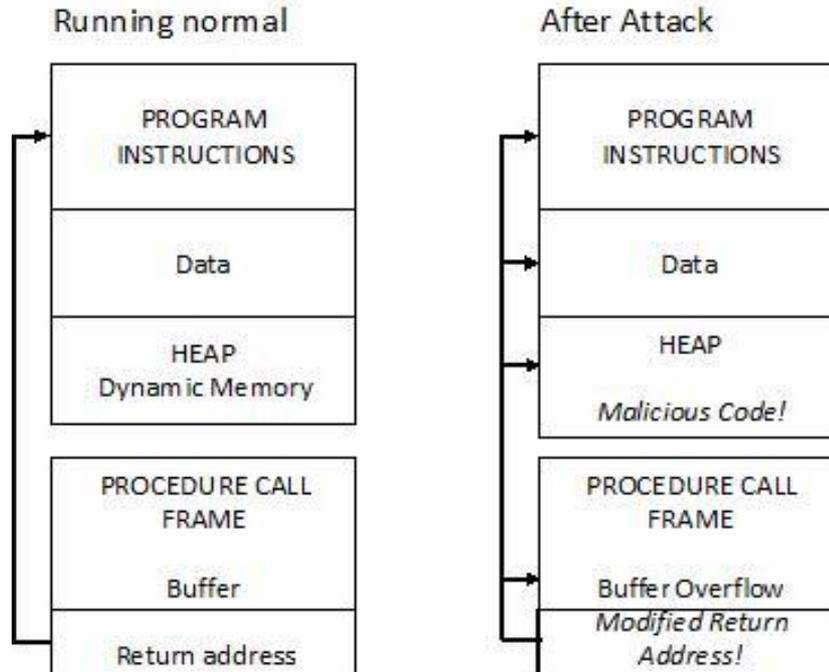
- Distributed systems security can be compromised on any layer
  - → and remember: any security breach potentially renders the entire system insecure
- The following is just a small set of examples of what can go wrong
  - All of the following things happen in practice **all the time**



# Buffer Overflows (1)

- Common security problem in unmanaged programming languages (e.g., C / C++)
  - Input data larger than reserved heap space
  - Hence data **flows over** into next frame, allowing an attacker to overwrite the return address pointer of a procedure call with a custom address
  - Hence allowing the attacker to execute arbitrary code

# Buffer Overflows (2)



Attacker plants code that overflows buffer and corrupts the return address. Instead of returning to the appropriate calling procedure, the modified return address returns control to malicious code, located elsewhere in process memory.

Source: <http://cis1.towson.edu/~cssecinj/modules/cs2/buffer-overflow-cs2-c/>

# SQL Injection Attack (1)

- Some web applications do not sufficiently check data received from users before issuing SQL queries

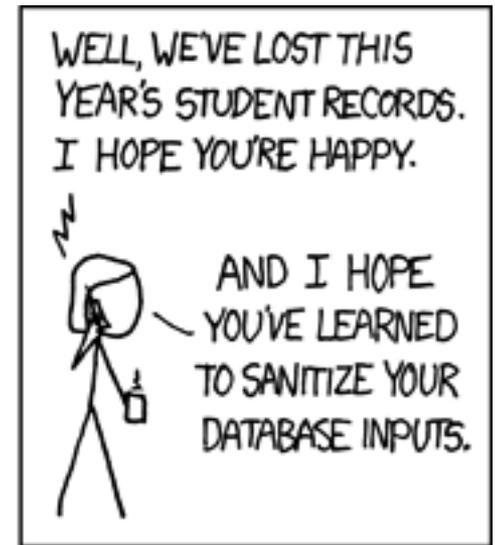
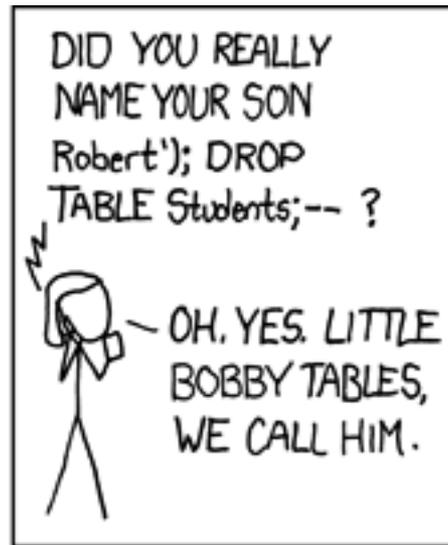
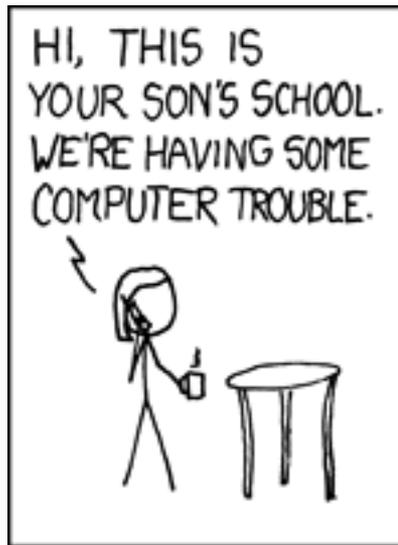
- `select * from users where user = $username and pw = md5($pw)`

- now assume e.g., `$username = ';' drop table users; --'`

- Example: First name:

Last name:

# SQL Injection Attack (2)



# Cross-Side Scripting Attack (XSS)

- Similar principle to SQL injection
- Allows attackers to **inject arbitrary scripts** into a legit (trustable) web site
  - Example: assume you have a blog with a comment function. The comment function accepts arbitrary HTML code.

## Leave a Reply

```

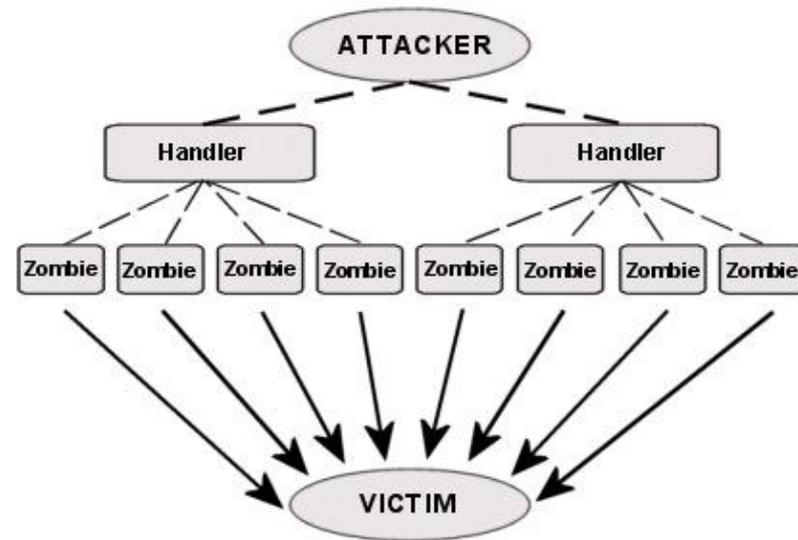
Very interesting article!

<script type="text/javascript">
<!--
  window.location="http://62.178.71.105";
//-->
</script>
    
```



# Distributed Denial-of-Service Attack (DDoS)

- Attacker uses a network of hacked machines (bots, zombies) to **overload** the resources of the target with requests
  - Produces costs and load
  - Server crashes
- Difficult to protect against
- Difficult to identify the attacker
  - All requests come from unassuming zombies



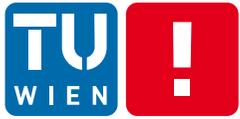
- Attacks that ignore the technical security mechanisms by finding out the **secret** that the mechanism was based on
- E.g.:
  - Phishing attackers steal passwords or keys
  - NSA demanding private keys from certification authorities
  - Hackers reverse-engineering keys in embedded devices by measuring energy consumption



- Catch-all term for various sidechannel attacks that target the **human** behind the (secure?) technical system
- Usual assumption: people are easily manipulated by a well-prepared talker
- Most common: **Phishing**
  - E.g., calling a user and convincing her/him to tell you his account data (*“Hey, I’m Joe from IT. I understand you have had troubles logging in today?”*)

# FURTHER LECTURES

- Internet Security
  - <https://tiss.tuwien.ac.at/course/courseDetails.xhtml?windowId=faa&courseNr=188366&semester=2014S>
- Advanced Internet Security
  - <https://tiss.tuwien.ac.at/course/courseDetails.xhtml?windowId=faa&courseNr=183222&semester=2013W>
- Organizational Aspects of IT Security
  - <https://tiss.tuwien.ac.at/course/courseDetails.xhtml?windowId=faa&courseNr=188312&semester=2013W>
- Seminar aus Security
  - <https://tiss.tuwien.ac.at/course/courseDetails.xhtml?windowId=faa&courseNr=183606&semester=2013W>



# Distributed Systems – Fault Tolerance

Dr. Stefan Schulte  
Distributed Systems Group  
Vienna University of Technology

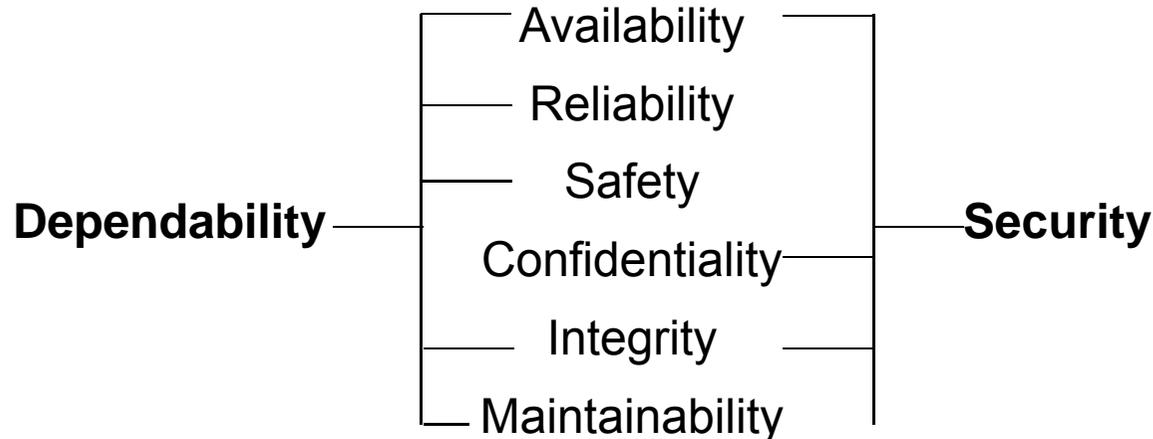
[schulte@infosys.tuwien.ac.at](mailto:schulte@infosys.tuwien.ac.at)

1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

# Dependability

- Basics: In Distributed Systems (DS), *components provide services to clients*
  - To provide services, the component may require services from other components,
  - This means: It *depends* on some other component
  - More specific: The *correctness* of the component in question depends on the *correctness* of another component
- *Dependability* is therefore a core objective in DS

# Dependability: Attributes



- Availability: Immediate readiness for correct service
- Reliability: Continuity of correct service
- Safety: Absence of catastrophic consequences
- Integrity: Absence of improper system alterations
- Maintainability: Ability to undergo modifications

# Threats to Dependability

- *Failure*: Delivered service deviates from correct service, i.e., the system functionality is not delivered anymore
- *Error*: Deviation of the actual system state from the perceived one
- *Fault*: Cause of an error

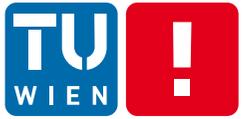
*Fault* → *Error* → *Failure*

# Example I: Failures, Errors, Faults

- Fault: Software bug in a particular method  
(so far, the fault is *dormant*: As long as nobody calls the method, it will not become *active*)
- Error: The method is called (fault becomes *active*), leading to calculation of wrong value
- Failure: If there is no mechanism to identify the error, it will lead to incorrect service of the component calling the method

## Example II: Failures, Errors, Faults

- Fault: Defect USB port of external drive  
(as long as you don't make use of the drive: fault is dormant, your computer is still working)
- Error: Input/Output operation started; bit errors occur
- Failure: It is not possible to correctly copy files from/to the external drive



# Fault Classes

- Development faults
- Operational faults
- Hardware faults
- Software faults
- Malicious faults
- Accidental faults
- Incompetence faults
- ...

# Failure Models

- **Crash Failure:** Component halts, but is working correctly until that moment.
- **Omission Failure:** Component fails to respond
- **Timing Failure:** Answer to request is too late  
(Performance Failure)
- **Response Failure:** Reproducible failures with correct input but wrong output  
(Common-Mode Failure)
- **Arbitrary Failure:** Arbitrary failures at arbitrary times  
(Byzantine Failures)



- Distributed Systems (DS) can become very complex:
  - The question is not *IF* something will go wrong, the question is *WHEN* this will happen: Faults are inevitable!
- However, a DS should not completely fail if a failure occurs:
  - *Partial Failure*: A failure in one component does not have to lead to a failure in another component or the whole system

# What to do about Faults?

- Fault Prevention:
  - Prevent the occurrence of a fault
- Fault Forecasting:
  - Estimate present and future faults and their consequences
- **Fault Tolerance:**
  - **Avoid that service failures occur from faults, i.e., *masks* the presence of faults**
  - **Service provision is continued!**
- Fault Removal:
  - Reduce the number and severity of faults

# Approaches to Fault Tolerance

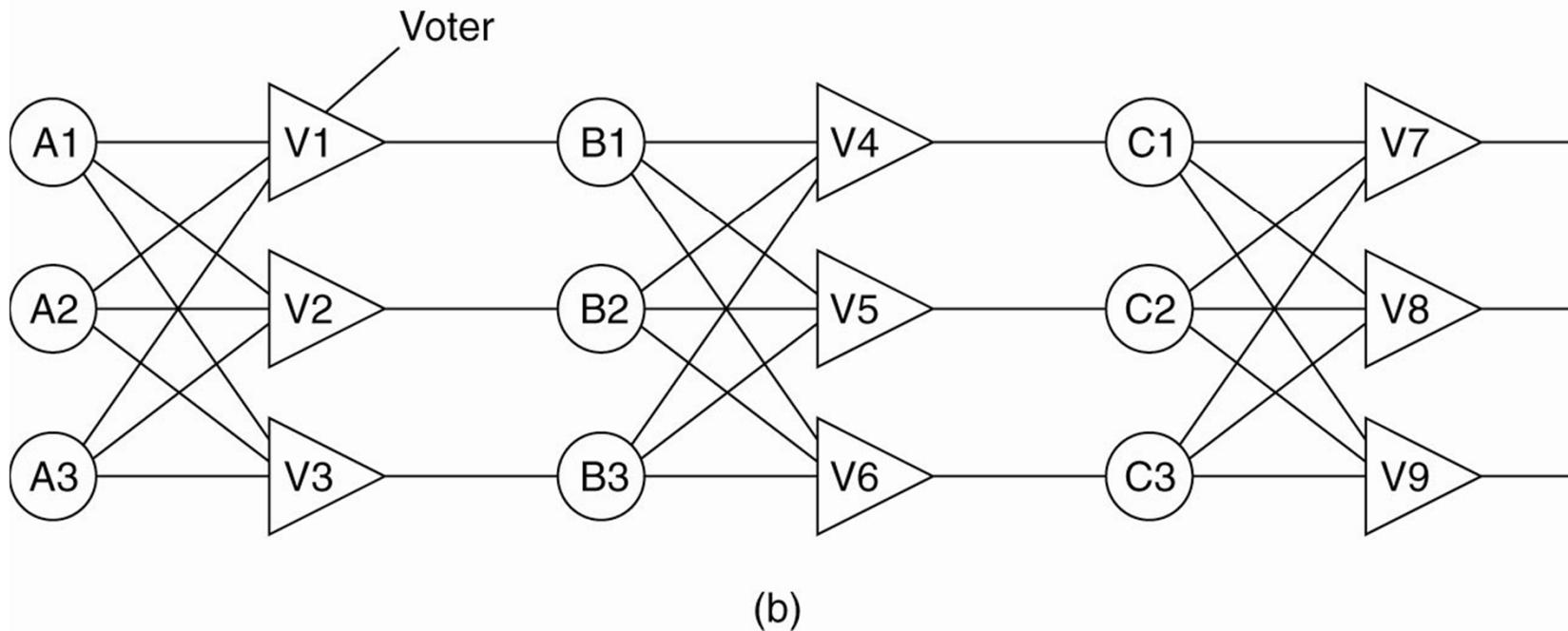
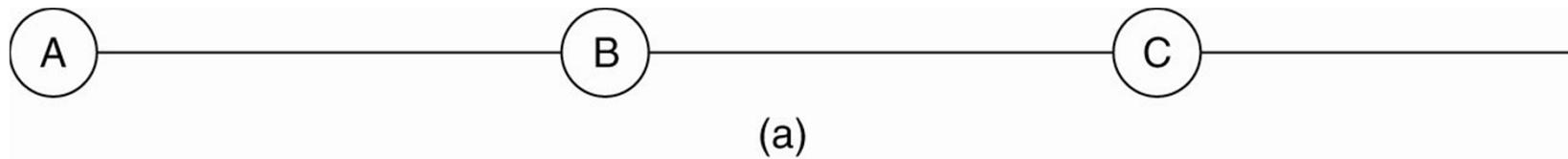
- “No Fault Tolerance Without Redundancy” (Gärtner, 1999)
  - Use redundancy to mask a failure, i.e., hide the occurrence of a fault
- Failure Masking by Redundancy:
  - Information Redundancy: Add extra information
  - Time Redundancy: Repeat request
  - Physical Redundancy: Add additional components

# Redundancy – Examples

- Information Redundancy:
  - Add a parity bit
  - Error Correcting Codes (memory)
  - Hard disks in a RAID 4+5
- Time Redundancy:
  - Retransmissions in TCP/IP
  - Call method again
- Physical Redundancy:
  - Backup server
  - Hard disks in a RAID 1
  - But also: Different implementations of same functionality

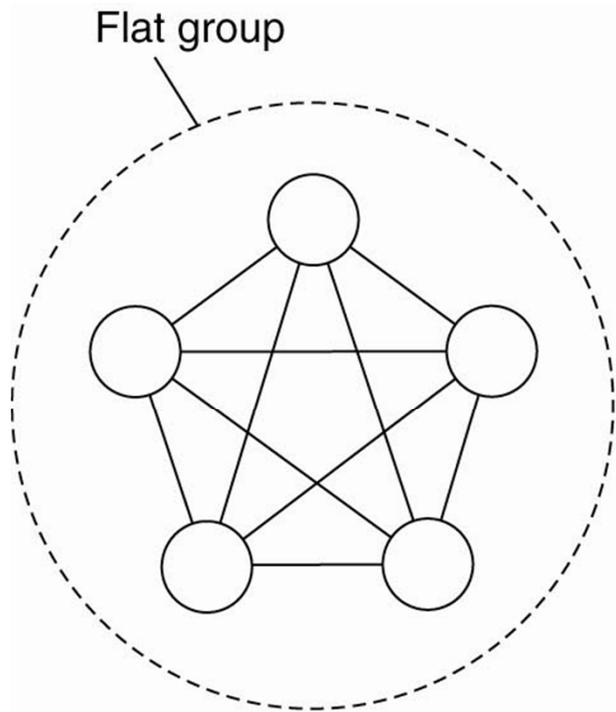
# Physical Redundancy – Example

Electronic circuit with Triple Modular Redundancy:



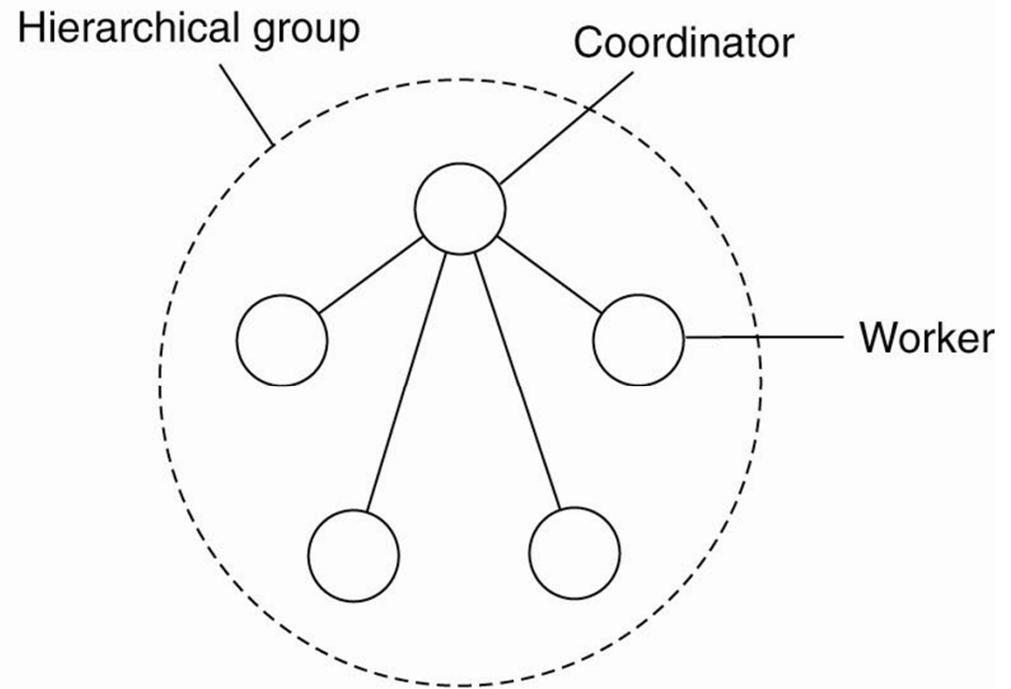
1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

- How to tolerate faulty processes?
  - “No Fault Tolerance Without Redundancy”
    - Organize several identical processes into a group



DS WS 2013 (a)

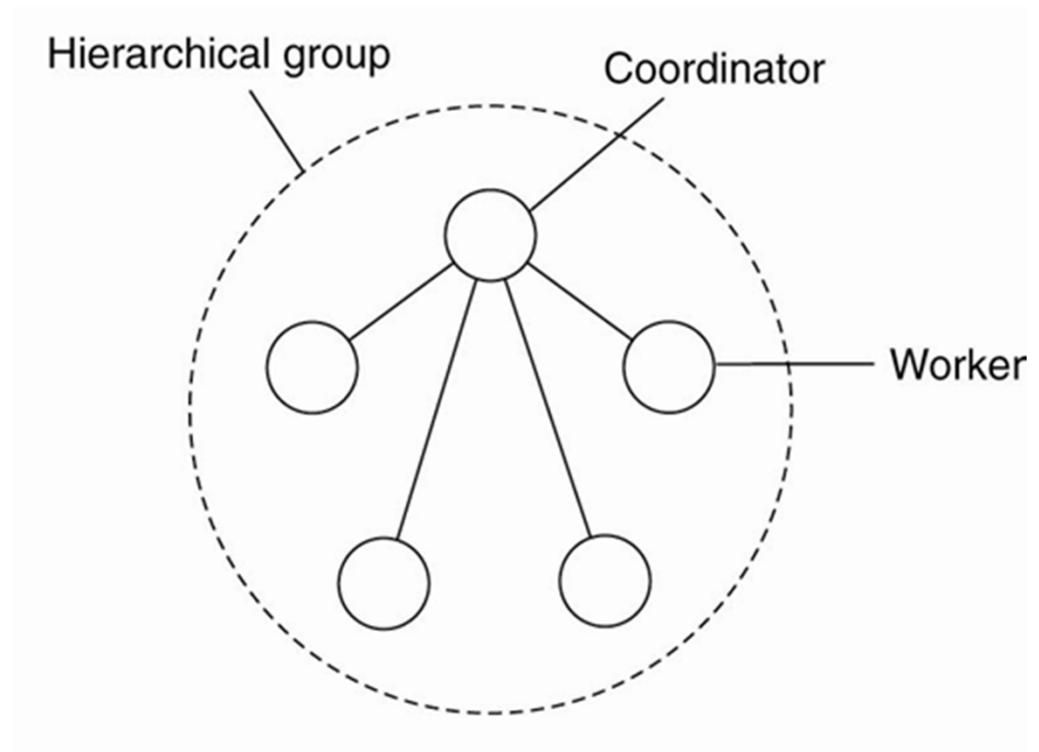
16



(b)

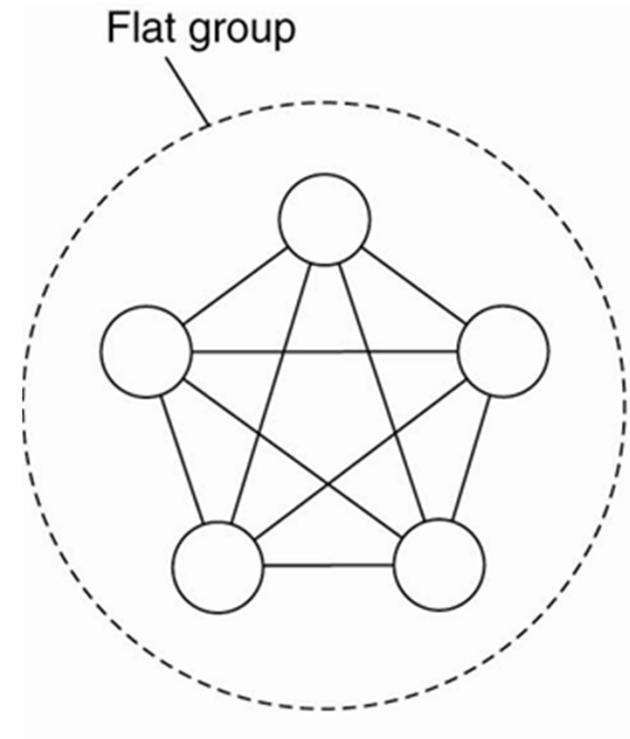
# Communication in Hierarchical Groups

- Hierarchical Groups:
  - Communication through a single coordinator
  - Not really fault tolerant or scalable
  - However, easier to implement



# Communication in Flat Groups

- Flat Groups:
  - Good for fault tolerance as information exchange immediately occurs with all group members
  - May impose overhead as control is completely distributed, and voting needs to be carried out
  - Harder to implement

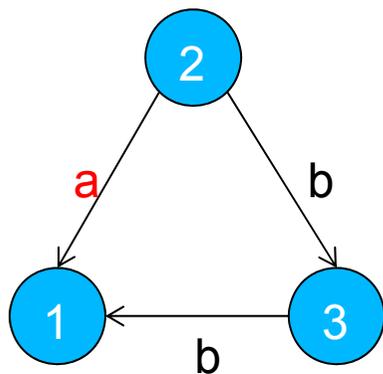


# Groups and Failure Masking

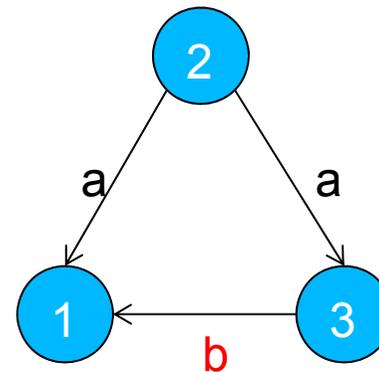
- k-fault tolerant group:
  - Group is able to mask any  $k$  concurrent member failures
- How large does a k-fault tolerant group need to be?
  - Crash/performance failure models (i.e., components don't answer anymore):  $k+1$  are necessary
  - Arbitrary/Byzantine failure model:  $2k+1$  components are necessary
- Assumptions: All members are identical and process all input in the same order

# Groups and Failure Masking II

- Scenario (distributed computation):
  - At least one group member different from the others
  - Non-faulty members should have to reach agreement on the same value



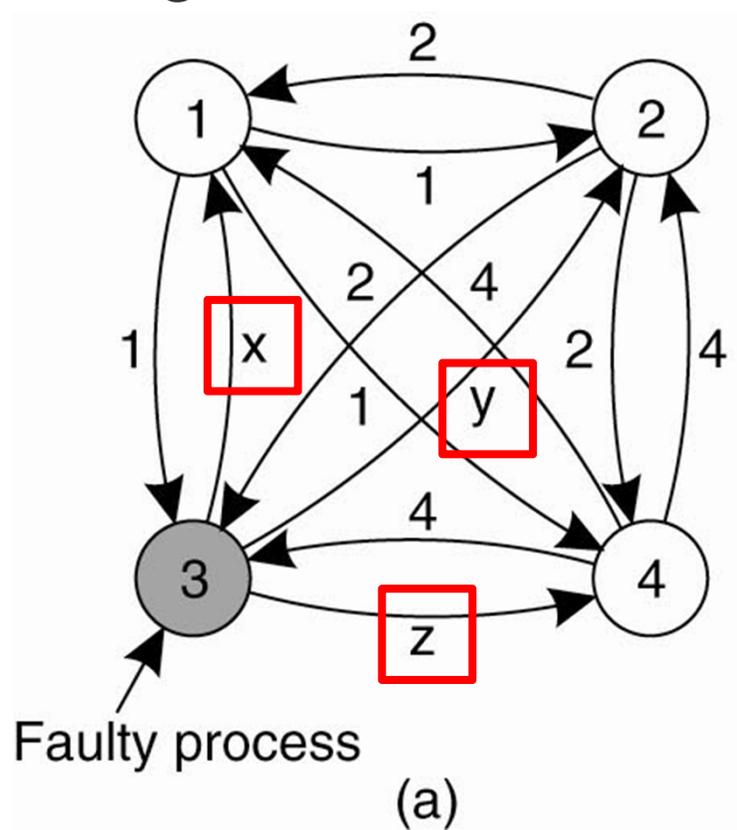
Process 2 tells  
different things



Process 3 passes  
a different value

# Byzantine Agreement Problem I

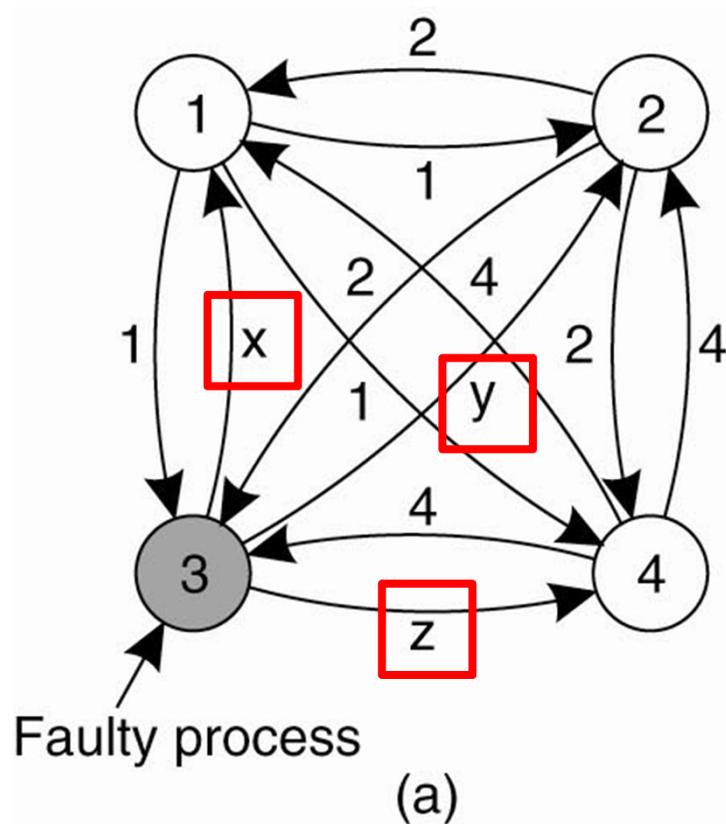
- Byzantine Agreement Problem:



- Assumptions:
  - Unicast messages
  - Ordered message delivery
  - Synchronous processes
  - Bounded communication delay

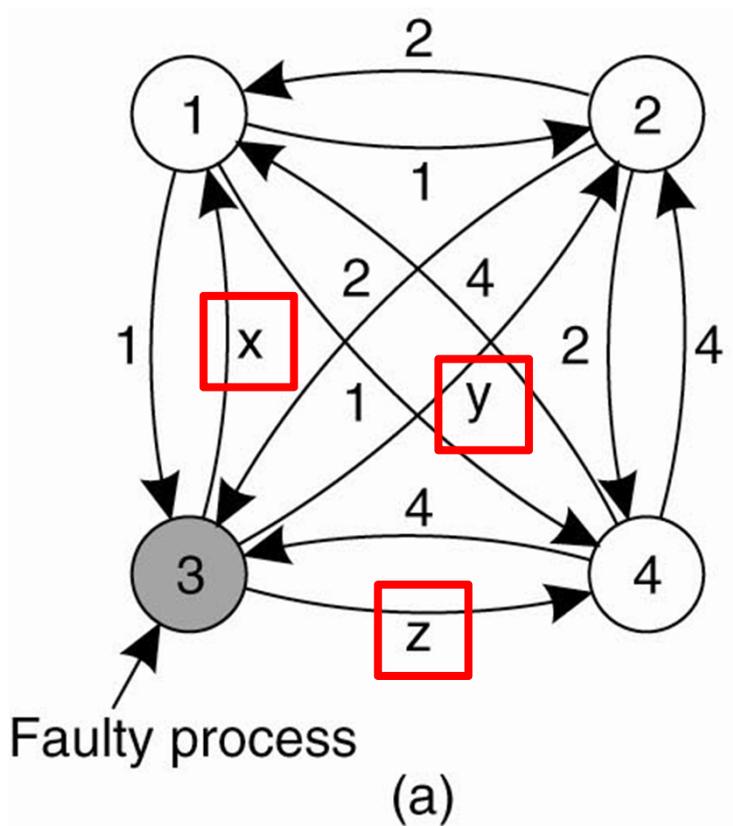
# Byzantine Agreement Problem II

- $N$  processes
- Each process sends value  $v_i$  to the others
- Each process builds a vector  $V$  from the values
- If process  $i$  is non-faulty,  $V[i]=v_i$



# Byzantine Agreement Problem III

- Step 1: Messages are sent



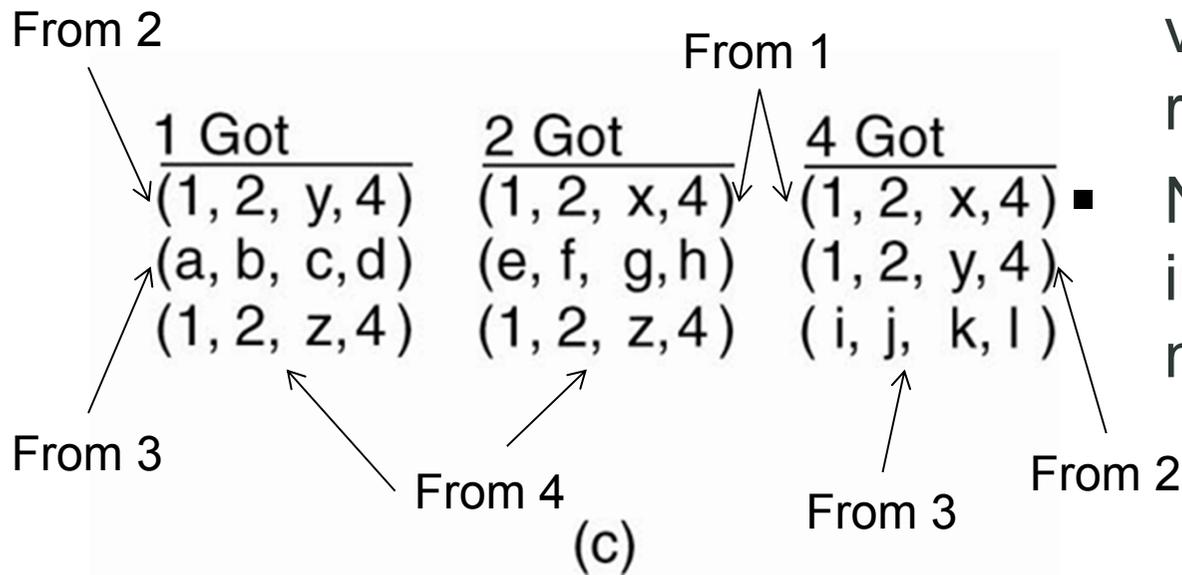
- Step 2: Results - Individual V

1 Got(1, 2  $x$ , 4)  
 2 Got(1, 2  $y$ , 4)  
 3 Got(1, 2  $z$ , 4)  
 4 Got(1, 2  $z$ , 4)

(b)

# Byzantine Agreement Problem IV

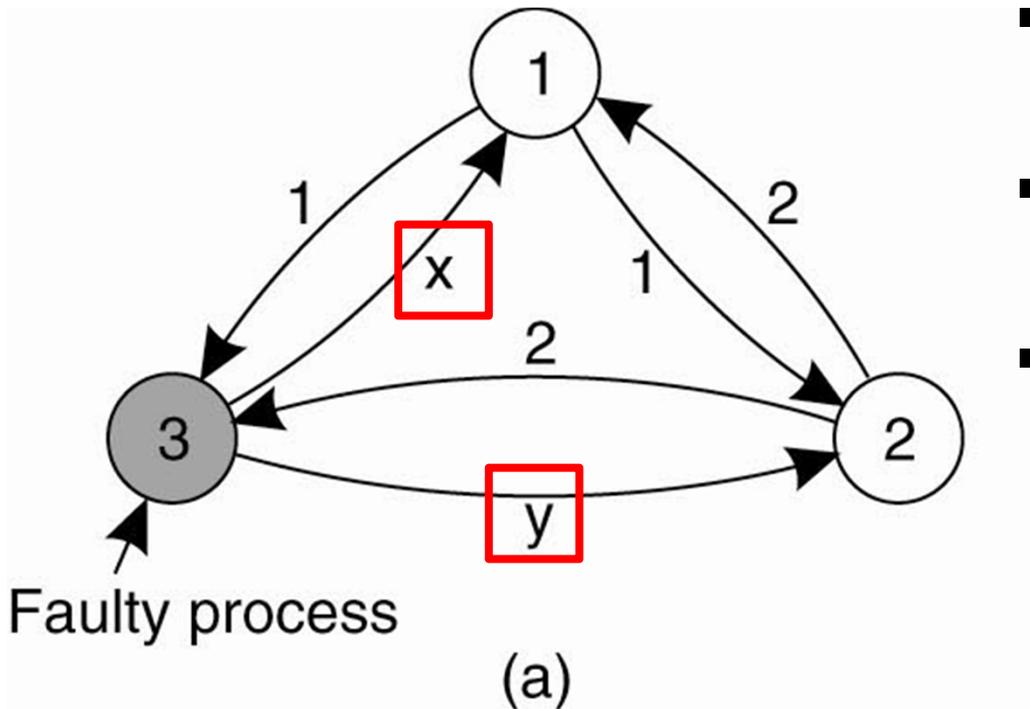
- Step 3:
  - Every process passes its vector  $V$
  - Process 3 „lies“ to everyone
- Step 4:
  - Each process examines  $i$ th element of *received* vectors
  - If there is a majority, value is put into resulting vector
  - No majority: element in result vector is marked *UNKNOWN*



# Byzantine Agreement Problem V

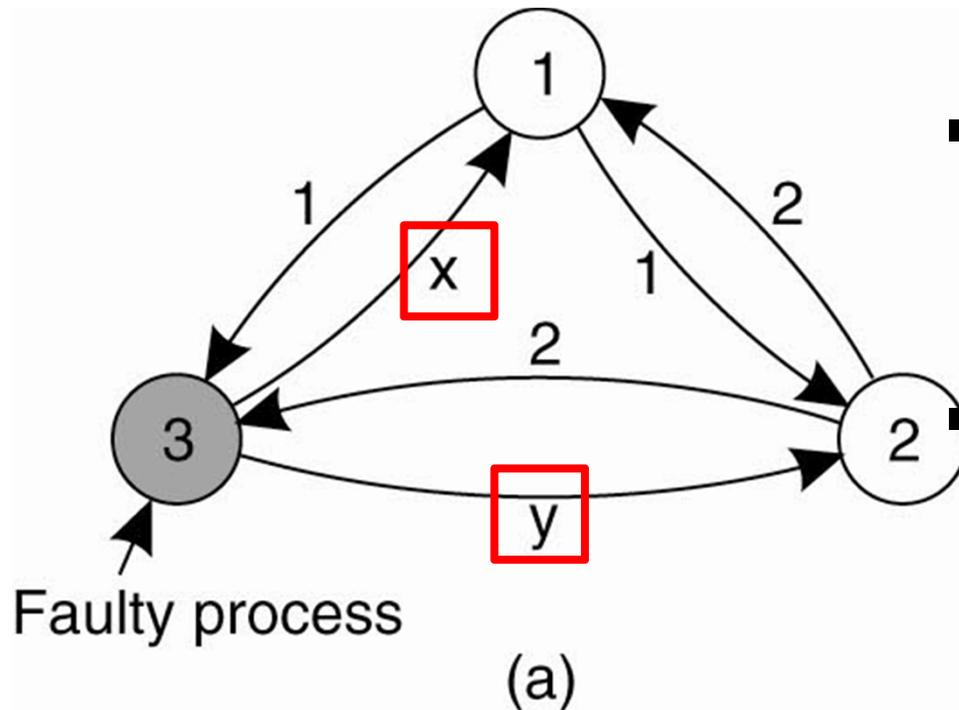
- Byzantine Agreement Problem:

- Assumptions:
  - Unicast messages
  - Ordered message delivery
  - Synchronous processes
  - Bounded communication delay



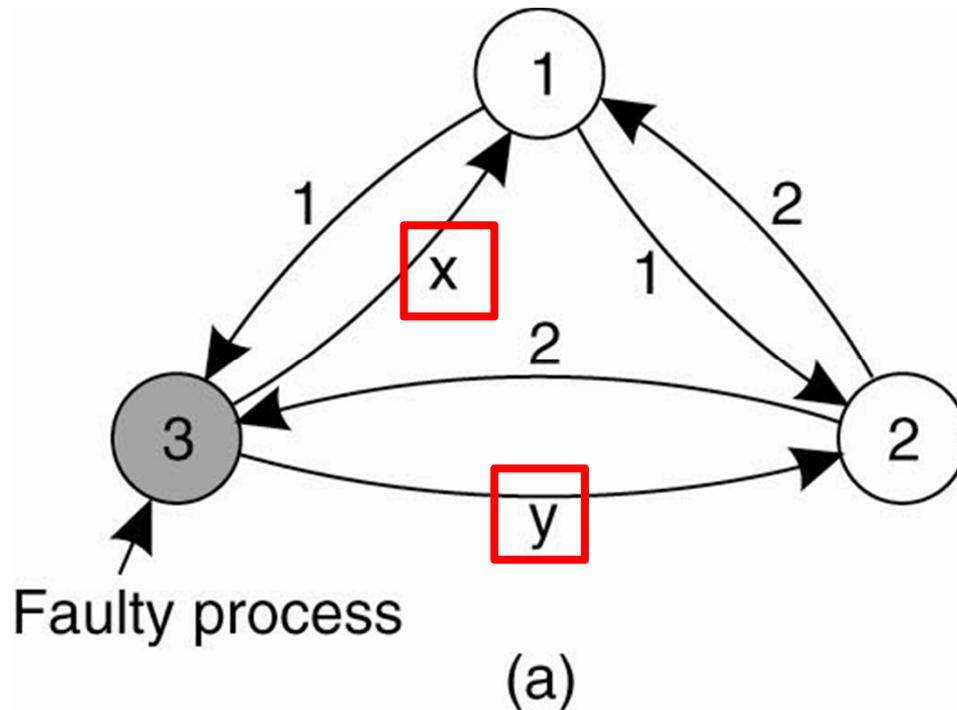
# Byzantine Agreement Problem VI

- $N$  processes
  - Each process sends value  $v_i$  to the others
  - Each process builds a vector  $V$  from the values
- If process  $i$  is non-faulty,  $V[i]=v_i$



# Byzantine Agreement Problem VII

- Step 1: Messages are sent
- Step 2: Results - Individual V



```

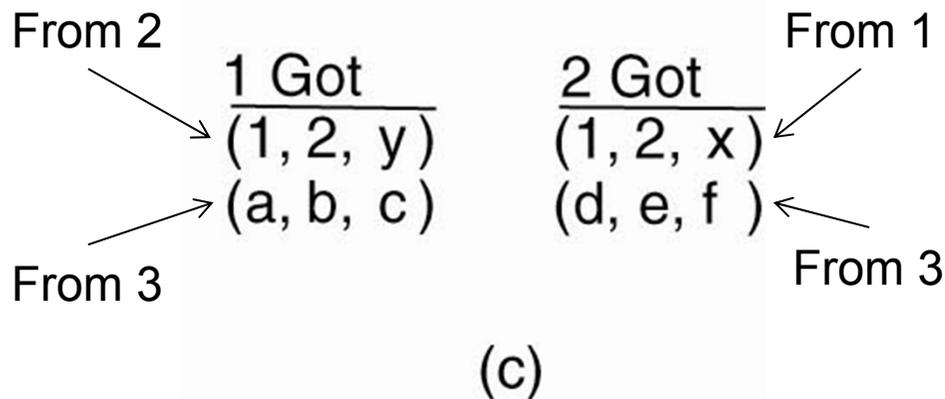
1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

```

(b)

# Byzantine Agreement Problem VIII

- Step 3:
  - Every process passes its vector  $V$
  - Process 3 „lies“ to everyone
- Step 4:
  - Each process examines  $i$ th element of *received* vectors
  - If there is a majority, value is put into resulting vector
  - No majority: element in result vector is marked *UNKNOWN*



# Byzantine Agreement Problem IX

- 4 Processes:

<u>1 Got</u>	<u>2 Got</u>	<u>4 Got</u>
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

- Agreement for  $v_1, v_2, v_4$

- 3 Processes:

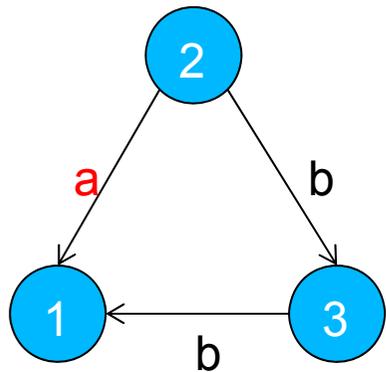
<u>1 Got</u>	<u>2 Got</u>
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

- No agreement possible!
- $2k+1$  non-faulty processes are necessary for  $k$ -fault tolerance

1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

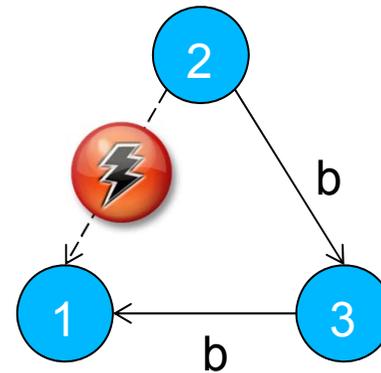
# Reliable Client-Server Communication

- So far: Process Resilience

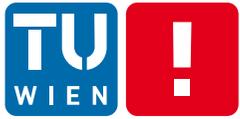


Process 2 tells different things

- But what about reliable communication channels?



Connection between Process 2 and Process 1 fails



# Remote Procedure Calls: What can go wrong?

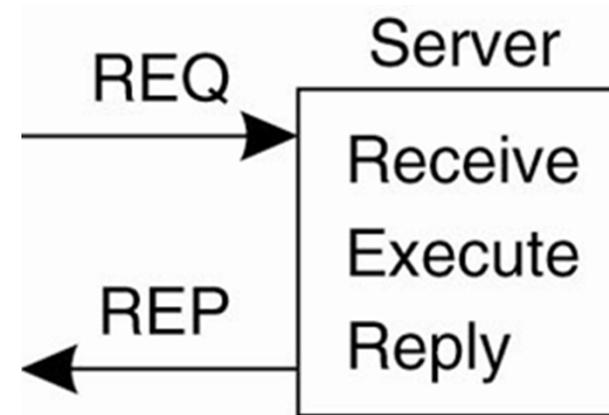
1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes (after request has been sent)

# Remote Procedure Calls: Solutions I

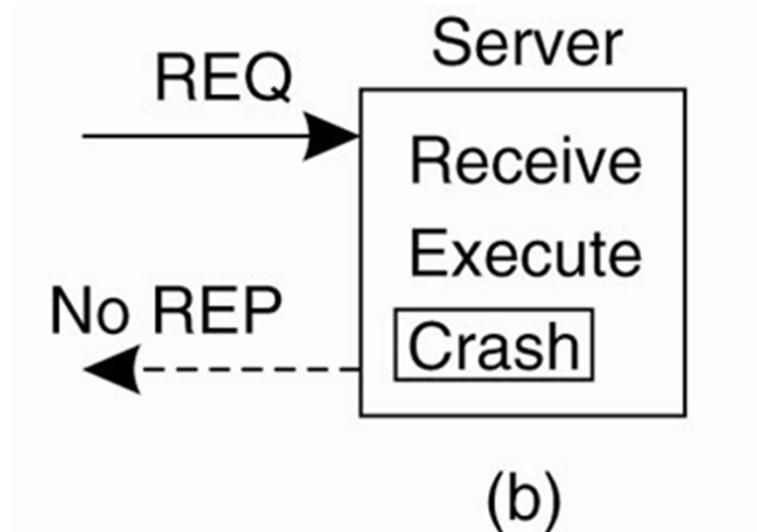
1. Client cannot locate server
  - Just report back to client
  - Client has to take care of it (e.g., exception handling)
  
2. Client request is lost
  - Resend request message
  - Server won't know difference between original and retransmission

# Remote Procedure Calls: Solutions II

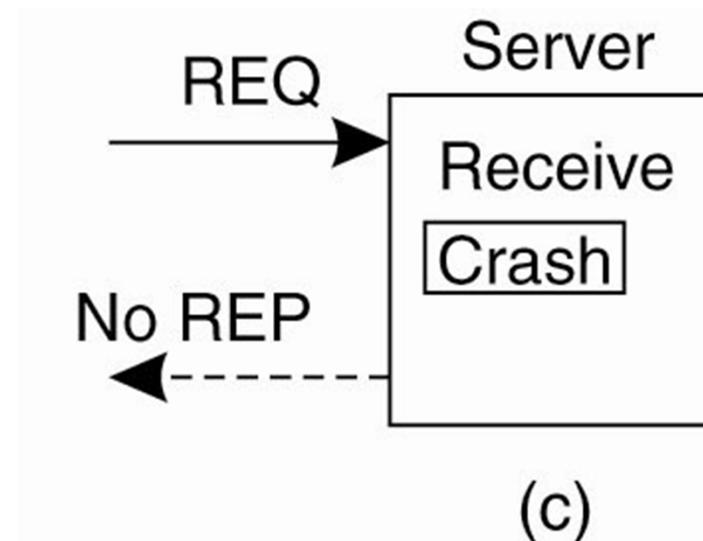
3. Server crashes
  - a) Normal case
  - b) Crash *after* execution
  - c) Crash *before* execution



(a)



(b)



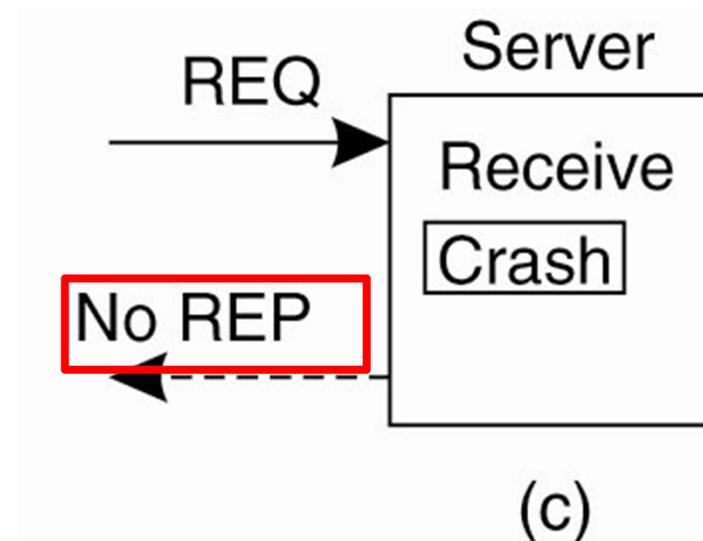
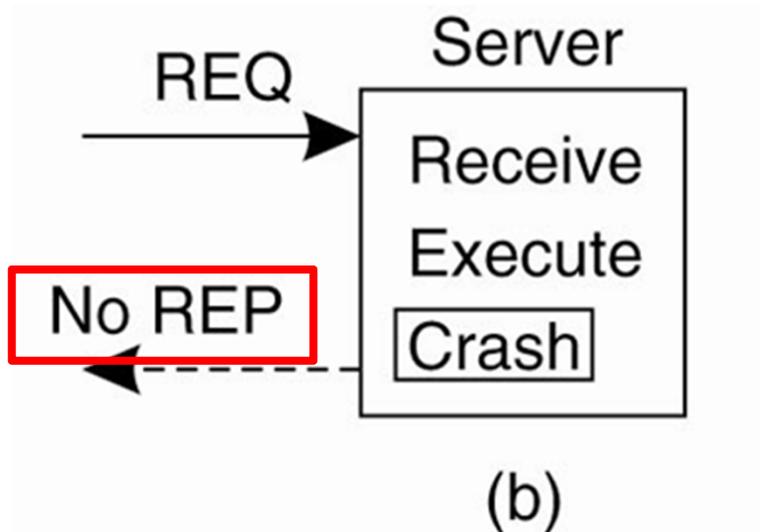
(c)

# Remote Procedure Calls: Solutions II

## 3. Server crashes

- a) Normal case – no crash
- b) Crash *after* execution
- c) Crash *before* execution

The client is not able to see the difference



# Remote Procedure Calls: Solutions III

## 3. Server crashes

- Correct behaviour of Client depends on behaviour of Server
  1. *At-least-once-semantics*: The Server guarantees it will carry out an operation at *least* once, no matter what
  2. *At-most-once-semantics*: The Server guarantees it will carry out an operation at *most* once.
- And the Client? (if not receiving a reply, but a message that the server has rebooted)
  1. *Always* reissues a request
  2. *Never* reissues a request
  3. Reissue a request only if it did not receive an ACK (that request has been delivered)
  4. Reissue a request only if it did receive an ACK

# Remote Procedure Calls: Solutions IV

## 3. Server crashes

- 8 possible combinations of strategies
- Example: Client sends printing request to Print Server
  - Three events may happen at the Server:
    - (M) Send the completion message (ACK)
    - (P) Print the text
    - (C) Crash
- There is no combination of server and client strategies that will work correctly under all possible event sequences.

# Remote Procedure Calls: Solutions V

## 3. Server crashes

- These events can occur in six different sequences:
  1.  $M \rightarrow P \rightarrow C$ : A crash occurs after sending the completion message and printing the text.
  2.  $M \rightarrow C (\rightarrow P)$ : A crash happens after sending the completion message, but before the text could be printed.
  3.  $P \rightarrow M \rightarrow C$ : A crash occurs after sending the completion message and printing the text.
  4.  $P \rightarrow C (\rightarrow M)$ : The text printed, after which a crash occurs before the completion message could be sent.
  5.  $C (\rightarrow P \rightarrow M)$ : A crash happens before the server could do anything.
  6.  $C (\rightarrow M \rightarrow P)$ : A crash happens before the server could do anything.

# Remote Procedure Calls: Solutions VI

## 3. Server crashes

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once  
 DUP = Text is printed twice  
 ZERO = Text is not printed at all

M = Send the completion message, P = Print, C = Crash

Example: Client wrongly assumes Print hasn't been carried out

# Remote Procedure Calls: Solutions VII

## 4. Server response is lost

- How do we know that the server has not crashed?
- Once again: Has the server carried out the operation?
- Repeat request:  
→ In case of real-world impact? Transfer from your banking account carried out twice?
- *No real solution!* Except making operations idempotent, i.e., repeatable without any harm

# Remote Procedure Calls: Solutions VIII

5. Client crashes (after request has been sent)
  - Server executes requests anyway and sends response (called orphan computation)
  - Different Solutions:
    1. Orphan is killed by Client if it is received
    2. Reincarnation: Client tells Servers that it has rebooted; Server kills orphans
    3. Expiration: Require computations to complete in  $T$  time units. Old ones are simply removed.



1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

# Recovery

- So far: Tolerate faults
- But what if a failure occurs nevertheless?
  - Recovery is complicated as processes need to cooperate to identify a *consistent state* from where to recover
- Bring the system into an error-free state:
  - Forward error recovery: Find a new state from which the system can continue operation
  - Backward error recovery: Bring the system back into a previous error-free state
    - Usually applied

# Backward Recovery

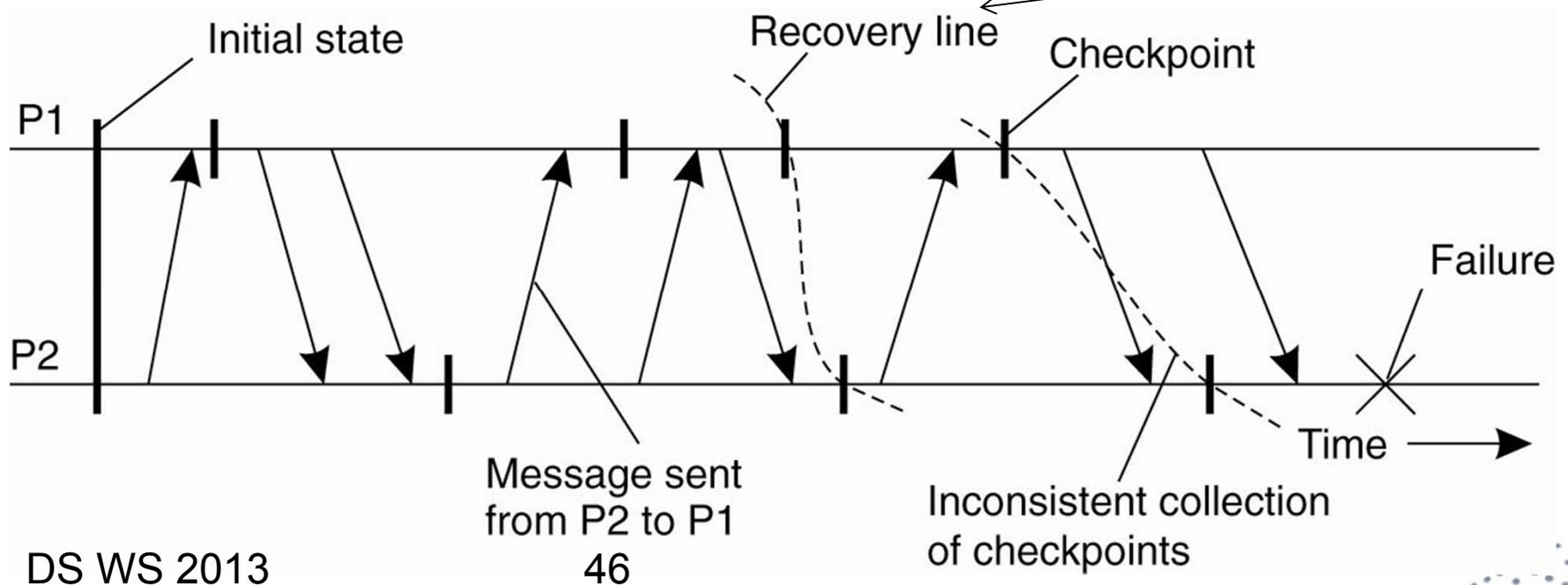
- Bring system from its present erroneous state to a previously correct state:
  - Makes it necessary to record the system's state from time to time, i.e., *checkpointing*
- Benefit: Generally applicable method
- Drawbacks:
  - Relatively costly
  - No guarantee that the same failure won't happen again
  - Some things are simply irreversible



# Independent Checkpointing

- Distributed nature of checkpointing (each process records local state from time to time) makes it difficult to find a *recovery line*

Distributed Snapshot  
(Most recent consistent collection of snapshots)



# Coordinate Checkpointing

- As the name implies: Each process takes a checkpoint after a globally coordinated action
  - Coordinator necessary!
- Two-phase blocking protocol:
  1. Coordinator multicasts a *checkpoint request* message
  2. When participant receives this message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
  3. When all checkpoints have been confirmed at the coordinator, the latter broadcasts a *checkpoint done*
  4. Processes continue

# Message Logging

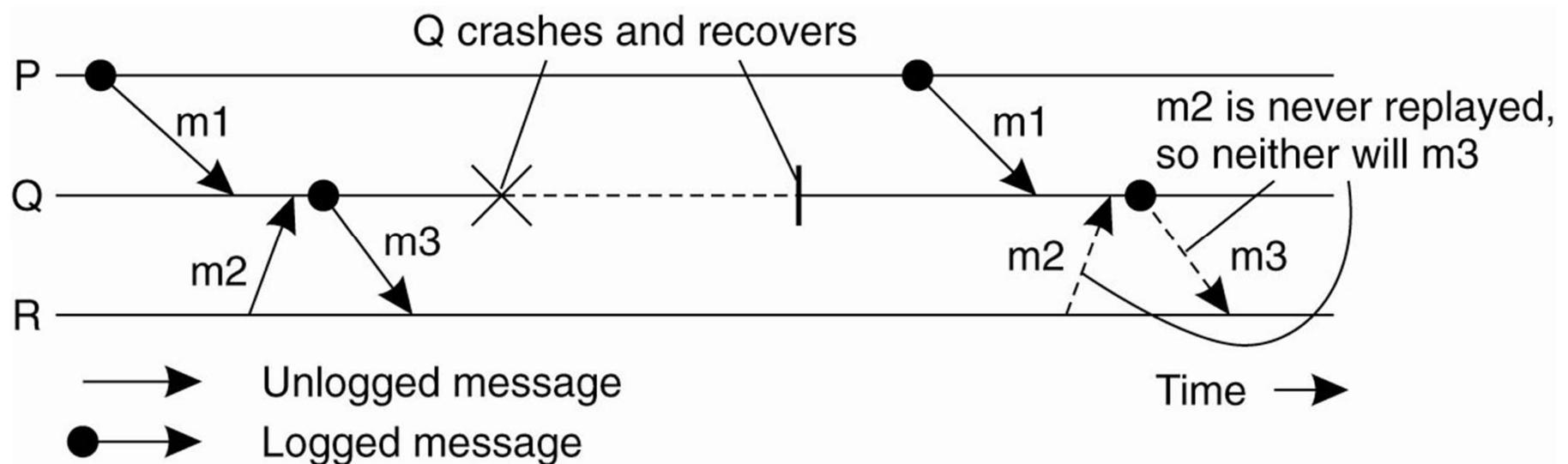
- Alternative to checkpointing
  - Less costly than checkpointing
  - Nevertheless needs some checkpoints
- Instead of taking a checkpoint, try to *replay* communication behaviour from the most recent checkpoint

# Message Logging – Basic Assumption

- Piecewise deterministic execution model:
  - The execution of each process can be considered as a sequence of state intervals
  - Each state interval starts with a nondeterministic event (e.g., message receipt)
  - Execution in a state interval is completely deterministic
- If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

# Message Logging – Avoid Orphans

- Example:
  - Process Q has just received and subsequently delivered messages  $m_1$  and  $m_2$
  - Assume that  $m_2$  is never logged.
  - After delivering  $m_1$  and  $m_2$ , Q sends message  $m_3$  to process R
  - Process R receives and subsequently delivers  $m_3$



## Further Readings

- Tanenbaum, van Steen: Distributed Systems – Principles and Paradigms, 2nd edition, 2007.
- Jalote: Fault Tolerance in Distributed Systems, 1998.
- Avizienis, Laprie, Randell, Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, 1(1), 2004.
- Gärtner: Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments, ACM Computing Surveys, 31(1), 1999.



# Distributed Systems – Current Trends in Distributed Systems

Dr. Stefan Schulte  
Distributed Systems Group  
Vienna University of Technology

[schulte@infosys.tuwien.ac.at](mailto:schulte@infosys.tuwien.ac.at)



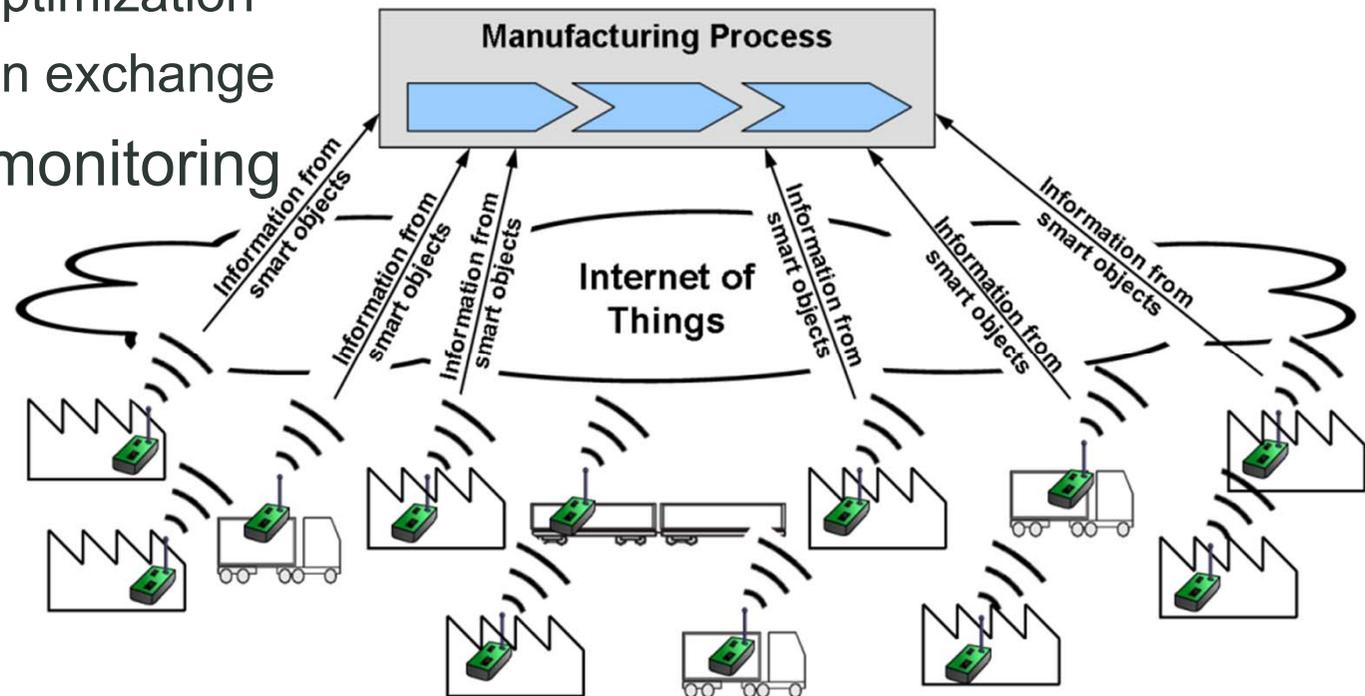
1. Overview
2. Peer-to-Peer Computing
3. Service-oriented Computing
4. Cloud Computing
5. Epilogue

# Major Trends in Distributed Systems I

- Internet of Things (IoT):
  - Physical objects are seamlessly integrated into the information network
  - Physical objects become active participants in business processes
  - Physical objects become “Smart Objects”
  - Technologies: RFID, sensor networks, Internet Protocol version 6 (IPv6)

# IoT – Example: Factories of the Future

- Combining the power of independent factories
- Achieving complex manufacturing processes
- Providing concrete tools for
  - Process creation
  - Process optimization
  - Information exchange
- Real-time monitoring



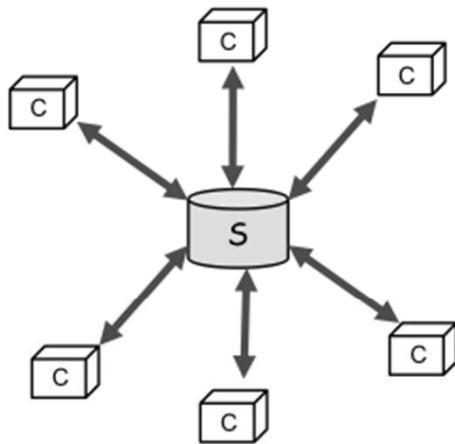
# Major Trends in Distributed Systems II

- Internet of Services (IoS):
  - Software services are provided through the Internet
  - Technologies: REST, WSDL, SOAP
  - Foundation for Cloud Computing
- Service-oriented Architectures vs. IoS:
  - IoS = Global SOA?
  - SOA: Originally a concept to organize IT architectures in a company

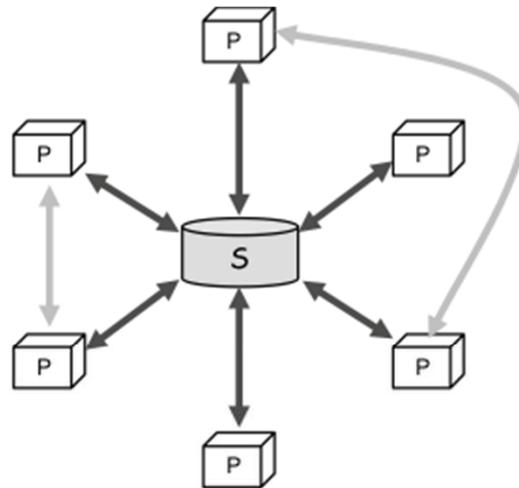
1. Overview
2. Peer-to-Peer Computing
3. Service-oriented Computing
4. Cloud Computing
5. Epilogue

Slides are based on the book “Peer-to-Peer Systems and Applications”, LNCS Vol. 3485 Springer and lecture “Peer-to-Peer Systems and Applications” (TU Darmstadt)

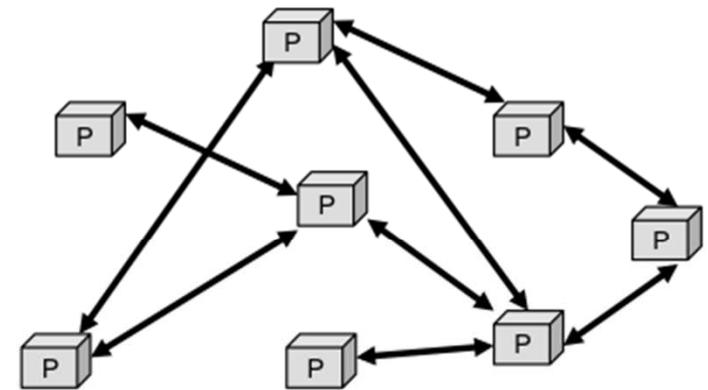
# Peer-to-Peer: Overview



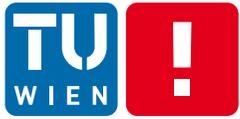
(a)  
Client/Server



(b) Hybrid



(c) Peer-to-Peer



# Peer-to-Peer

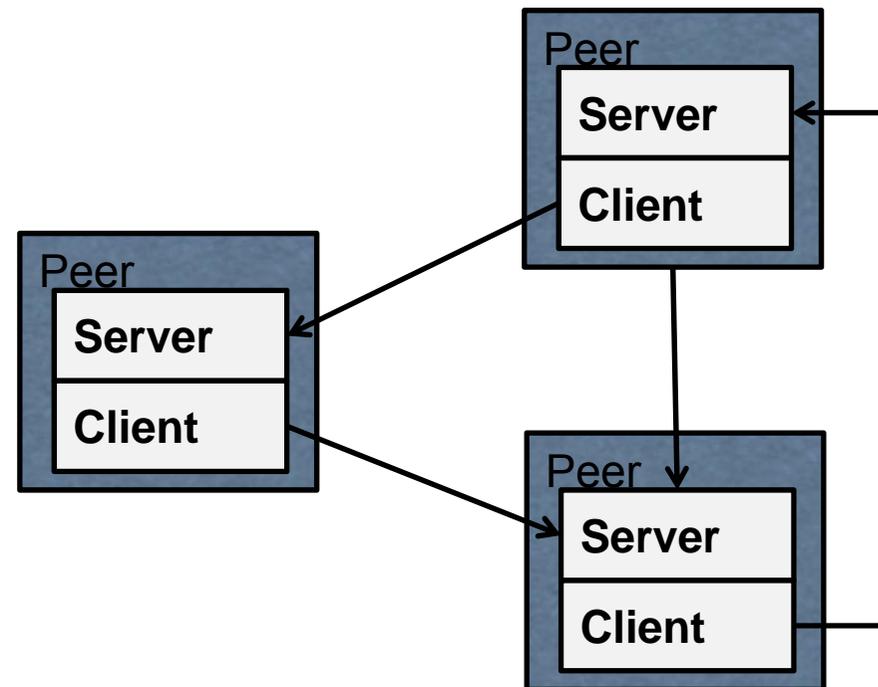
- Components directly interact as peers by exchanging services
- Request/reply interaction without the asymmetry found in the client-server pattern – all peers are equal
- Each peer component provides and consumes similar services

# What is P2P?

- Definition according to Oram et al.:
  - A Peer-to-Peer (P2P) system is „a self-organizing system of equal, autonomous entities (peers) [which] aims for the shared usage of distributed resources in a networked environment avoiding central services.“
  - „A system with completely decentralized self-organization and resource usage.“
- Derived key characteristics of a P2P system:
  - Equality – All peers are equal (peer = gleichgestellt)
  - Autonomy – No central control
  - Decentralization – No centralized services
  - Self-organization – No coordination from outside
  - Shared resources – Peers may use resources provided by other peers

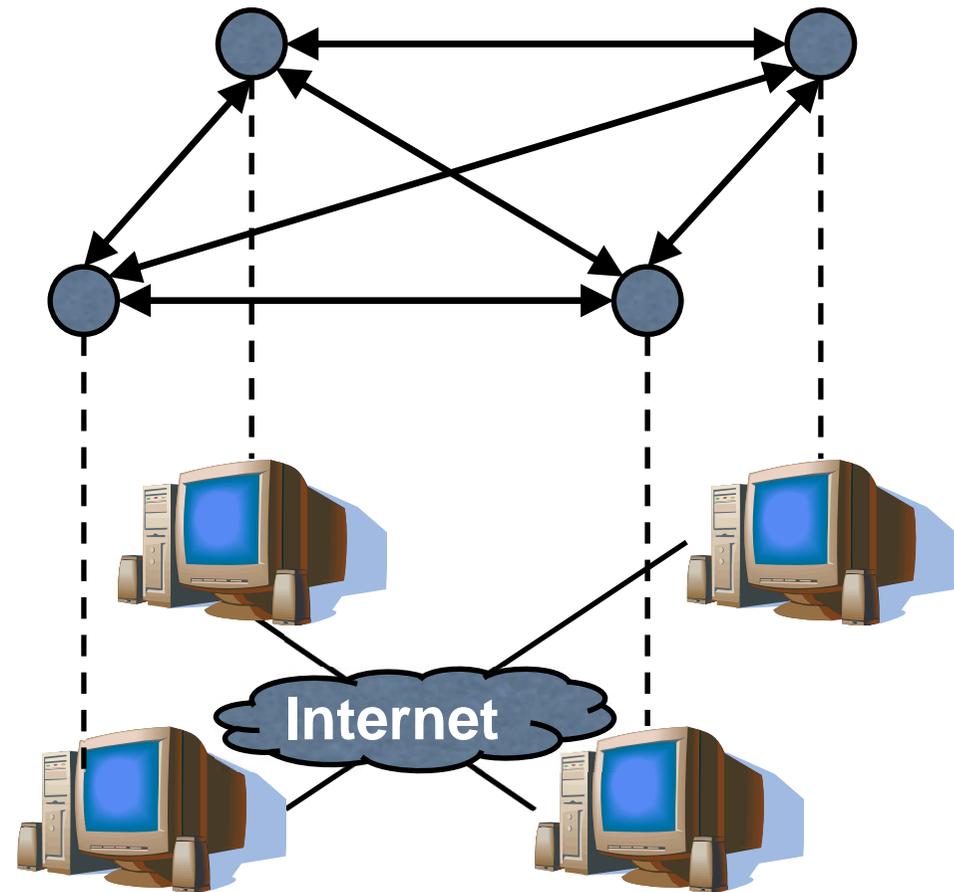


- Peers
  - Are nodes running in some P2P overlay
  - Have all the same capabilities (ability to act in any role)
  - Can act as “clients” and “servers” at the same time



# Overlay-Network

- Composed of direct connections between peers
- Typically an “overlay“ network on top of a network (e.g., the Internet)
- But completely independent from physical network, due to abstraction of the TCP/IP layer
- Separate addressing scheme



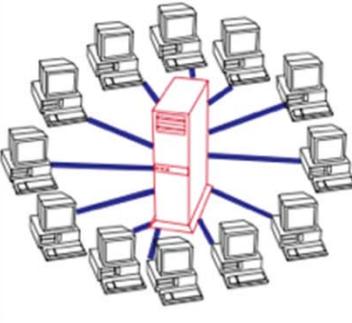
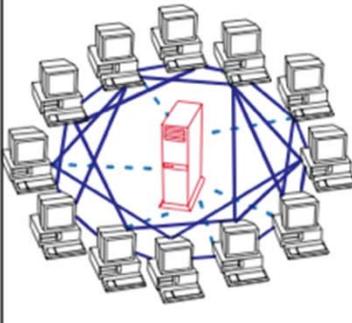
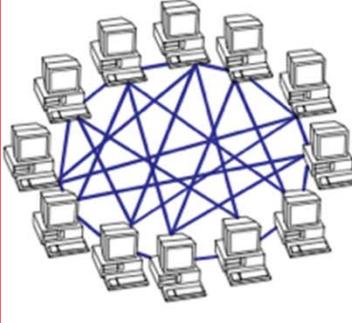
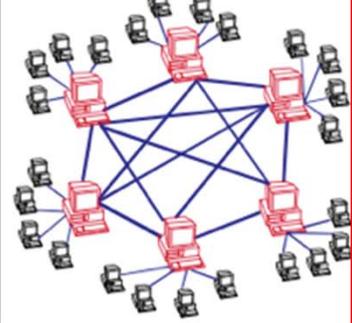
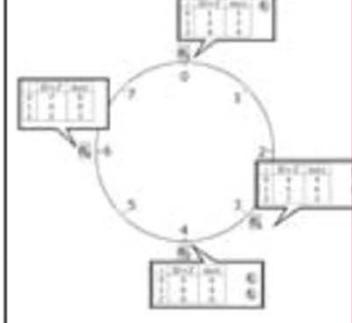
# P2P: Application Areas

- Several application areas:
  - VoIP (Skype/FastTrack)
  - Media streaming (Joost)
- In 2006, P2P made up 70% of the Internet traffic (CacheLogic Research):
  - P2P accounts for ~19% of fixed access traffic in North America according to Sandvine (2010/11)
  - Bittorrent is the single biggest application regarding upstream traffic in North America in 2010/11 (52%)
- Obviously, File Sharing is one area where P2P is heavily applied:
  - Napster (1st Generation Centralized P2P)
  - Gnutella 0.4 (1st Generation Pure P2P)
  - Gnutella 0.6, FastTrack/KaZaA (2nd Generation Hybrid P2P)
  - Kademlia (foundation for trackerless BitTorrent and eDonkey) → Structured P2P

## Reasons for Application of P2P

- Costs: Computing/Storage can be outsourced (this is the major reason why Skype applies P2P)
- High Extensibility (easy to add further resources)
- High Scalability (system can grow to a very large number of peers)
- Fault Tolerance: If one peer fails, the overall system will nevertheless work
- Resistance to lawsuits...



<b>Client-Server</b>	<b>Peer-to-Peer</b>			
	<ol style="list-style-type: none"> <li>1. Resources are shared between the peers</li> <li>2. Resources can be accessed directly from other peers</li> <li>3. Peer is provider and requestor (Servent concept)</li> </ol>			
	<b>Unstructured P2P</b>			<b>Structured P2P</b>
	<b>1st Generation</b>		<b>2nd Generation</b>	
<ol style="list-style-type: none"> <li>1. Server is the central entity and only provider of service and content. → Network managed by the Server</li> <li>2. Server as the higher performance system.</li> <li>3. Clients as the lower performance system</li> </ol> <p>Example: WWW</p>	<p><i>Centralized P2P</i></p> <ol style="list-style-type: none"> <li>1. All features of Peer-to-Peer included</li> <li>2. Central entity is necessary to provide the service</li> <li>3. Central entity is some kind of index/group database</li> </ol> <p>Example: Napster</p>	<p><i>Pure P2P</i></p> <ol style="list-style-type: none"> <li>1. All features of Peer-to-Peer included</li> <li>2. Any terminal entity can be removed without loss of functionality</li> <li>3. → No central entities</li> </ol> <p>Examples: Gnutella 0.4, Freenet</p>	<p><i>Hybrid P2P</i></p> <ol style="list-style-type: none"> <li>1. All features of Peer-to-Peer included</li> <li>2. Any terminal entity can be removed without loss of functionality</li> <li>3. → dynamic central entities</li> </ol> <p>Example: Gnutella 0.6, JXTA</p>	<p><i>DHT-Based</i></p> <ol style="list-style-type: none"> <li>1. All features of Peer-to-Peer included</li> <li>2. Any terminal entity can be removed without loss of functionality</li> <li>3. → No central entities</li> <li>4. Connections in the overlay are "fixed"</li> </ol> <p>Examples: Chord, CAN</p>
				

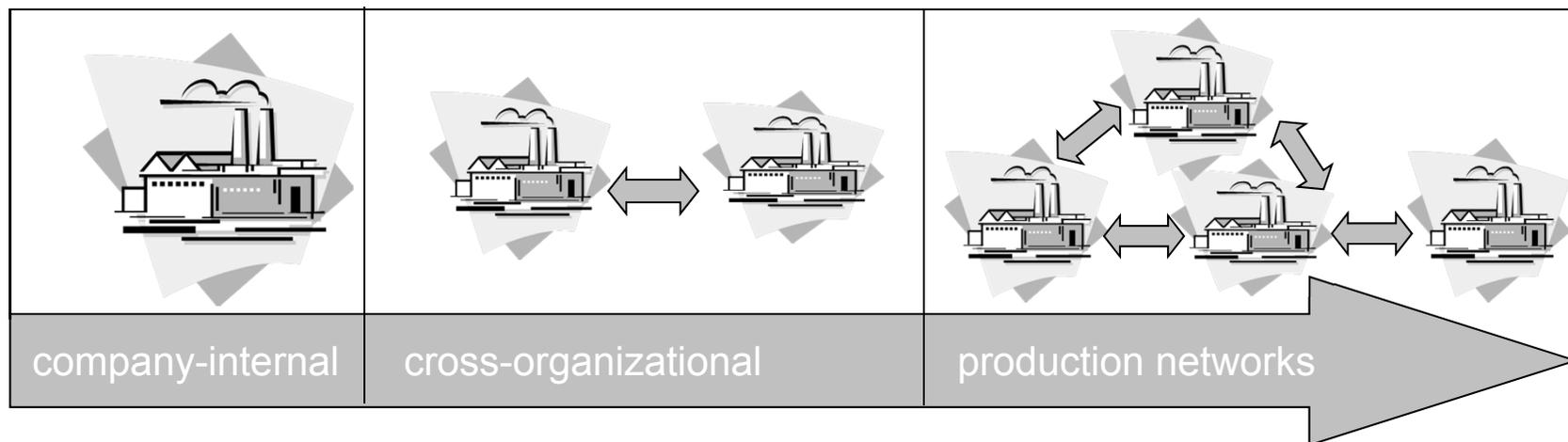




1. Overview
2. Peer-to-Peer Computing
3. Service-oriented Computing
4. Cloud Computing
5. Epilogue

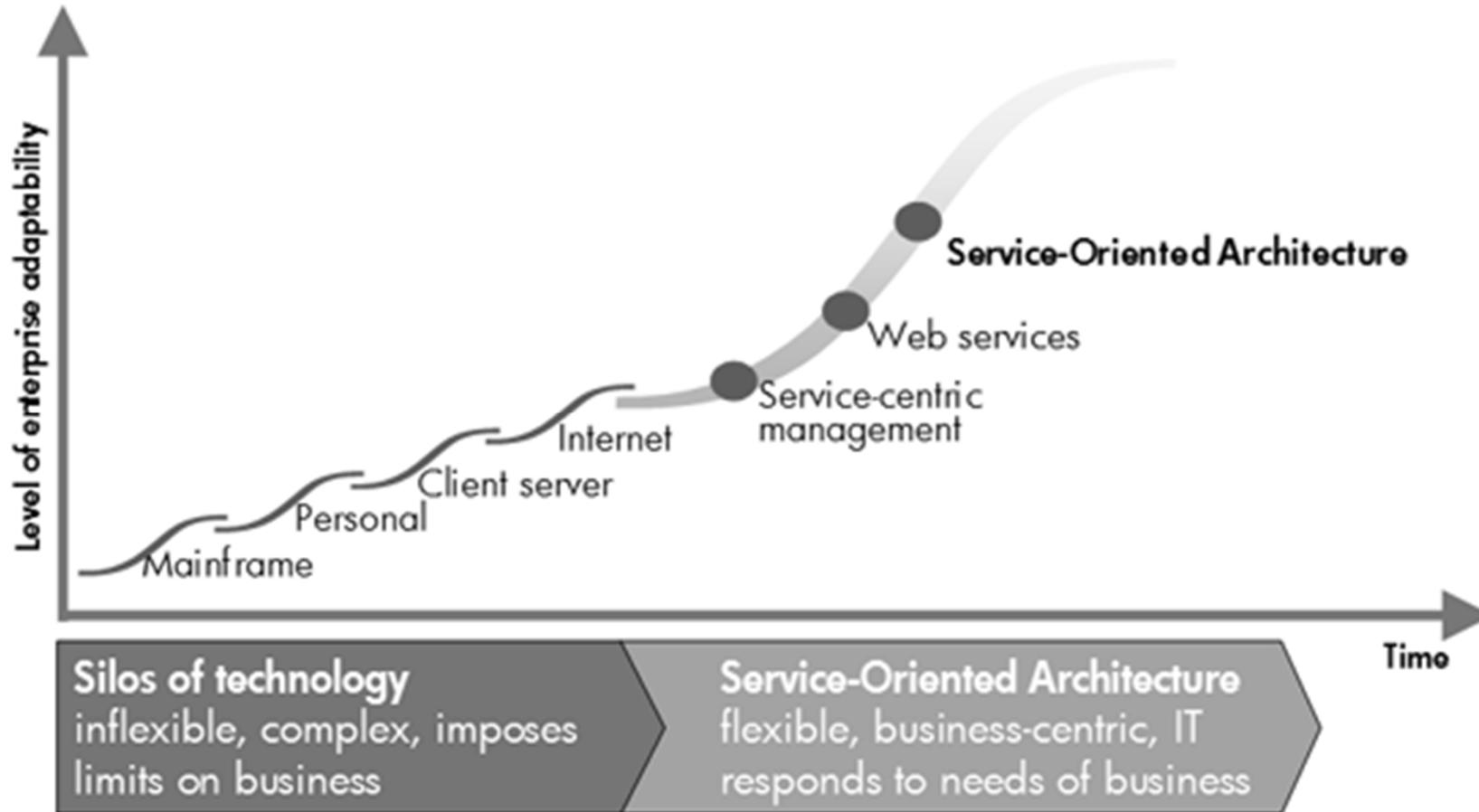
# Motivation

- Major Trend since the 1990s:
  - Globalization, deregulation of markets
  - Cross-organizational workflows and business processes are of major importance
  - Business Process Outsourcing (BPO)
  - Flexibility of business processes is a key success factor



- Flexible IT architectures are a major requirement:
  - Integration of legacy systems
  - Coupling to IT systems of business partners

# Motivation – A Shift of Paradigms

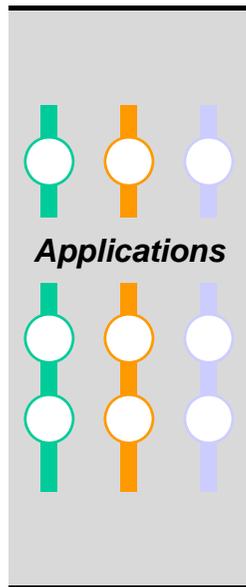


# Vision of a Service-oriented Architecture

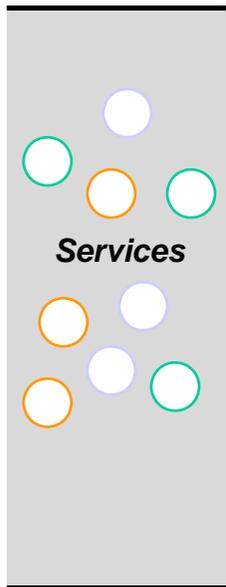
“Loosely Coupled, Process Driven Services and Components”

## Tomorrow

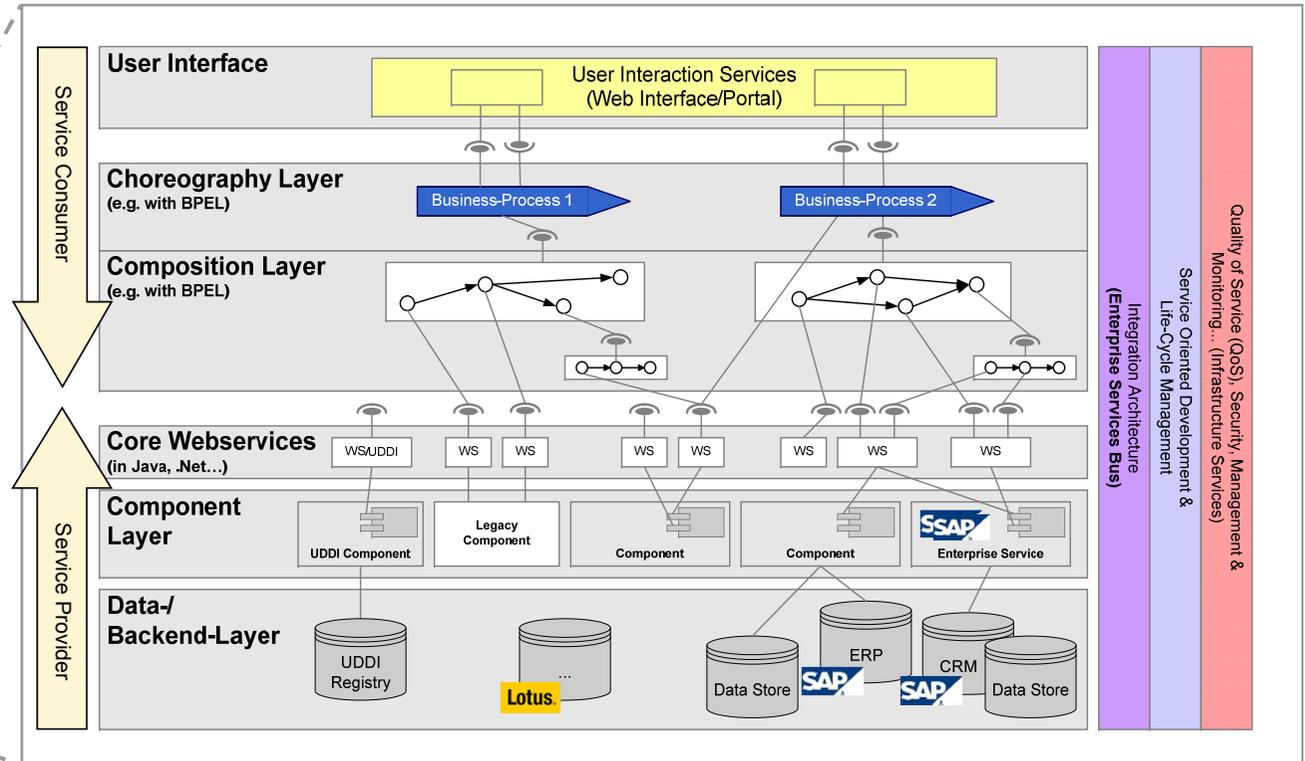
### Today



Discrete Applications



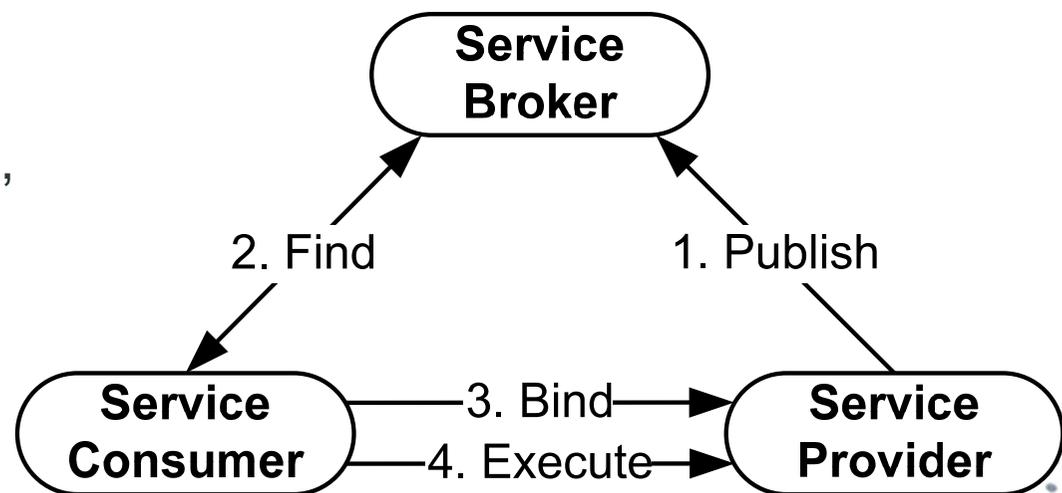
Services



Source: IBM 2007

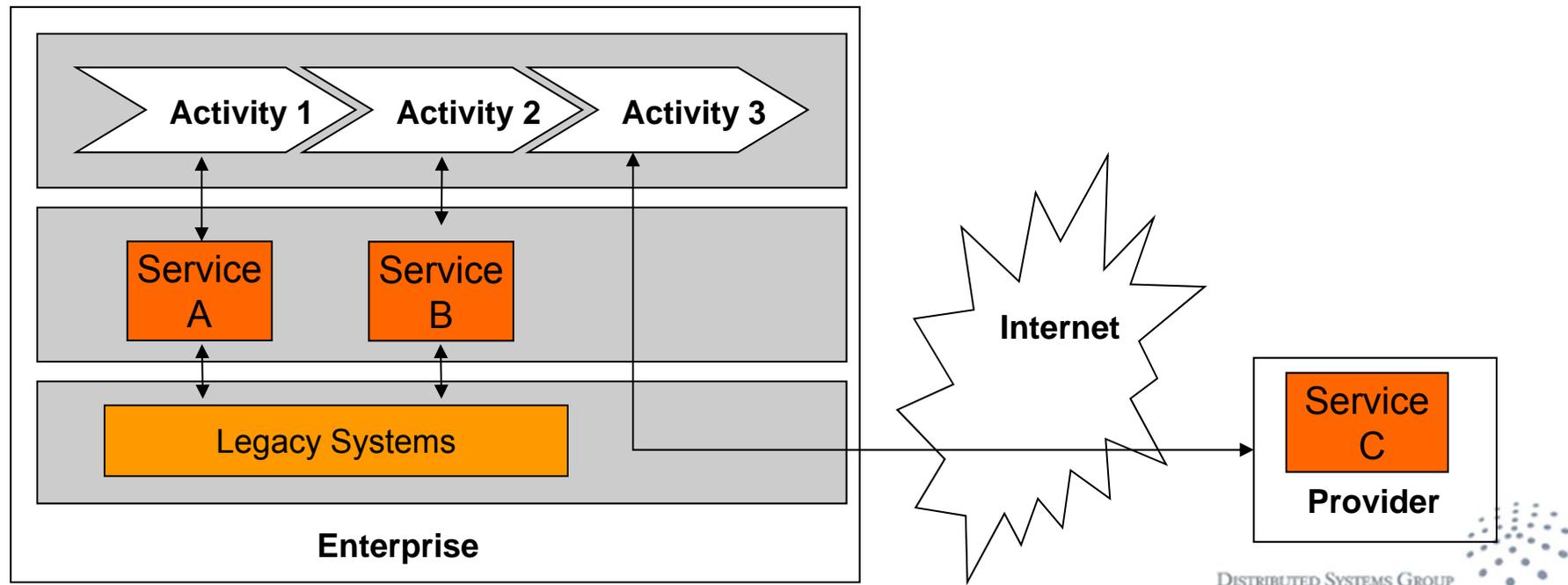
# SOA – Overview and Roles

- Service-oriented Architectures:
  - IT architecture made up from single services, i.e., self-contained software components with a distinct functionality
  - Complex applications arise from the coupling of single services, e.g.,
    - Service-based workflows
    - Mashups
  - However, it is also possible to invoke single services
- Roles in a Service-oriented Architecture
  - Service Provider
  - Service Consumer
  - Intermediary (optional), e.g., Service Broker



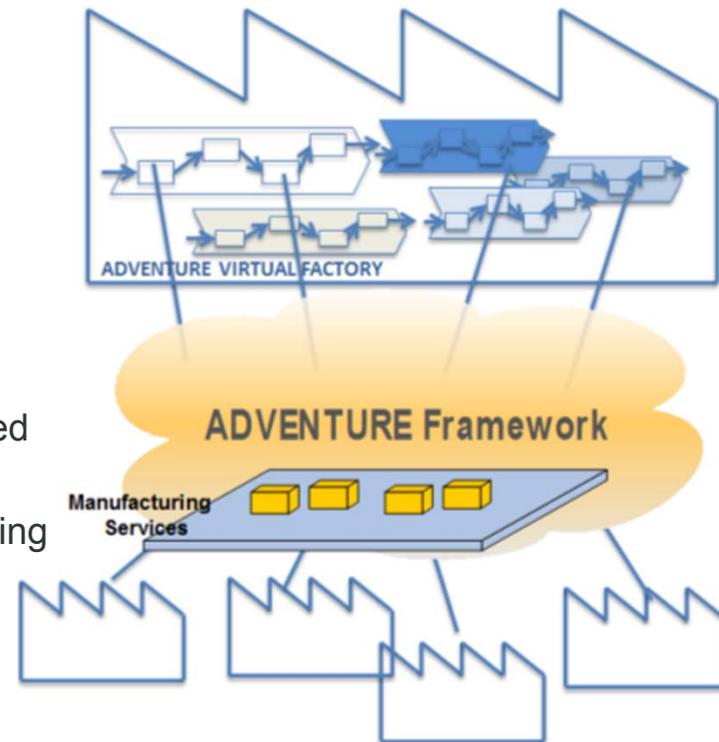
# Workflows and Services

- Workflows and Services:
  - Workflows are IT-enabled business processes
  - Services can be composed to workflows (2-level-programming)
  - Services wrap functionality of legacy systems (e.g. Service A/B)
  - Integration external services (e.g. Service C)
- Services support rapid composition of distributed workflows



# Example for IoT and IoS: ADVENTURE – The Plug-and-Play Virtual Factory

- Virtual Factory
  - Multiple factories may form a virtual factory
  - Integrated ICT
    - Leverage information exchange
  - Interoperability at a deeper technical level
    - Ensuring that factories can be technically connected
  
- Plug
  - Factories provide information
    - Semantically enriched descriptions of offered manufacturing capabilities and products
    - Sensor technologies to monitor manufacturing processes
  
- Play
  - Factories model manufacturing process
    - Manufacturing processes modeled as composition of services
  - Identify particular partners who offer a distinct product



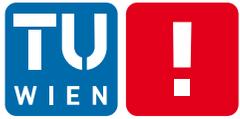
1. Overview
2. Peer-to-Peer Computing
3. Service-oriented Computing
4. Cloud Computing
5. Epilogue

Slides are based on “A View of Cloud Computing”, Armbrust et al., Communications of the ACM, Vol. 53, No. 4, April 2010 and The NIST Definition of Cloud Computing

# Motivation – Want milk?



- Buy a cow:
  - High upfront investment
  - High maintenance cost
  - Produces a more or less fixed amount of milk
  - Stepwise (discrete) scaling
- Buy bottled milk:
  - Pay-per-use
  - Lower maintenance cost
  - Linear (continuous) scaling
  - Fault-tolerant

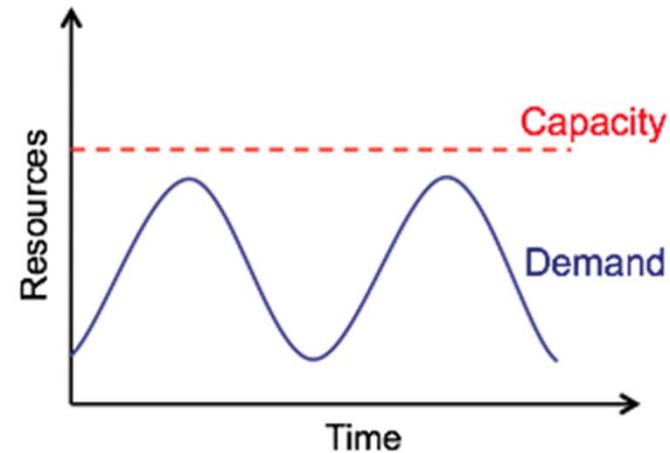


# Use Cases for Cloud Computing

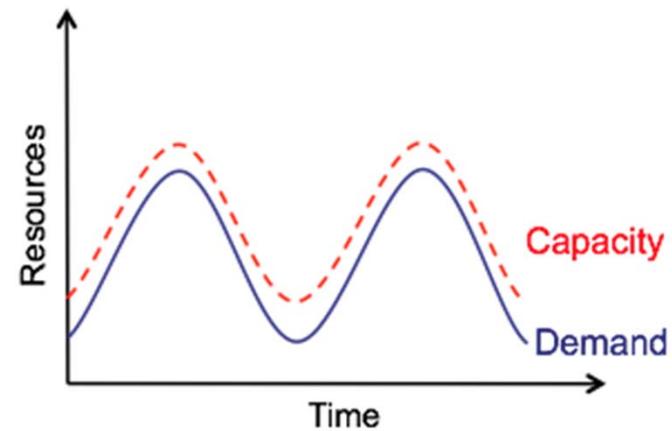
- Demand for a service varies with time
  - e.g., Peak loads
- Demand is unknown in advance
  - e.g., for new startup
- Batch analytics
  - e.g., 1000 EC2 instances for one hour cost the same as one instance for 1000 hours

# Traditional Datacenter vs Cloud

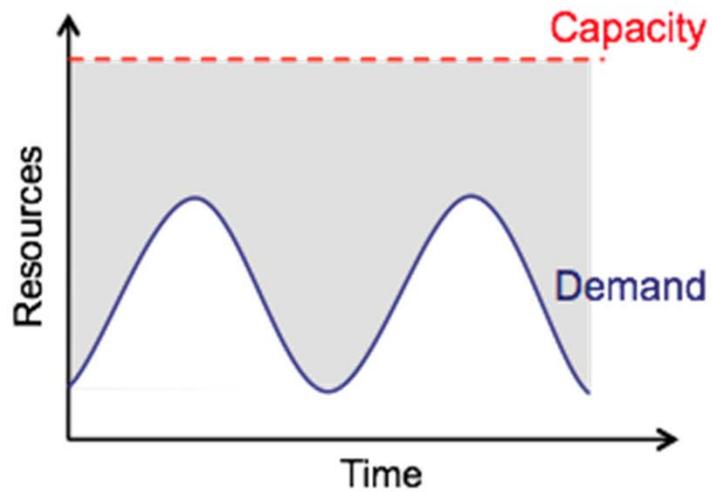
- Traditional datacenter

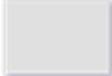


- Virtual datacenter in the cloud

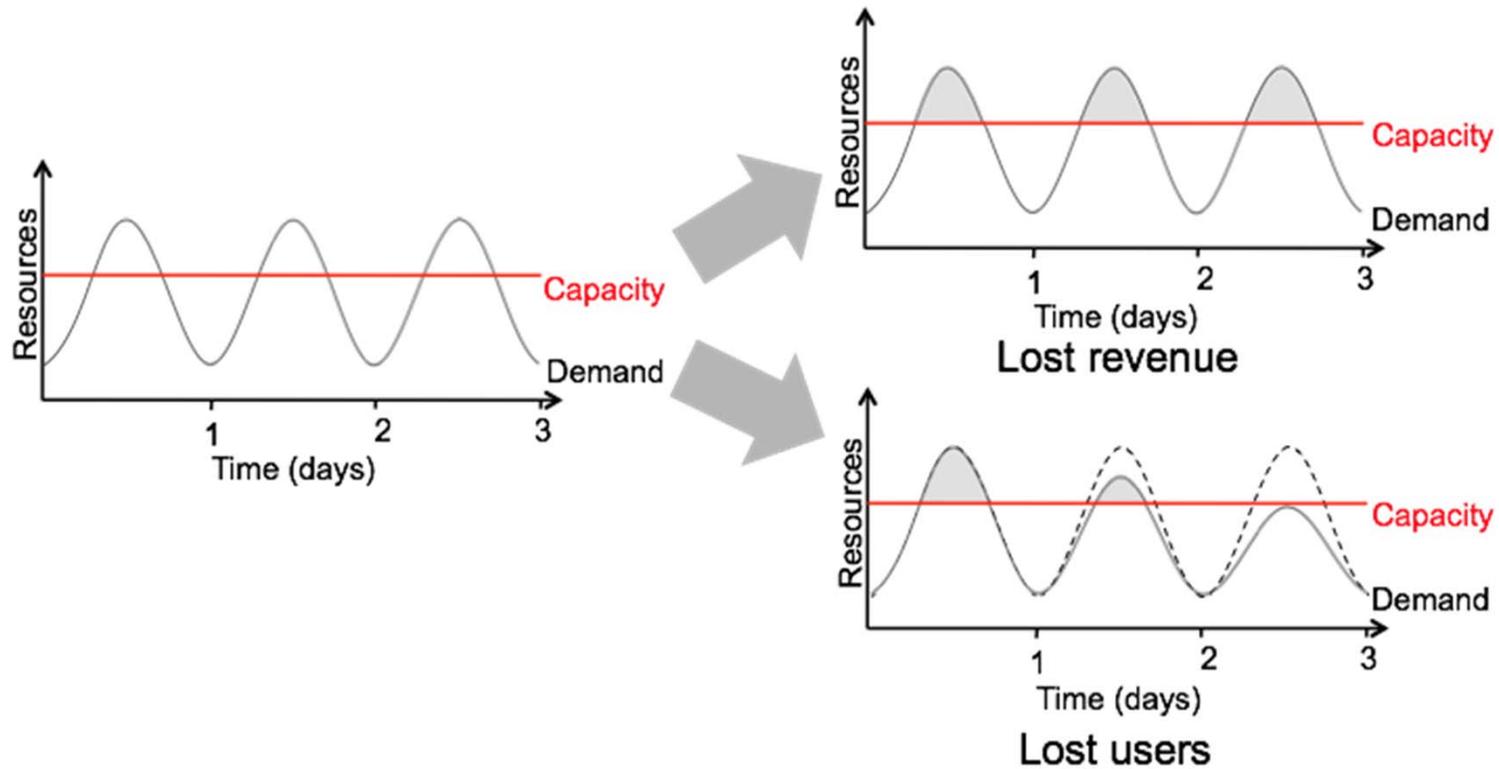


# Risk of Overprovisioning



 Unused resources

# Risks of Underprovisioning



## Definition

- According to the National Institute of Standards and Technology (NIST):
  - On-demand self services: Quick, automated rental of capacity using Web interfaces
  - Broad network access
  - Resource pooling: Use of virtualization techniques
  - Rapid elasticity: Virtually unlimited capacity and scalability
  - Measured service: Pay-as-you-go

# NIST: 3 Service Models (1)

- Cloud Infrastructure as a Service (IaaS)
  - Deliver computer infrastructure as a service (Virtual Machines, storage, ...)
  - Example: Amazon EC2, Amazon S3
- Cloud Platform as a Service (PaaS)
  - Deliver computing platform and solution stack as a service (execution environment/framework)
  - Example: Google App Engine
- Cloud Software as a Service (SaaS)
  - Example: ERP software as a service, Salesforce.com

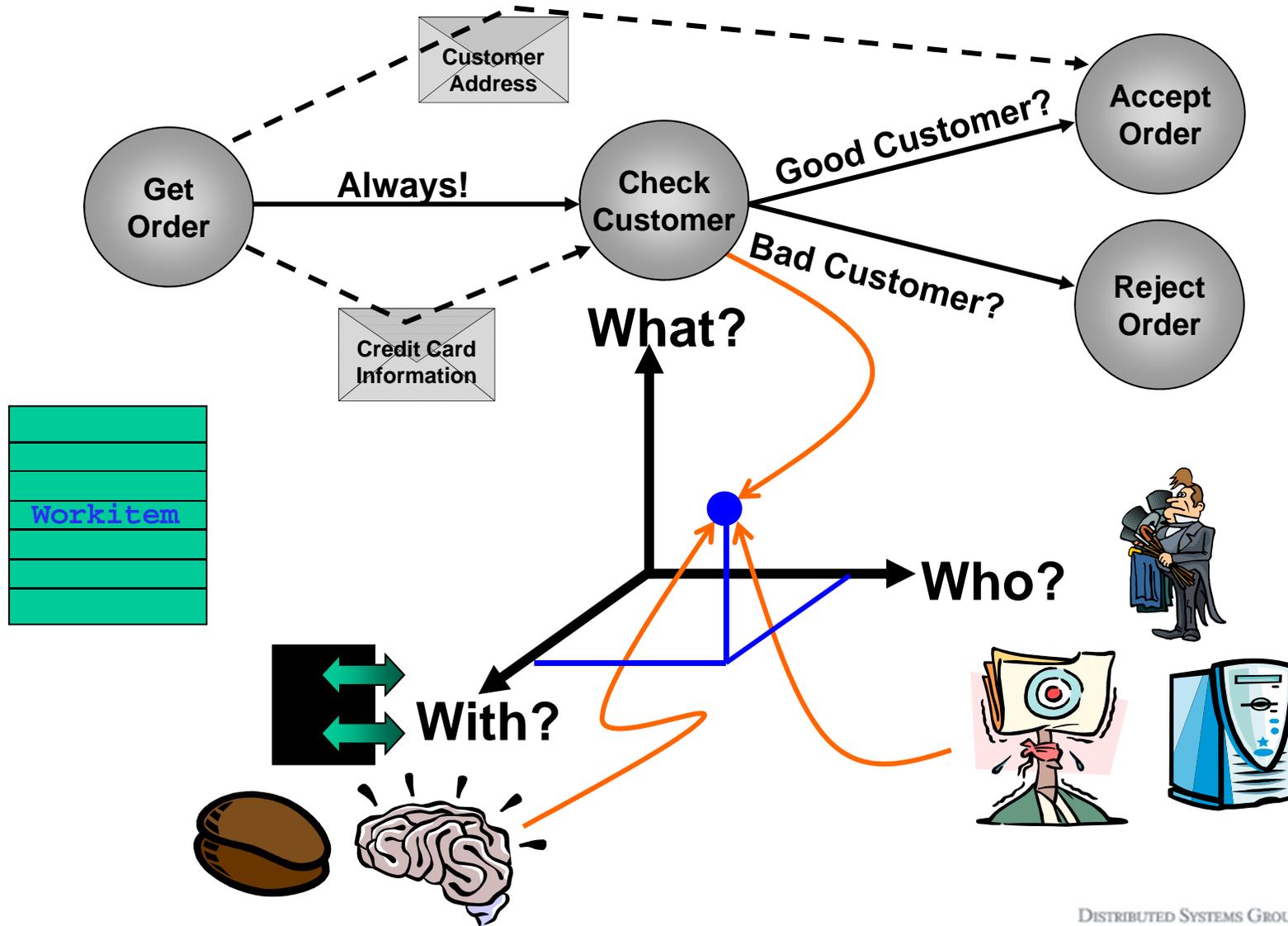
# NIST: 4 Deployment Models

- Private Cloud: Operated solely for one single organization
- Community Cloud: Shared by several organizations
- Public Cloud: Open to general public, owned by an organization selling Cloud services
- Hybrid Cloud: Composition of two or more Cloud deployment models (private, community, public)



1. Overview
2. Peer-to-Peer Computing
3. Service-oriented Computing
4. Cloud Computing
5. Epilogue

# Business Processes and Workflows

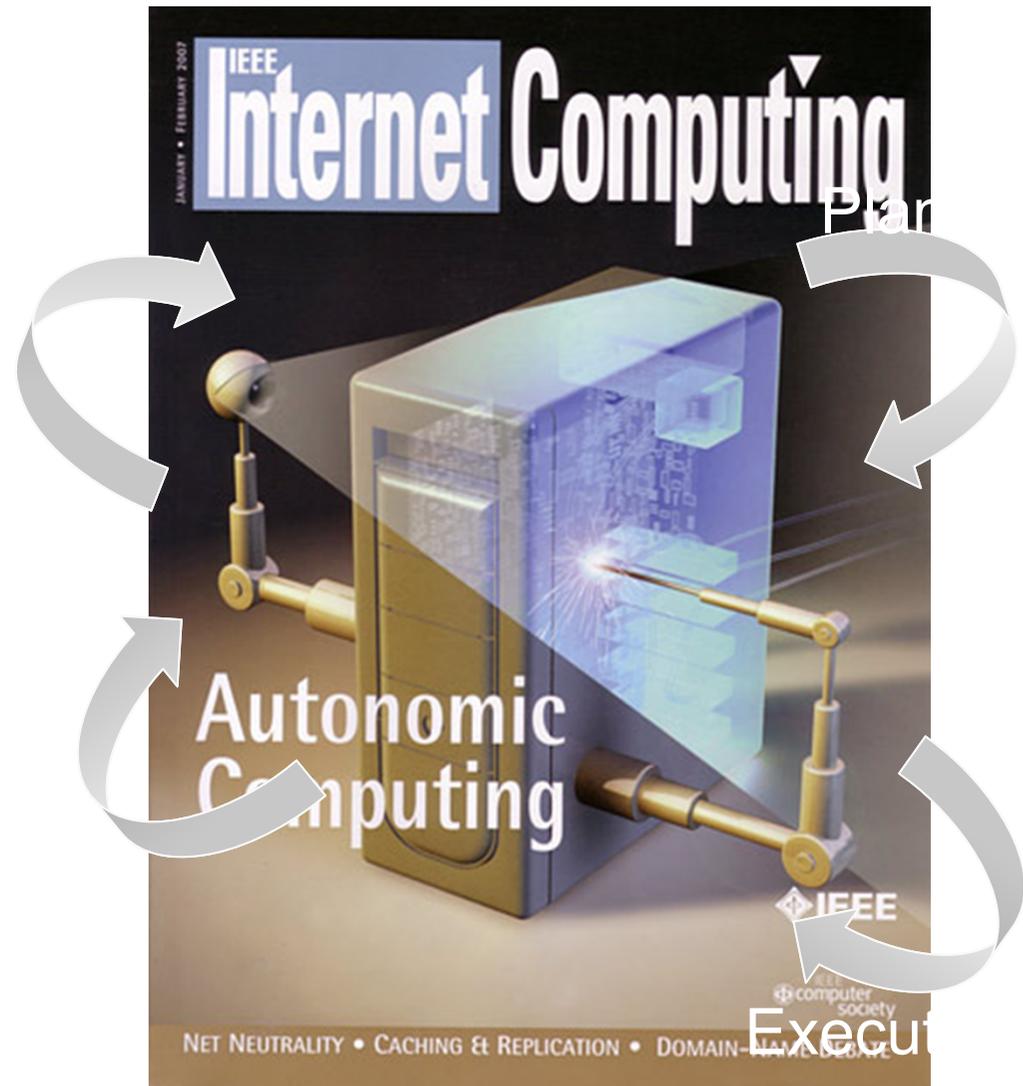


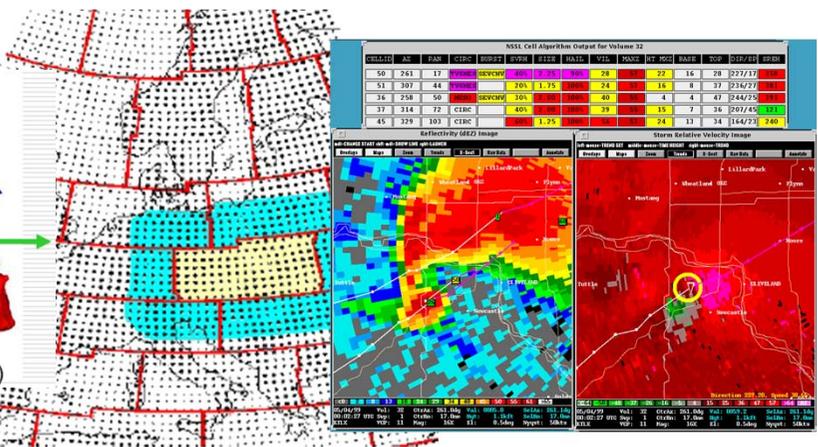
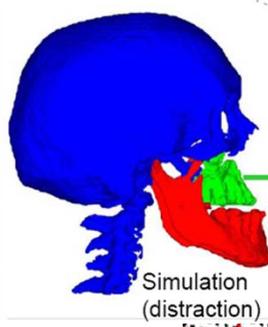
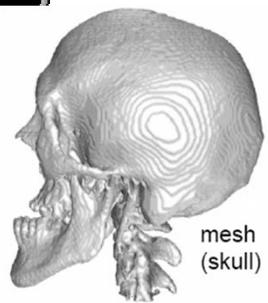
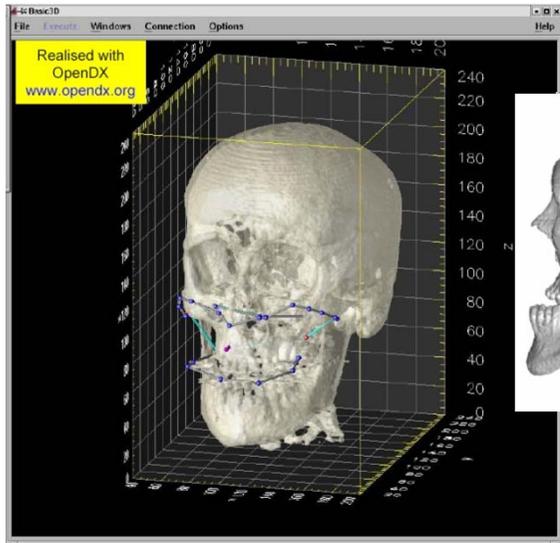
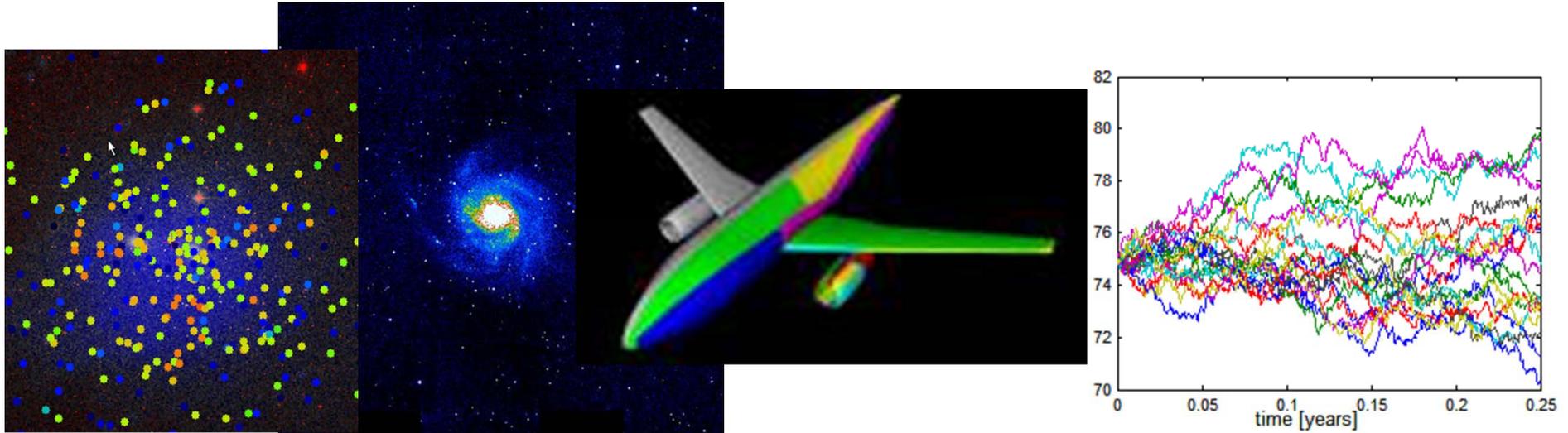
# Mobile & Context-aware Computing



# Autonomic Computing

- Goals:
  - Self-Configuring
  - Self-Healing
  - Self-Optimizing
  - Self-Protecting





# Cloud Computing



[Amazon Web Services](#) » AWS Simple Monthly Calculator

Data Transfer-out:  GB

[Customer Sample 1 \(Amazon S3 only\)](#)   
 [Customer Sample 2 \(Amazon EC2 only\)](#)   
 [Customer Sample 3 \(Amazon SQS only\)](#)  
[Customer Sample 4 \(Amazon S3 + EC2\)](#)   
 [Customer Sample 5 \(Amazon EC2 + SQS\)](#)   
 [Customer Sample 6 \(Amazon SQS + EC2 + S3\)](#)

Amazon S3 (US)  
  Amazon S3 (EUR)  
  Amazon EC2  
  Amazon SQS

Amazon S3 (US) Storage:  GB-months  
 Amazon S3 (US) Data Transfer-in:  GB  
 Amazon S3 (US) Data Transfer-out:  GB  
 Amazon S3 (US) PUT/LIST Requests:  Requests  
 Amazon S3 (US) Other Requests:  Requests  
 Amazon S3 (EUR) Storage:  GB-months  
 Amazon S3 (EUR) Data Transfer-in:  GB

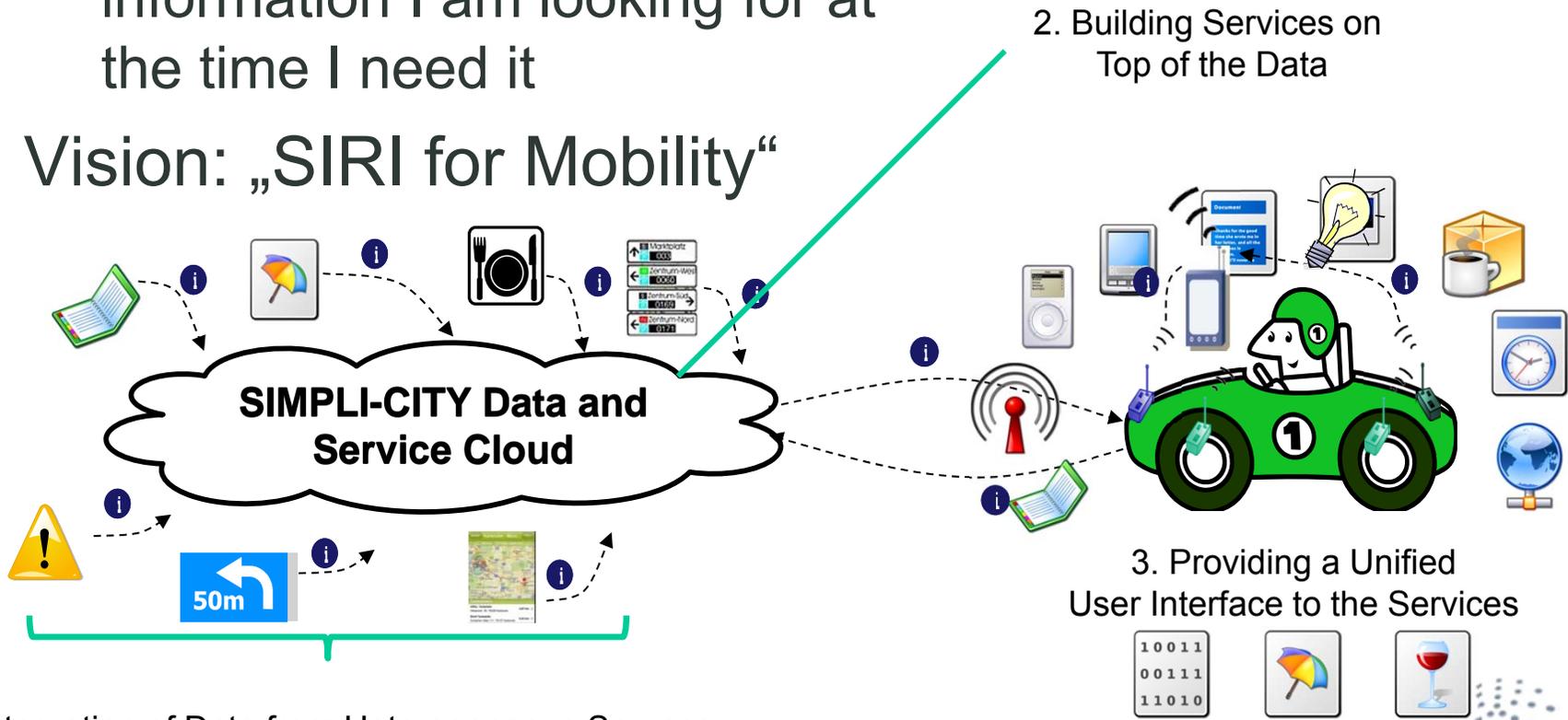
### Estimate of Your Monthly Bill

Amazon S3 (US)	Storage	\$	<input type="text" value="1.50"/>	
	Data Transfer	\$	<input type="text" value="2.73"/>	
	Requests	\$	<input type="text" value="0.02"/>	
<b>Amazon S3 (US) Bill:</b>			\$	<input type="text" value="4.25"/>
Amazon S3 (EUR)	Storage	\$	<input type="text" value="0.00"/>	
	Data Transfer	\$	<input type="text" value="0.00"/>	
	Requests	\$	<input type="text" value="0.00"/>	
<b>Amazon S3 (EUR) Bill:</b>			\$	<input type="text" value="0.00"/>
Amazon EC2	Compute	\$	<input type="text" value="0.00"/>	
	Data Transfer	\$	<input type="text" value="0.00"/>	
	EBS Volumes	\$	<input type="text" value="0.00"/>	
	EBS Snapshots	\$	<input type="text" value="0.00"/>	
	<b>Amazon EC2 Bill:</b>			\$
Amazon SQS	Messaging	\$	<input type="text" value="0.00"/>	
	Data Transfer	\$	<input type="text" value="0.00"/>	
<b>Amazon SQS Bill:</b>			\$	<input type="text" value="0.00"/>
<b>Total Monthly Payment:</b>			\$	<input type="text" value="4.25"/>

Computing Power as a configurable, payable Service

# The Road User Information System of the Future

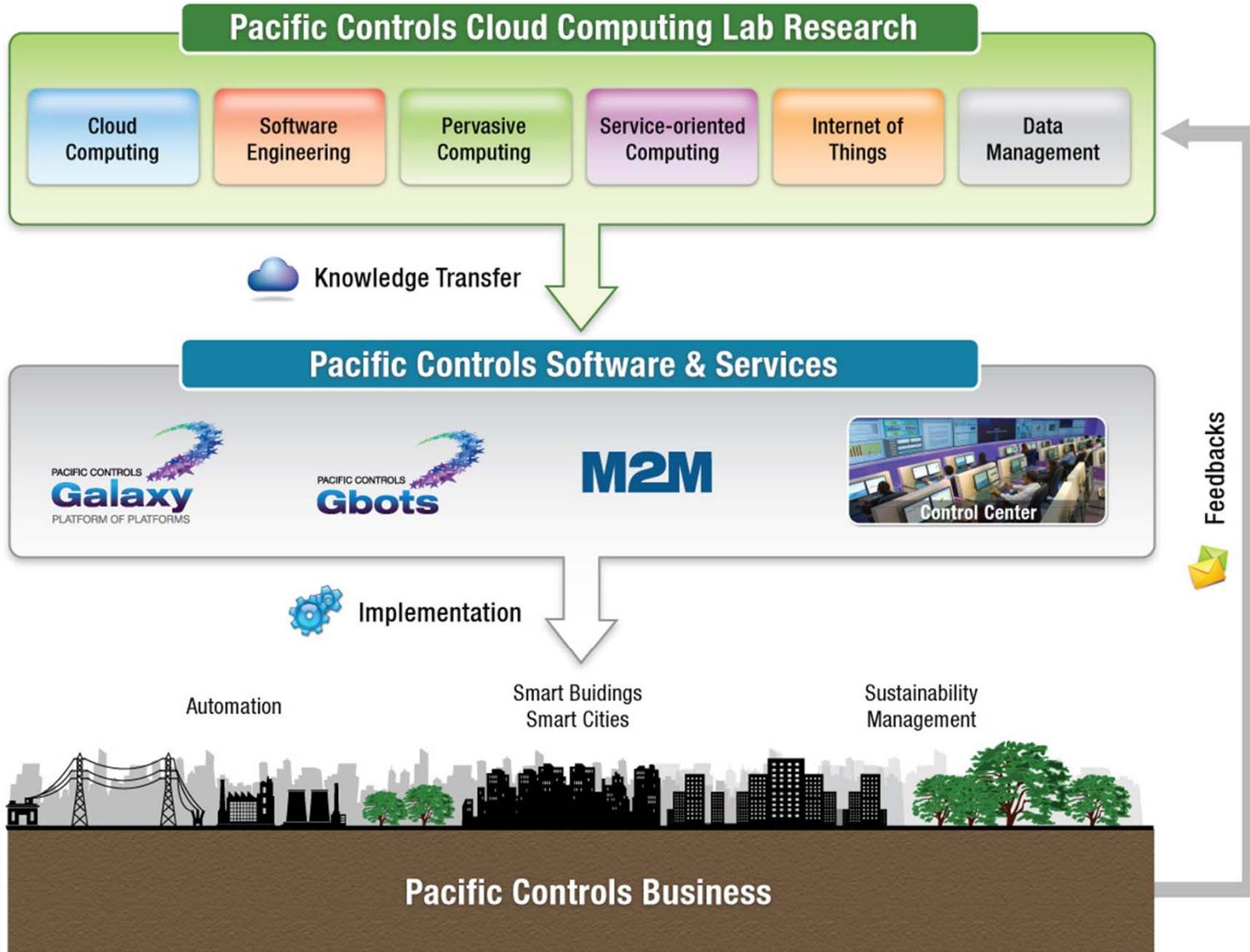
- Ubiquitous Web access makes a multitude of information sources available to drivers:
  - Makes it difficult to get exactly the information I am looking for at the time I need it
- Vision: „SIRI for Mobility“



1. Integration of Data from Heterogeneous Sources

2. Building Services on Top of the Data

3. Providing a Unified User Interface to the Services



# Some Final Words

- We are always looking for motivated students:
  - Bachelor theses
  - Master theses
  - International internships
- Topics:
  - Cloud Computing
  - Service-oriented Computing
  - Elastic Processes



FAKULTÄT  
FÜR INFORMATIK  
Faculty of Informatics



Distributed  
Systems Group

Dr.-Ing.  
**Stefan Schulte**  
Research Assistant

Vienna University of Technology  
Institute of Information Systems  
Argentinierstrasse 8/184-1, 1040 Vienna, Austria  
T: +43 1 58801-18417 F: +43 1 58801-18491  
E: [s.schulte@infosys.tuwien.ac.at](mailto:s.schulte@infosys.tuwien.ac.at)  
[www.infosys.tuwien.ac.at](http://www.infosys.tuwien.ac.at)



## Further Readings

- Armbrust et al.: A View of Cloud Computing, Communications of the ACM, 53(4), 2010.
- Papazoglou, Traverso, Dustdar, Leymann: Service-oriented Computing: State of the art and research challenges, Computer, 40(11), 2007.
- Steinmetz, Wehrle: Peer-to-Peer Systems and Applications, 2005.