CC529: DISTRIBUTED SYSTEM

Final Revision

CHAPTER 1: INTRODUCTION

Definition, Goals, DS Types

DS Goals

- Resource Sharing
- Transparency
- Openness
- Scalability
- Hardware Concepts
  - Multiprocessors
  - Homogenous Multicomputers
- Heterogeneous Multicomputers

Hardware Concept

- Multiprocessors
- Homogenous Multicomputers
- Heterogeneous Multicomputers

Definition of a Distributed System (1)

A distributed system is:

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

Definition of a Distributed System (2)

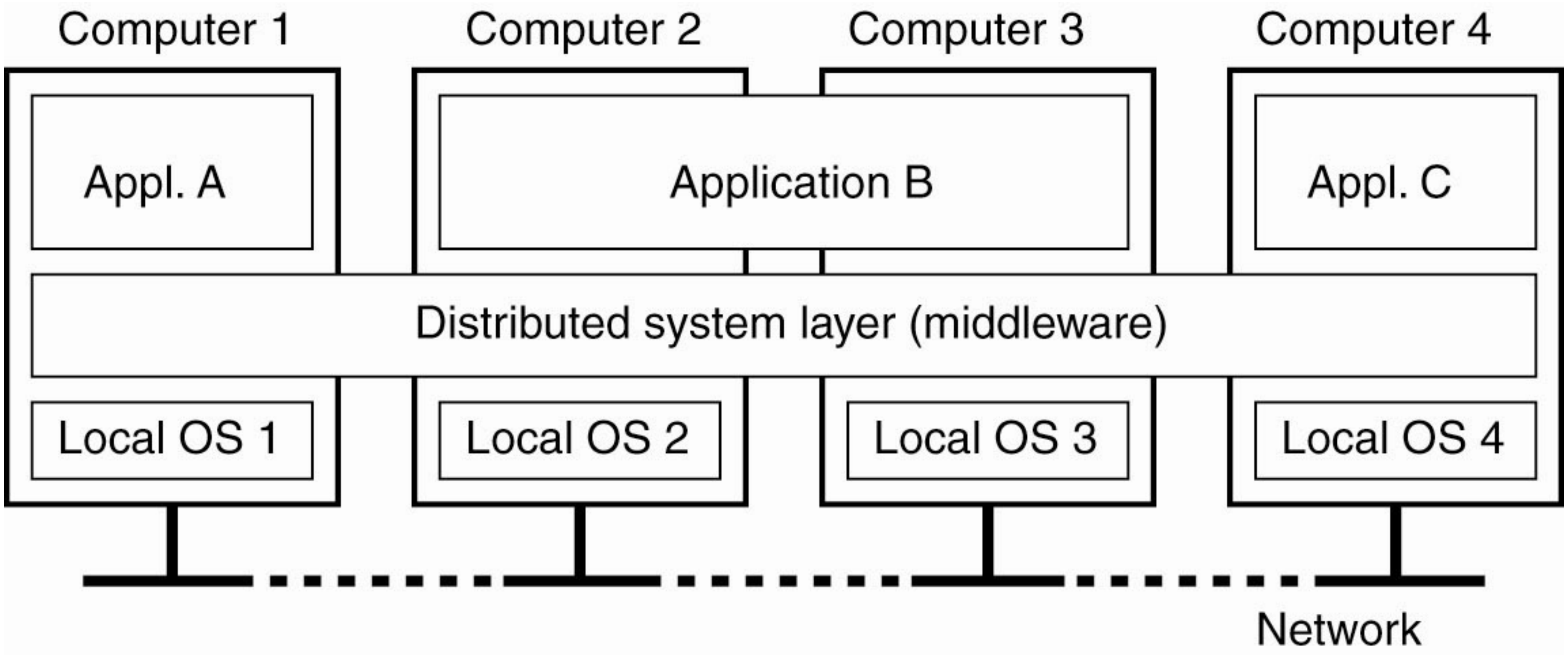
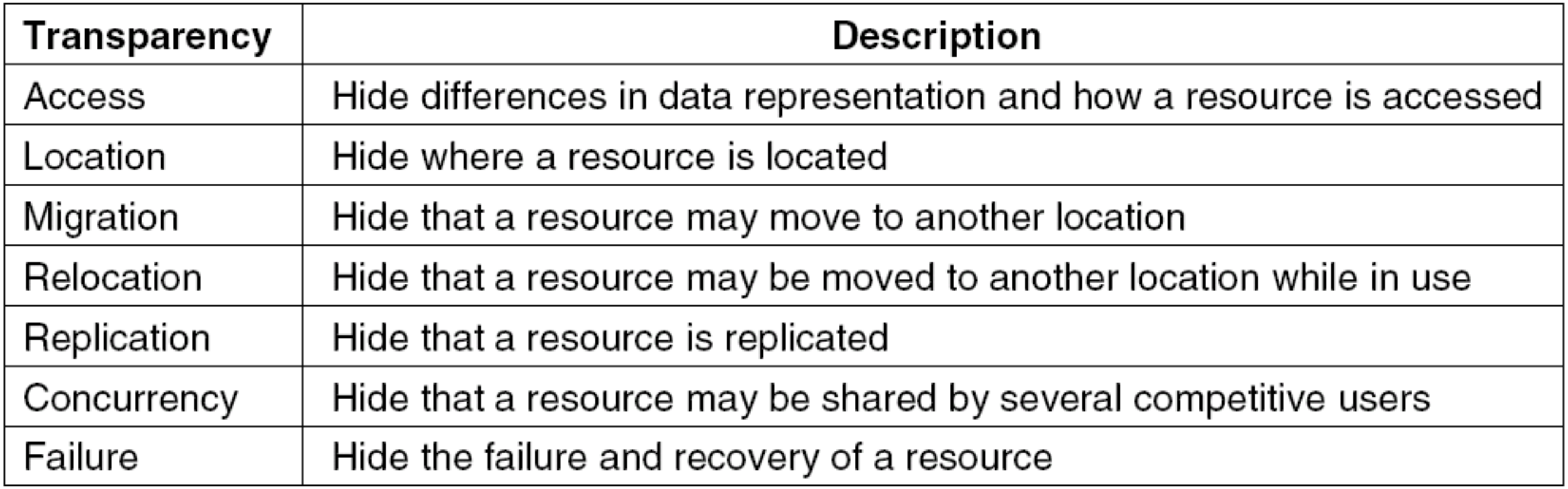


Figure 1-1. A distributed system organised as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

Transparency in a Distributed System



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

Figure 1-2. Different forms of transparency in a distributed system (ISO, 1995).

Scalability Problems

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Figure 1-3. Examples of scalability limitations.

Scalability Problems

Characteristics of decentralised algorithms:

- No machine has complete information about the system state.
- Machines make decisions based only on local information.
- Failure of one machine does not ruin the algorithm.
- There is no implicit assumption that a global clock exists.

Pitfalls when Developing Distributed Systems

False assumptions made by first time developer:

- The network is reliable.
- The network is secure.
- The network is homogeneous.
- The topology does not change.
- Latency is zero.
- Bandwidth is infinite.
- Transport cost is zero.
- There is one administrator.

TYPES OF DS

- Distributed Computing System
- Distributed Information System
- Distributed Pervasive System

Cluster Computing Systems

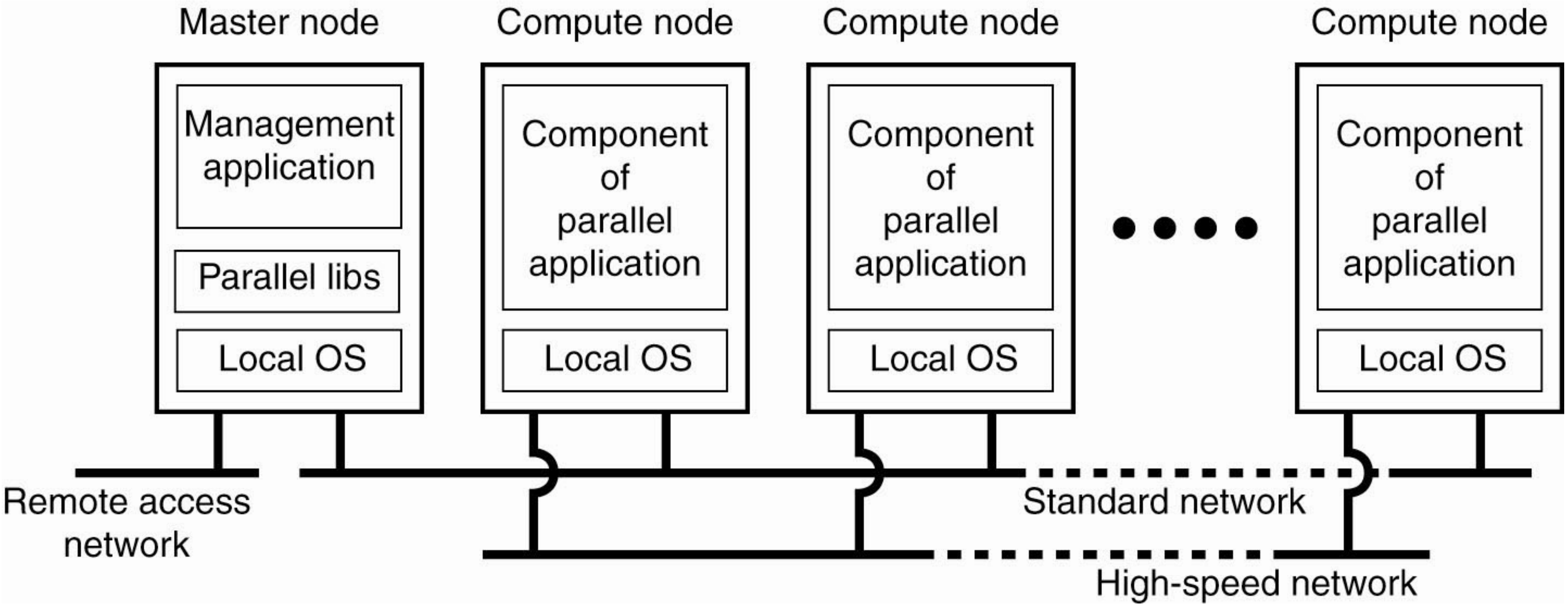
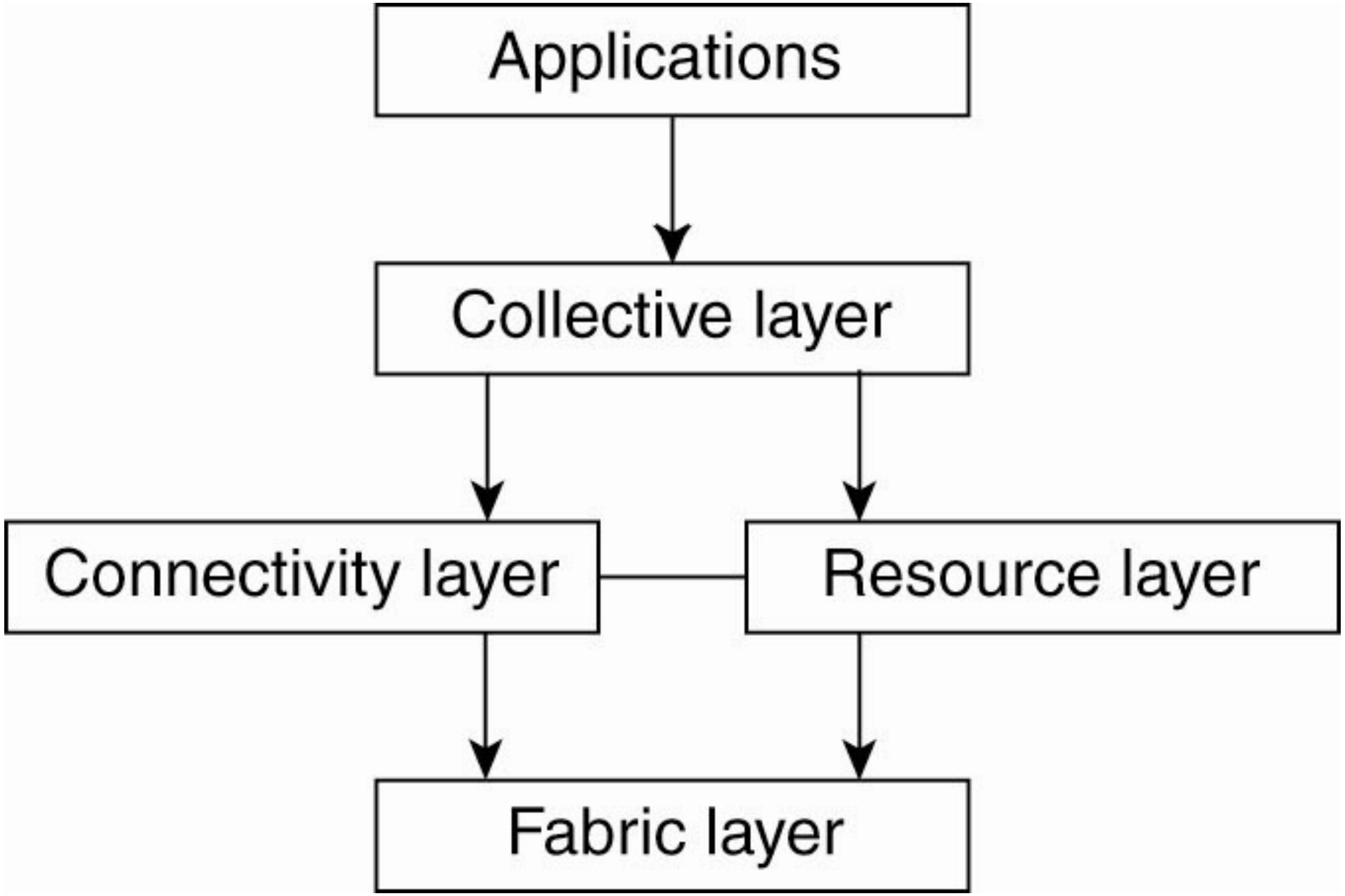


Figure 1-6. An example of a cluster computing system.

Grid Computing Systems


Applications

Collective layer

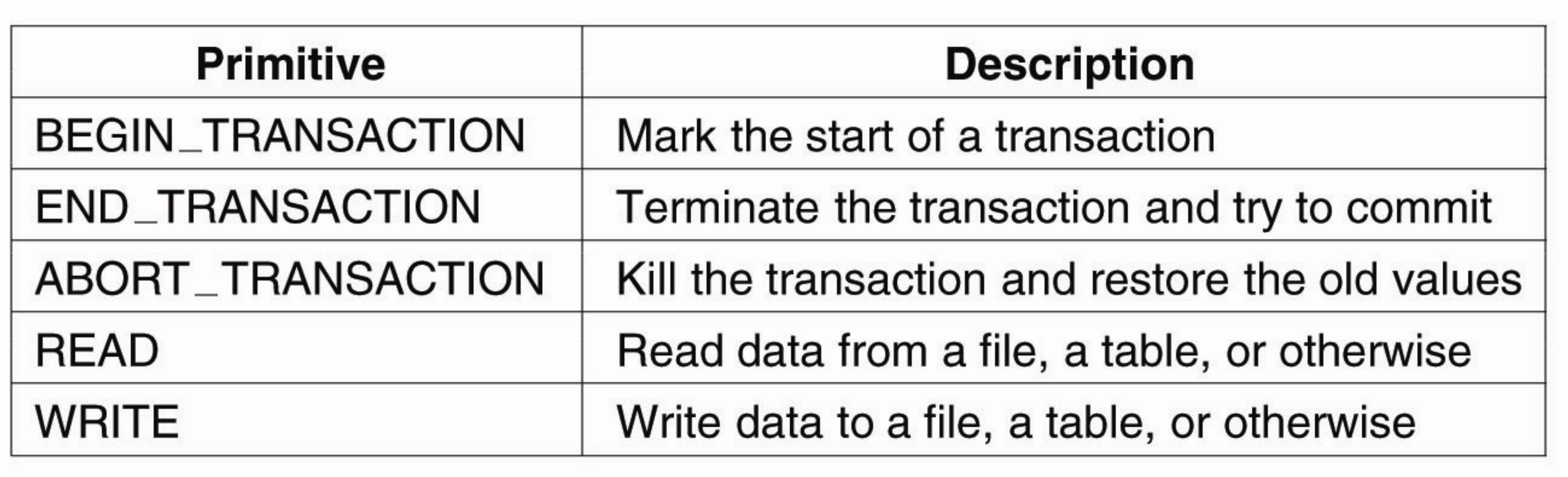
Connectivity layer

Resource layer

Fabric layer

Figure 1-7. A layered architecture for grid computing systems.

Transaction Processing Systems (1)



Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Figure 1-8. Example primitives for transactions.

Transaction Processing Systems (2)

Characteristic properties of transactions:

- Atomic: To the outside world, the transaction happens indivisibly.
- Consistent: The transaction does not violate system invariants.
- Isolated: Concurrent transactions do not interfere with each other.
- Durable: Once a transaction commits, the changes are permanent.

Transaction Processing Systems (3)

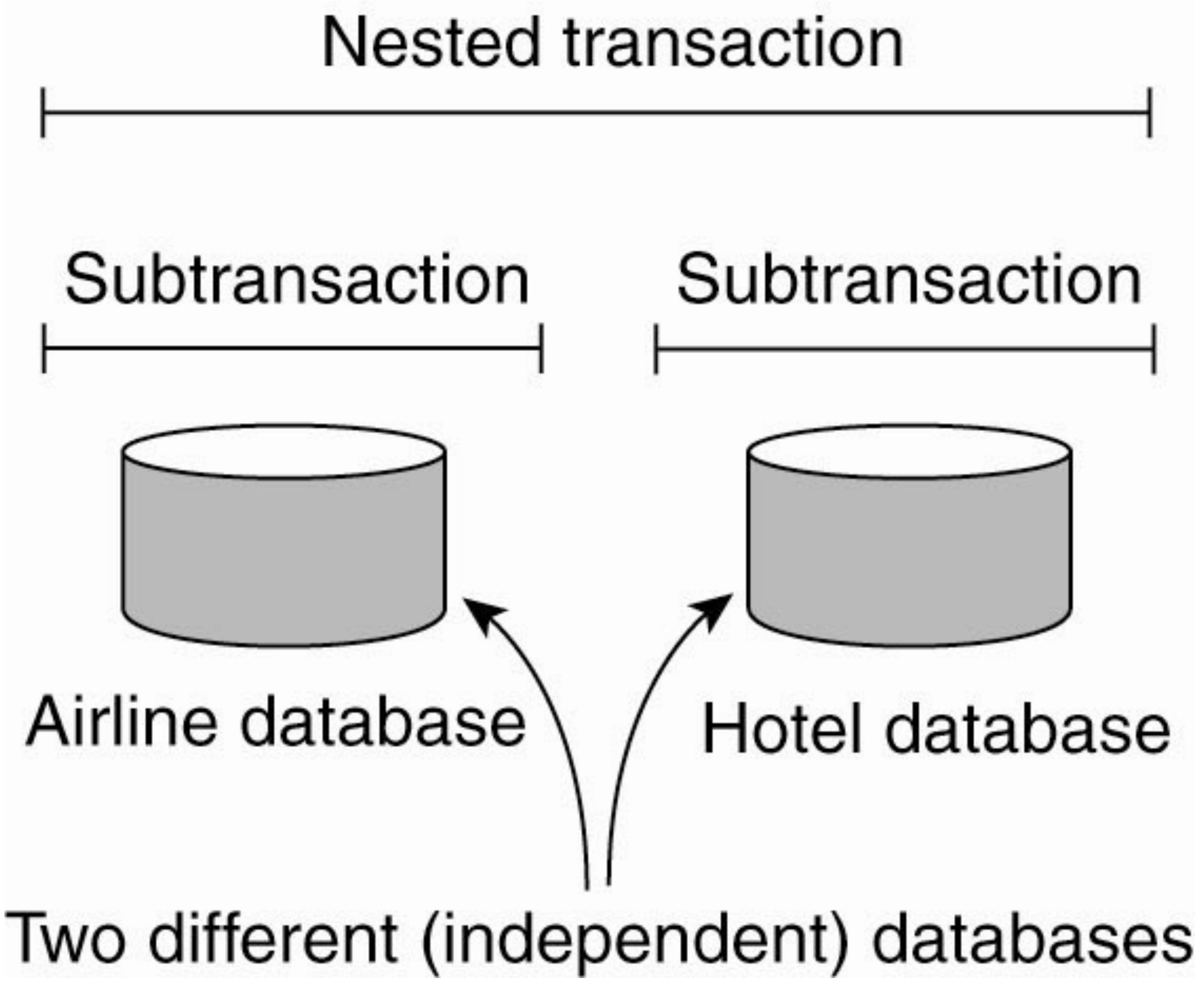


Figure 1-9. A nested transaction.

Transaction Processing Systems (4)

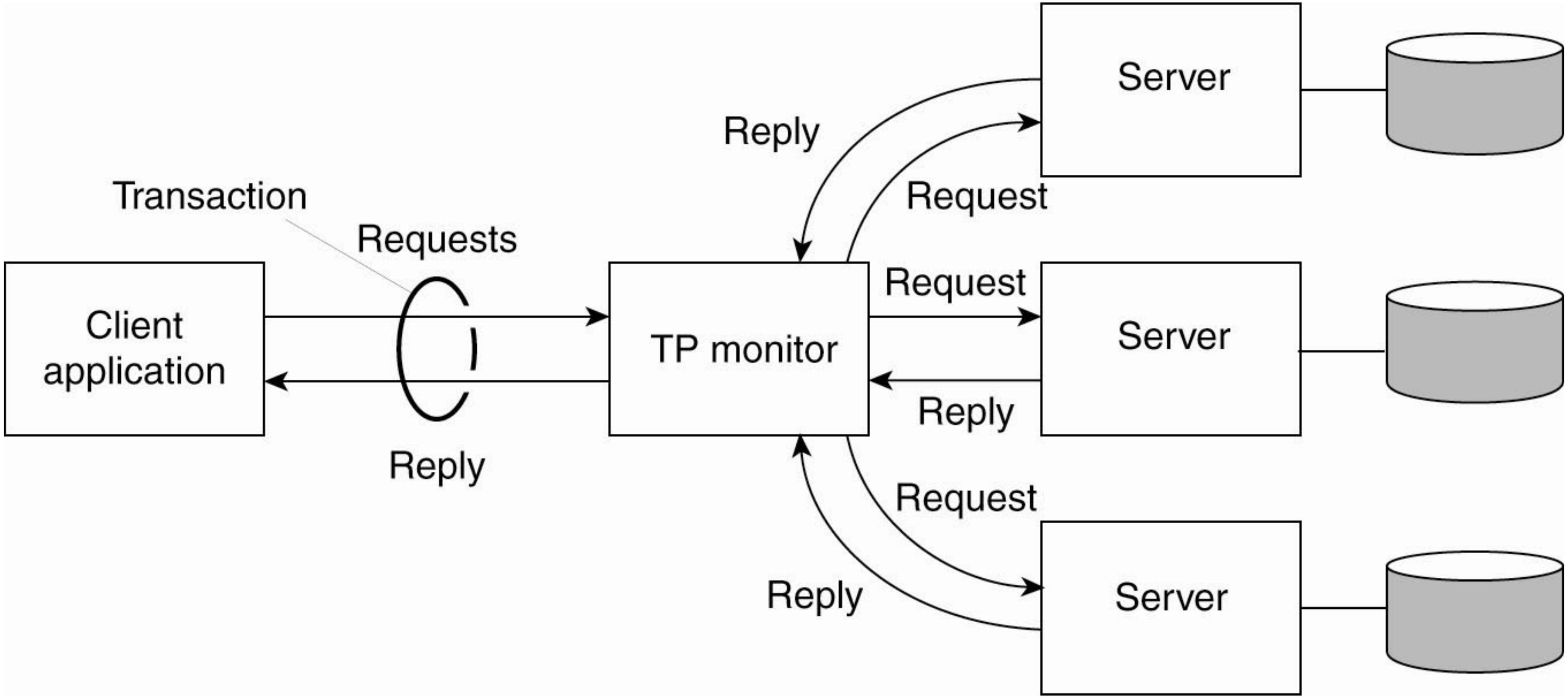


Figure 1-10. The role of a TP monitor in distributed systems.

Enterprise Application Integration

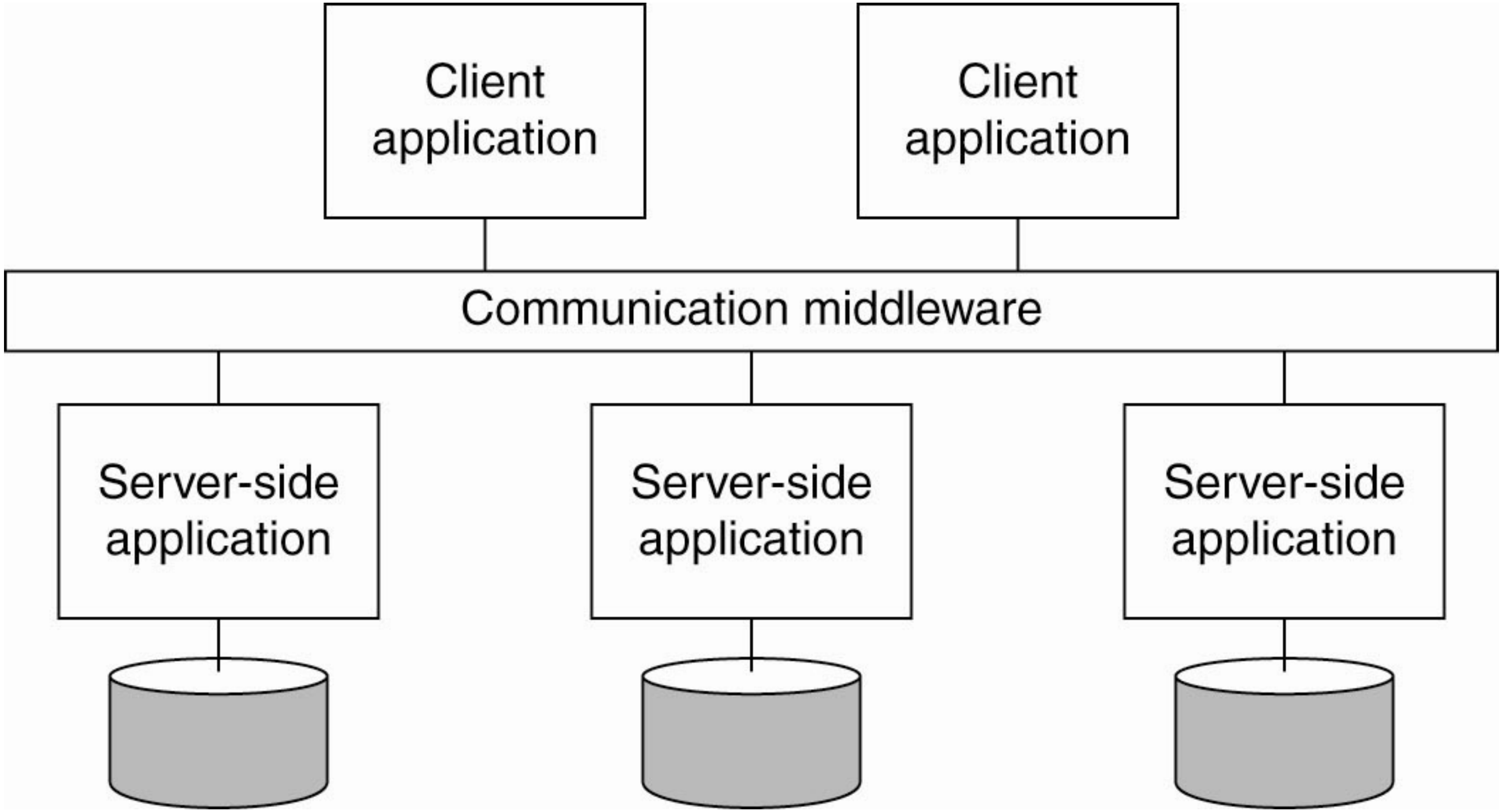


Figure 1-11. Middleware as a communication facilitator in enterprise application integration.

Distributed Pervasive Systems

Requirements for pervasive systems

- Embrace contextual changes.
- Encourage ad hoc composition.
- Recognize sharing as the default.

CHAPTER 2: ARCHITECTURE

Architecture Styles
Centralised : Client Server, Multitier
Decentralised: Peer to Peer, Structured Peer to Peer

Architectural Styles (1)

Important styles of architecture for distributed systems

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

Architectural Styles (2)

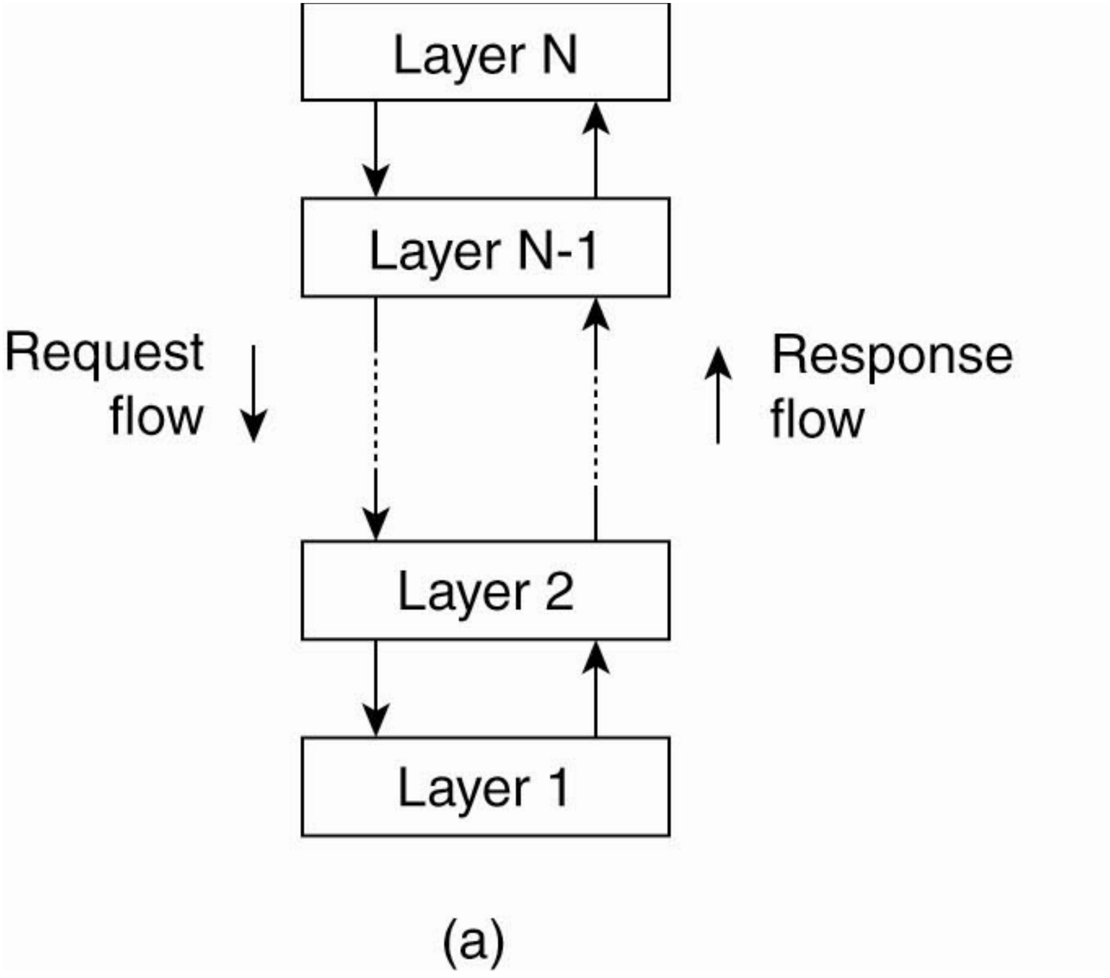


Figure 2-1. The (a) layered architectural style and …

Architectural Styles (3)

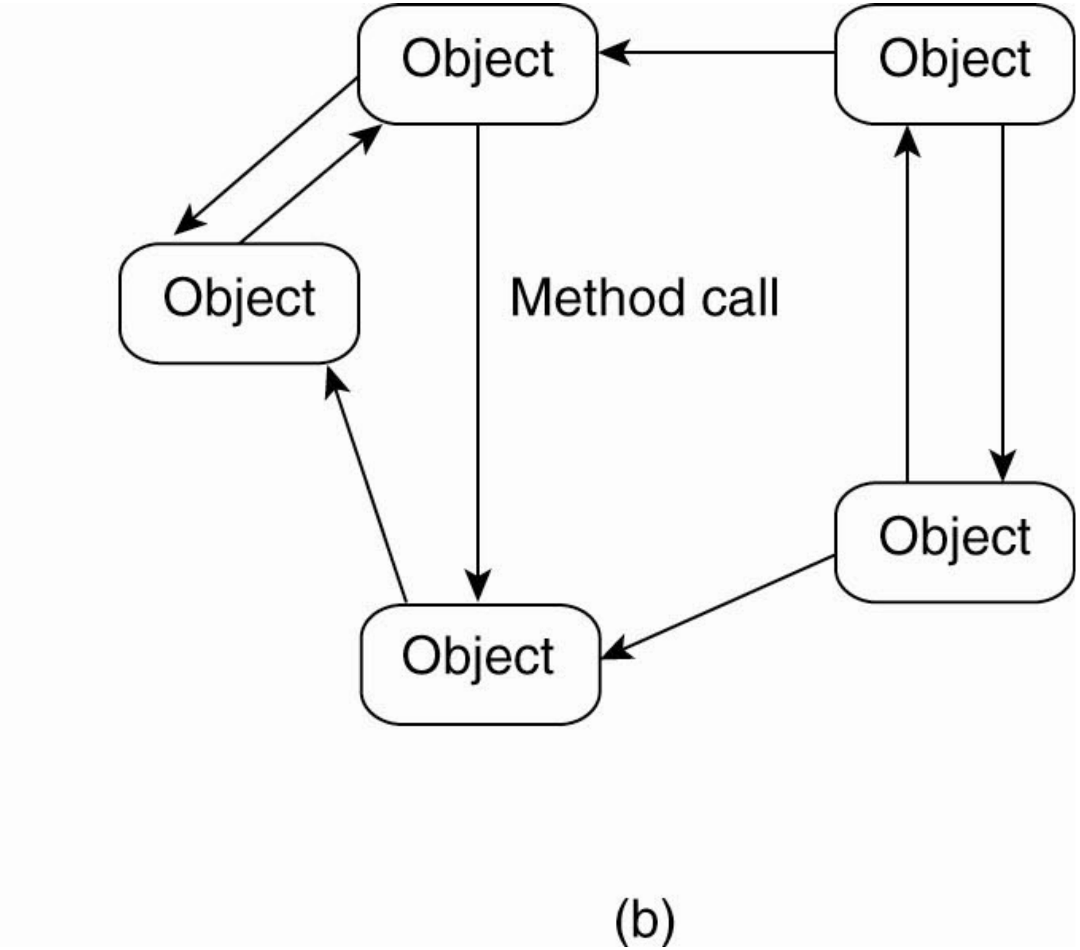


Figure 2-1. (b) The object-based architectural style.

Architectural Styles (4)

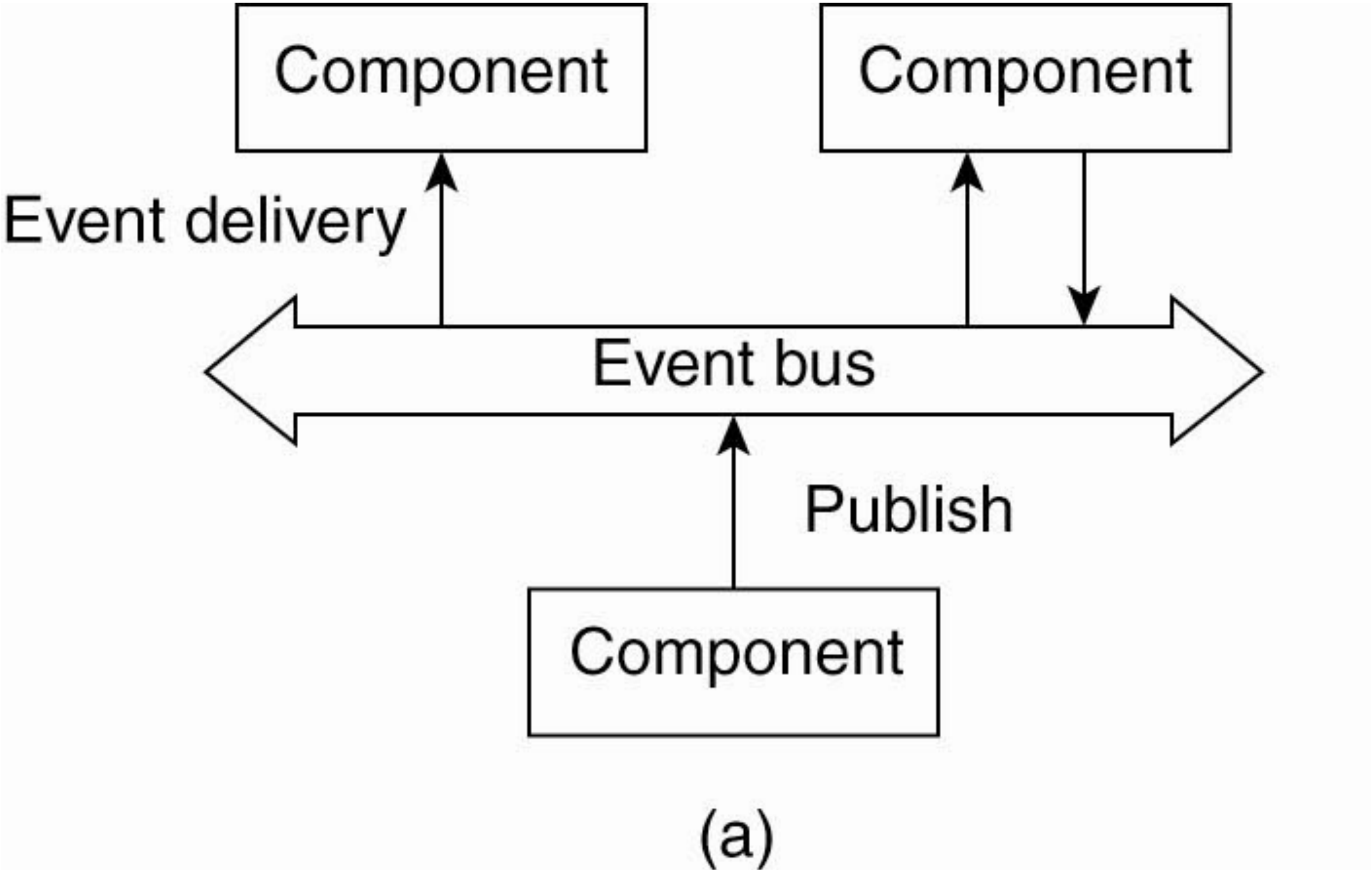


Figure 2-2. (a) The event-based architectural style and …

Architectural Styles (5)

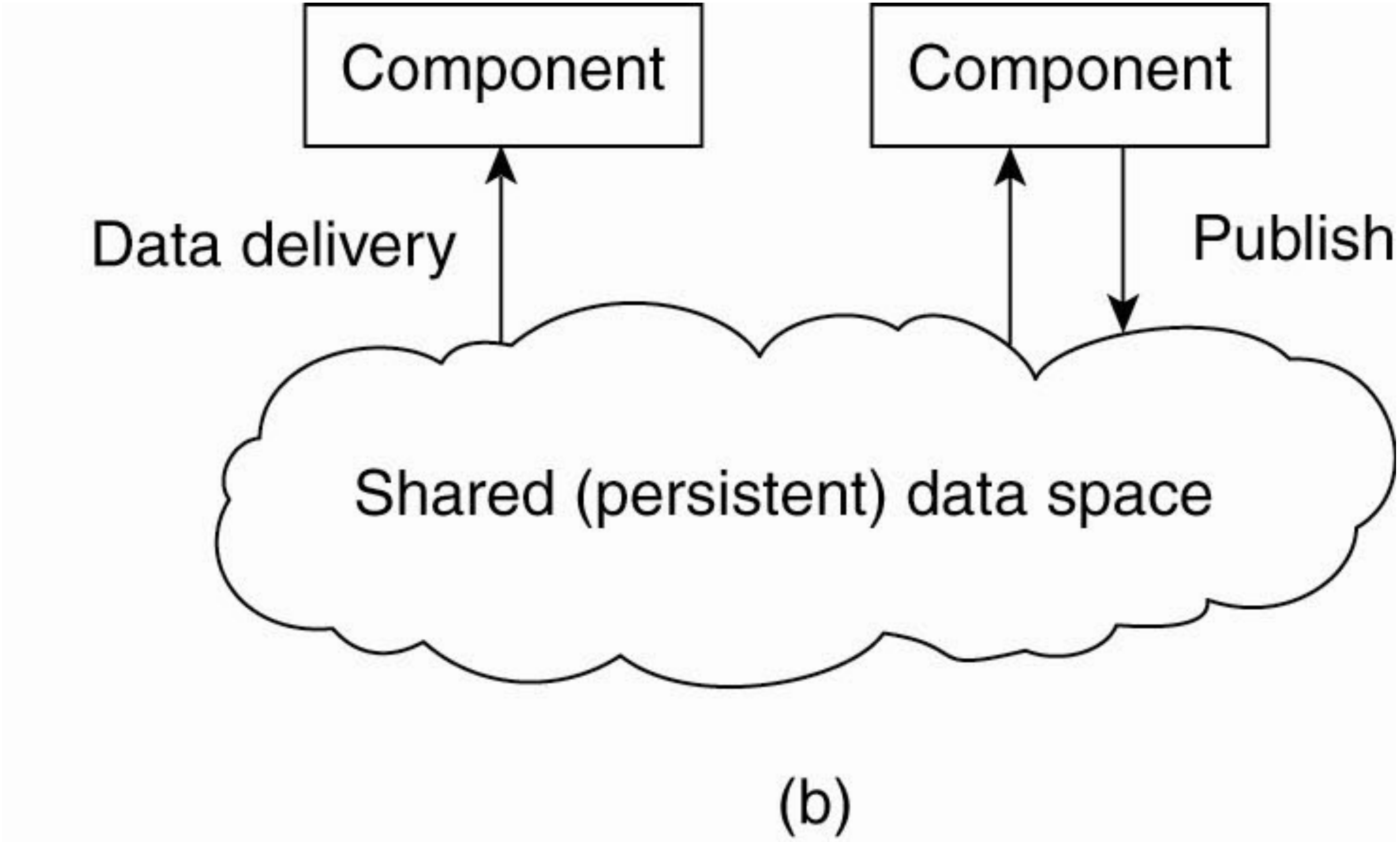


Figure 2-2. (b) The shared data-space architectural style.

Application Layering (2)

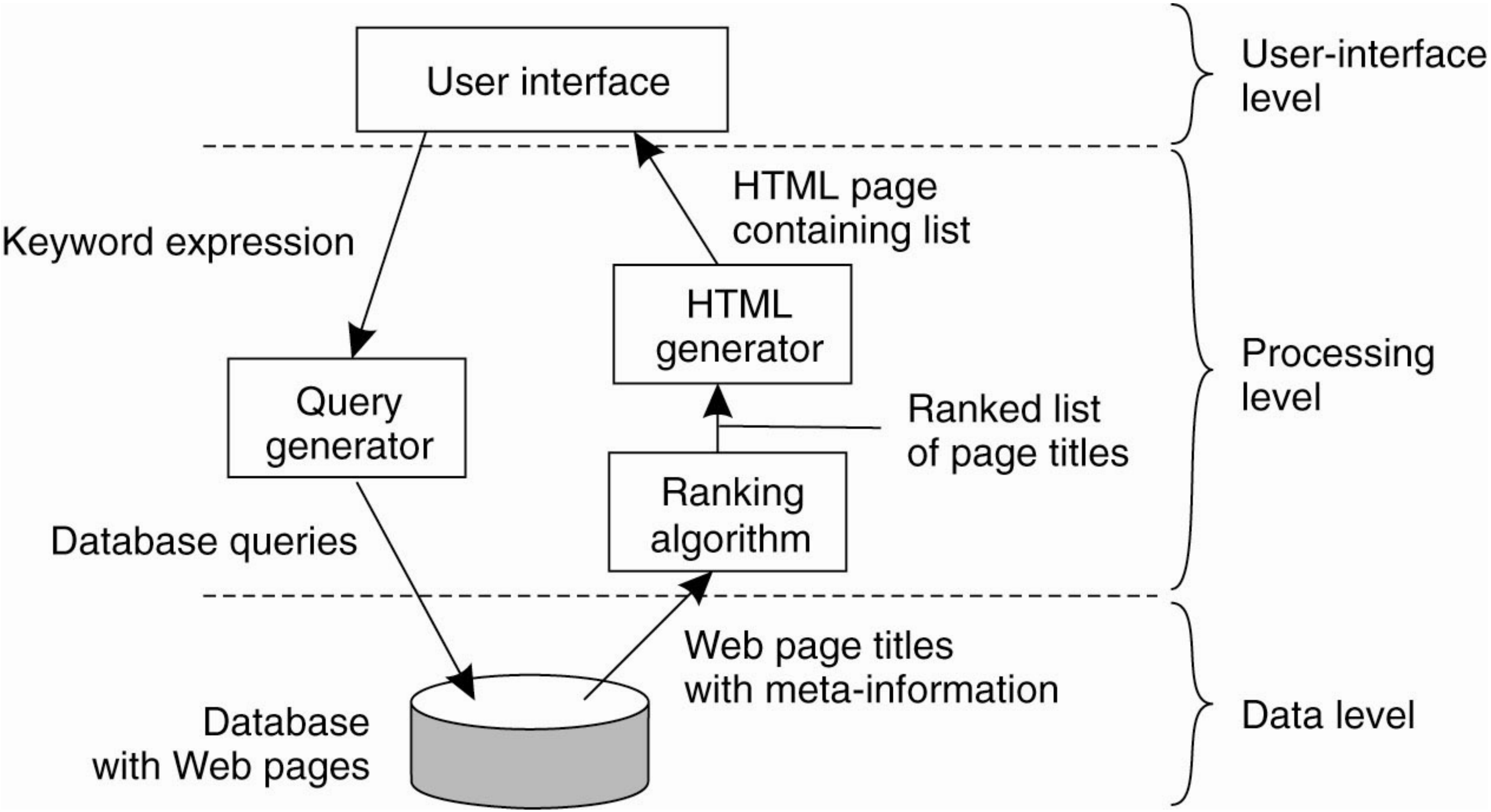


Figure 2-4. The simplified organization of an Internet search engine into three different layers.

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

Multitiered Architectures (2)

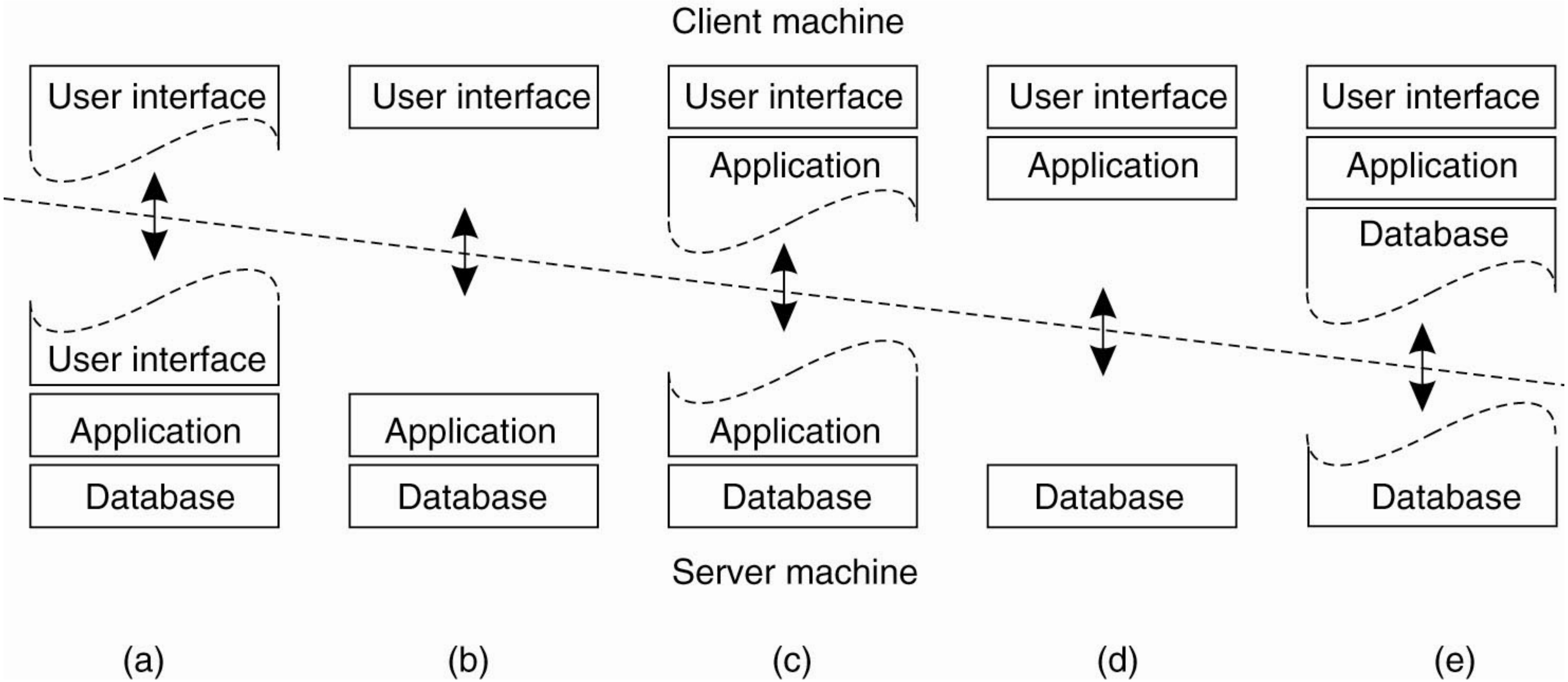


Figure 2-5. Alternative client-server organizations (a)–(e).

Multitiered Architectures (3)

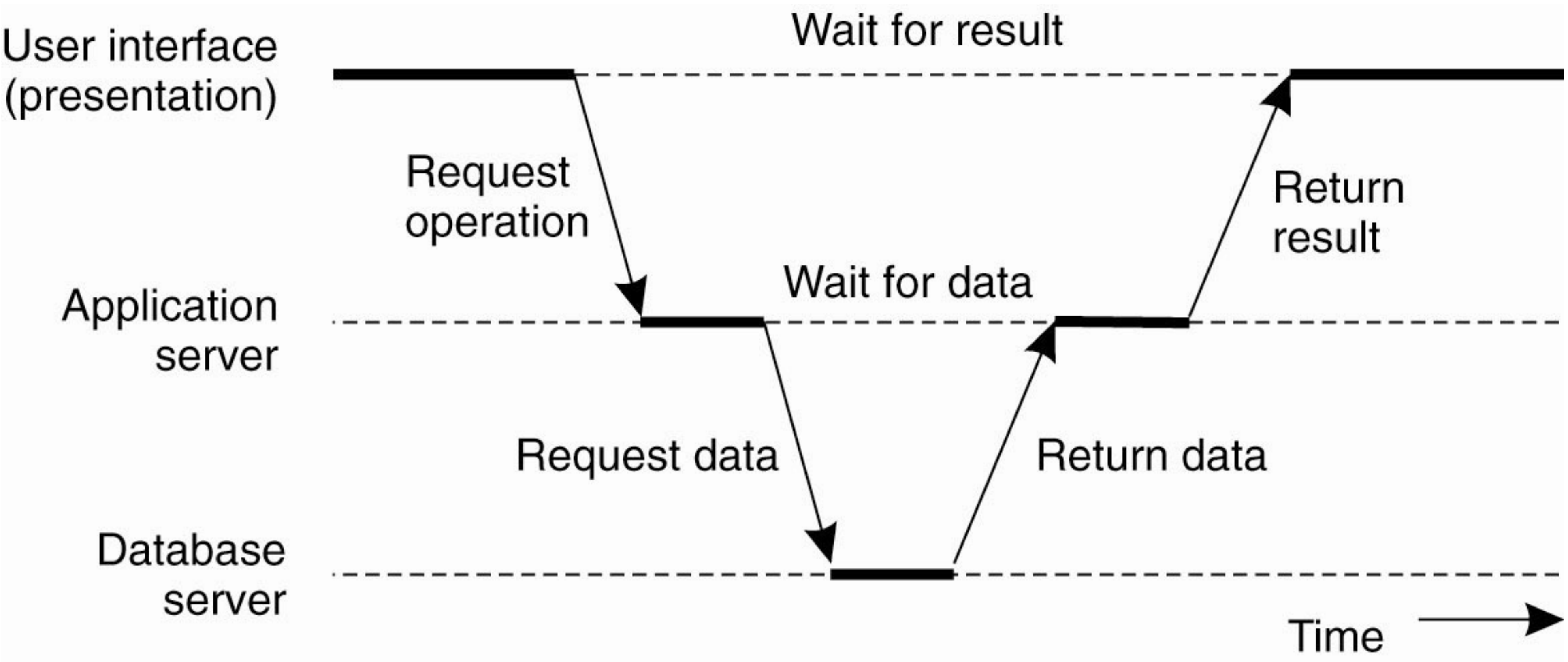


Figure 2-6. An example of a server acting as client.

Structured Peer-to-Peer Architectures (1)

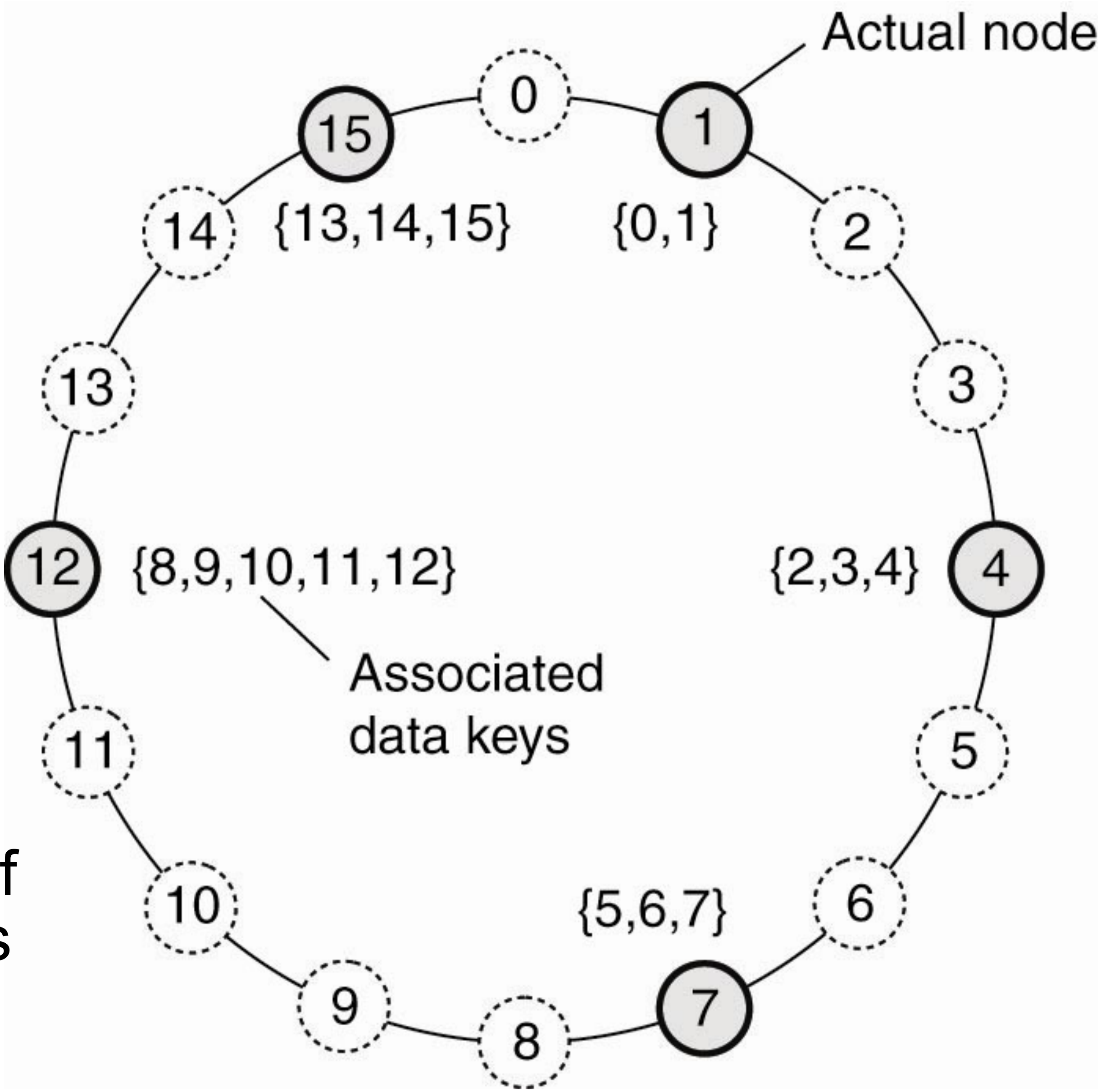


Figure 2-7. The mapping of data items onto nodes in Chord.

Structured Peer-to-Peer Architectures (2)

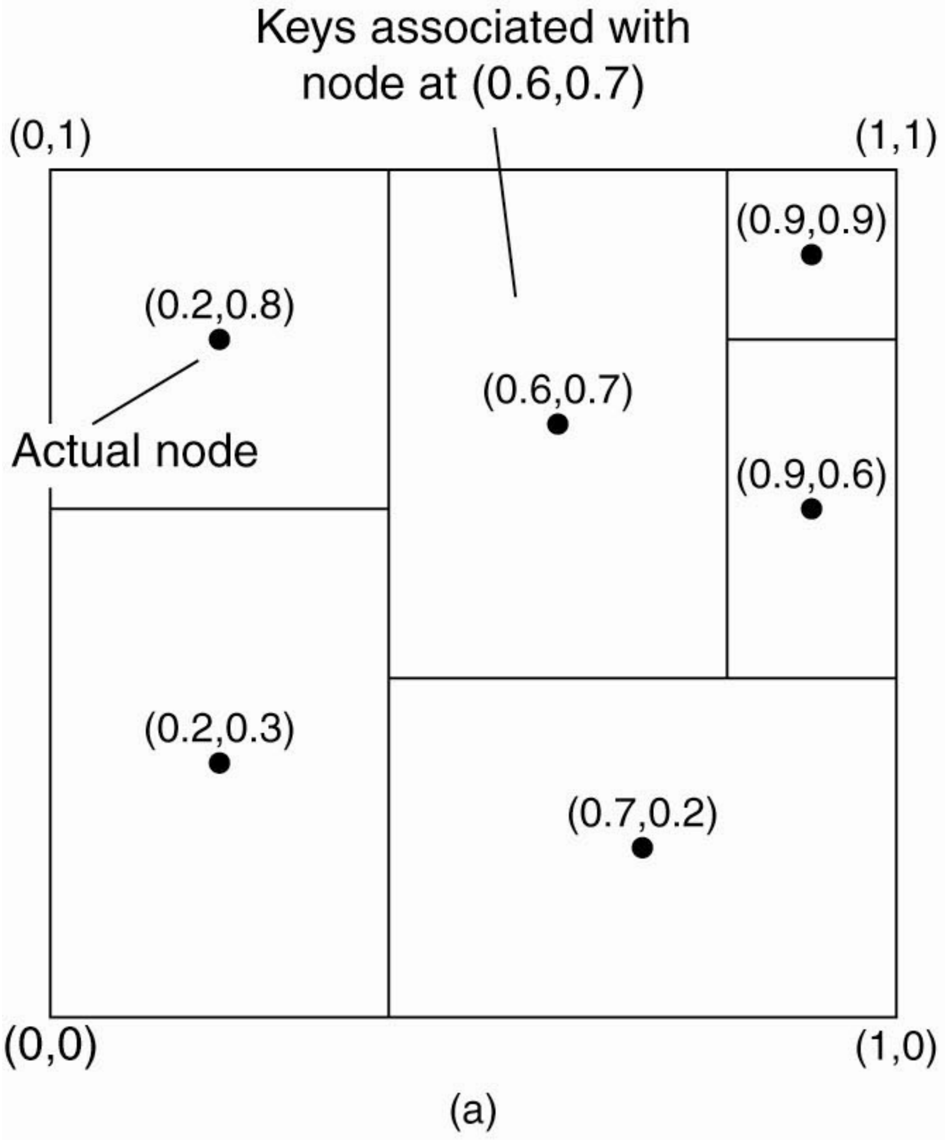


Figure 2-8. (a) The mapping of data items onto nodes in CAN.

Structured Peer-to-Peer Architectures (3)

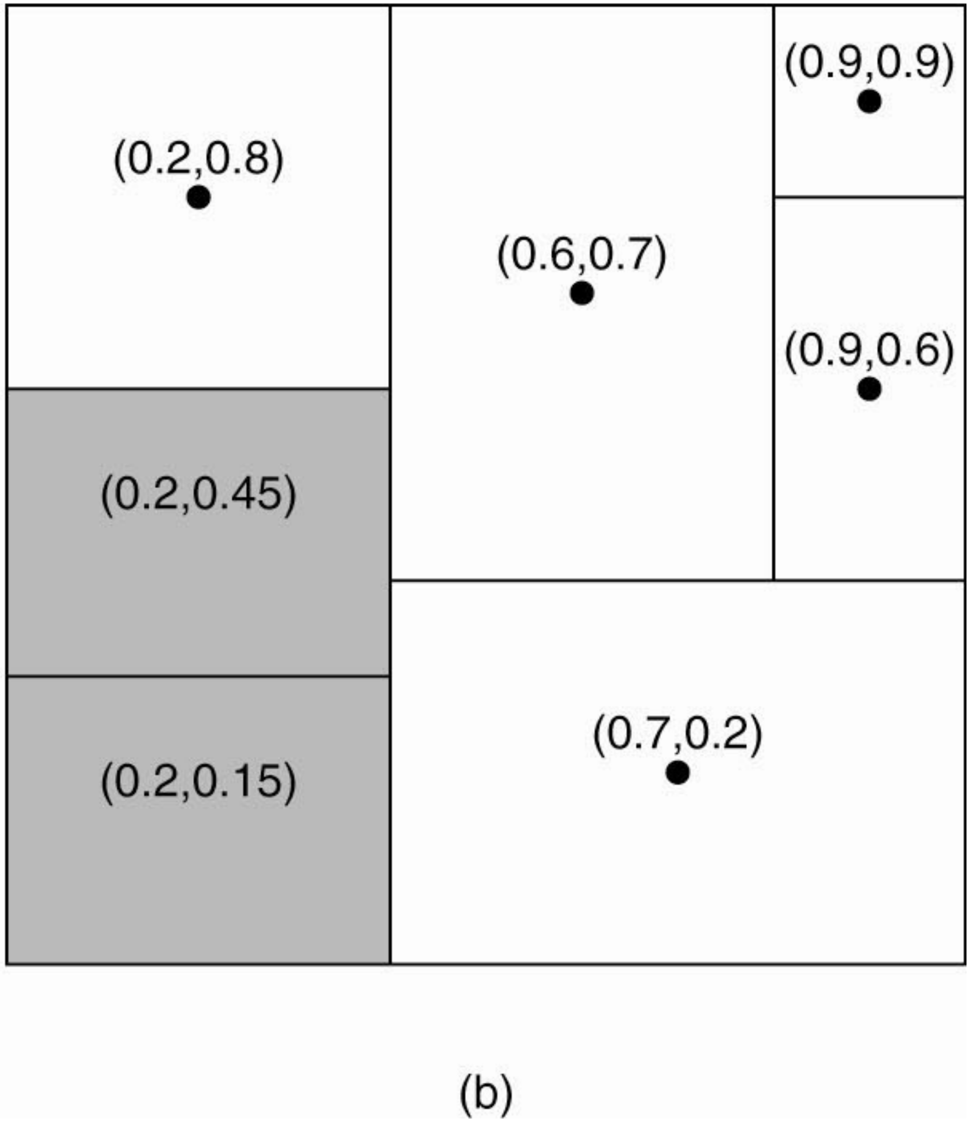


Figure 2-8. (b) Splitting a region when a node joins.

Superpeers

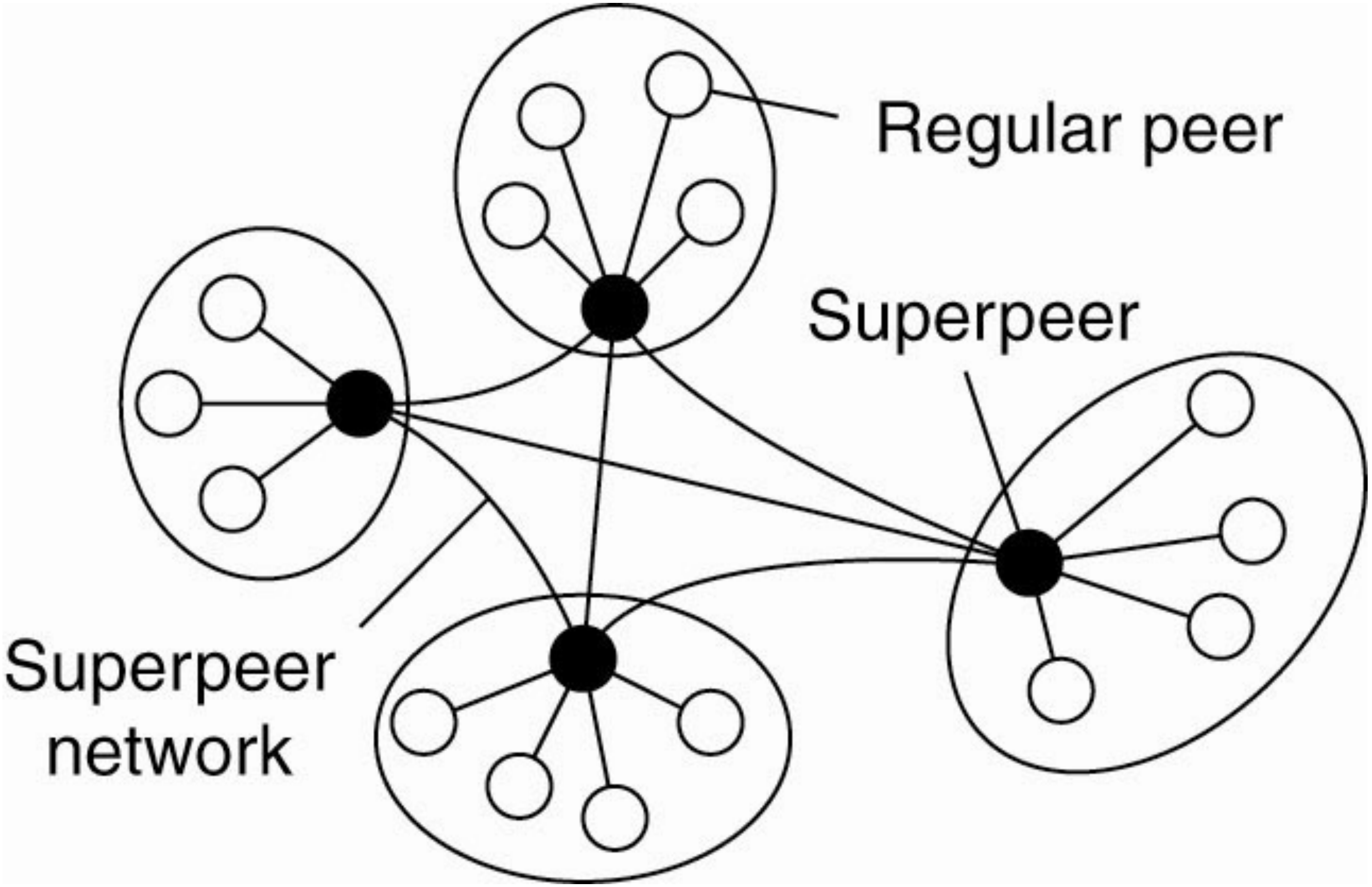


Figure 2-12. A hierarchical organization of nodes into a superpeer network.

General Approaches to Adaptive Software

Three basic approaches to adaptive software:

- Separation of concerns
- Computational reflection
- Component-based design

CHAPTER 3: PROCESSES

Processes, Threads, Processors
Virtualisation
Client/Server

Introduction to Threads

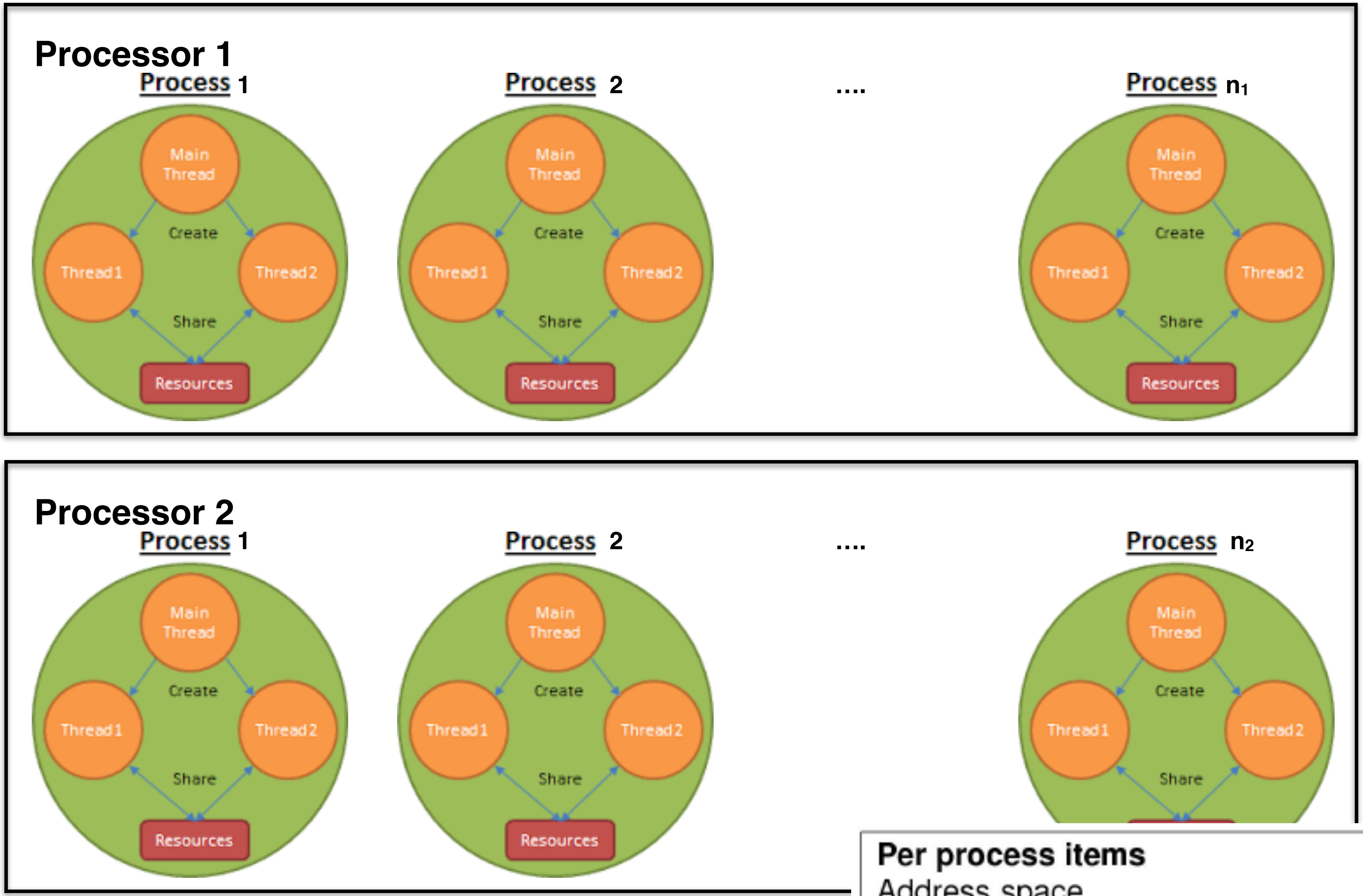
Basic idea

We build virtual processors in software, on top of physical processors:

Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.

Thread: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.

Process: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.



.

.

.

Processor n_3

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Items shared by all threads in a process
- Items private to each thread

Context Switching

Contexts

- Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
- Thread context: The minimal collection of values stored in registers and memory, used for the execution of a series of instructions (i.e., processor context, state).
- Process context: The minimal collection of values stored in registers and memory, used for the execution of a thread (i.e., thread context, but now also at least MMU register values).

Context Switching

Observations

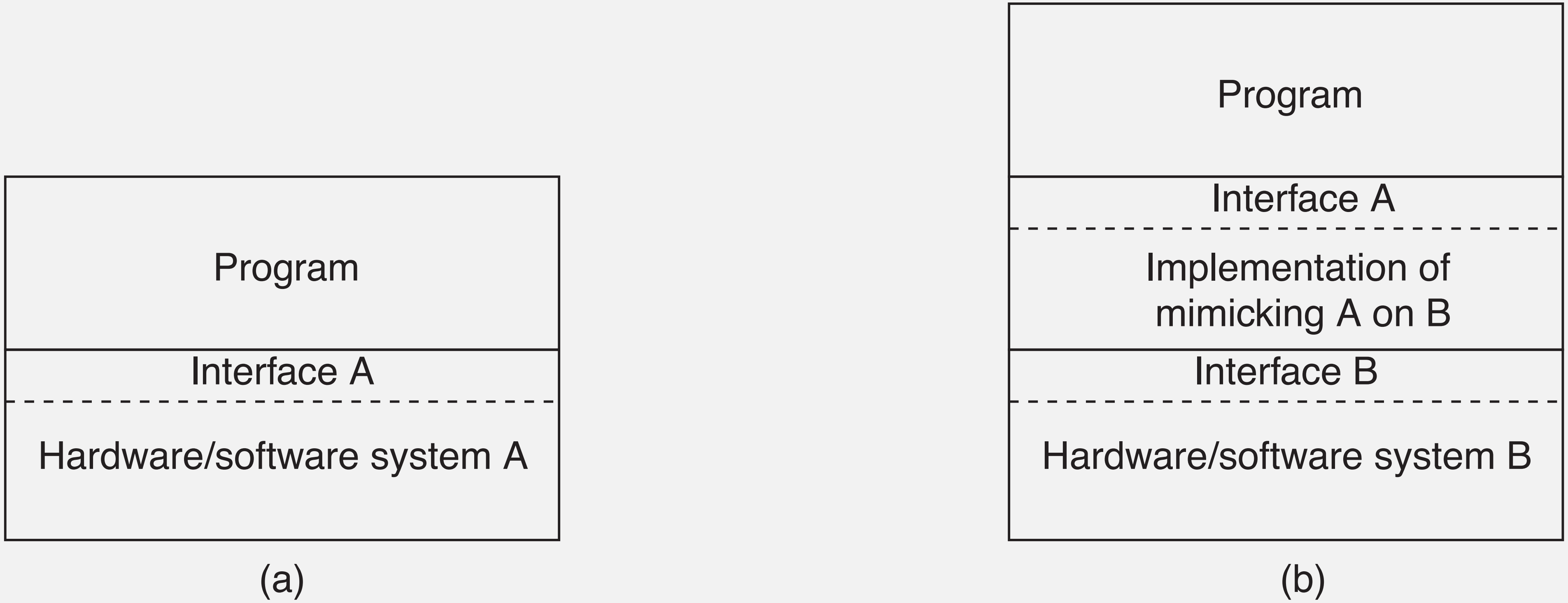
1. Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
2. Process switching is generally more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.
3. Creating and destroying threads is much cheaper than doing so for processes.

Virtualization

Observation

Virtualization is becoming increasingly important:

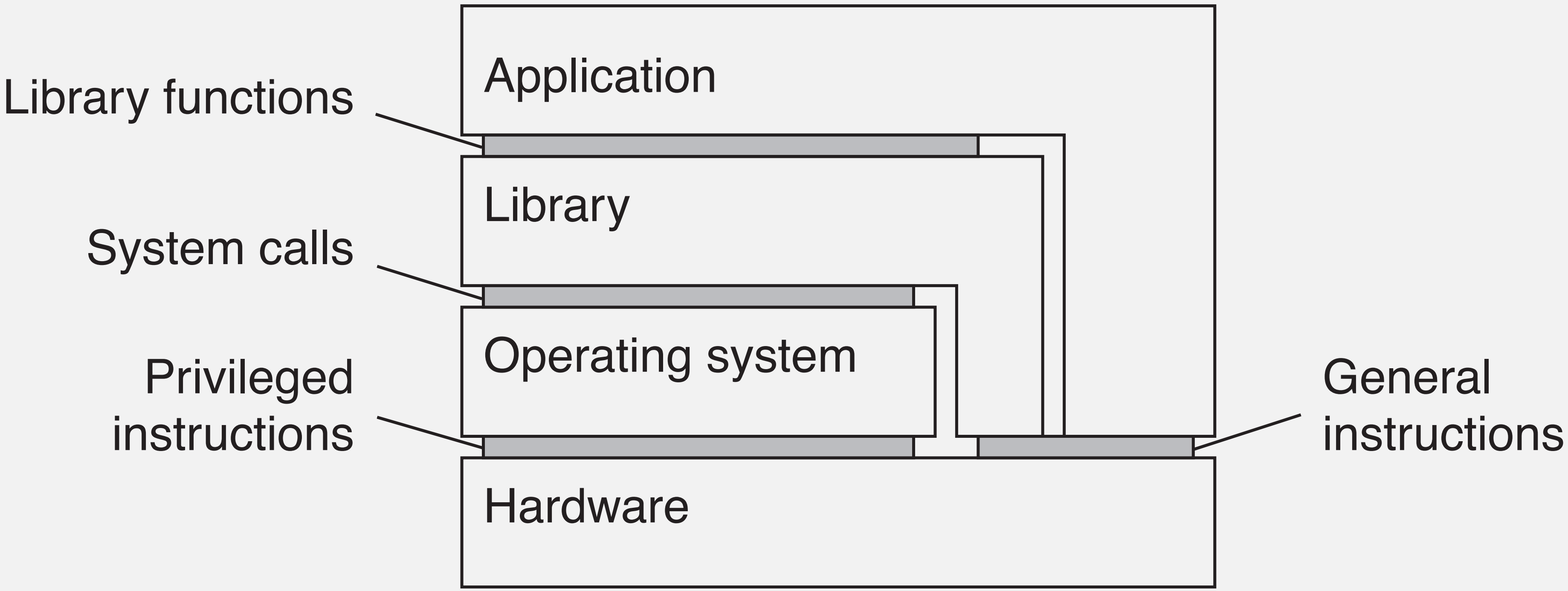
- Hardware changes faster than software
- Ease of portability and code migration
- Isolation of failing or attacked components


Program
Interface A
Hardware/software system A
(a)
Program
Interface A
Implementation of mimicking A on B
Interface B
Hardware/software system B
(b)

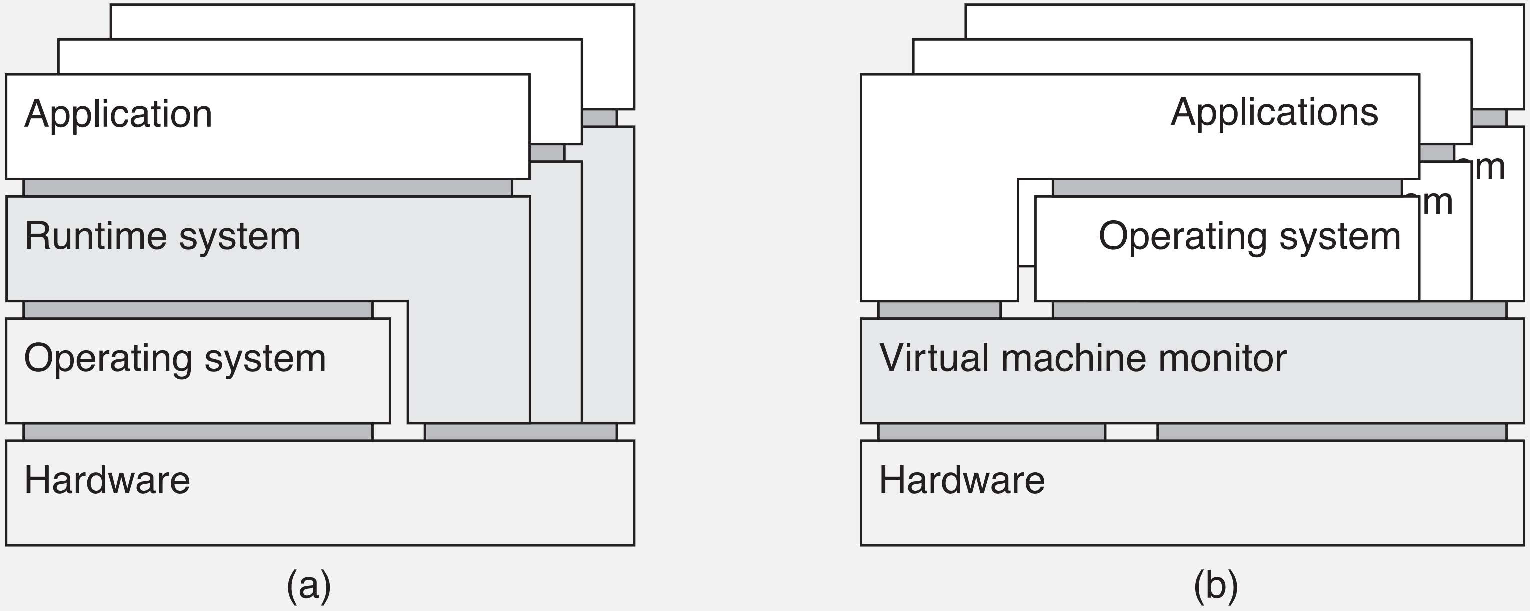
Architecture of VMs

Observation

Virtualization can take place at very different levels, strongly depending on the interfaces as offered by various systems components:



Process VMs versus VM Monitors

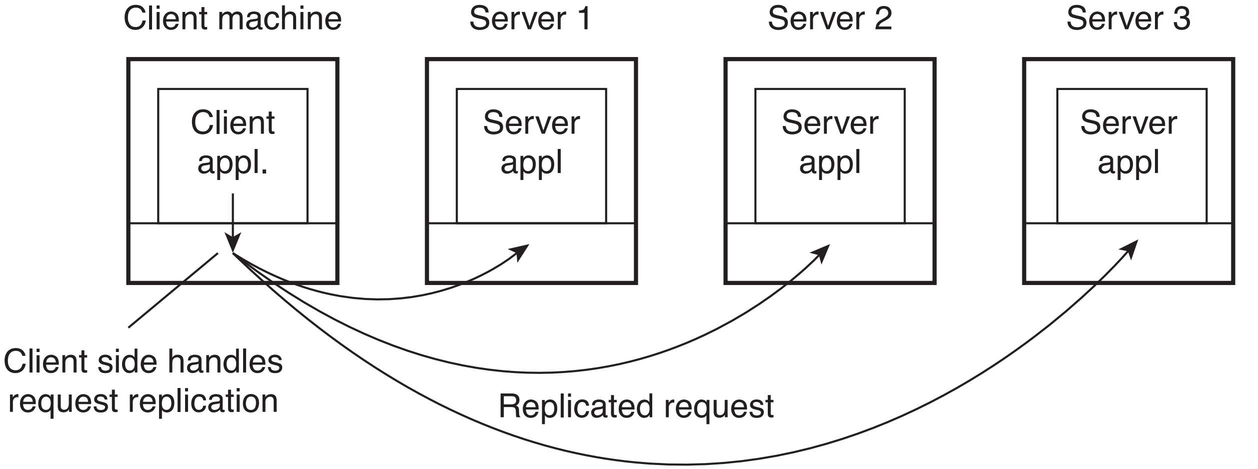


- **Process VM:** A program is compiled to intermediate (portable) code, which is then executed by a runtime system (Example: Java VM).
- **VM Monitor:** A separate software layer mimics the instruction set of hardware ⇒ a complete operating system and its applications can be supported (Example: VMware, VirtualBox).

Client-Side Software

Generally tailored for distribution transparency

- access transparency: client-side stubs for RPCs
- location/migration transparency: let client-side software keep track of actual location
- replication transparency: multiple invocations handled by client stub:



- failure transparency: can often be placed only at client (we're trying to mask server and communication failures).

Servers: General organization

Basic model

A server is a process that waits for incoming service requests at a specific transport address. In practice, there is a one-to-one mapping between a port and a service.

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
	24	any private mail system
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)
courier	530	Xerox RPC

Servers: General organization

Type of servers

Superservers: Servers that listen to several ports, i.e., provide several independent services. In practice, when a service request comes in, they start a subprocess to handle the request (UNIX *inetd*)

Iterative vs. concurrent servers: Iterative servers can handle only one client at a time, in contrast to concurrent servers

Servers and state

Stateless servers

Never keep accurate information about the status of a client after having handled a request:

- Don’t record whether a file has been opened (simply close it again after access)
- Don’t promise to invalidate a client’s cache
- Don’t keep track of your clients

Consequences

- Clients and servers are completely independent
- State inconsistencies due to client or server crashes are reduced
- Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Servers and state

Stateful servers

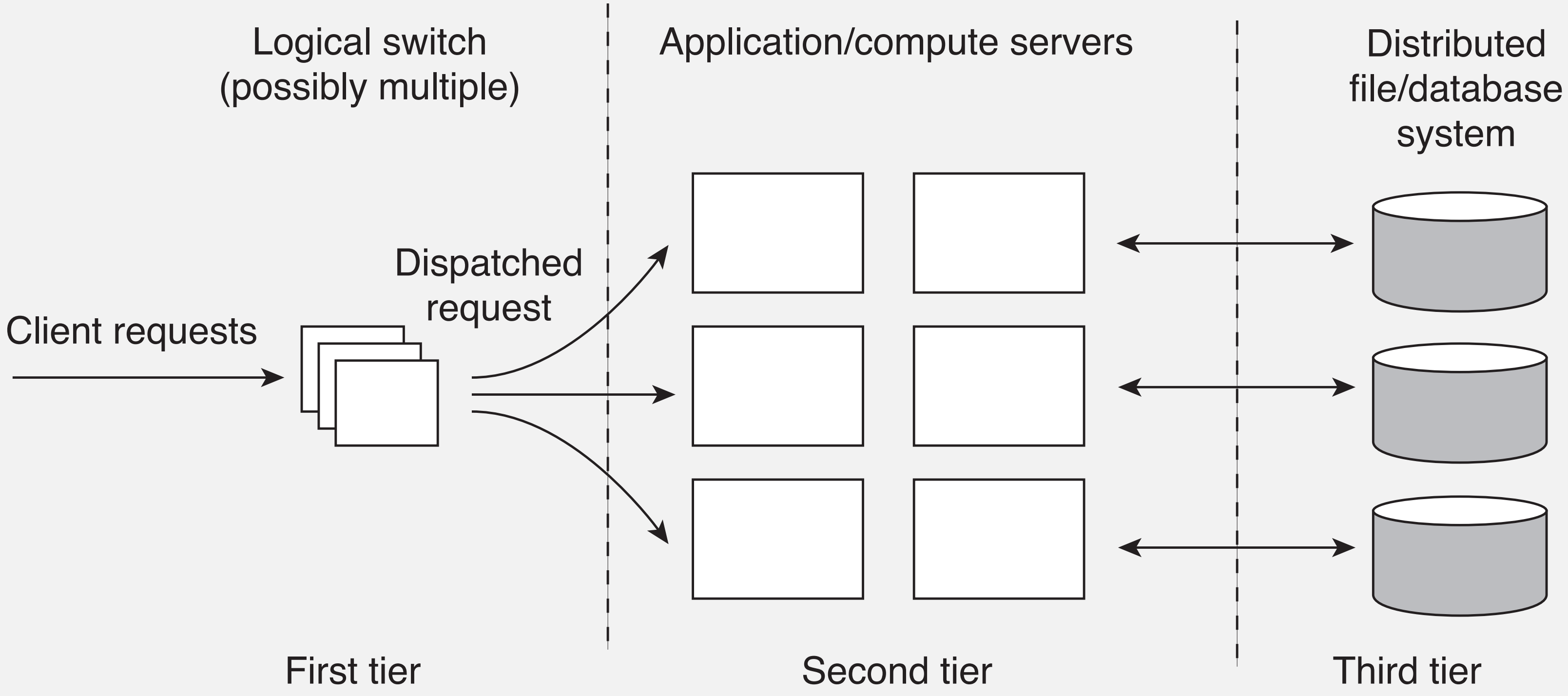
Keeps track of the status of its clients:

- Record that a file has been opened, so that prefetching can be done
- Knows which data a client has cached, and allows clients to keep local copies of shared data

Observation

The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is not a major problem.

Server clusters: three different tiers



Crucial element

The first tier is generally responsible for passing requests to an appropriate server.

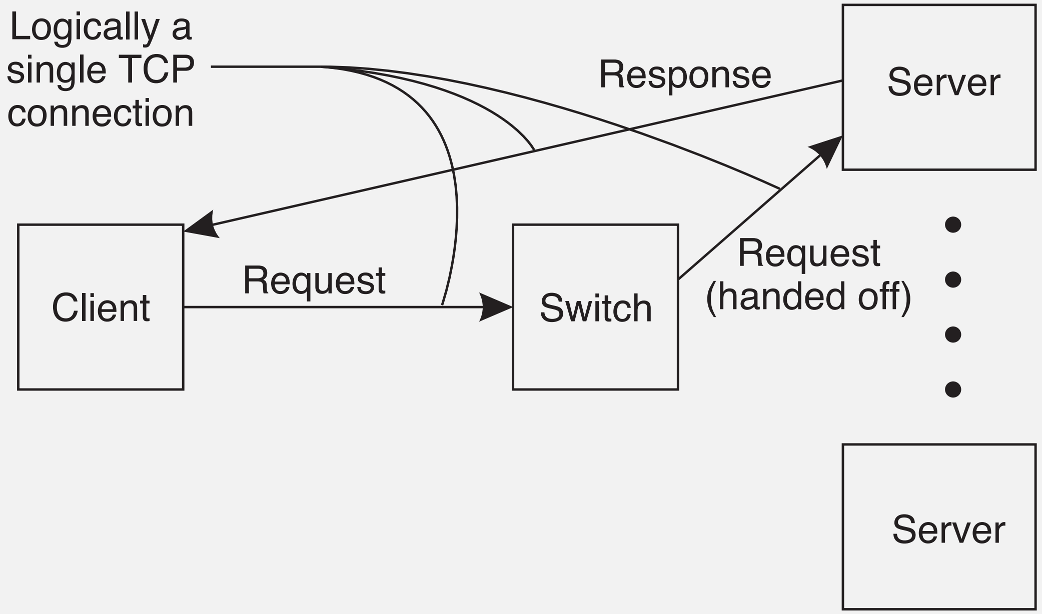
Request Handling

Observation

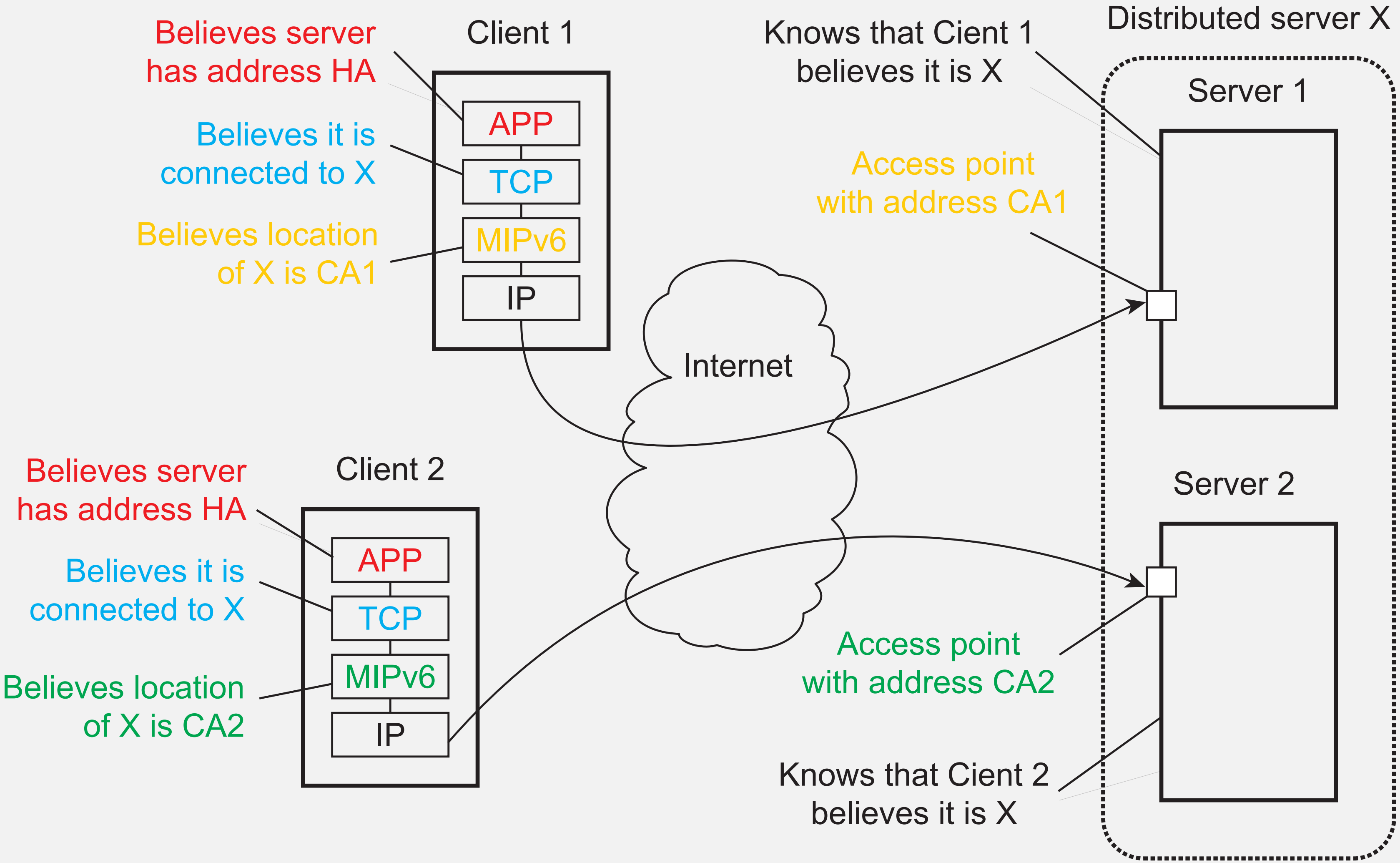
Having the first tier handle all communication from/to the cluster may lead to a bottleneck.

Solution

Various, but one popular one is TCP-handoff



Distributed servers with stable IPv6 address(es)


Believes server has address HA
Client 1
APP
Believes it is connected to X
TCP
Believes location of X is CA1
MIPv6
IP
Knows that Cient 1 believes it is X
Distributed server X
Server 1
Access point with address CA1
Internet
Believes server has address HA
Client 2
APP
Believes it is connected to X
TCP
Believes location of X is CA2
MIPv6
IP
Server 2
Access point with address CA2
Knows that Cient 2 believes it is X

Distributed servers: addressing details

Essence

Clients having MobileIPv6 can transparently set up a connection to any peer:

- Client *C* sets up connection to IPv6 home address *HA*
- *HA* is maintained by a (network-level) home agent, which hands off the connection to a registered care-of address *CA*.
- *C* can then apply route optimization by directly forwarding packets to address *CA* (i.e., without the handoff through the home agent).

Collaborative CDNs

Origin server maintains a home address, but hands off connections to address of collaborating peer ⇒ Origin server and peer appear as one server.

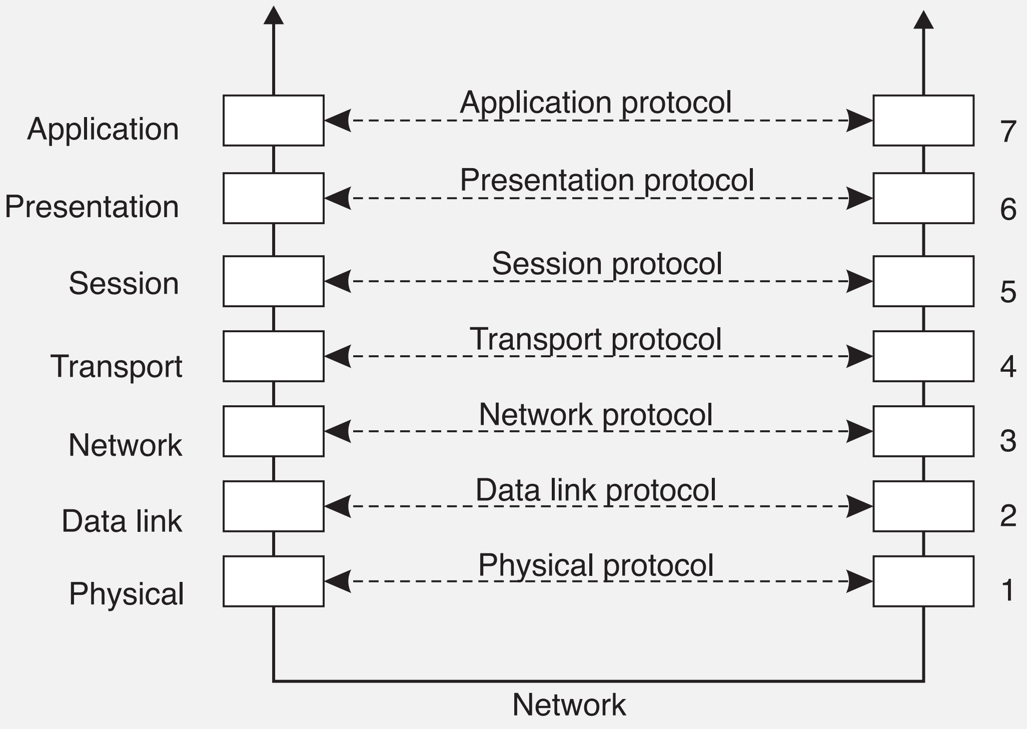
CHAPTER 4: COMMUNICATION

Communication Types, RPC, MPI, Message Oriented, Stream Communication

Layered Protocols

- Low-level layers
- Transport layer
- Application layer
- Middleware layer

Basic networking model



Drawbacks

- Focus on message-passing only
- Often unneeded or unwanted functionality
- Violates access transparency

Low-level layers

Recap

- Physical layer: contains the specification and implementation of bits, and their transmission between sender and receiver
- Data link layer: prescribes the transmission of a series of bits into a frame to allow for error and flow control
- Network layer: describes how packets in a network of computers are to be routed.

Observation

For many distributed systems, the lowest-level interface is that of the network layer.

Transport Layer

Important

The transport layer provides the actual communication facilities for most distributed systems.

Standard Internet protocols

- TCP: connection-oriented, reliable, stream-oriented communication
- UDP: unreliable (best-effort) datagram communication

Note

IP multicasting is often considered a standard available service (which may be dangerous to assume).

Middleware Layer

Observation

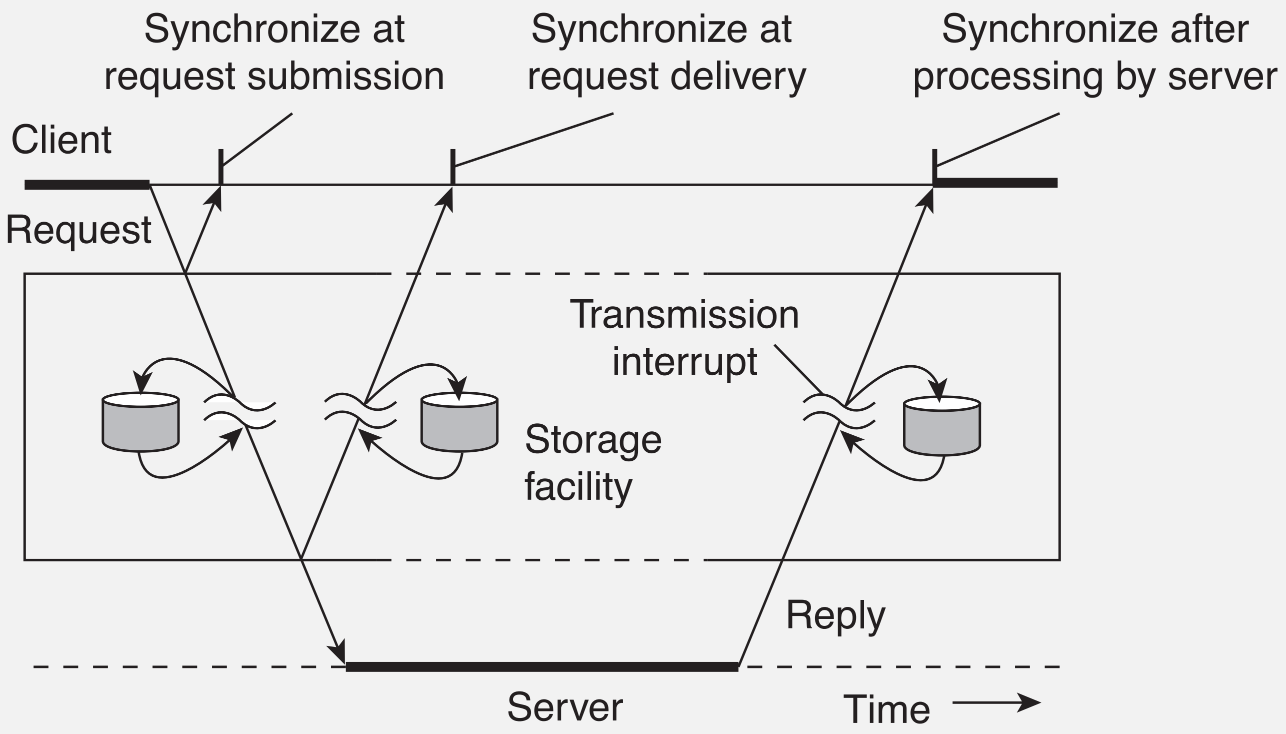
Middleware is invented to provide common services and protocols that can be used by many different applications

- A rich set of communication protocols
- (Un)marshaling of data, necessary for integrated systems
- Naming protocols, to allow easy sharing of resources
- Security protocols for secure communication
- Scaling mechanisms, such as for replication and caching

Note

What remains are truly application-specific protocols...
such as?

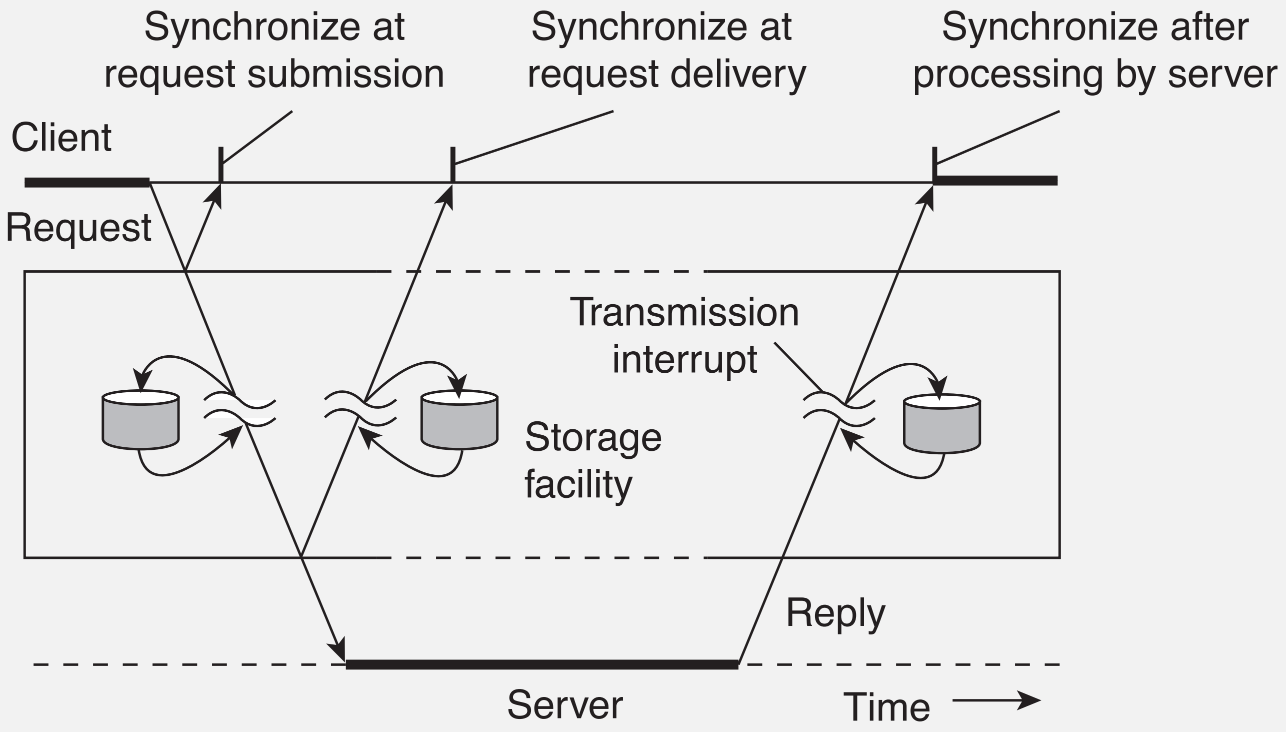
Types of communication


Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

Distinguish

- Transient versus persistent communication
- Asynchrounous versus synchronous communication

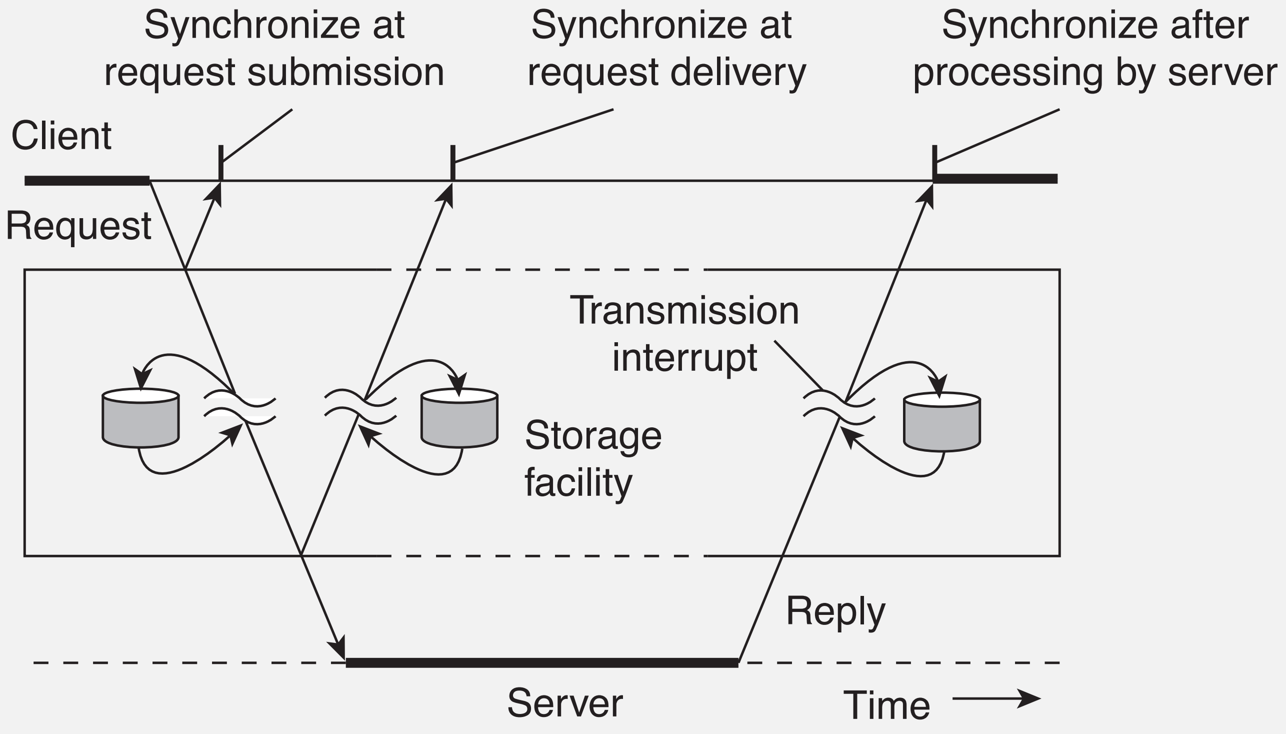
Types of communication


Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

Transient versus persistent

- Transient communication: Comm. server discards message when it cannot be delivered at the next server, or at the receiver.
- Persistent communication: A message is stored at a communication server as long as it takes to deliver it.

Types of communication


Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

Places for synchronization

- At request submission
- At request delivery
- After request processing

Client/Server

Some observations

Client/Server computing is generally based on a model of transient synchronous communication:

- Client and server have to be active at time of commun.
- Client issues request and blocks until it receives reply
- Server essentially waits only for incoming requests, and subsequently processes them

Drawbacks synchronous communication

- Client cannot do any other work while waiting for reply
- Failures have to be handled immediately: the client is waiting
- The model may simply not be appropriate (mail, news)

Messaging

Message-oriented middleware

Aims at high-level persistent asynchronous communication:

- Processes send each other messages, which are queued
- Sender need not wait for immediate reply, but can do other things
- Middleware often ensures fault tolerance

Remote Procedure Call (RPC)

- Basic RPC operation
- Parameter passing
- Variations

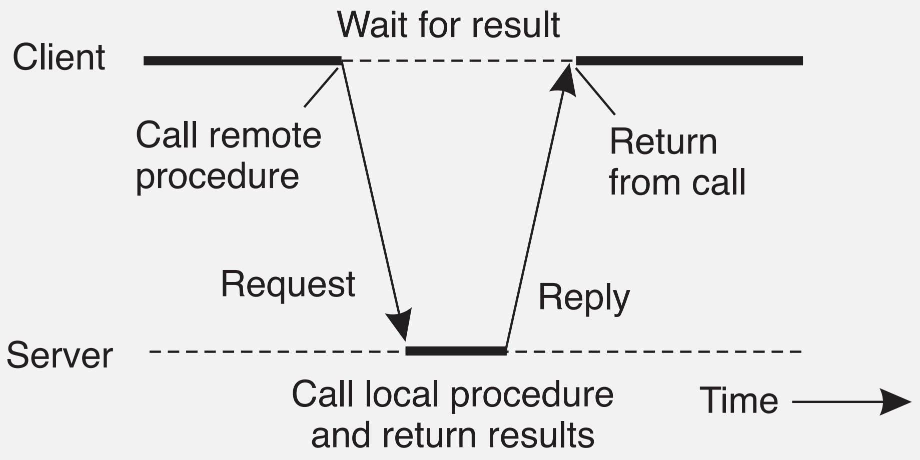
Basic RPC operation

Observations

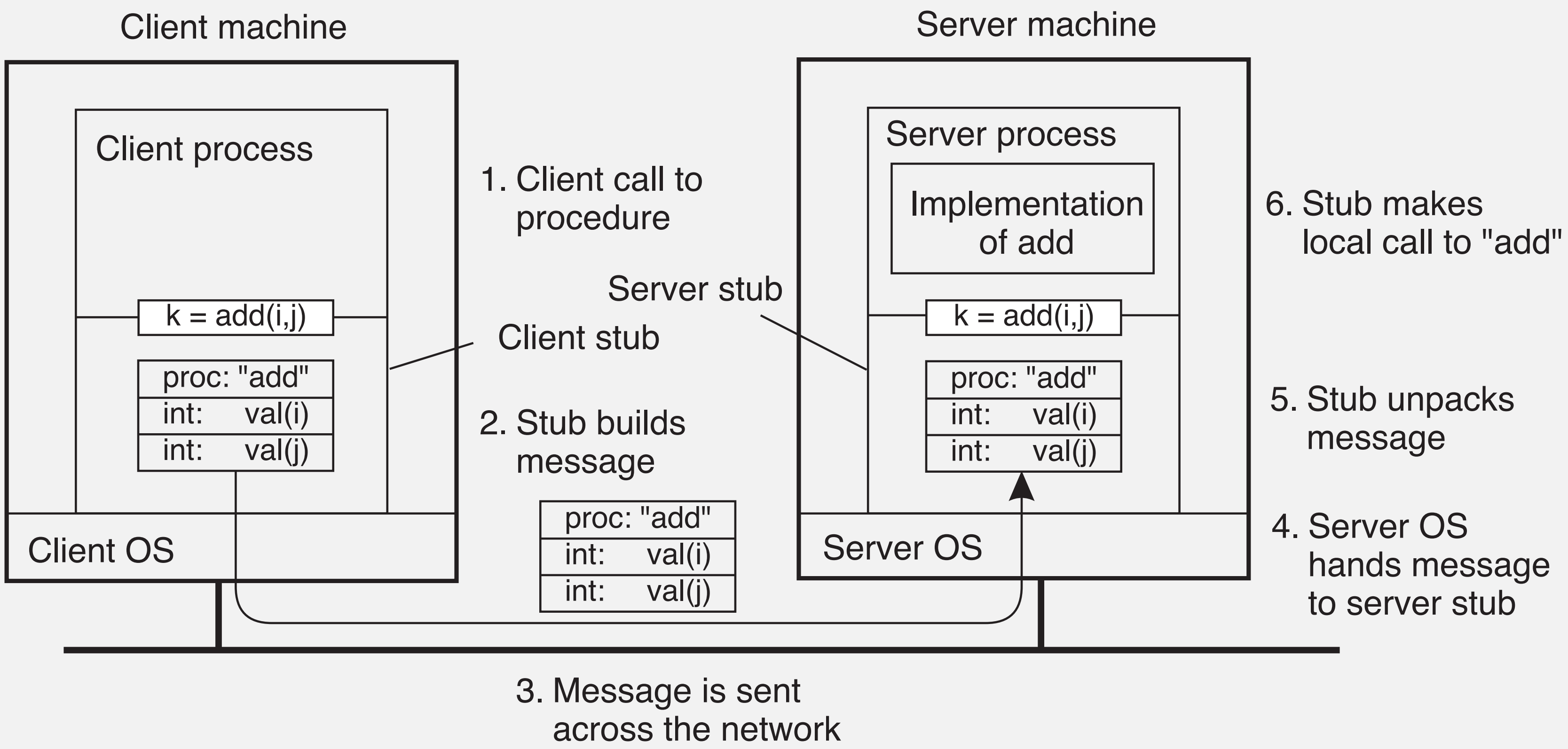
- Application developers are familiar with simple procedure model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on separate machine

Conclusion

Communication between caller & callee can be hidden by using procedure-call mechanism.



Basic RPC operation



1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters and calls server.
6. Server makes local call and returns result to stub.
7. Stub builds message; calls OS.
8. OS sends message to client’s OS.
9. Client’s OS gives message to stub.
10. Client stub unpacks result and returns to the client.

RPC: Parameter passing

Parameter marshaling

There's more than just wrapping parameters into a message:

- Client and server machines may have different data representations (think of byte ordering)
- Wrapping a parameter means transforming a value into a sequence of bytes
- Client and server have to agree on the same encoding:
  - How are basic data values represented (integers, floats, characters)
  - How are complex data values represented (arrays, unions)
- Client and server need to properly interpret messages, transforming them into machine-dependent representations.

RPC: Parameter passing

RPC parameter passing: some assumptions

- Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
- All data that is to be operated on is passed by parameters. Excludes passing references to (global) data.

Conclusion

Full access transparency cannot be realized.

Observation

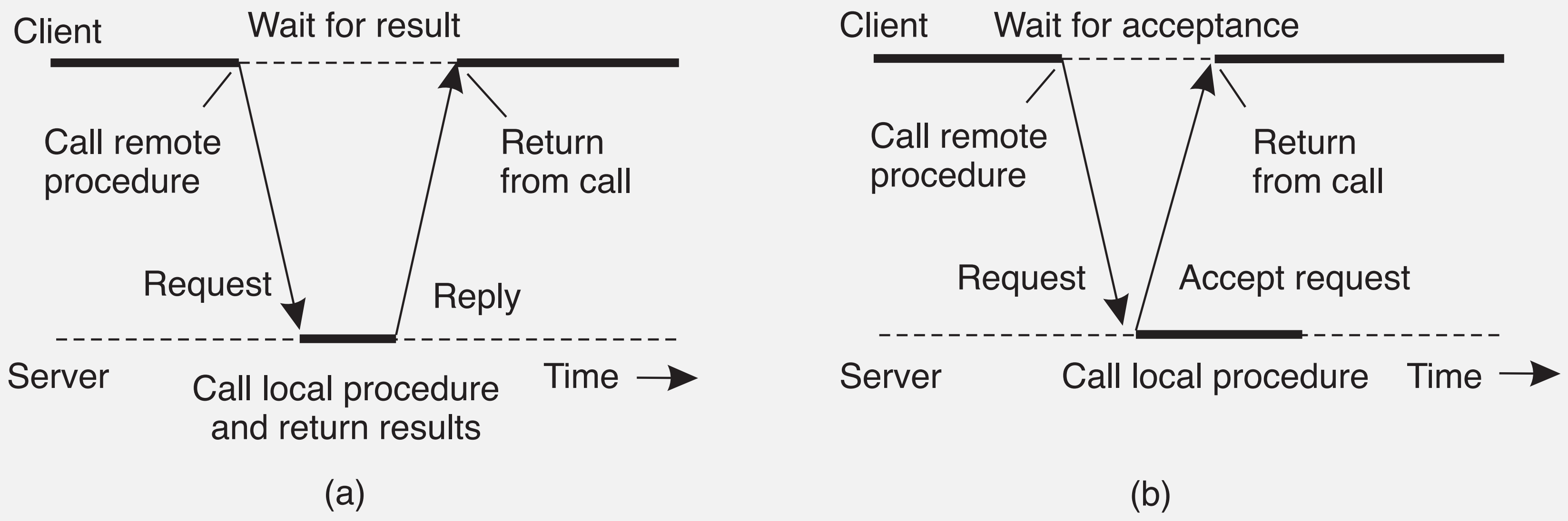
A remote reference mechanism enhances access transparency:

- Remote reference offers unified access to remote data
- Remote references can be passed as parameter in RPCs

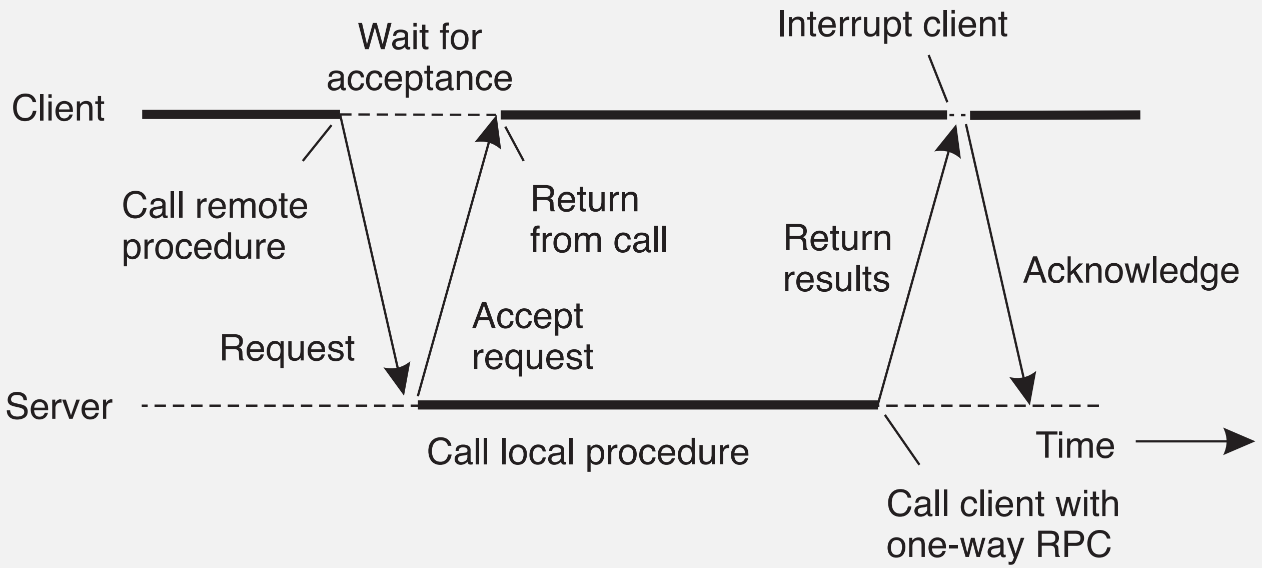
Asynchronous RPCs

Essence

Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.



Deferred synchronous RPCs



Variation

Client can also do a (non)blocking poll at the server to see whether results are available.

Message-Oriented Communication

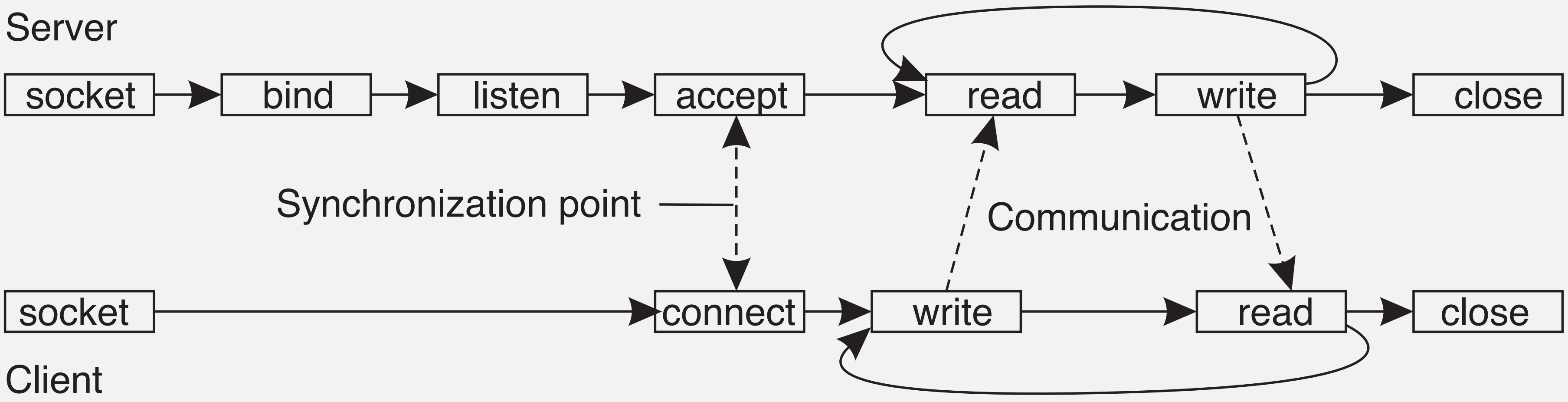
- Transient Messaging
- Message-Queuing System
- Message Brokers
- Example: IBM Websphere

Transient messaging: sockets

Berkeley socket interface

SOCKET	Create a new communication endpoint
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept *N* connections
ACCEPT	Block until request to establish a connection
CONNECT	Attempt to establish a connection
SEND	Send data over a connection
RECEIVE	Receive data over a connection
CLOSE	Release the connection

Transient messaging: sockets



Message Passing Interface

Basic send with user-provided buffering

Performs a blocking send

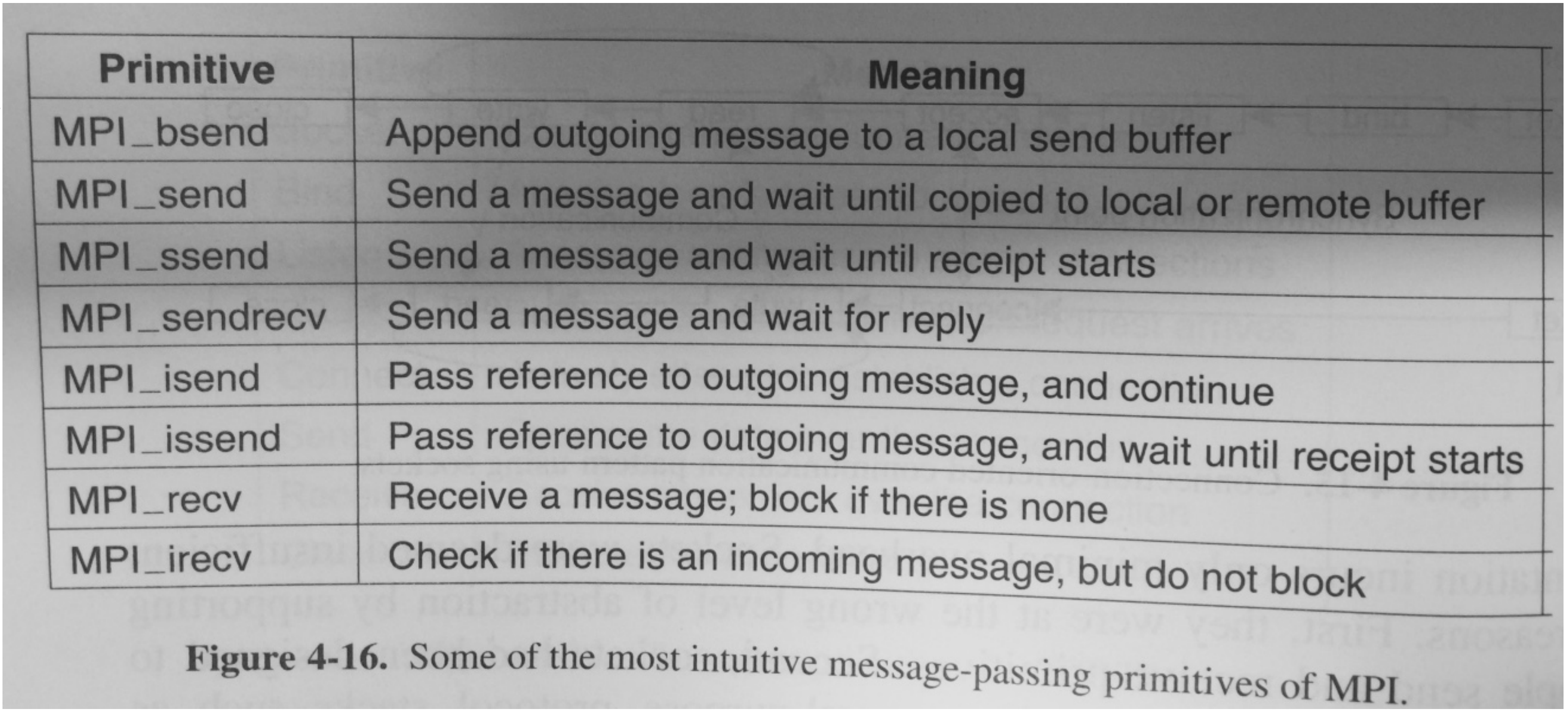
Blocking synchronous send

Begins a nonblocking send

Starts a nonblocking synchronous send

Blocking receive for a message

Begins a nonblocking receive



Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isend	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

Figure 4-16. Some of the most intuitive message-passing primitives of MPI.

Message-oriented middleware

Essence

Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers.

PUT	Append a message to a specified queue
GET	Block until the specified queue is nonempty, and remove the first message
POLL	Check a specified queue for messages, and remove the first. Never block
NOTIFY	Install a handler to be called when a message is put into the specified queue

Message broker

Observation

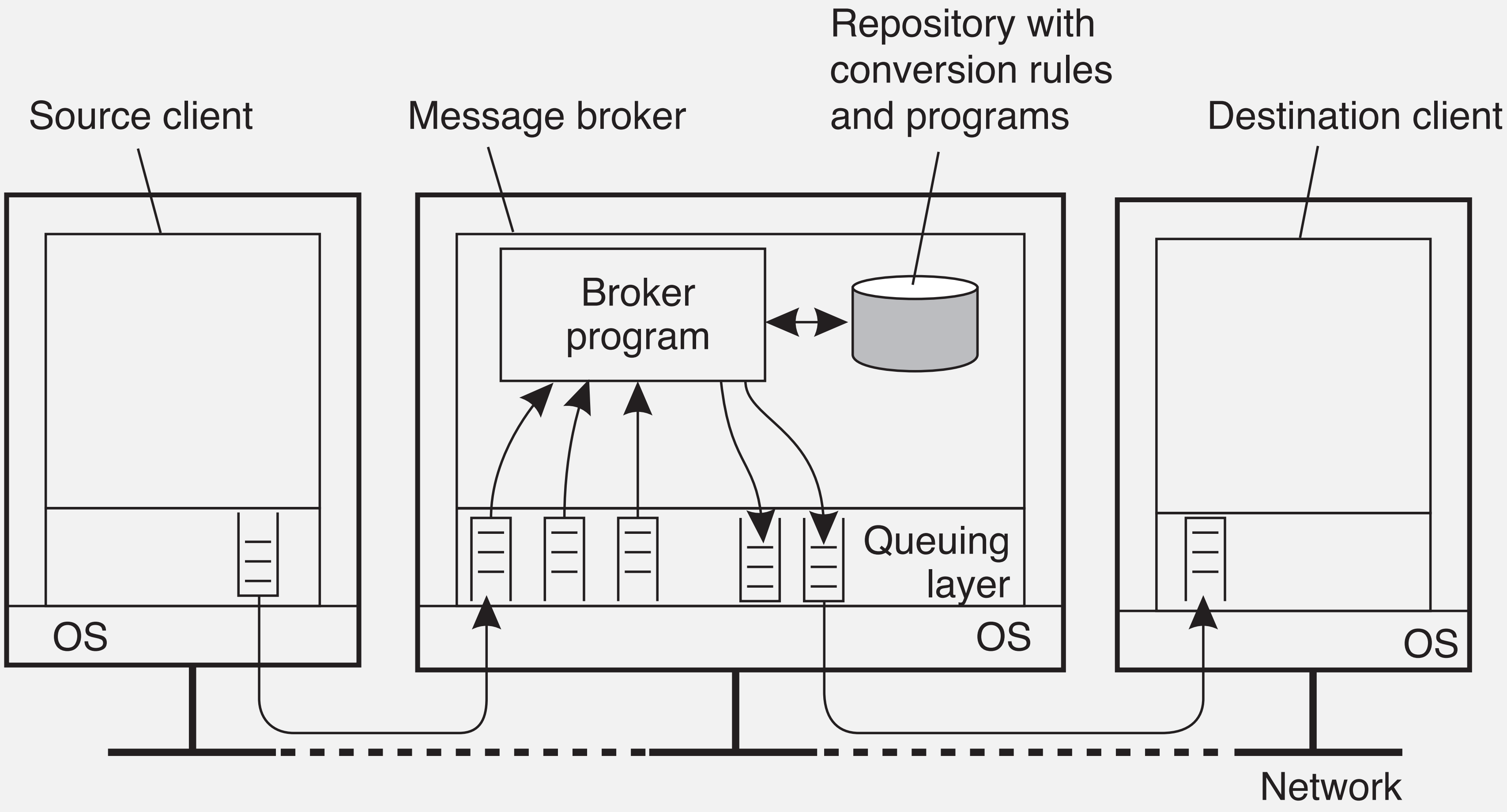
Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)

Message broker

Centralized component that takes care of application heterogeneity in an MQ system:

- Transforms incoming messages to target format
- Very often acts as an application gateway
- May provide subject-based routing capabilities ⇒ Enterprise Application Integration

Message broker



Stream-oriented communication

- Support for continuous media
- Streams in distributed systems
- Stream management

Continuous media

Observation

All communication facilities discussed so far are essentially based on a discrete, that is time-independent exchange of information

Continuous media

Characterized by the fact that values are time dependent:

- Audio
- Video
- Animations
- Sensor data (temperature, pressure, etc.)

Continuous media

Transmission modes

Different timing guarantees with respect to data transfer:

- Asynchronous: no restrictions with respect to **when** data is to be delivered
- Synchronous: define a maximum end-to-end delay for individual data packets
- Isochronous: define a maximum and minimum end-to-end delay (jitter is bounded)

Stream

Definition

A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission.

Some common stream characteristics

- Streams are unidirectional
- There is generally a single source, and one or more sinks
- Often, either the sink and/or source is a wrapper around hardware (e.g., camera, CD device, TV monitor)
- Simple stream: a single flow of data, e.g., audio or video
- Complex stream: multiple data flows, e.g., stereo audio or combination audio/video

Streams and QoS

Essence

Streams are all about timely delivery of data. How do you specify this Quality of Service (QoS)? Basics:

- The required bit rate at which data should be transported.
- The maximum delay until a session has been set up (i.e., when an application can start sending data).
- The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
- The maximum delay variance, or jitter.
- The maximum round-trip delay.

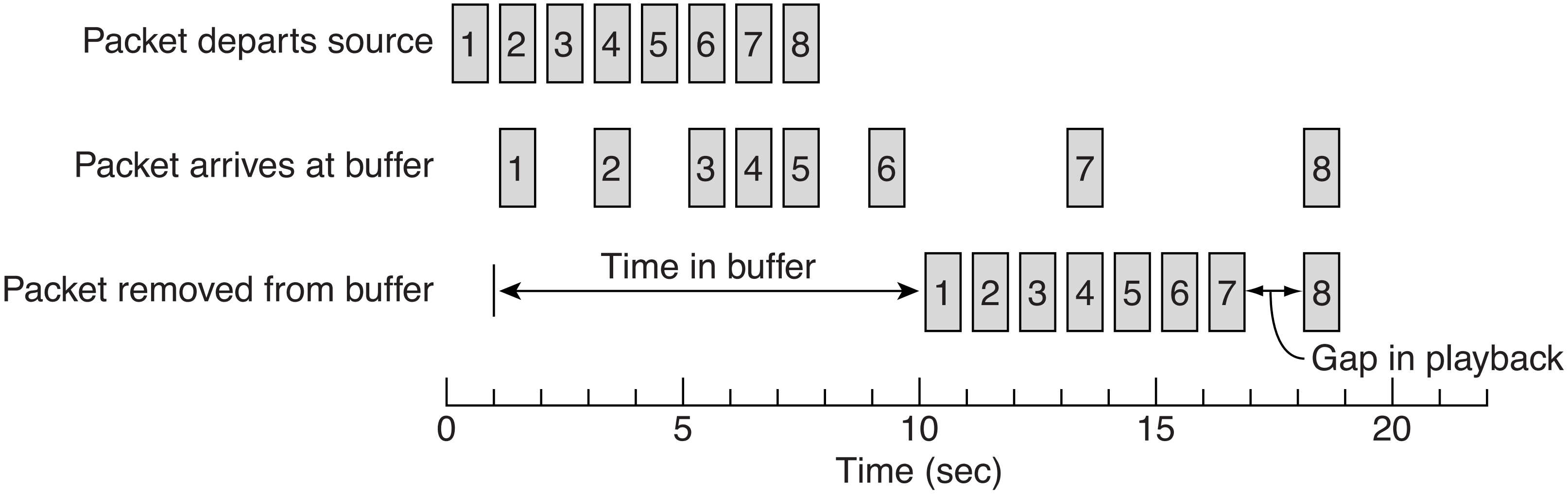
Enforcing QoS

Observation

There are various network-level tools, such as differentiated services by which certain packets can be prioritized.

Also

Use buffers to reduce jitter:

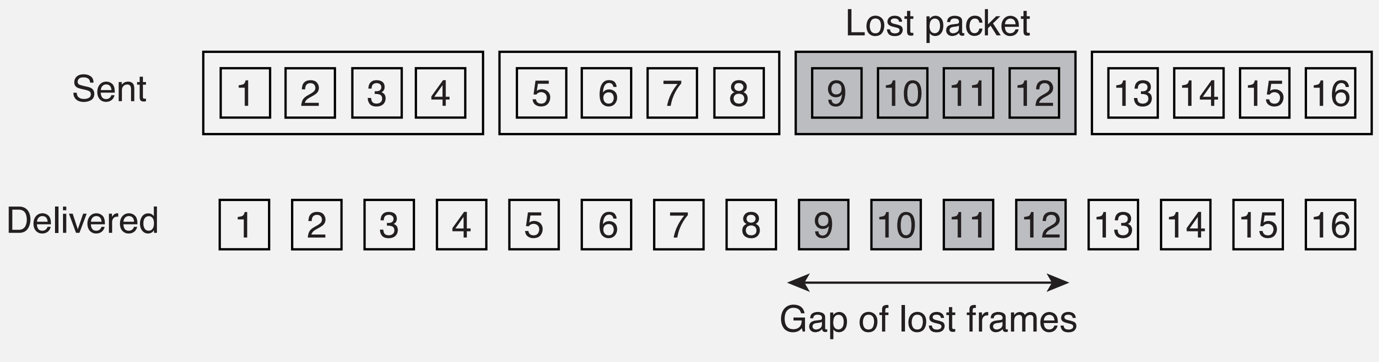


Enforcing QoS

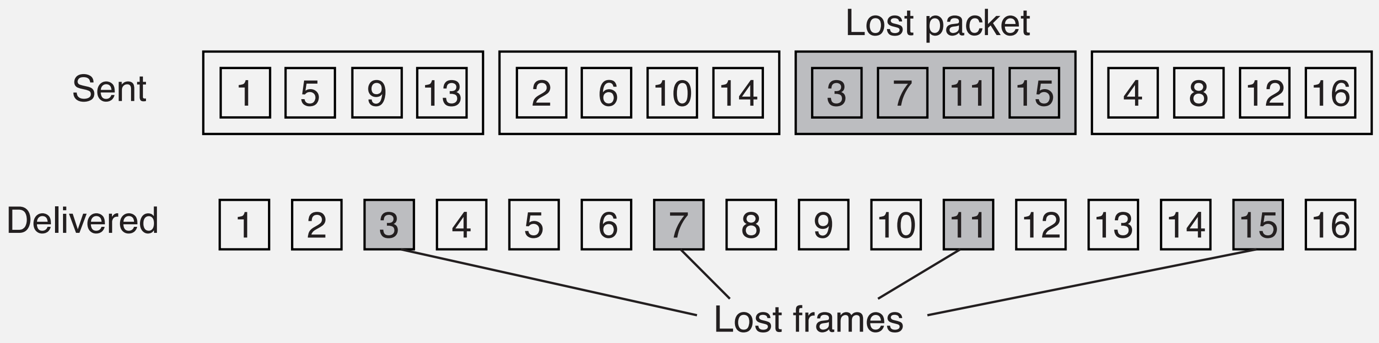
Problem

How to reduce the effects of packet loss (when multiple samples are in a single packet)?

Enforcing QoS



(a)



(b)

Stream synchronization

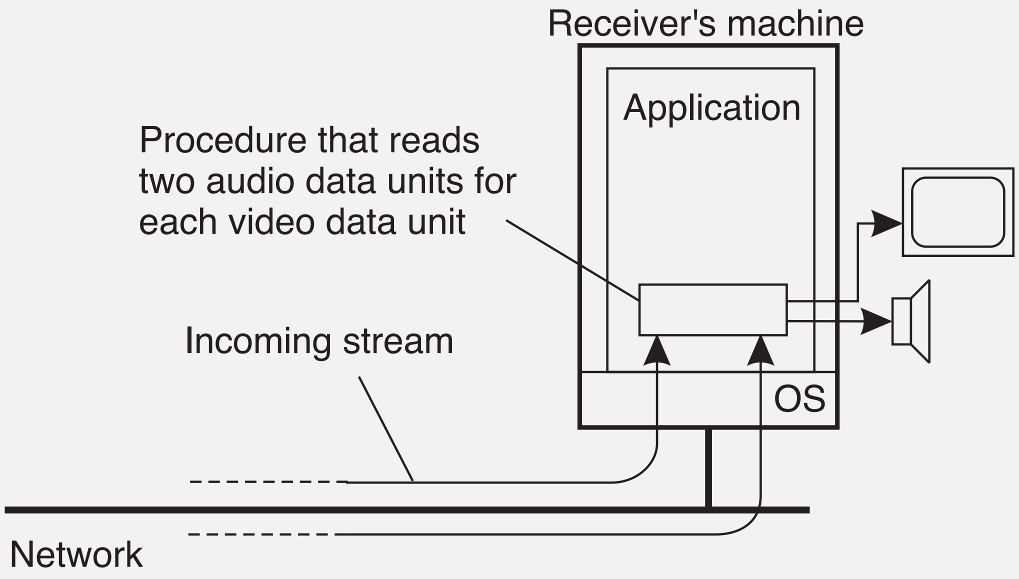
Problem

Given a complex stream, how do you keep the different substreams in synch?

Example

Think of playing out two channels, that together form stereo sound. Difference should be less than 20–30 μsec!

Stream synchronization


Receiver's machine
Application
Procedure that reads two audio data units for each video data unit
Incoming stream
OS
Network

Alternative

Multiplex all substreams into a single stream, and demultiplex at the receiver. Synchronization is handled at multiplexing/demultiplexing point (MPEG).

CHAPTER 5: NAMING

Names, Flat Names, Structured, Attribute Based

Naming

Essence

Names are used to denote entities in a distributed system. To operate on an entity, we need to access it at an access point. Access points are entities that are named by means of an address.

Note

A location-independent name for an entity E, is independent from the addresses of the access points offered by E.

Identifiers

Pure name

A name that has no meaning at all; it is just a random string. Pure names can be used for comparison only.

Identifier

A name having the following properties:

- P1: Each identifier refers to at most one entity
- P2: Each entity is referred to by at most one identifier
- P3: An identifier always refers to the same entity (prohibits reusing an identifier)

Observation

An identifier need not necessarily be a pure name, i.e., it may have content.

Flat naming

Problem

Given an essentially unstructured name (e.g., an identifier), how can we locate its associated access point?

- Simple solutions (broadcasting)
- Home-based approaches
- Distributed Hash Tables (structured P2P)
- Hierarchical location service

Simple solutions

1 **Broadcasting**

Broadcast the ID, requesting the entity to return its current address.

- Can never scale beyond local-area networks
- Requires all processes to listen to incoming location requests

2 **Forwarding pointers**

When an entity moves, it leaves behind a pointer to its next location

- Dereferencing can be made entirely transparent to clients by simply following the chain of pointers
- Update a client's reference when present location is found
- Geographical scalability problems (for which separate chain reduction mechanisms are needed):
  - Long chains are not fault tolerant
  - Increased network latency at dereferencing

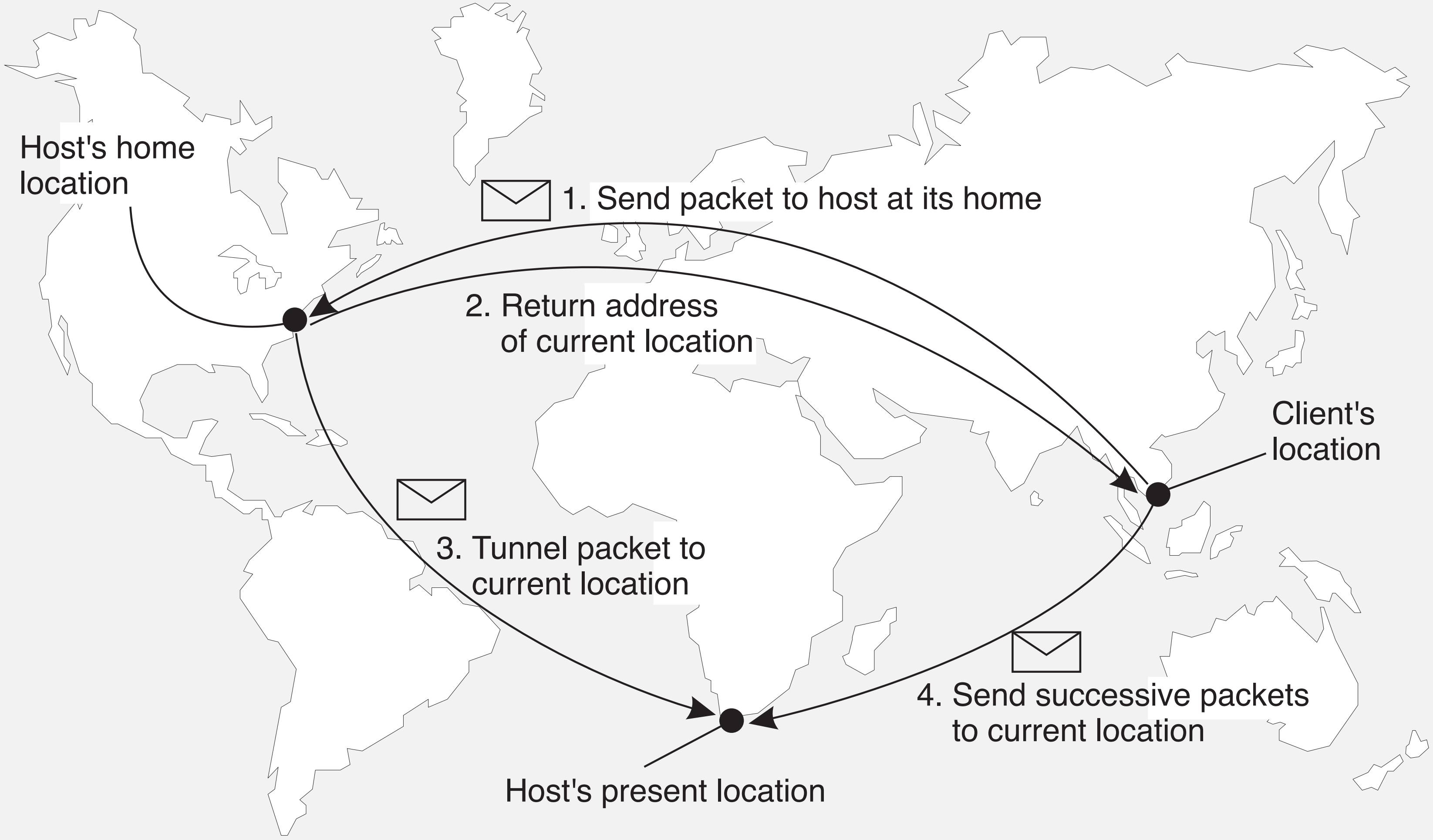
3 Home-based approaches

Single-tiered scheme

Let a home keep track of where the entity is:

- Entity's home address registered at a naming service
- The home registers the foreign address of the entity
- Client contacts the home first, and then continues with foreign location

Home-based approaches



Host's home location

1. Send packet to host at its home

2. Return address of current location

Client's location

3. Tunnel packet to current location

4. Send successive packets to current location

Host's present location

Home-based approaches

Two-tiered scheme

Keep track of visiting entities:

- Check local visitor register first
- Fall back to home location if local lookup fails

Problems with home-based approaches

- Home address has to be supported for entity's lifetime
- Home address is fixed ⇒ unnecessary burden when the entity permanently moves
- Poor geographical scalability (entity may be next to client)

Question

How can we solve the "permanent move" problem?

Distributed Hash Tables (DHT)

Chord

Consider the organization of many nodes into a logical ring

- Each node is assigned a random m-bit identifier.
- Every entity is assigned a unique m-bit key.
- Entity with key k falls under jurisdiction of node with smallest $id \geq k$ (called its successor).

Nonsolution

Let node id keep track of $succ(id)$ and start linear search along the ring.

DHTs: Finger tables

Principle

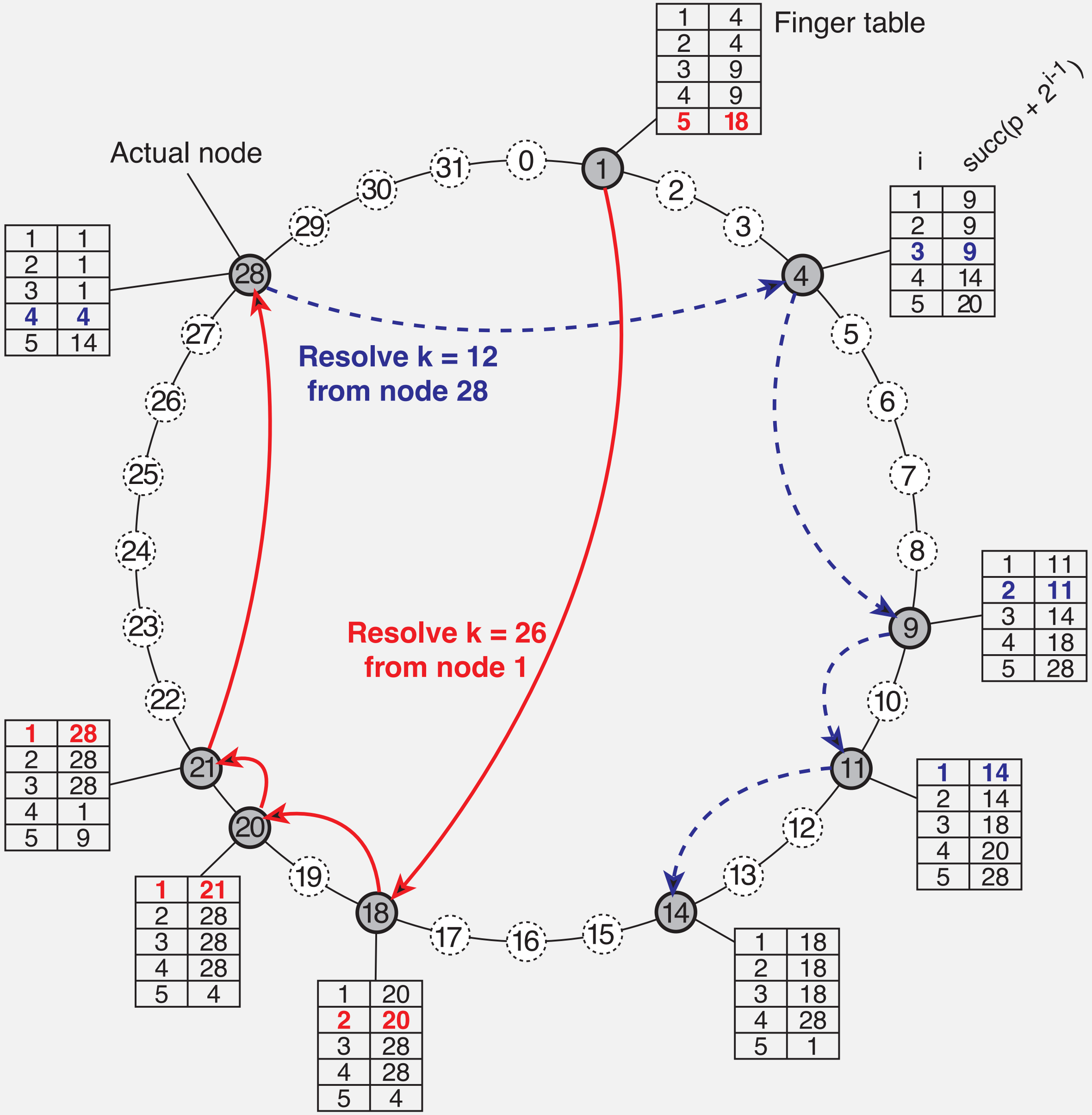
- Each node p maintains a finger table $FT_p[]$ with at most m entries:

  $$FT_p[i] = succ(p + 2^{i-1})$$

  Note: $FT_p[i]$ points to the first node succeeding p by at least 2^{i-1}.
- To look up a key k, node p forwards the request to node with index j satisfying

  $$q = FT_p[j] \leq k < FT_p[j+1]$$
- If $p < k < FT_p[1]$, the request is also forwarded to $FT_p[1]$

DHTs: Finger tables



Resolve k = 12 from node 28

Resolve k = 26 from node 1

1	4
2	4
3	9
4	9
5	**18**

i	succ(p + 2^{i-1})
1	9
2	9
3	**9**
4	14
5	20

1	11
2	**11**
3	14
4	18
5	28

1	**14**
2	14
3	18
4	20
5	28

1	18
2	18
3	18
4	28
5	1

1	20
2	**20**
3	28
4	28
5	4

1	**21**
2	28
3	28
4	28
5	4

1	**28**
2	28
3	28
4	1
5	9

1	1
2	1
3	1
4	**4**
5	14

Exploiting network proximity

Problem

The logical organization of nodes in the overlay may lead to erratic message transfers in the underlying Internet: node k and node $succ(k+1)$ may be very far apart.

Topology-aware node assignment: When assigning an ID to a node, make sure that nodes close in the ID space are also close in the network. Can be very difficult.

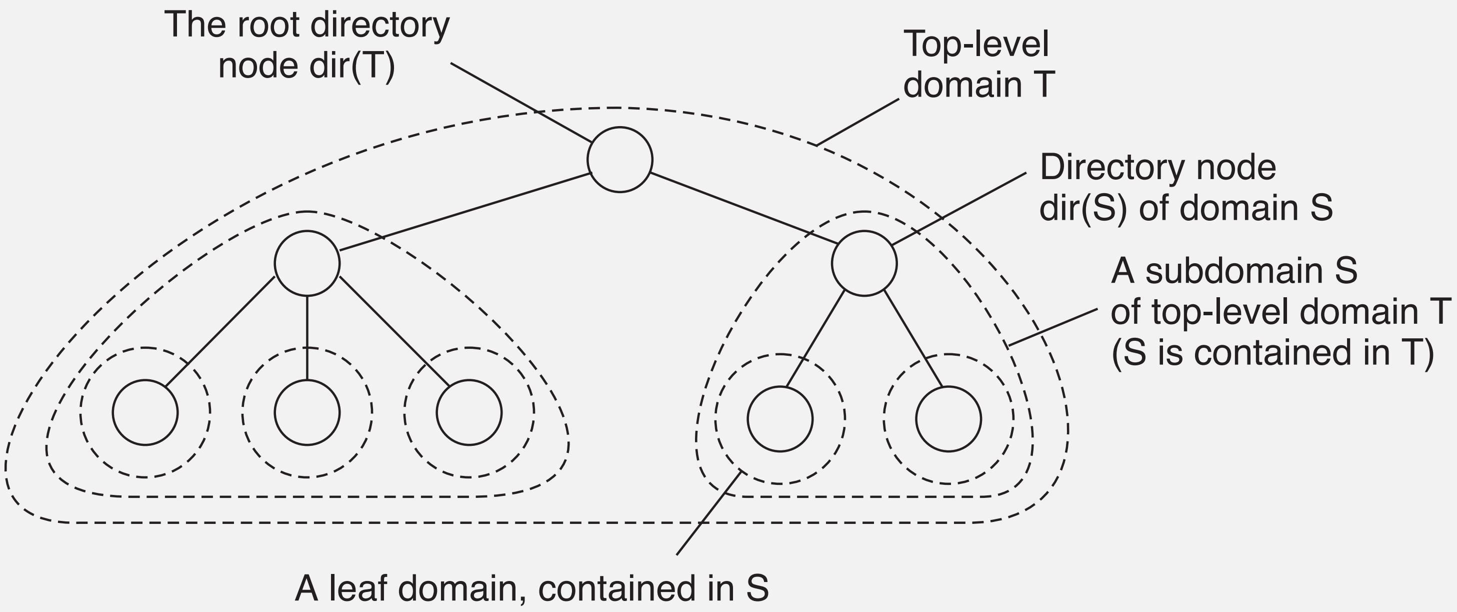
Proximity routing: Maintain more than one possible successor, and forward to the closest.
Example: in Chord $FT_p[i]$ points to first node in $INT = [p+2^{i-1}, p+2^i - 1]$. Node p can also store pointers to other nodes in INT.

Proximity neighbor selection: When there is a choice of selecting who your neighbor will be (not in Chord), pick the closest one.

Hierarchical Location Services (HLS)

Basic idea

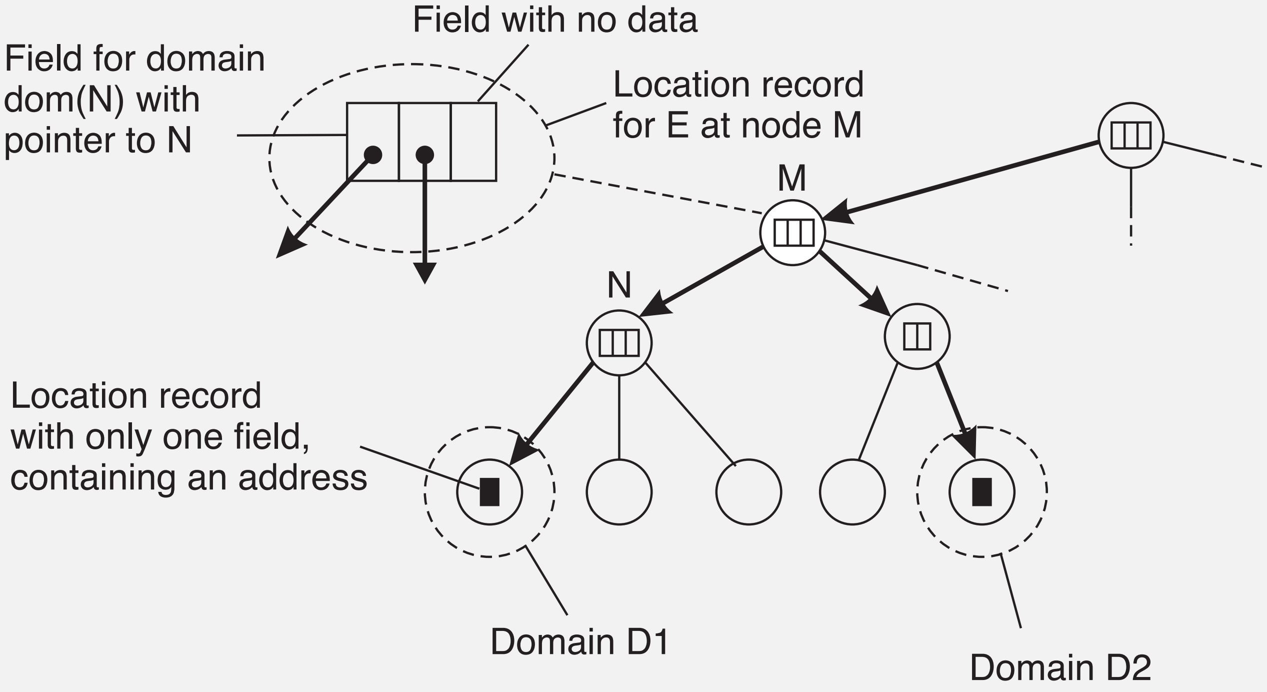
Build a large-scale search tree for which the underlying network is divided into hierarchical domains. Each domain is represented by a separate directory node.


The root directory node dir(T)
Top-level domain T
Directory node dir(S) of domain S
A subdomain S of top-level domain T (S is contained in T)
A leaf domain, contained in S

HLS: Tree organization

Invariants

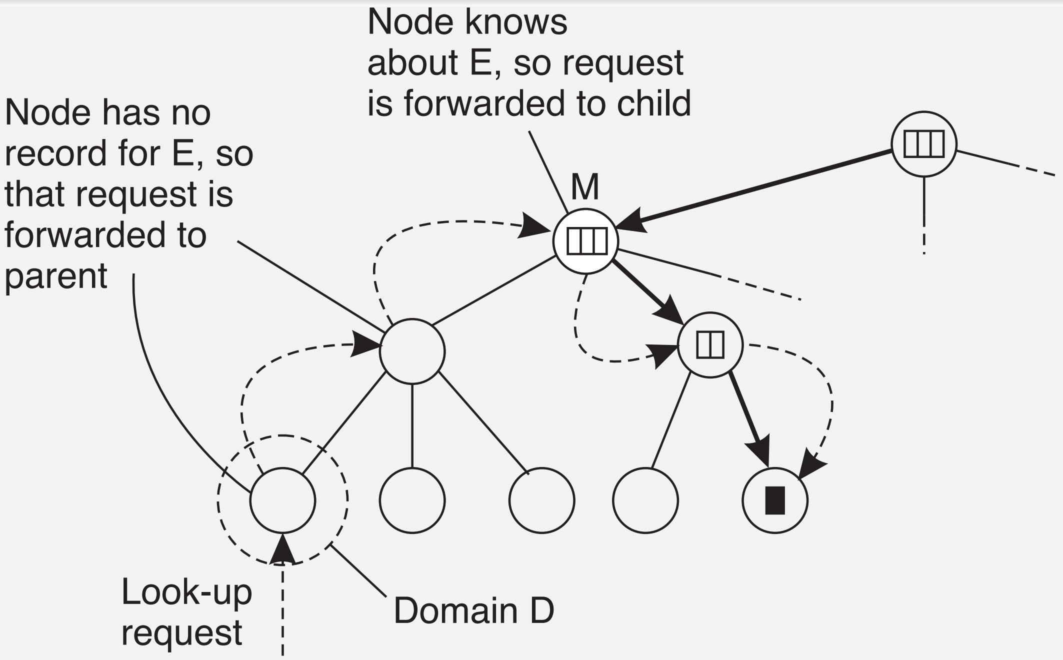
- Address of entity *E* is stored in a leaf or intermediate node
- Intermediate nodes contain a pointer to a child iff the subtree rooted at the child stores an address of the entity
- The root knows about all entities


Field with no data
Field for domain dom(N) with pointer to N
Location record for E at node M
M
N
Location record with only one field, containing an address
Domain D1
Domain D2

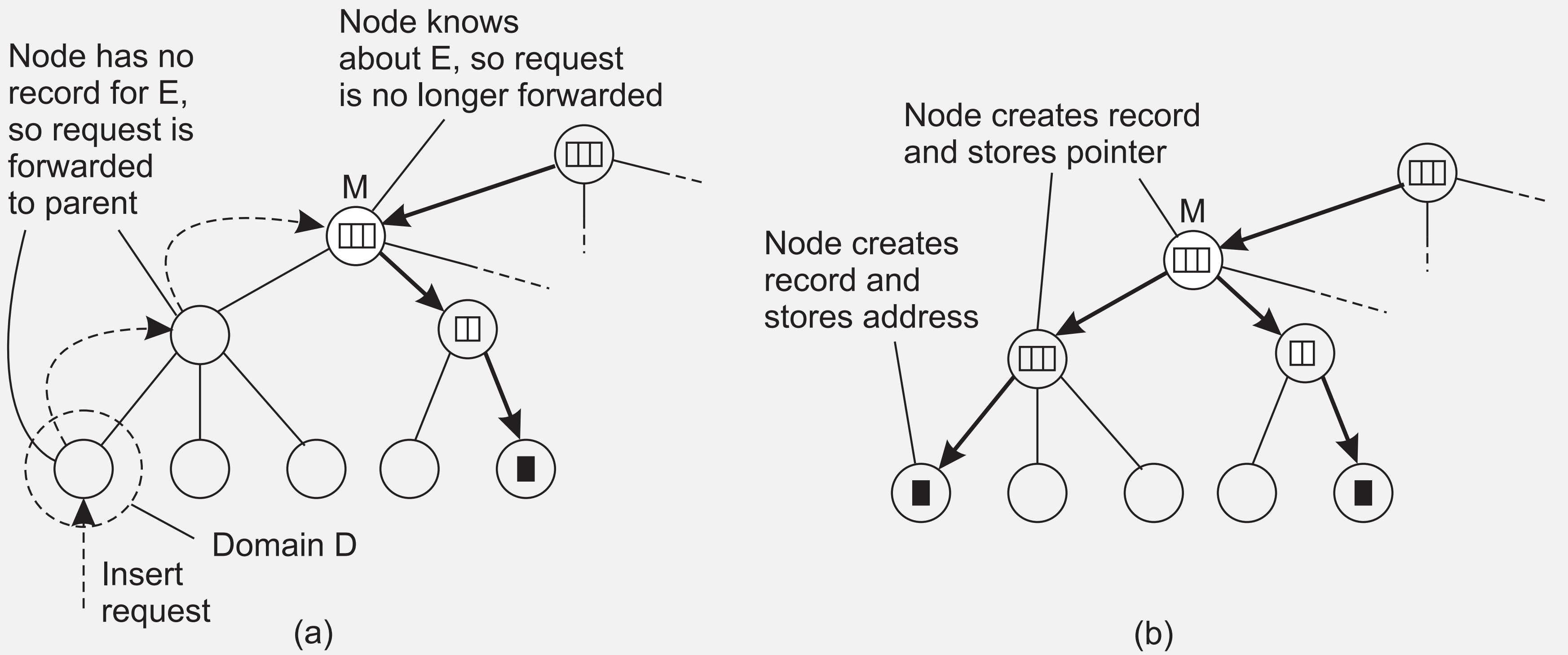
HLS: Lookup operation

Basic principles

- Start lookup at local leaf node
- Node knows about E $\Rightarrow$ follow downward pointer, else go up
- Upward lookup always stops at root



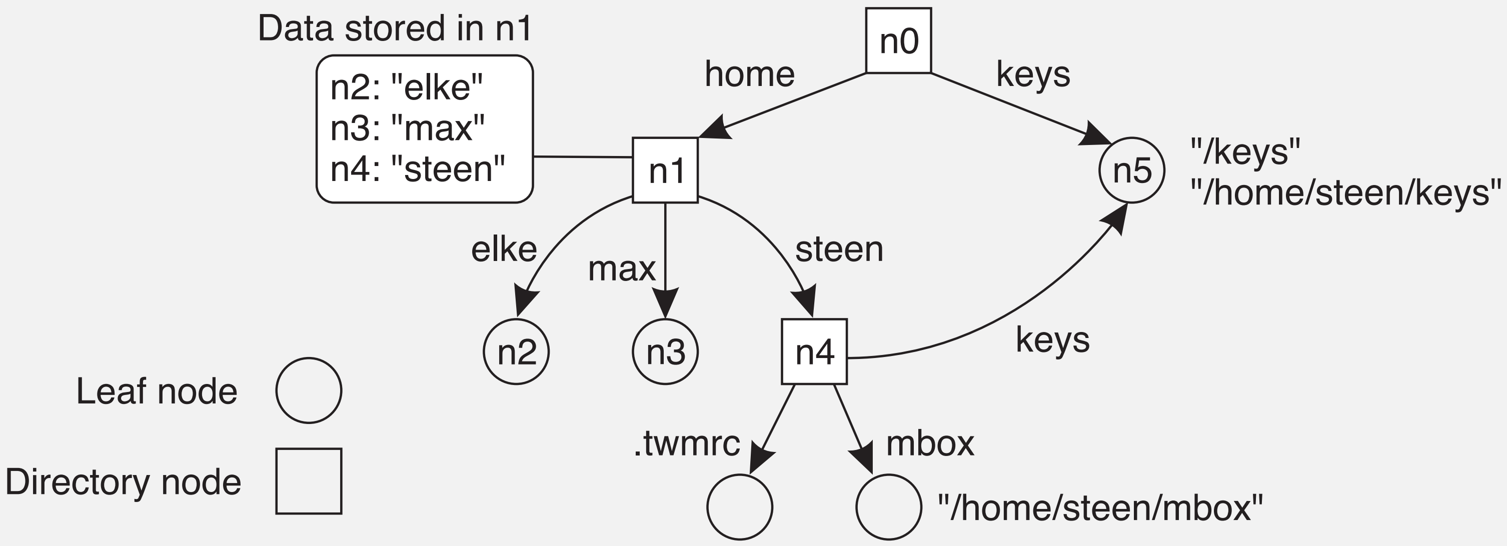
HLS: Insert operation


Node has no record for E, so request is forwarded to parent
Node knows about E, so request is no longer forwarded
M
Domain D
Insert request
(a)
Node creates record and stores pointer
Node creates record and stores address
M
(b)

Name space

Essence

A graph in which a leaf node represents a (named) entity. A directory node is an entity that refers to other nodes.



Note

A directory node contains a (directory) table of *(edge label, node identifier)* pairs.

Name space

Observation

We can easily store all kinds of attributes in a node, describing aspects of the entity the node represents:

- Type of the entity
- An identifier for that entity
- Address of the entity’s location
- Nicknames
- ...

Note

Directory nodes can also have attributes, besides just storing a directory table with *(edge label, node identifier)* pairs.

Name resolution

Problem

To resolve a name we need a directory node. How do we actually find that (initial) node?

Closure mechanism

The mechanism to select the implicit context from which to start name resolution:

- *www.cs.vu.nl*: start at a DNS name server
- */home/steen/mbox*: start at the local NFS file server (possible recursive search)
- *0031204447784*: dial a phone number
- *130.37.24.8*: route to the VU's Web server

Question

Why are closure mechanisms always implicit?

Name-space implementation

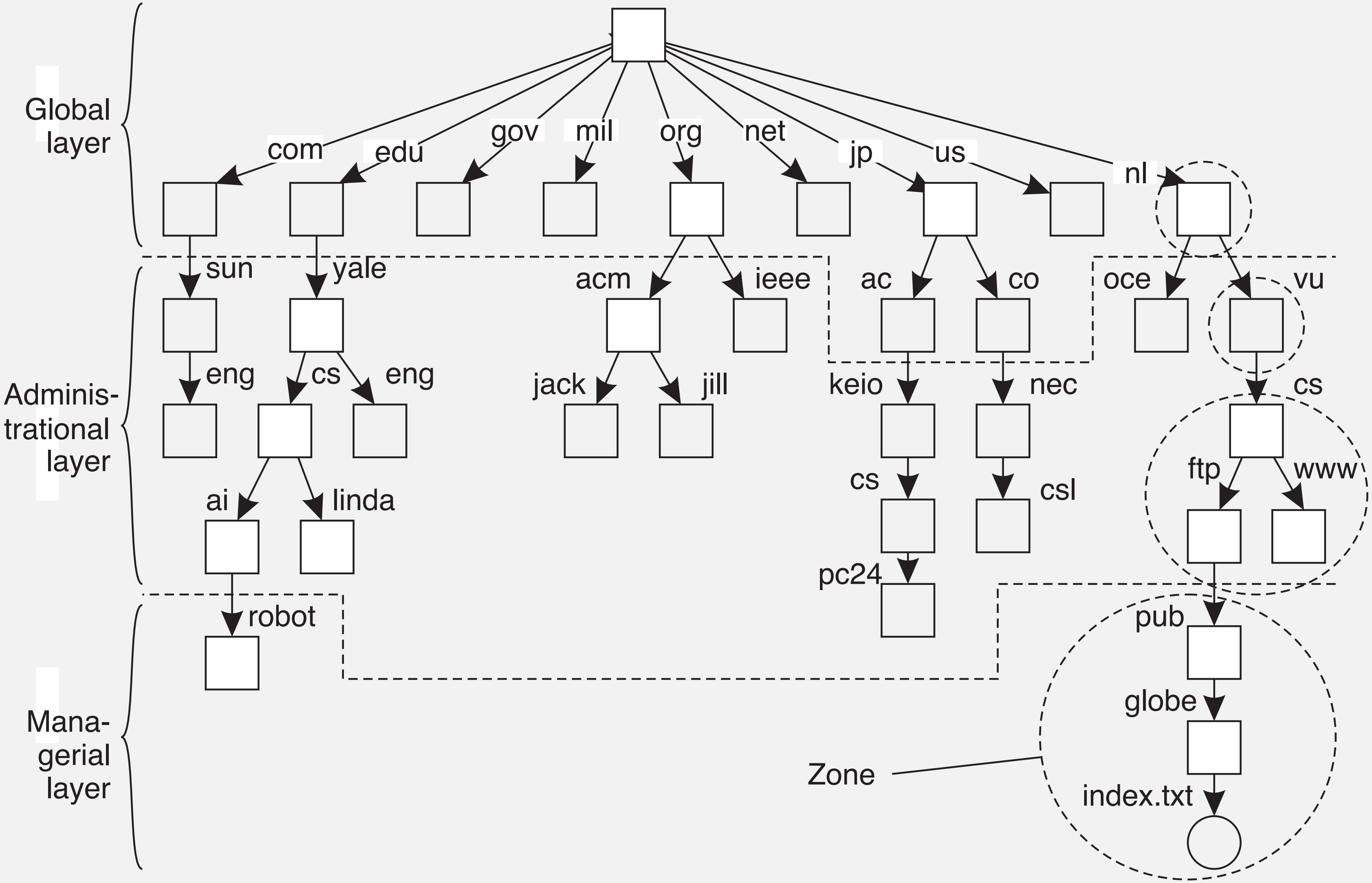
Basic issue

Distribute the name resolution process as well as name space management across multiple machines, by distributing nodes of the naming graph.

Distinguish three levels

- Global level: Consists of the high-level directory nodes. Main aspect is that these directory nodes have to be jointly managed by different administrations
- Administrational level: Contains mid-level directory nodes that can be grouped in such a way that each group can be assigned to a separate administration.
- Managerial level: Consists of low-level directory nodes within a single administration. Main issue is effectively mapping directory nodes to local name servers.

Name-space implementation



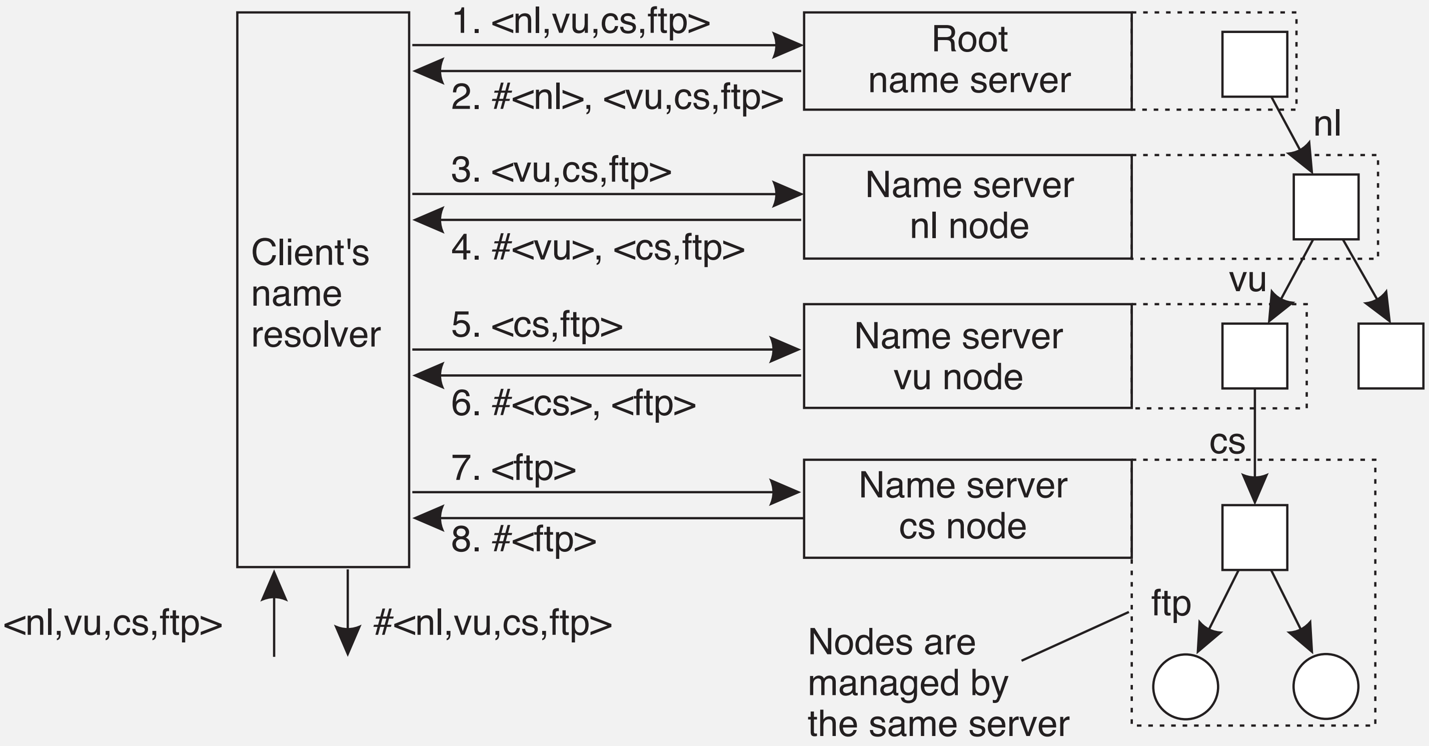
Name-space implementation

Item	Global	Administrational	Managerial
1	Worldwide	Organization	Department
2	Few	Many	Vast numbers
3	Seconds	Milliseconds	Immediate
4	Lazy	Immediate	Immediate
5	Many	None or few	None
6	Yes	Yes	Sometimes

1: Geographical scale	4: Update propagation
2: # Nodes	5: # Replicas
3: Responsiveness	6: Client-side caching?

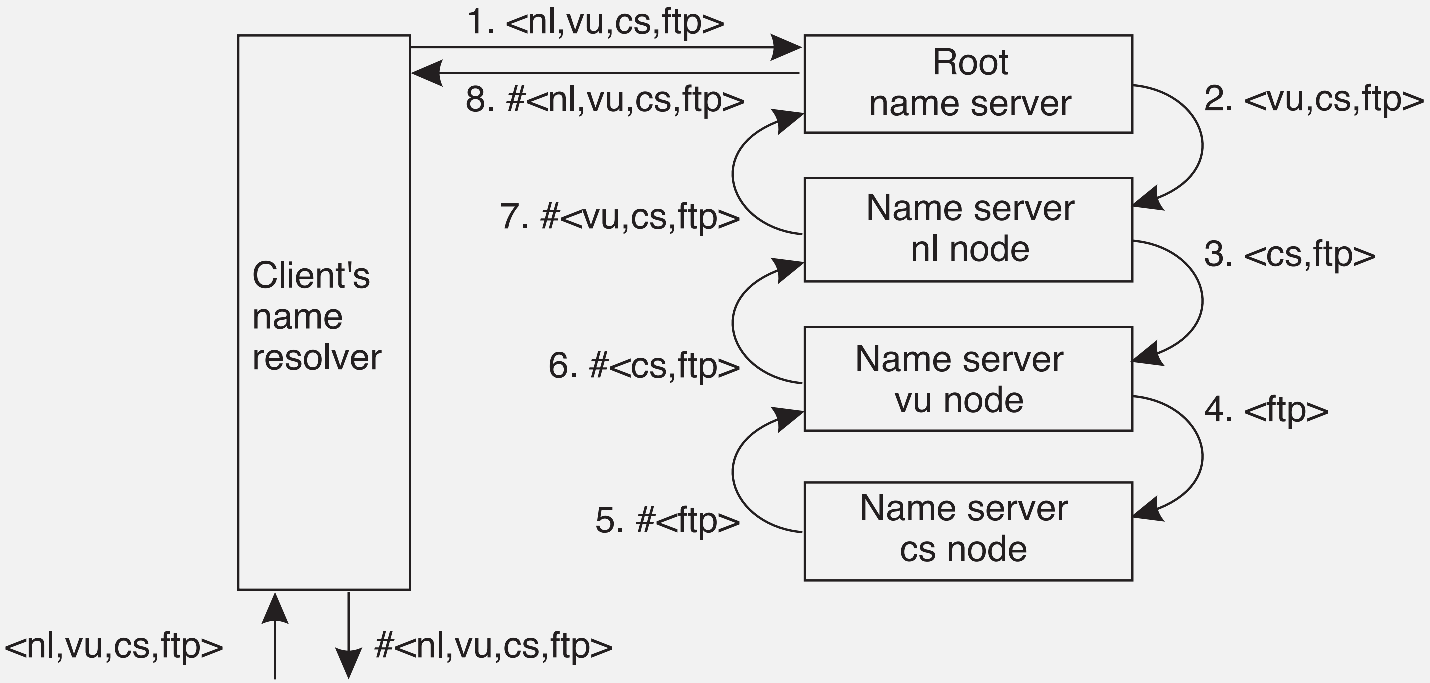
Iterative name resolution

1. *resolve(dir,[name1,...,nameK])* sent to *Server0* responsible for *dir*
2. *Server0* resolves *resolve(dir,name1)* → *dir1*, returning the identification (address) of *Server1*, which stores *dir1*.
3. Client sends *resolve(dir1,[name2,...,nameK])* to *Server1*, etc.



Recursive name resolution

1. *resolve(dir,[name1,...,nameK])* sent to *Server0* responsible for *dir*
2. *Server0* resolves *resolve(dir,name1)* → *dir1*, and sends *resolve(dir1,[name2,...,nameK])* to *Server1*, which stores *dir1*.
3. *Server0* waits for result from *Server1*, and returns it to client.



Caching in recursive name resolution

Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	<ftp>	#<ftp>	—	—	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

Scalability issues

Size scalability

We need to ensure that servers can handle a large number of requests per time unit ⇒ high-level servers are in big trouble.

Solution

Assume (at least at global and administrational level) that content of nodes hardly ever changes. We can then apply extensive replication by mapping nodes to multiple servers, and start name resolution at the nearest server.

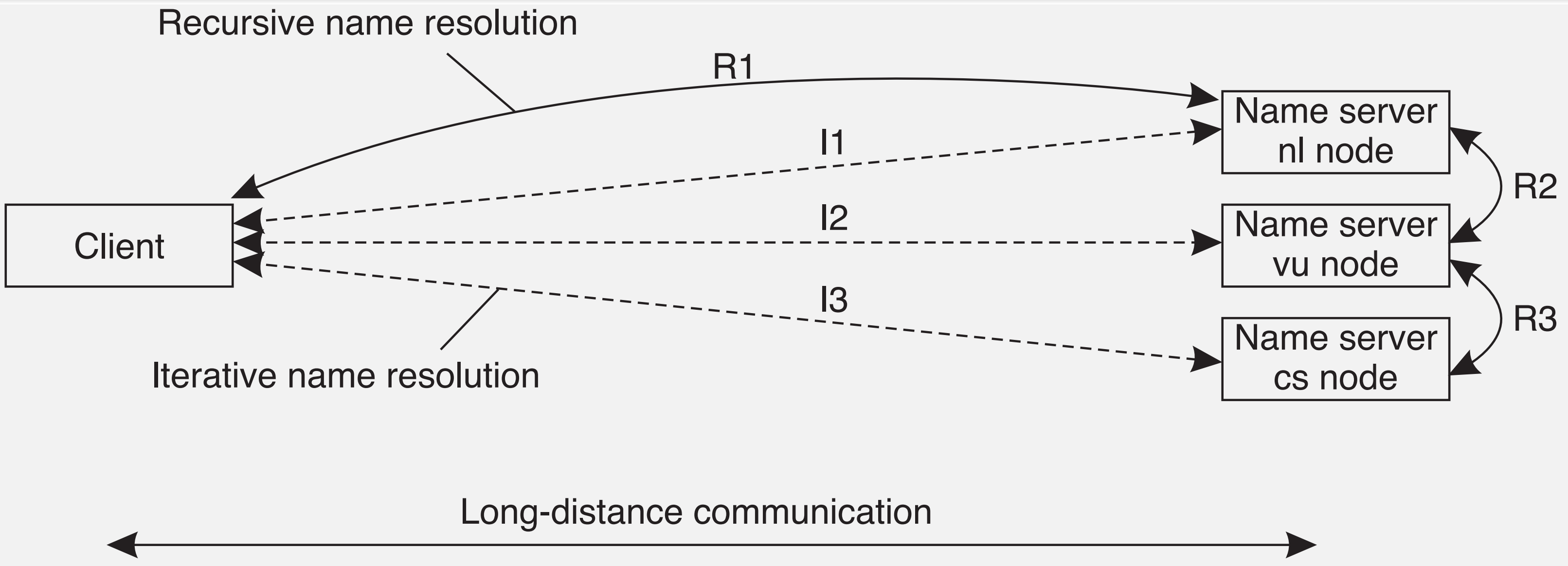
Observation

An important attribute of many nodes is the address where the represented entity can be contacted. Replicating nodes makes large-scale traditional name servers unsuitable for locating mobile entities.

Scalability issues

Geographical scalability

We need to ensure that the name resolution process scales across large geographical distances.


Recursive name resolution
R1
I1
I2
I3
Client
Name server nl node
Name server vu node
Name server cs node
R2
R3
Iterative name resolution
Long-distance communication

Problem

By mapping nodes to servers that can be located anywhere, we introduce an implicit location dependency.

CHAPTER 6: SYNCHRONISATION

Physical Clocks, Logical Clocks, Vector Clocks

Physical clocks

Problem

Sometimes we simply need the exact time, not just an ordering.

Solution

Universal Coordinated Time (UTC):

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium-clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

Note

UTC is broadcast through short wave radio and satellite. Satellites can give an accuracy of about ± 0.5 ms.

Physical clocks

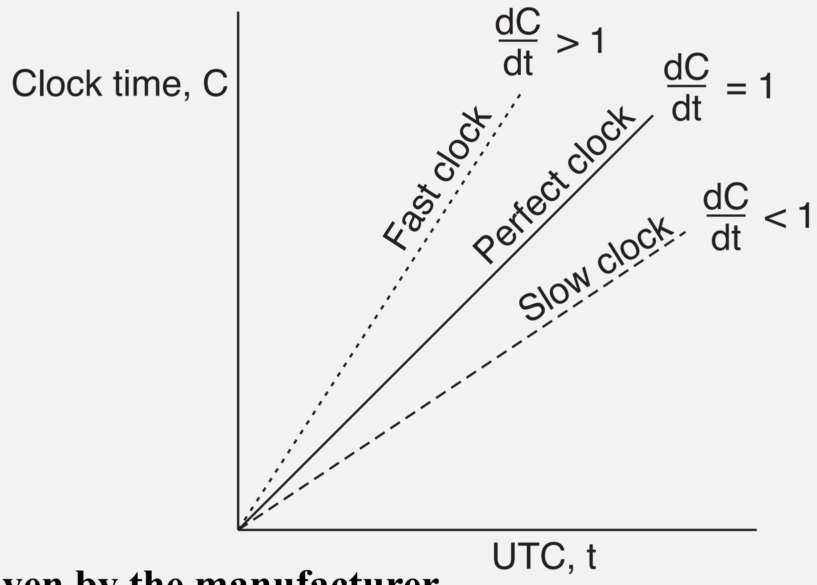
Problem

Suppose we have a distributed system with a UTC-receiver somewhere in it ⇒ we still have to distribute its time to each machine.

Basic principle

- Every machine has a timer that generates an interrupt H times per second.
- There is a clock in machine p that ticks on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.
- Ideally, we have that for each machine p, $C_p(t) = t$, or, in other words, $dC/dt = 1$.

Physical clocks



ρ is the maximum drift rate given by the manufacturer

In practice: $1-\rho \leq \frac{dC}{dt} \leq 1+\rho$.

Goal

Never let two clocks in any system differ by more than δ time units $\Rightarrow$ synchronize at least every $\delta/(2\rho)$ seconds.

Clock synchronization principles

Principle I

Every machine asks a time server for the accurate time at least once every $\delta/(2\rho)$ seconds (Network Time Protocol).

Note

Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.

Clock synchronization principles

Principle II

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Note

Okay, you'll probably get every machine in sync. You don't even need to propagate UTC time.

Fundamental

You'll have to take into account that setting the time back is never allowed ⇒ smooth adjustments.

The Happened-before relationship

Problem

We first need to introduce a notion of ordering before we can order anything.

The happened-before relation

- If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.
- If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Note

This introduces a partial ordering of events in a system with concurrently operating processes.

Logical clocks

Problem

How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Solution

Attach a timestamp $C(e)$ to each event e, satisfying the following properties:

P1 If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.

P2 If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Problem

How to attach a timestamp to an event when there's no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process.

Logical clocks

Solution

Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

1: For any two successive events that take place within P_i, C_i is incremented by 1.

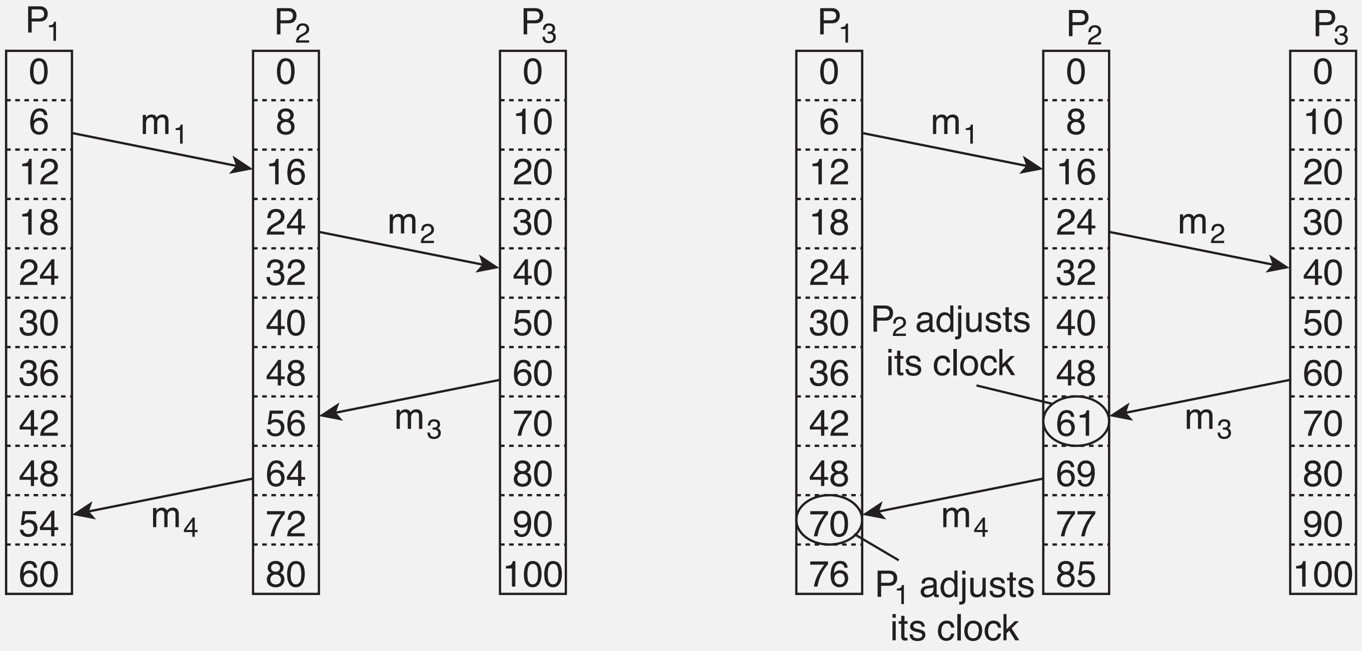
2: Each time a message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$.

3: Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

Logical clocks – example



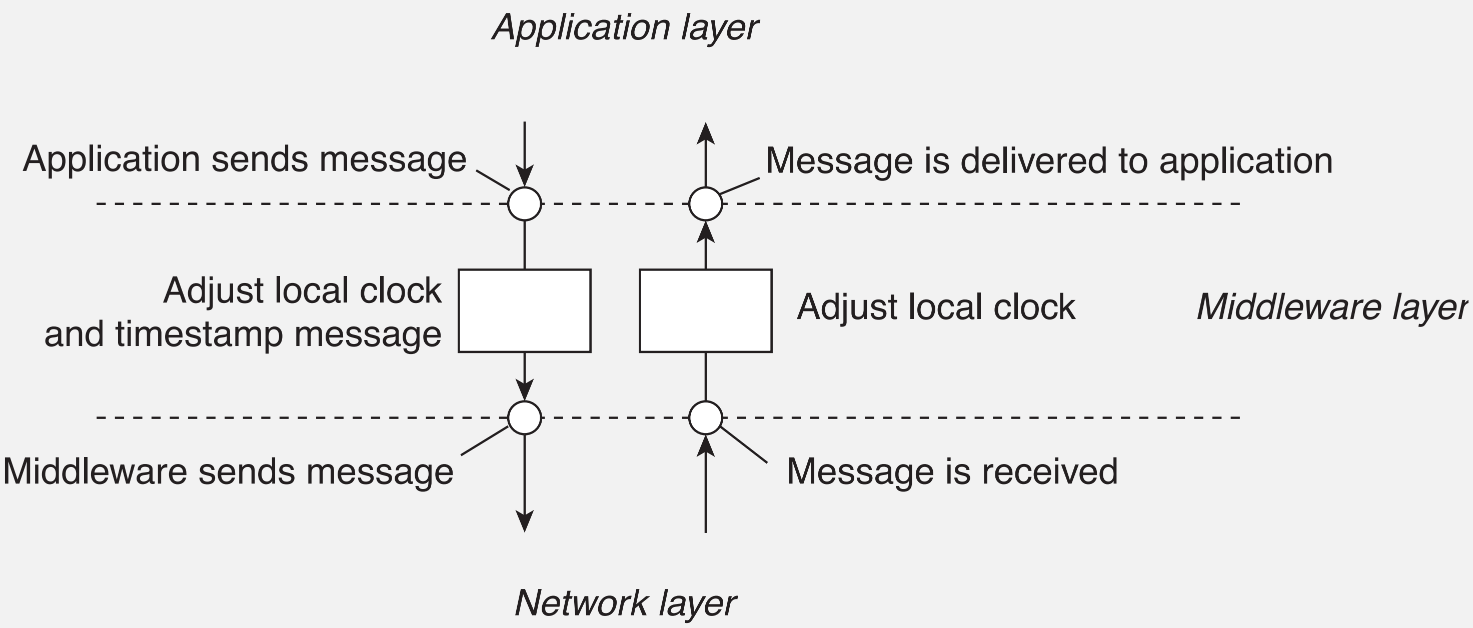
(a)

(b)

Logical clocks – example

Note

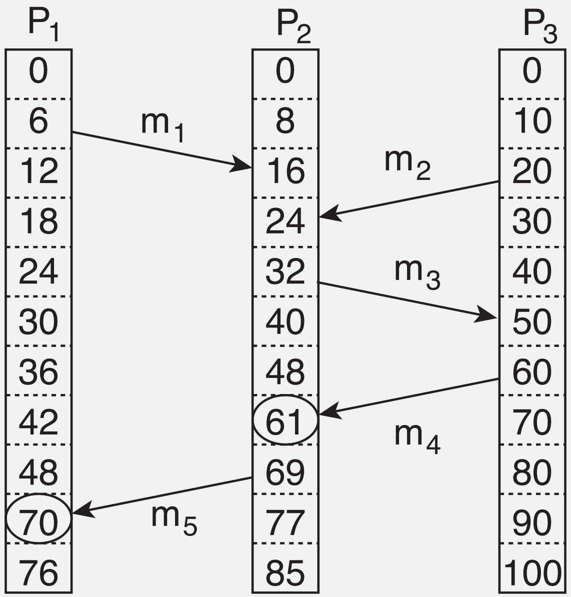
Adjustments take place in the middleware layer



Vector clocks

Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ that a causally preceded b



P_1	P_2	P_3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

Observation

Event a: m_1 is received at $T = 16$;
Event b: m_2 is sent at $T = 20$.

Note

We cannot conclude that a causally precedes b.

Vector clocks

Solution

- Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j.
- When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector timestamp $vt(m)$. Result: upon arrival, recipient knows P_i's timestamp.
- When a process P_j delivers a message m that it received from P_i with vector timestamp $ts(m)$, it

  (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$ for all k = 1 to n
  (2) increments $VC_j[j]$ by 1.

Question

What does $VC_i[j] = k$ mean in terms of messages sent and received?

Causally ordered multicasting

Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

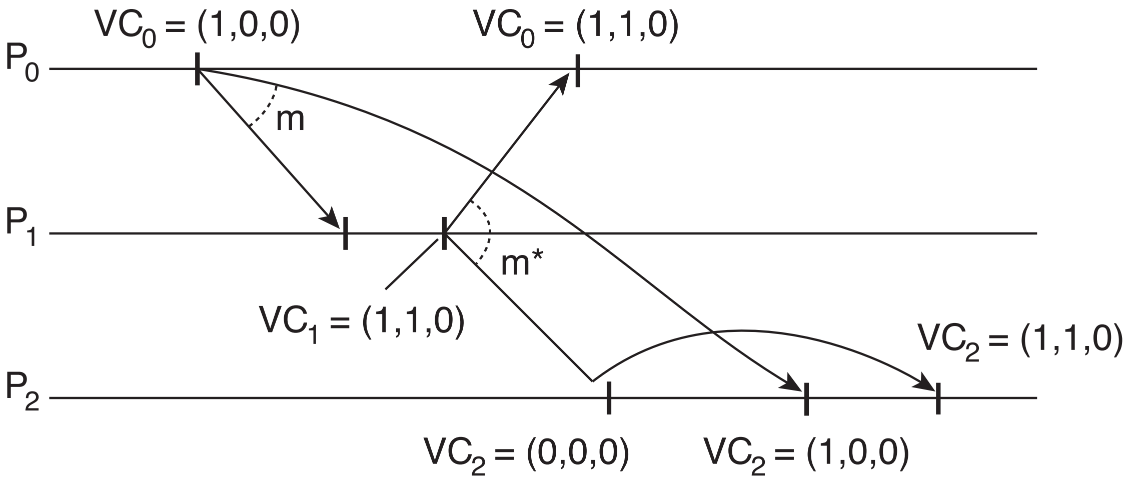
P_i increments $VC_i[i]$ only when sending a message, and P_j “adjusts” VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).

P_j postpones delivery of m until:

- $ts(m)[i] = VC_j[i] + 1$.
- $ts(m)[k] \leq VC_j[k]$ for $k \neq i$.

Causally ordered multicasting

Example



Example

Take $VC_2 = [0,2,2]$, $ts(m) = [1,3,0]$ from P_0. What information does P_2 have, and what will it do when receiving m (from P_0)?

CHAPTER 7: CONSISTENCY & REPLICATION

Week & Strict Consistency
Data Centric Consistency
Client Centric Consistency

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

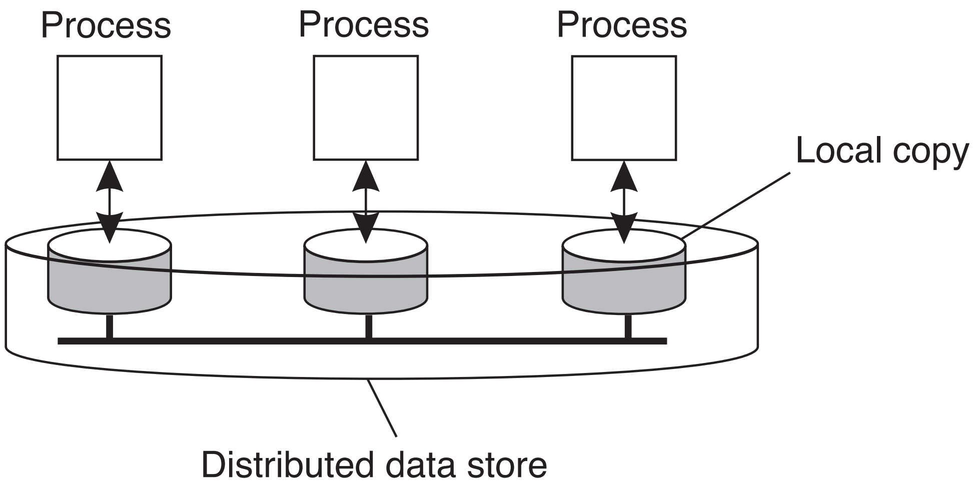
Data-centric consistency models

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:



Continuous Consistency

Observation

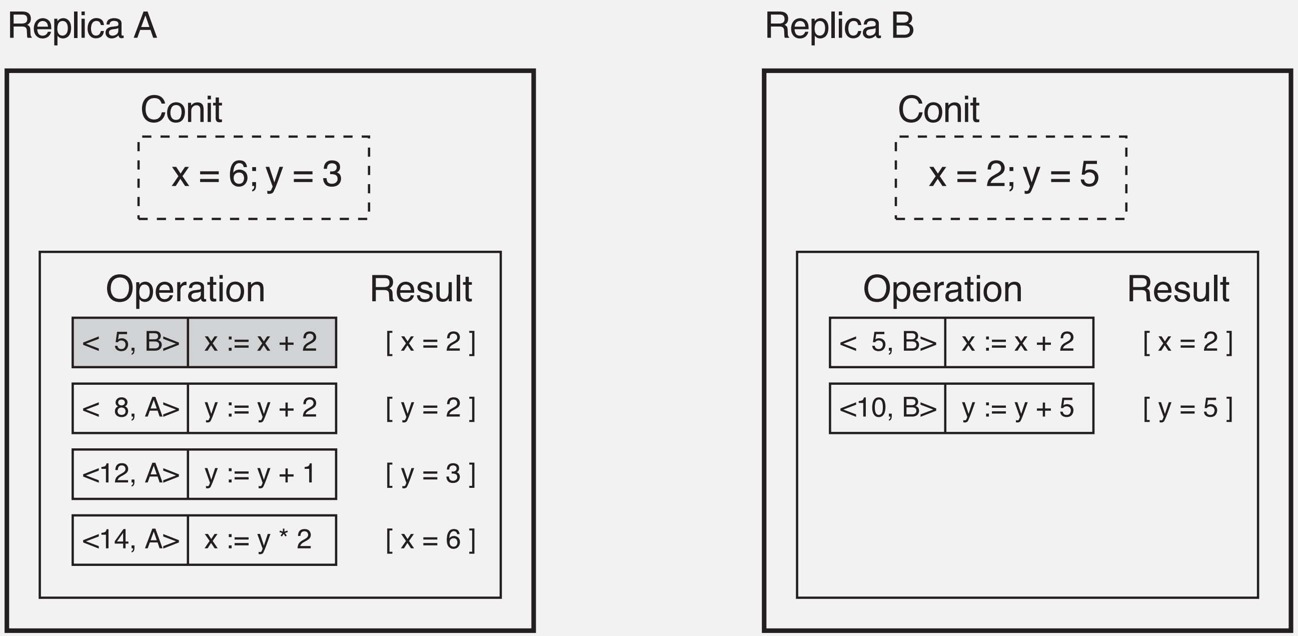
We can actually talk a about a degree of consistency:

- 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 ⇒ specifies the data unit over which consistency is to be measured.

Example: Conit



Replica A

Conit

x = 6; y = 3

Operation		Result
< 5, B>	x := x + 2	[x = 2]
< 8, A>	y := y + 2	[y = 2]
<12, A>	y := y + 1	[y = 3]
<14, A>	x := y * 2	[x = 6]

Replica B

Conit

x = 2; y = 5

Operation		Result
< 5, B>	x := x + 2	[x = 2]
<10, B>	y := y + 5	[y = 5]

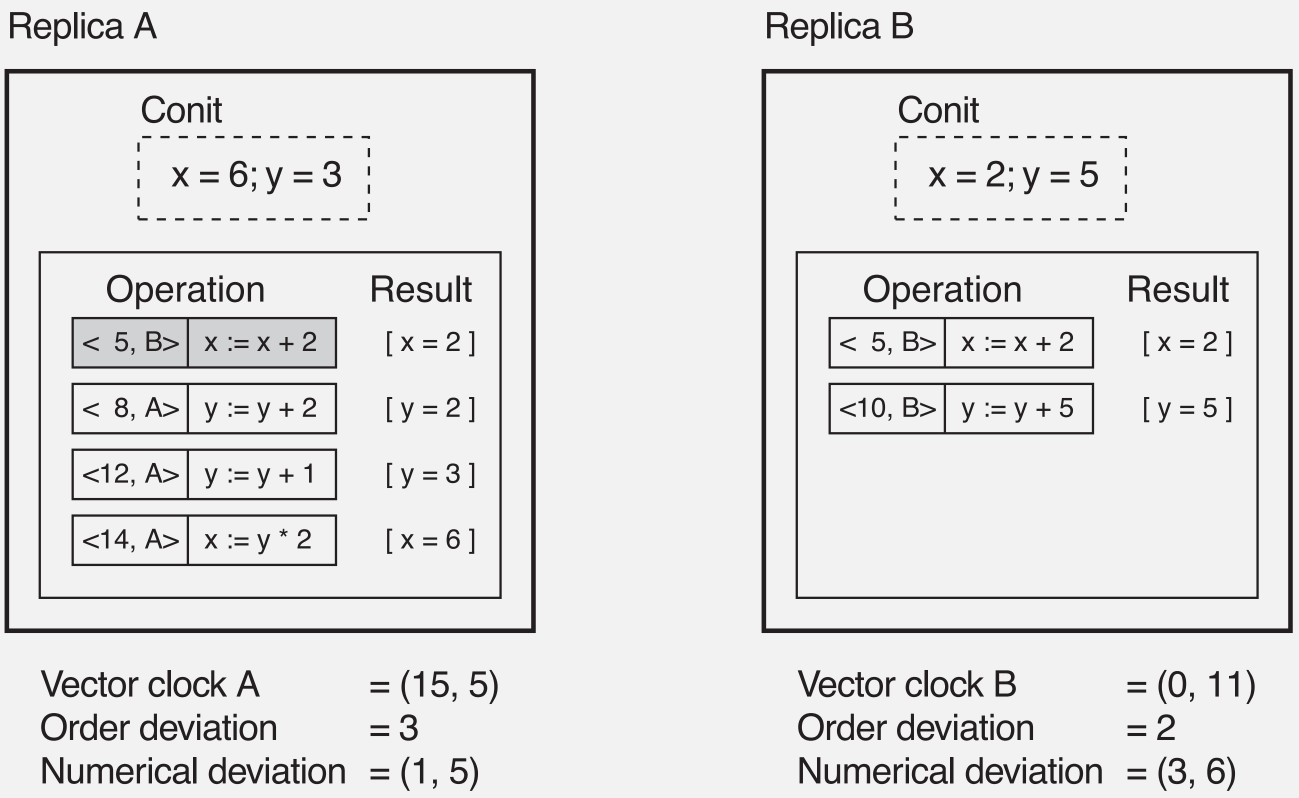
Vector clock A = (15, 5)
Order deviation = 3
Numerical deviation = (1, 5)

Vector clock B = (0, 11)
Order deviation = 2
Numerical deviation = (3, 6)

Conit (contains the variables x and y)

- Each replica has a vector clock: ([known] time @ A, [known] time @ B)
- B sends A operation $[\langle 5, B\rangle: x := x + 2]$; A has made this operation permanent (cannot be rolled back)

Example: Conit



Replica A

Conit

x = 6; y = 3

Operation		Result
< 5, B>	x := x + 2	[x = 2]
< 8, A>	y := y + 2	[y = 2]
<12, A>	y := y + 1	[y = 3]
<14, A>	x := y * 2	[x = 6]

Vector clock A = (15, 5)
Order deviation = 3
Numerical deviation = (1, 5)

Replica B

Conit

x = 2; y = 5

Operation		Result
< 5, B>	x := x + 2	[x = 2]
<10, B>	y := y + 5	[y = 5]

Vector clock B = (0, 11)
Order deviation = 2
Numerical deviation = (3, 6)

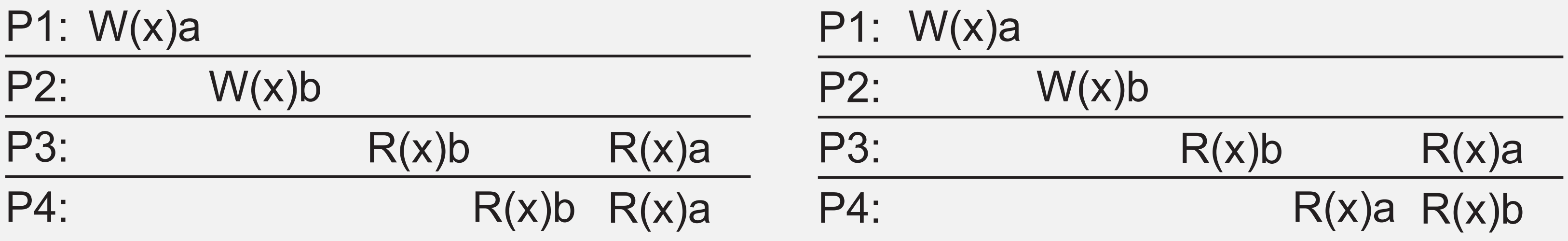
Conit (contains the variables *x* and *y*)

- *A* has three pending operations ⇒ order deviation = 3
- *A* has missed one operation from *B*, yielding a max diff of 5 units ⇒ $(1, 5)$

Sequential consistency

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)

Causal consistency

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				
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)

Grouping operations

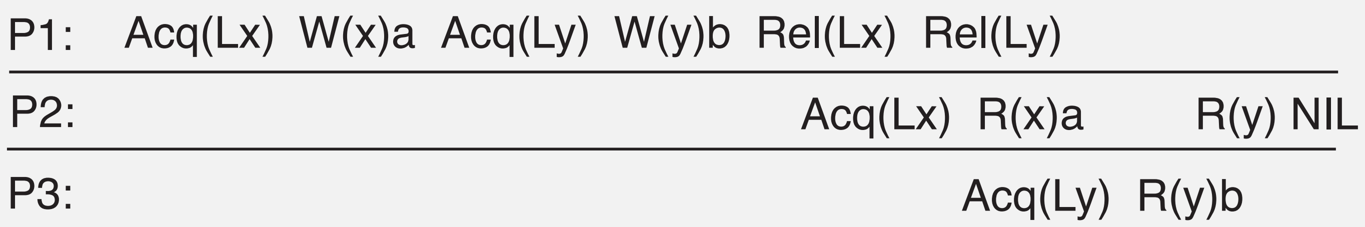
Definition

- 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.

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

What would be a convenient way of making this consistency more or less transparent to programmers?

Client-centric consistency models

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.

Consistency for mobile users

Example

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

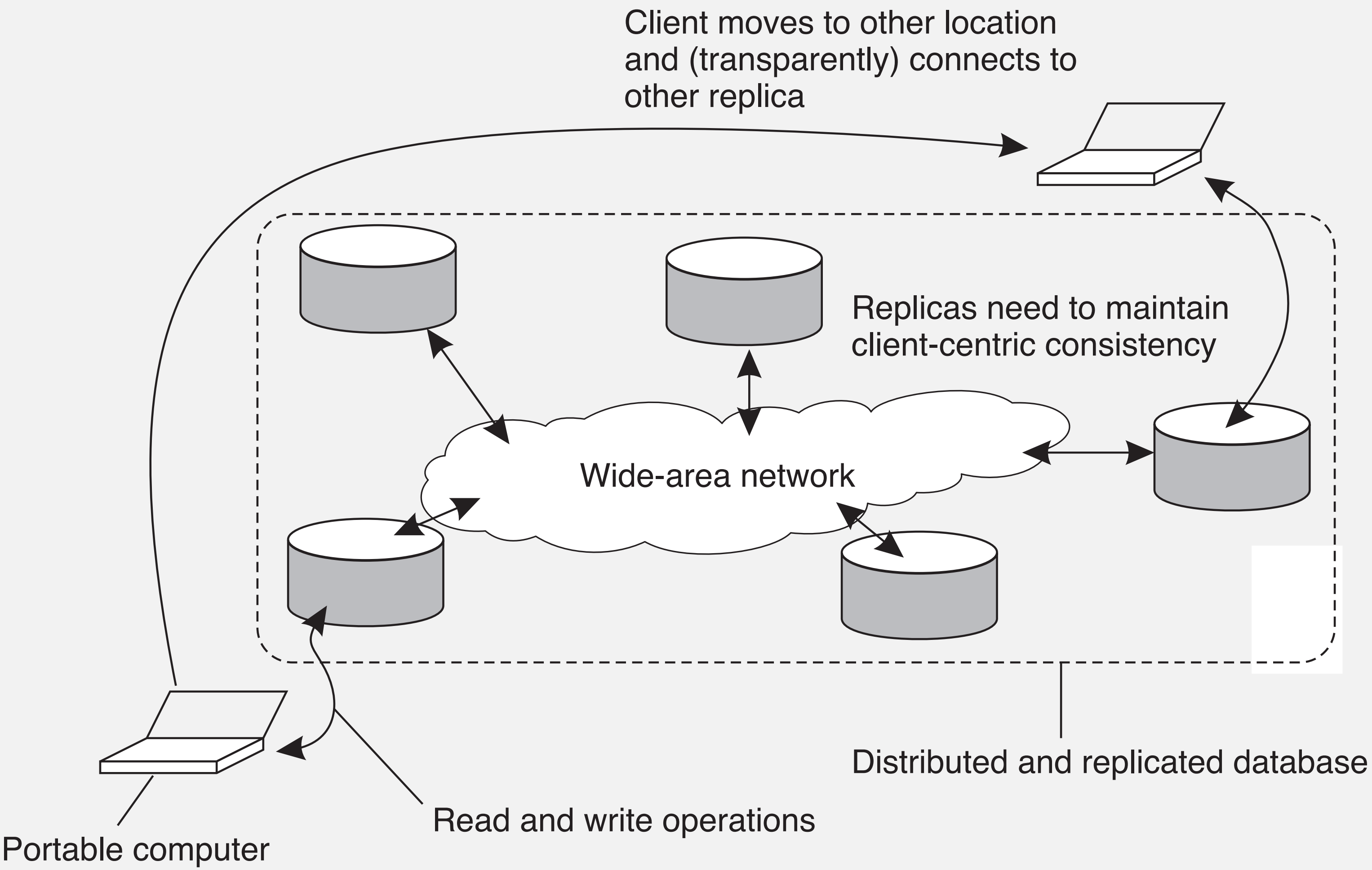
- At location A you access the database doing reads and updates.
- At location B you continue your work, but unless you access the same server as the one at location A, you may detect inconsistencies:
  - your updates at A may not have yet been propagated to B
  - you may be reading newer entries than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

Basic architecture



Client moves to other location and (transparently) connects to other replica

Replicas need to maintain client-centric consistency

Distributed and replicated database

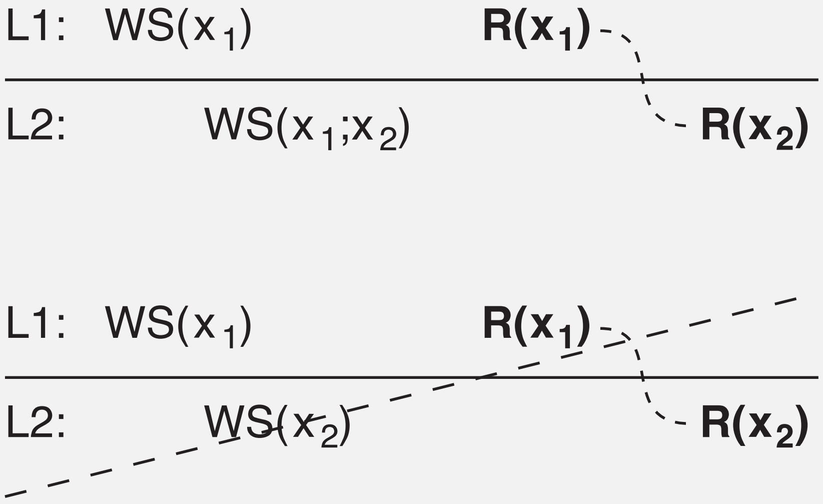
Read and write operations

Portable computer

Monotonic reads

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.



L1: WS(x_1) **R(x_1)**

L2: WS(x_1;x_2) **R(x_2)**

L1: WS(x_1) **R(x_1)**

L2: WS(x_2) **R(x_2)**

Client-centric consistency: notation

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.

Monotonic reads

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.

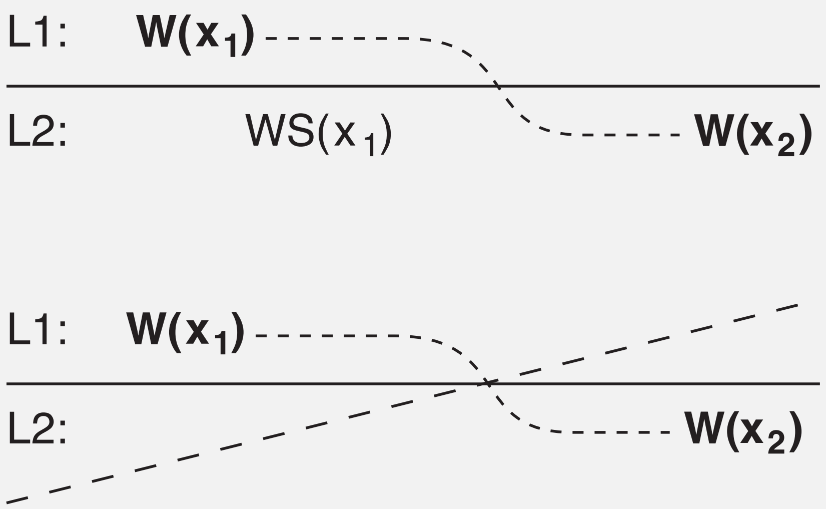
Example

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.

Monotonic writes

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.



Monotonic writes

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.

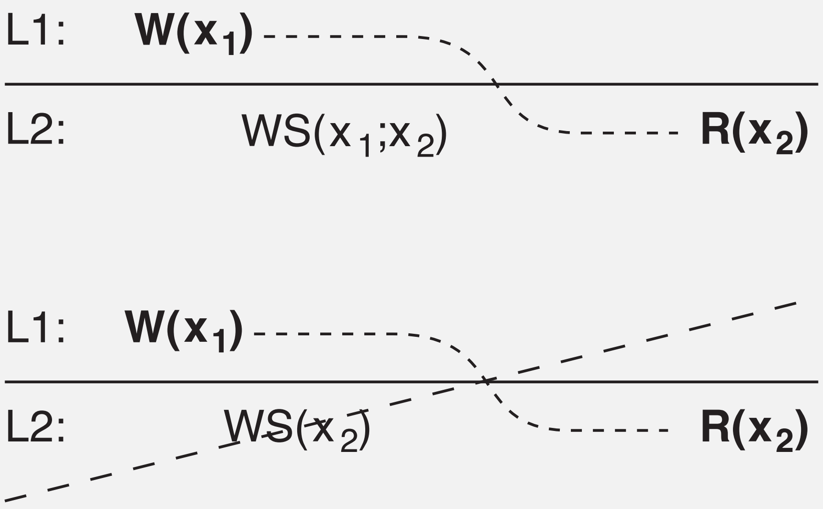
Example

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

Read your writes

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.



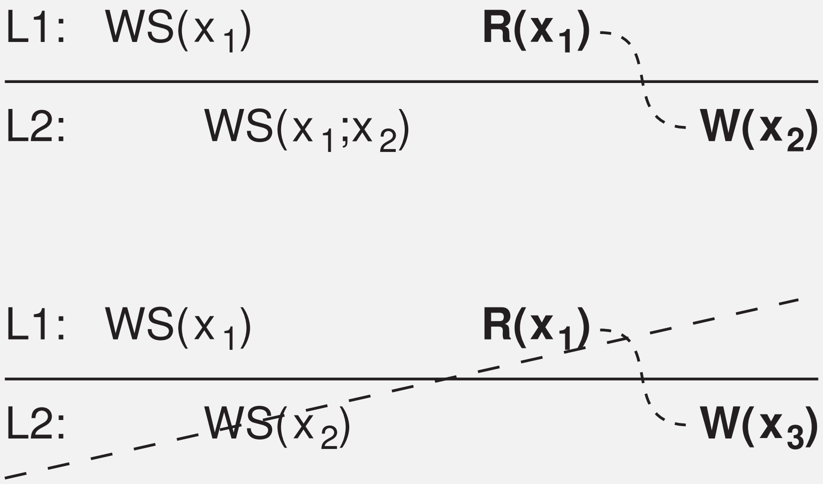
Example

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

Writes follow reads

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.



L1: WS(x_1) **R(x_1)**

L2: WS(x_1;x_2) **W(x_2)**

L1: WS(x_1) **R(x_1)**

L2: WS(x_2) **W(x_3)**

Example

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

CHAPTER 8: FAULT TOLERANCE

Failure Models
Failure Masking & Replication
Agreement in a Faulty System
Recover by Checkpointing

Failure models

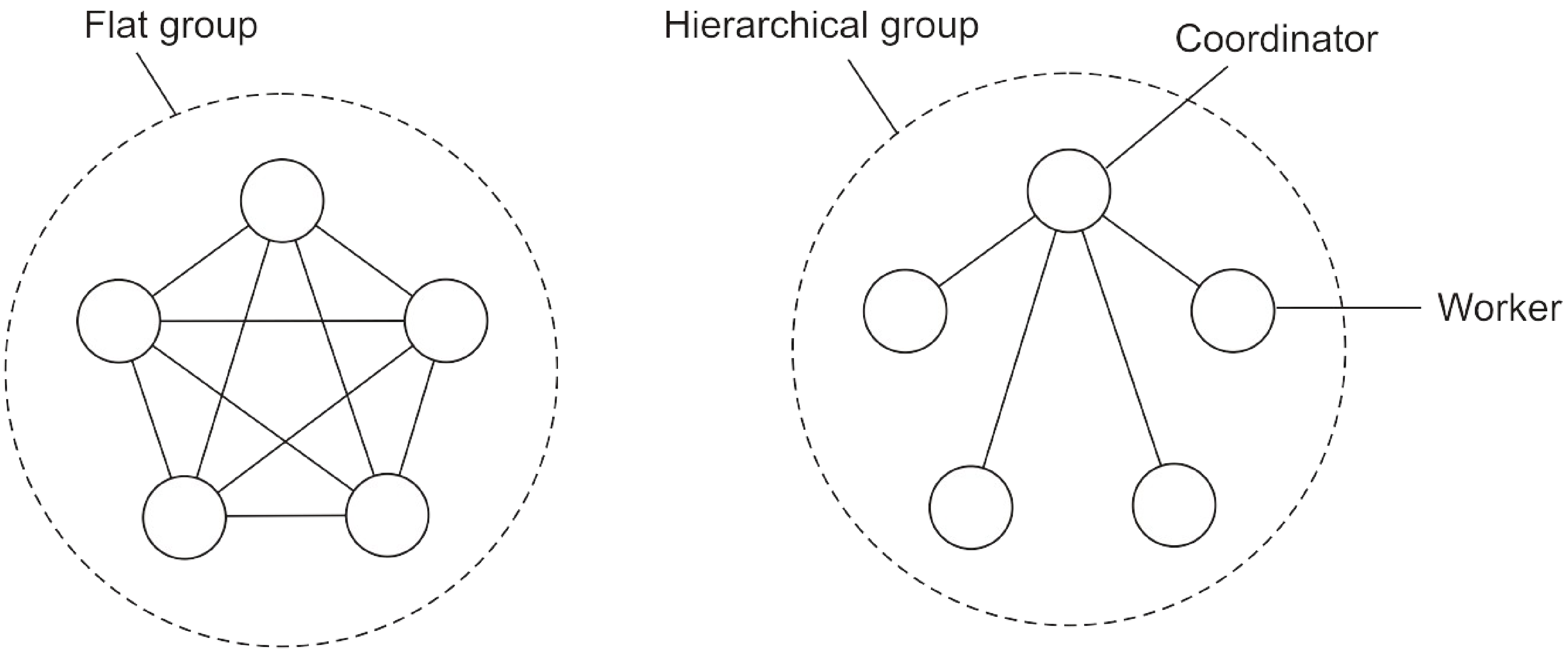
- Crash failures: Halt, but correct behavior until halting
- General omission failures: failure in sending or receiving messages
  - Receiving omissions: sent messages are not received
  - Send omissions: messages are not sent that should have
- Timing failures: correct output, but provided outside a specified time interval.
  - Performance failures: the component is too slow
- Response failures: incorrect output, but cannot be accounted to another component
  - Value failures: wrong output values
  - State transition failures: deviation from correct flow of control (Note: this failure may initially not even be observable)
- Arbitrary failures: any (combination of) failure may occur, perhaps even unnoticed

Halting failures

- Assumptions we can make:
  - Fail-stop: Crash failures, but reliably detectable
  - Fail-noisy: Crash failures, eventually reliably detectable
  - Fail-silent: Omission or crash failures: clients cannot tell what went wrong.
  - Fail-safe: Arbitrary, yet benign failures (can't do any harm).
  - Fail-arbitrary: Arbitrary, with malicious failures

Process reslience

- Basic idea: protect yourself against faulty processes through process replication:



Groups and failure masking

- k-Fault-tolerant group: When a group can mask any *k* concurrent member failures (*k* is called degree of fault tolerance).
- How large must a k-fault-tolerant group be:
  - With halting failures (crash/omission/timing failures): we need *k+1* members: no member will produce an incorrect result, so the result of one member is good enough.
  - With arbitrary failures: we need *2k+1* members: the correct result can be obtained only through a majority vote.

Groups and failure masking

- Important:
  - All members are identical
  - All members process commands in the same order
- Result:
  - Only then do we know that all processes are programmed to do exactly the same thing.

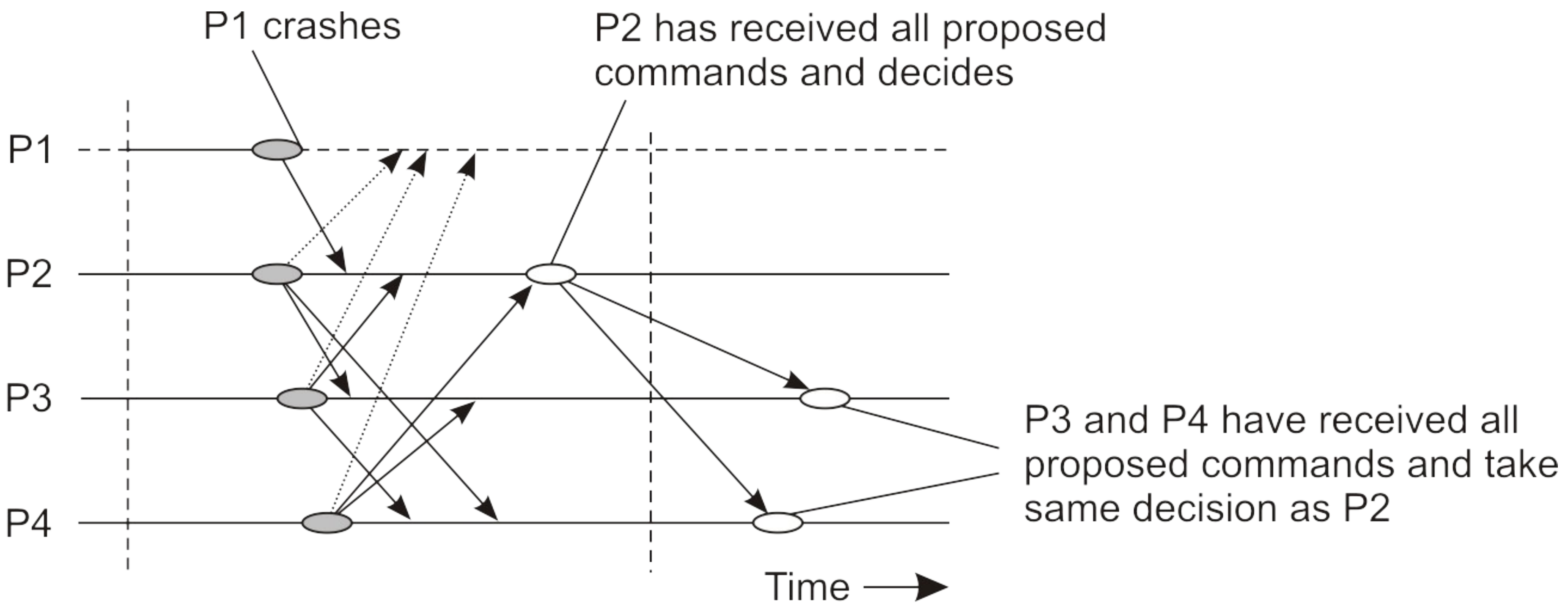
Observation

The processes need to have consensus on which command to execute next

Flooding-based consensus

- Assume:
  - Fail-crash semantics
  - Reliable failure detection
  - Unreliable communication
- Basic idea:
  - Processes multicast their proposed operations
  - All apply the same selection procedure → all process will execute the same if no failures occur
- Problem:
  - Suppose a process crashes before completing its multicast

Flooding-based consensus


P1 crashes
P2 has received all proposed commands and decides
P1
P2
P3
P4
P3 and P4 have received all proposed commands and take same decision as P2
Time

Failure detection

Issue

How can we reliably detect that a process has actually crashed?

- General model:
  - Each process is equipped with a failure detection module
  - A process *p* probes another process *q* for a reaction
  - *q* reacts → *q* is alive
  - *q* does not react within *t* time units → *q* is suspected to have crashed
- Note: in a synchronous system:
  - a suspected crash is a known crash
  - Referred to as a perfect failure detector

Failure detection

- Practice: the eventually perfect failure detector
- Has two important properties:
  - Strong completeness: every crashed process is eventually suspected to have crashed by every correct process.
  - Eventual strong accuracy: eventually, no correct process is suspected by any other correct process to have crashed.
- Implementation:
  - If *p* did not receive heartbeat from *q* within time *t* → *p* suspects *q*.
  - If *q* later sends a message (received by *p*):
  - *p* stops suspecting *q*
  - *p* increases timeout value *t*
  - Note: if *q* does crash, *p* will keep suspecting *q.*

Reliable communication

So far

Concentrated on process resilience (by means of process groups). What about reliable communication channels?

Error detection

- Framing of packets to allow for bit error detection
- Use of frame numbering to detect packet loss

Error correction

- Add so much redundancy that corrupted packets can be automatically *corrected*
- Request retransmission of lost, or last N packets

Reliable RPC

RPC communication: 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

RPC communication: Solutions

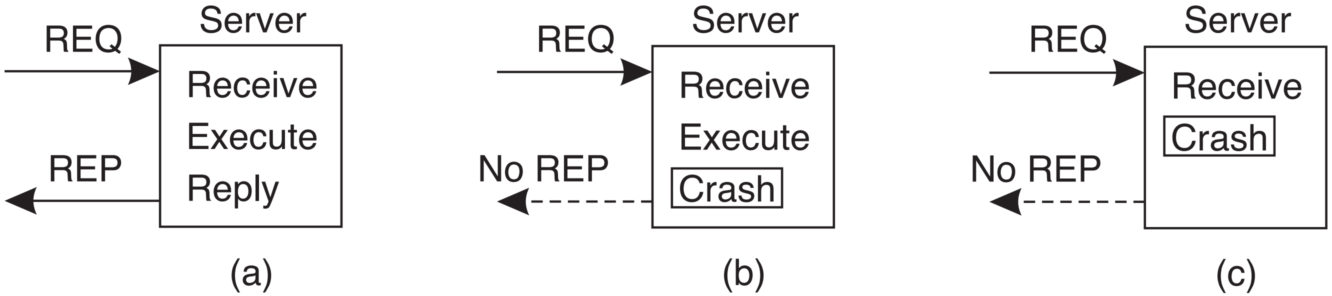
1: Relatively simple – just report back to client
2: Just resend message

Reliable RPC

RPC communication: Solutions

Server crashes

3: Server crashes are harder as you don't what it had already done:



Reliable RPC

Problem

We need to decide on what we expect from the server

- At-least-once-semantics: The server guarantees it will carry out an operation at least once, no matter what.
- At-most-once-semantics: The server guarantees it will carry out an operation at most once.

Reliable RPC

RPC communication: Solutions

Server response is lost

4: Detecting lost replies can be hard, because it can also be that the server had crashed. You don't know whether the server has carried out the operation
Solution: None, except that you can try to make your operations idempotent: repeatable without any harm done if it happened to be carried out before.

Reliable RPC

RPC communication: Solutions

Client crashes

5: Problem: The server is doing work and holding resources for nothing (called doing an orphan computation).

- Orphan is killed (or rolled back) by client when it reboots
- Broadcast new epoch number when recovering ⇒ servers kill orphans
- Require computations to complete in a T time units. Old ones are simply removed.

Question

What's the rolling back for?

Recovery

- Introduction
- Checkpointing
- Message Logging

Recovery: Background

Essence

When a failure occurs, we need to 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

Practice

Use backward error recovery, requiring that we establish recovery points

Observation

Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover

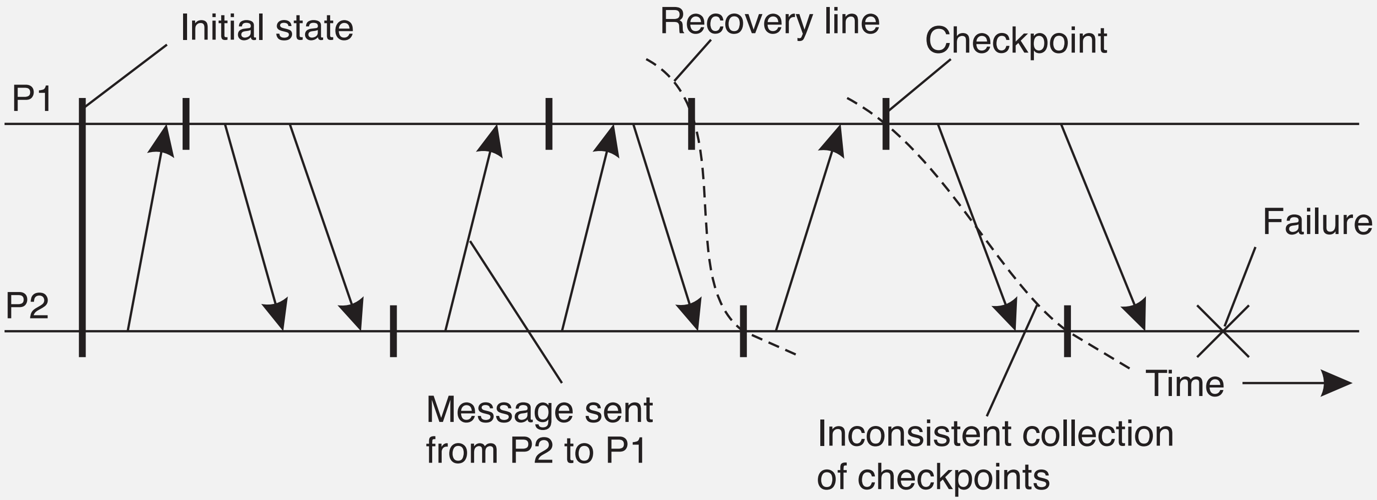
Consistent recovery state

Requirement

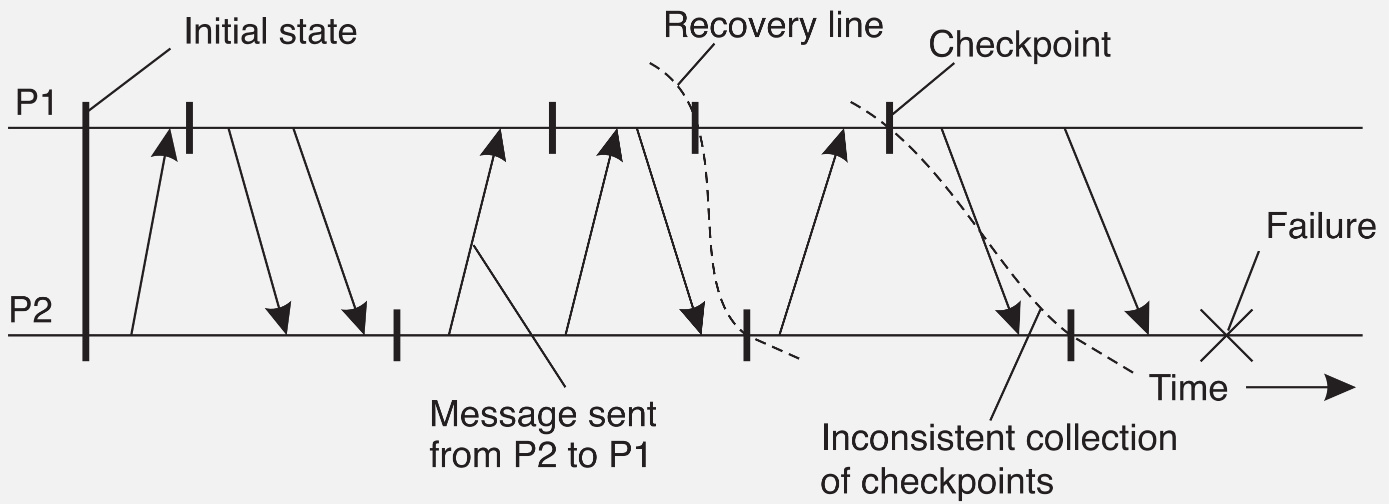
Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line

Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.


Initial state
Recovery line
Checkpoint
P1
P2
Failure
Time
Message sent from P2 to P1
Inconsistent collection of checkpoints

Consistent recovery state



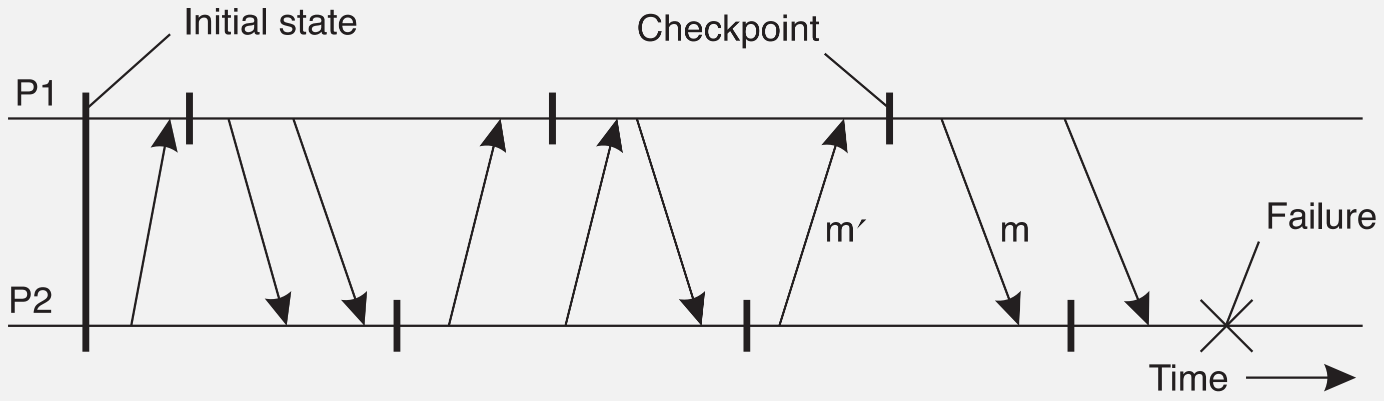
Observation

If and only if the system provides *reliable* communication, should sent messages also be received in a consistent state.

Cascaded rollback

Observation

If checkpointing is done at the “wrong” instants, the recovery line may lie at system startup time ⇒ cascaded rollback



Independent checkpointing

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$

Independent checkpointing

Observation

If process P_i rolls back to $CP[i](m-1)$, P_j must roll back to $CP[j](n-1)$.

Question

How can P_j find out where to roll back to?

Coordinated checkpointing

Essence

Each process takes a checkpoint after a globally coordinated action.

Question

What advantages are there to coordinated checkpointing?

Coordinated checkpointing

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a *checkpoint request* message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a *checkpoint done* message to allow all processes to continue

Observation

It is possible to consider only those processes that depend on the recovery of the coordinator, and ignore the rest

Message logging

Alternative

Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint ⇒ store messages in a log.

Assumption

We assume a 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 deterministic