

**DISTRIBUTED SYSTEMS**  
**Principles and Paradigms**

Second Edition  
**ANDREW S. TANENBAUM**  
**MAARTEN VAN STEEN**

**Chapter 1**  
**Introduction**

**Many modifications by Clark Elliott**

# CDK

- ▶ Some slides are courtesy of Coulouris, Dollimore, and Kindberg
- ▶ *Distributed Systems, concepts and designs*
- ▶ This is a good alternative book, and can give you a different perspective on the material.

# Keep your toolbox open this quarter...

- ▶ We will fill it with distributed systems techniques and tools during each lecture.
- ▶ Imagine you are in a room full of professionals. What expertise do you bring to the discussion because of your distributed systems background? Can you *teach* the concepts?



# Two themes throughout the quarter

- ▶ There is ***almost always a compromise.***  
Having distributed systems expertise means knowing the best compromise to make.  
Relevant to distributed system ***design.***
- ▶ There is ***no global clock*** for coordinating processes which affects the choice of ***algorithms*** in a distributed system.

# Add to your professional toolbox:

- ▶ Looking at the big picture, identify the pros and cons of the *compromises* being made in your distributed systems design.



# Distributed System—first view

A distributed system is:

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

# We will use a limited definition of *Distributed Systems*

- ▶ Software systems that operate on more than one computer via:
  - Internet
  - Intranet – like internet but managed by autonomous organization
  - Mobile computing
  - Wireless and other modern ad hoc network technologies

# Valid Distributed Systems areas, but we will *not* study...

- ▶ Parallel computing
- ▶ Grid computing
- ▶ Cluster computing
- ▶ Distributed operating systems
- ▶ [shared memory systems—brief discussion]
- ▶ [Distributed algorithms—touch on these]

# Motivation for Distributed Systems

- ▶ It's all distributed now anyway...
- ▶ Cloud computing: data and servers
- ▶ Resource sharing
- ▶ Geographical and legacy reasons
- ▶ Flexibility—buy, sell, add, drop

- ▶ Scalability—“from my desktop to the web”
- ▶ Robustness—one node fails, another is up
- ▶ Security—authenticated access to selective nodes
- ▶ System design – in CS it’s always a brave new world. What is design of the future?

# Computing of the future

- ▶ Moore's Law is running out. Was ~40% increase a year, now about 15% a year.
- ▶ More CPUs closely linked keeps total CPU power increase at 40% a year – but requires parallel computation. 8 processors? 1024? 4096?
- ▶ May lead to more distributed design at higher levels.

# Societies = distributed model

- ▶ People work in societies, and understand them.
  - Why not software?
  - The rise of agents—maintain their own state and goals
  - The rise of autonomous agents—especially as computers and programs become more intelligent and capable of independent decisions.
  - The human brain is coming online as processing nodes in computer networks, with brain/machine interfaces

# Intranets

- ▶ Like internet
- ▶ Locally administered
- ▶ Boundary allows policy enforcement, including for security
  - Firewall might allow/disallow packets based on remote source or content
- ▶ Possibly connected to Internet via router

# Distributed System—second view

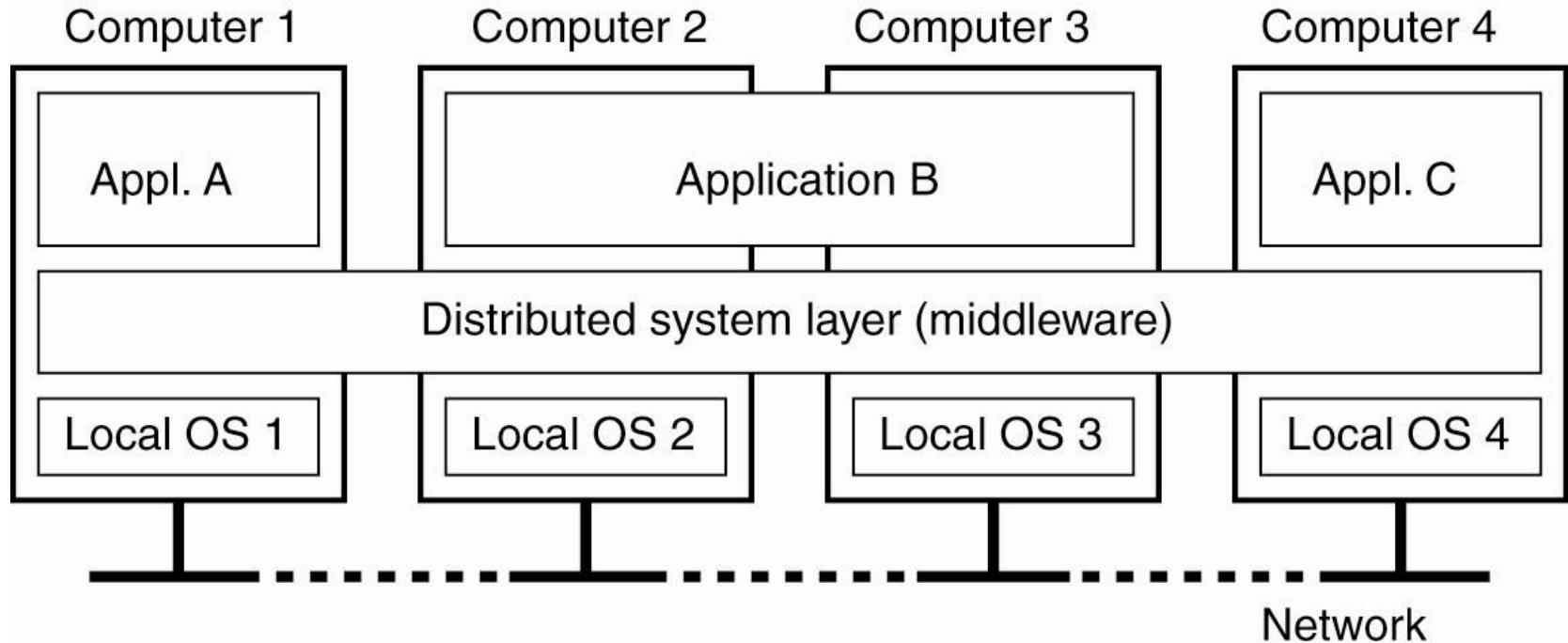


Figure 1-1. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

# Open Systems

- Non-proprietary system (don't have to pay for rights)
- Publicly known set of interfaces
- Anyone can write for, and use the defined standard interfaces to communicate with a system that adheres to the same standards.
- The interface is defined, the implementation is open
- Encourages development

# Open Systems

- Interoperability—IDLs define the syntax of the interfaces: names of functions, parameters, return values, exceptions raised on errors, etc.
- Semantics is defined differently and elsewhere—the behavior of the services and clients operating *through* the defined interfaces.
- Is your code portable to another installation?
- *Policy* separate from *implementation*. PC vs. Mac

# Four requirements

- (1) Fully defined, so that all vendors can work within the same framework
- (2) Stable over a reasonable length of time, so that the vendors have fixed development targets
- (3) Interfaces are publicly available
- (4) Are not under *arbitrary* control of any one firm or vendor.

<http://www.businessdictionary.com/definition/open-system.html>

# IDL - Interface Definition Language

[http://condor.depaul.edu/elliott/435/idlHello.idl.html](http://condor.depaul.edu/elliott/435/idl>Hello.idl.html)

- ▶ Specify the names of procedures or methods, and the precise bitstreams that go through the interface in each direction.
- ▶ Require a precise language that can be checked for syntax errors by a program.
- ▶ Allow any implementation that meets the specifications of the interface.
- ▶ BUT NOTE—Main purpose is to *define the interface*, and we could do this using an IDL with pencil and paper. The rest is serendipity for computer scientists.

- **Publish** the interface requirements – often captured in an IDL – Interface Definition Language. Anyone can write for (either side of) the interface: client side, server side, peer-to-peer.
- IDL – rules are so carefully specified that we can actually use them as a language fit for input to a compiler to write helper code (called stubs or skeletons) that gets us started. Methods, arguments, return values are all specified, based on their definition in the IDL.

# For \$200,000 could you write the code?

- ▶ You will write the simple server, Xu will write the simple client that allows the user to make twenty simple queries of various kinds of data.
- ▶ The two of you have pencil and paper only and invent an IDL: Together you write down all the methods, arguments and return values. All the data types are *fully* defined at the bit level. You agree on the semantics of what will be done by the server.
- ▶ You leave for Hawaii. Xu leaves for Manchester, UK. Each has a photocopy of the sheets of paper.
- ▶ No communication is allowed. Xu writes in Java, and you write in C++.
- ▶ At 3:12 AM on June 24<sup>th</sup> the server and client will come online. Do you want the job?

# Policy separated from Implementation

- ▶ Not only are the interfaces defined, but also the relationship of the parts of the system itself.
- ▶ We can replace or update either the local system, or the remote system, without affecting the other component.
- ▶ Note, for example that the programming language we use at either end is not important.
- ▶ Big picture is clear. Auditing? Politics?

# Arguments against...

- ▶ In many new arenas, start-up costs may run into the tens of millions of dollars. How do we encourage companies to invest if they are not guaranteed a return?
- ▶ Consider the development of many extremely useful drugs: suppose drug companies were not guaranteed a proprietary patent? A new drug costs \$100 million to bring to market.

- ▶ For large distributed systems, there may be many score sets of interacting and possibly conflicting policies. For example, the Firefox browser already has hundreds of configuration settings. Do we need to support all policies?

# Simple IDL example

Name of Remote Procedure:  
Hello-IDL-World

Return type of Remote Procedure:  
32-bit unsigned integer

Argument one:  
32-bit unsigned integer describing the length of argument two

Argument two:  
UTF-8 character string of the length contained in argument one.

# Client Stub

From the IDL we could automatically generate this client stub for calling the server:

```
char Hello-Message-String [] // char pointer to array
int Hello-IDL-World { // return int
    (unsigned int32) length, // arg one
    (char *) Hello-Message-String) // arg two
    <NO BODY OF CODE—YOU WRITE> // make the call:
    Remote-FB23AA(a1, a2, a3, length, Hello-Message-String));
}
```

Hides the actual operating system call to the “FB23AA” generic remote-call procedure.

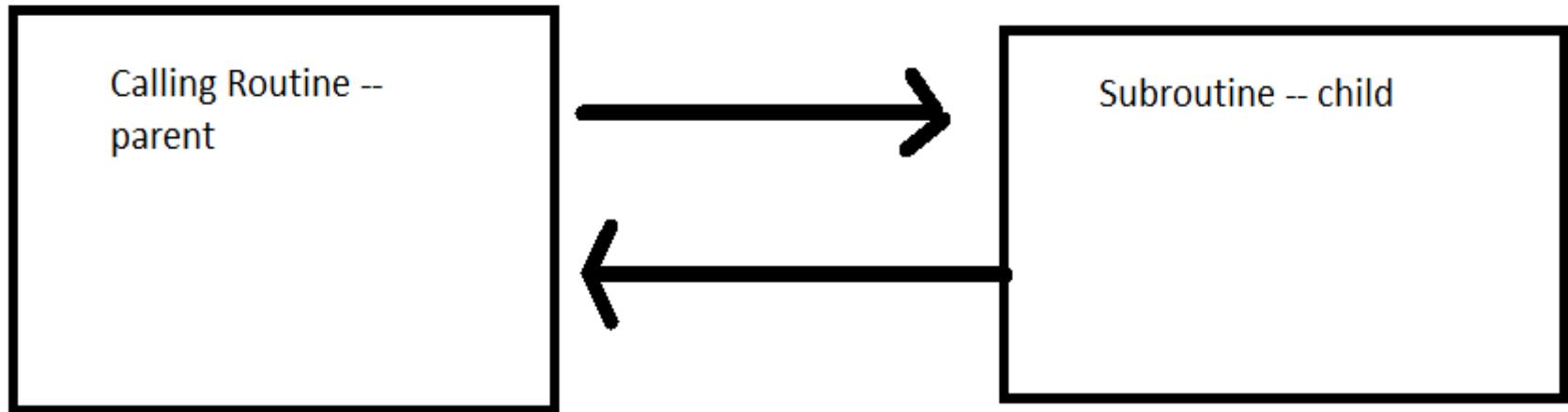
# Server Skeleton (Stub)

From the IDL we could automatically generate this server skeleton/stub to be called by the (un)marshaling RPC OpSys subsystem on the remote server:

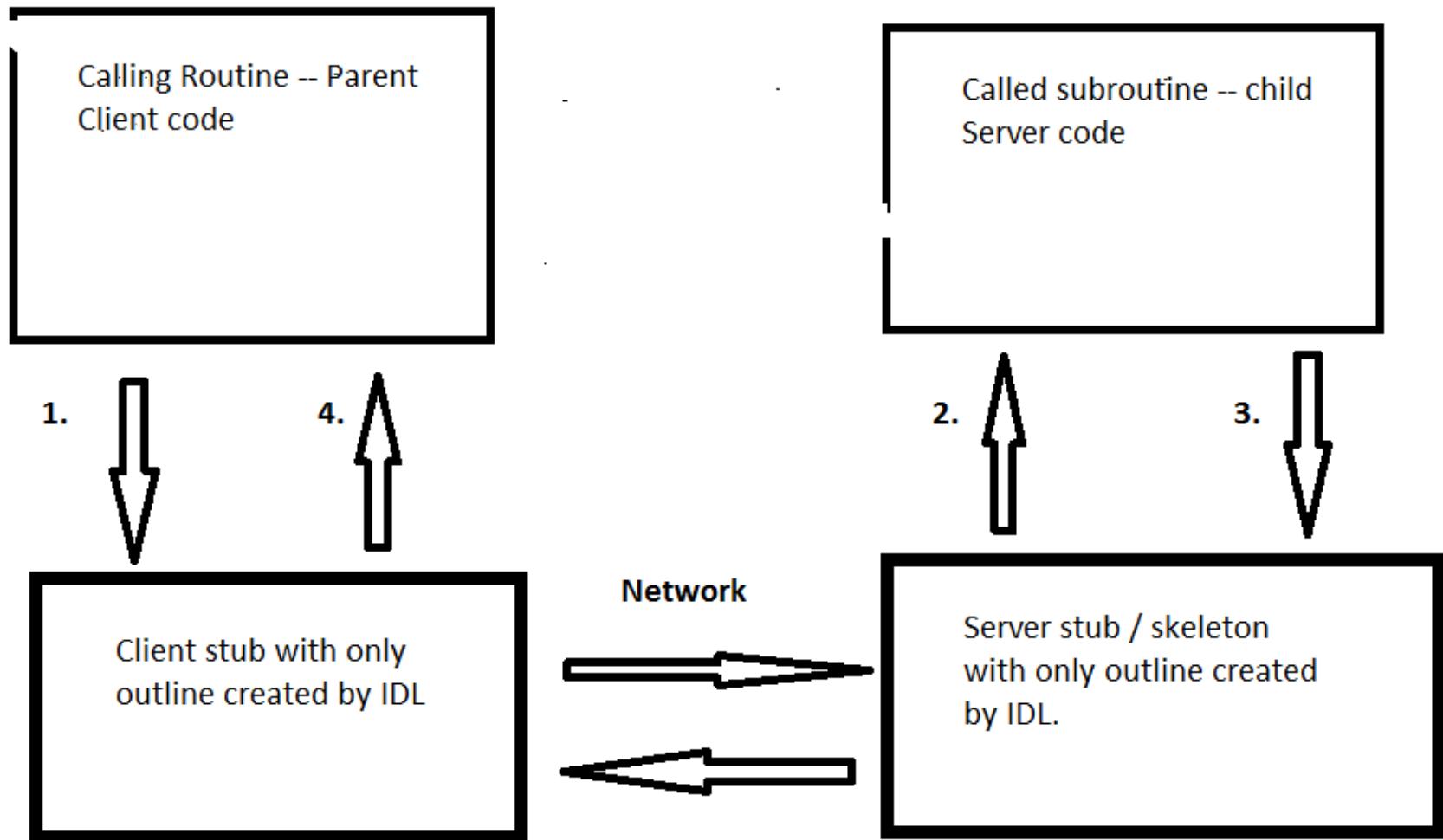
```
int Remote-FB23AA-Receive(int a1, int a2, int a3, unsigned int length,
                           char-string HelloMsg)){
    int return-val == 0;
    <NO BODY OF CODE-YOU WRITE>
    return (return-val);
}
```

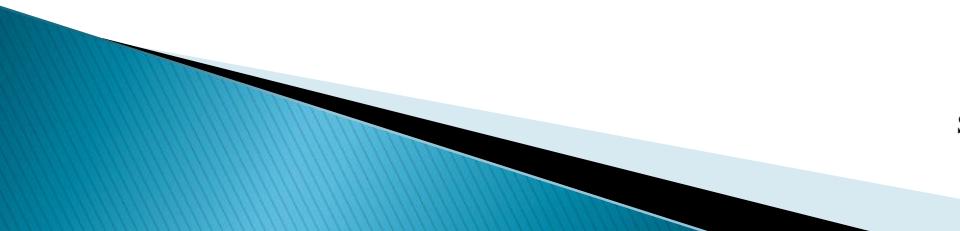
This procedure is called by the local server operating system after prompting by the remote system. An int return value is sent back to the caller.

# Local Call Sequence



# Remote call structured by IDL





Tanenbaum & Van Steen, Distributed  
Systems: Principles and Paradigms, 2e,  
(c) 2007 Prentice-Hall, Inc. All rights  
reserved. 0-13-239227-5

# Fully Distributed Algorithms

Characteristics of *fully decentralized (distributed) algorithms* which help with scalability:

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

- ▶ A formal area of study
- ▶ Largely research at this point, but valid study of algorithms that are immune to some problems of DSes
- ▶ We will not study these much. Could be a full quarter class.
- ▶ However – I *do* require that you understand their characteristics, and know what they are for.

# Scale-up issues in distributed systems

“From your desktop to the web...” *Inet!*

Size

Geography

***Administration – often not discussed***

# Example: scale-up of online course—size

How many OL students can the server handle?

- Bandwidth for the audio visual
- Disk space for resources and assignments
- Room in the grading links
- Etc

# Scaling Online class—*geography*

OL: what happens when OL becomes global?

- Timing problems: deadlines are in the middle of the night, fast turn-around email, foreign customs
- Suppose there is a lecture on security?  
DeFacto exportation of “munitions” Ouch!

# Scaling problem—*administration*

Administration is often *the* overlooked scale-up problem.

The critical issue is that the *design* of systems must often change rather than just the *size* of the resources

# Example one: online instruction grading

- 40 OL students: All grade assessments made by one person.
- 4,000 students: A hierarchical system of graders is now needed.
  - How do they coordinate? Meetings? Software? Groupware? Consistent responses to students?
  - How do they agree on edge condition grading?
  - Need graders, grader-supervisors, grade managers, grade directors.

# Example two: systems programmer authentication

- Single system: One local system administrator with secret keys to all local resources.
- Conglomerate of systems administered non-locally: Now have to coordinate among many administrators.
- Now must create and maintain a hierarchy, or set-based system of authentication for system programmers and supervisors.

- Administration is not a traditionally algorithm-based structure, so is often overlooked in scale-up discussions.
- But because it may require fundamental changes in the structure of the system design it is often the *most critical bottleneck* in scaling up a distributed system.

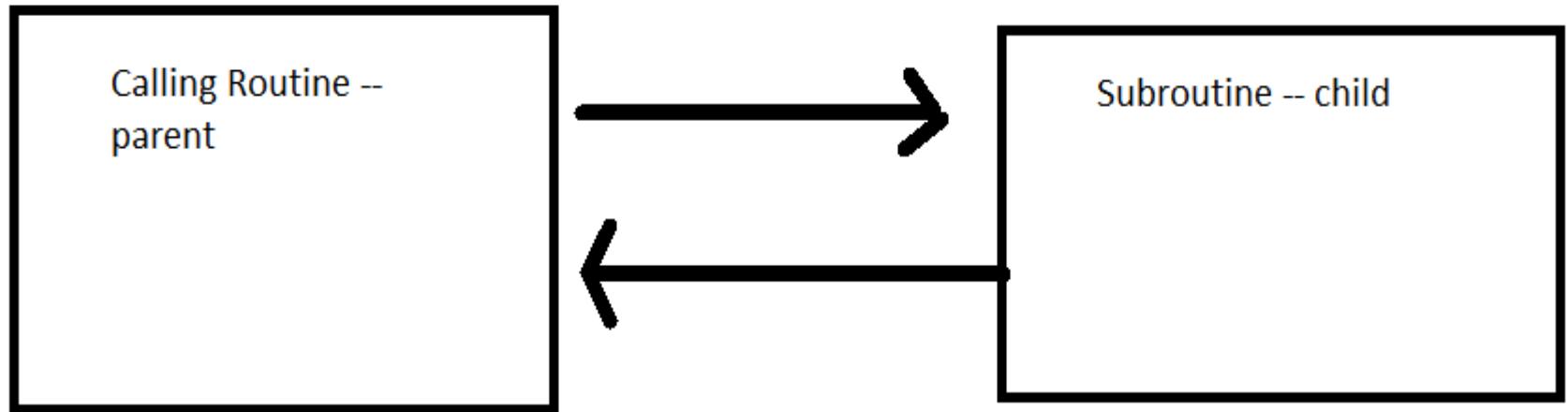
# Add to your professional's tool box

- When discussing current *or future* scale-up possibilities, have you examined difficulties with scaling up *administration*?
- Do you need to design now for future scale-up with regard to administration?



# Recall: Local Call Sequence

Both routines are in the same process, on the same computer:

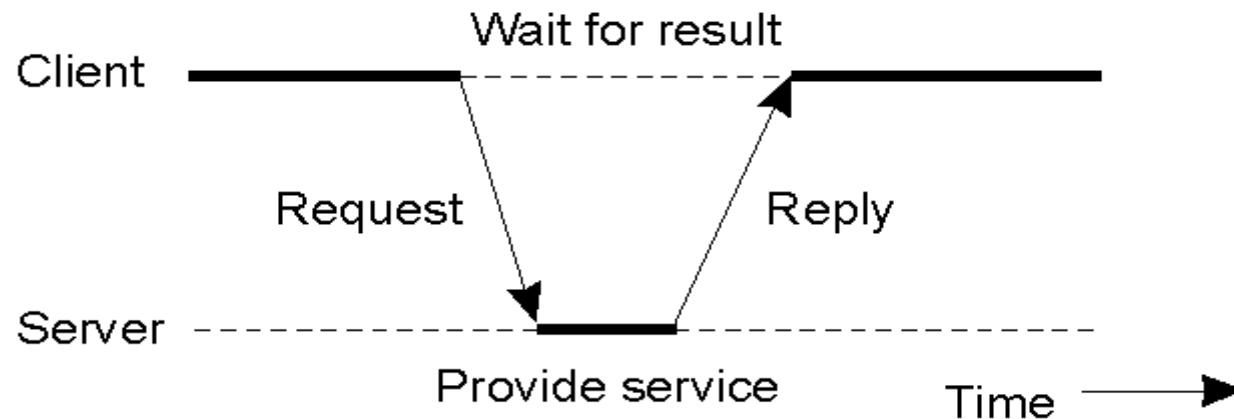


# Traditional *blocking* (synchronous) calls— works fine with local systems.

- Make a procedure call, and the caller does nothing until the call returns.
- Millions of a second to make the call.
- If the called procedure fails, the calling procedure is probably failing too because they are typically on the same computer in the same process. If one routine dies, the other dies too.
- Because the CPU is busy with the subroutine call it is not available to the caller routine anyway.

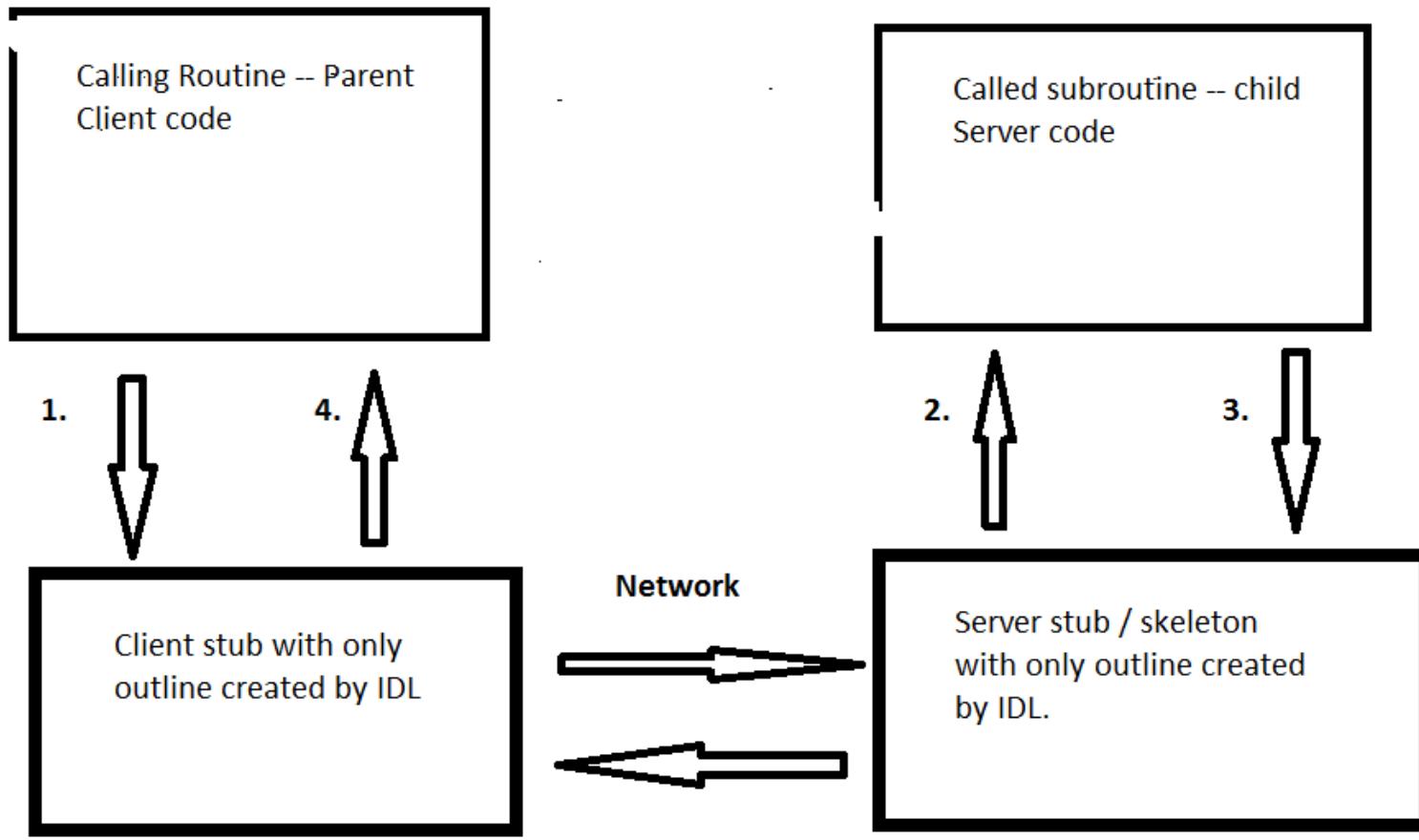
# Clients and Servers

General interaction between a client and a server.



# Recall: remote call structure

Calling routine and called routine on different computers:



# Traditional *blocking* calls—problems in a distributed system

- Call may take *seconds* over the network, a million times as slow.
- The caller process is blocked waiting and doing nothing, possibly for a long time even though the CPU and memory resources are available.
- Geography—more failures because of long network connections.
- The called routine on the remote system fails, or the network fails, but the caller routine still alive. It is blocked and can't query the remote system, or take remedial action (such as finding another server), even though there is no reason for it not to make queries and take actions.

# Synchronous vs. Asynchronous

- ▶ Blocking calls are *synchronous*—the caller and the called procedure coordinate in one process.
- ▶ A solution is non-blocking, or *asynchronous* calls, using two processes.
- ▶ New programming paradigm.
- ▶ Caller no longer has to wait, doing nothing.

# Synchronous vs. Asynchronous

- ▶ Logic is potentially *much* more complex.
- ▶ Idea is that the calling process continues, without waiting for the called process to return.
- ▶ While there is only one type of synchronous call, there are many types of asynchronous calls. We will study these later.

# Add to your toolbox

- ▶ Ask appropriately: should we make the compromise to take on extra complexity, so that we can free up our calling processes to:
  - Get simultaneous work done in both processes at once
  - Allow the callers to make queries when something goes wrong



# Transactions – essential concepts

- ▶ *Before/After:* To the outside world there is a *before*, and an *after*, and nothing in between. Think of closing on a house: they get the money, you get the house. When does the lighting bolt strike?
- ▶ *Critical Section*—May have to reserve resources for the duration of the transaction using *locks* on resources as part of a *critical section* [CS] of computer code inside the transaction.

- ▶ *Roll back*—something went wrong, so go back to the *before* state.
- ▶ *Commit*—everything is complete so continue or to other states, possibly discarding all rollback information
- ▶ *Checkpoint*—a saved state of the system to which you can return, discarding all subsequent transactions.

# The Critical-Section Problem

- ▶  $n$  processes are all competing to use some shared resource (e.g. data), but only one is allowed access at a time (e.g., for updates).
- ▶ Each process has a code segment, called the *critical section* in which the shared resource is accessed (e.g., open the file for update).
- ▶ Problem: ensure that when one process is executing in the critical section, no other process is allowed to execute in that critical section.

# Semaphores / Memory locations

Semaphores allowing access to a *critical section* of code:

- ▶ Usually just an integer memory location. TRUE (not null) means *wait, the CS is busy*. FALSE (null) means *go ahead into the CS*.
- ▶ If the CS is available [Semaphore is FALSE] set the semaphore to TRUE then proceed into the CS.
- ▶ When you are done, set the Semaphore to FALSE again so others can use the CS.
- ▶ Local systems have hardware/architecture support for semaphores.

# Example Critical Section

- ▶ Five different bank accounts belonging to two different customers.
- ▶ As part of a complex *transaction* move a bunch of money around for escrow, taxes, payments, refunds, etc.
- ▶ The total money *before* and *after* is always identical.
- ▶ But inside the critical section code you always have to chose to either delete from the sender account or add to the receiver first when you move it. So the total changes temporarily.

- ▶ During the transaction, the bank accounts are *locked* to all other processes. They cannot read or write those values.
- ▶ Why?
- ▶ To the outside world there are five accounts *before* the transaction, and five *after* the transaction. There is no access at all during the processing of the critical section.

## Characteristic properties of transactions:

- ▶ *Atomic*: To the outside world, the transaction happens indivisibly.
- ▶ *Consistent*: The transaction does not violate system invariants (rules) when complete.
- ▶ *Isolated*: Concurrent transactions do not interfere with one another.
- ▶ *Durable*: Once a transaction commits, the changes are permanent.

# Example invariants

- ▶ The *age-in-years* field may not exceed 120 when the transaction is complete.
- ▶ All the data must be sorted alphanumerically when the transaction is complete.
- ▶ The total money in the system must remain at a constant value.
- ▶ (Invariants may be temporarily violated within the critical section, during the execution of the transaction, as long as the external world cannot access them.)

# Example transaction

Transfer Sam's money from checking to savings:

- ▶ Remove \$3,000 from Sam's **checking** account.
- ▶ Add \$3,000 to Sam's **savings** account.
- ▶ During the process, there is a time when there is a missing \$3,000 in the bank. What if the system fails at exactly that moment, then restart outside the transaction? We violate the invariant that the total must remain constant.
- ▶ The business model requires a transaction: before and after, but nothing in between

# Distributed Transactions

- ▶ Built-in hardware support for critical sections is *not available*.
- ▶ The sending of messages is unreliable, and abort messages can go missing as well.
- ▶ Much more complex than local transactions
- ▶ *But we still need them!*

# Bank revisited.

- ▶ Five accounts are maintained by five different servers at five different banks in a distributed system.
- ▶ We must lock the accounts in five different databases on five different computers, managed by at least five different processes.
- ▶ Communication is by *messages* that travel over the network.

- ▶ Coordination messages can get lost.
  - ▶ Abort messages can get lost.
  - ▶ Commit messages can get lost.
  - ▶ Rollback messages can get lost.
- 
- ▶ Most important: where is the memory location for the semaphore managing the critical section? On which machine? Managed by which process?

# Real world example

- ▶ Getting a plane ticket, a hotel, and a rental car at the Orbitz online travel service.
- ▶ The transaction manager has to communicate with separate databases, on separate computers, managed by separate processes.

# Add to your toolbox

- ▶ Absolute crystal clear understanding of what a transaction is, and why it is much more complex in a distributed system.
- ▶ Be able to explain distributed transactions to a roomful of people.
- ▶ Understanding that there is hardware support for semaphores on a local system, but only software support via messages in a distributed system.



# Transaction Processing Systems (3)

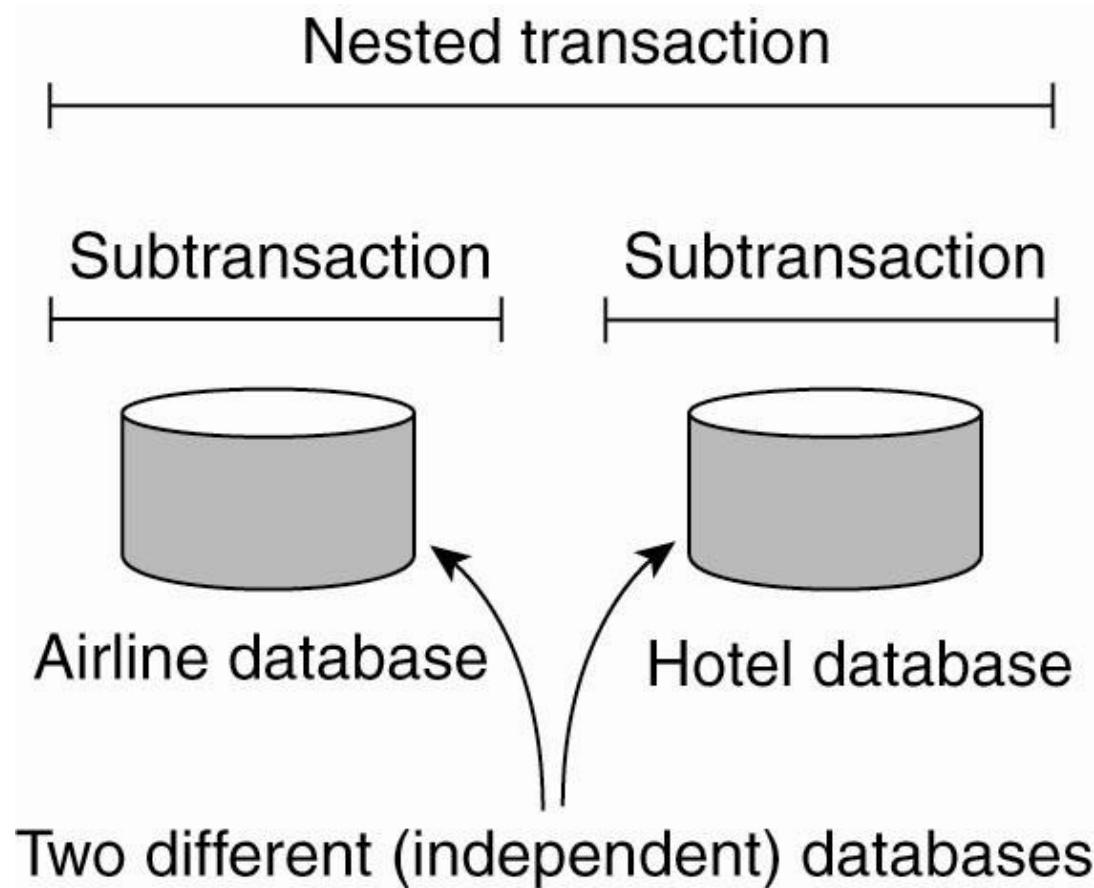
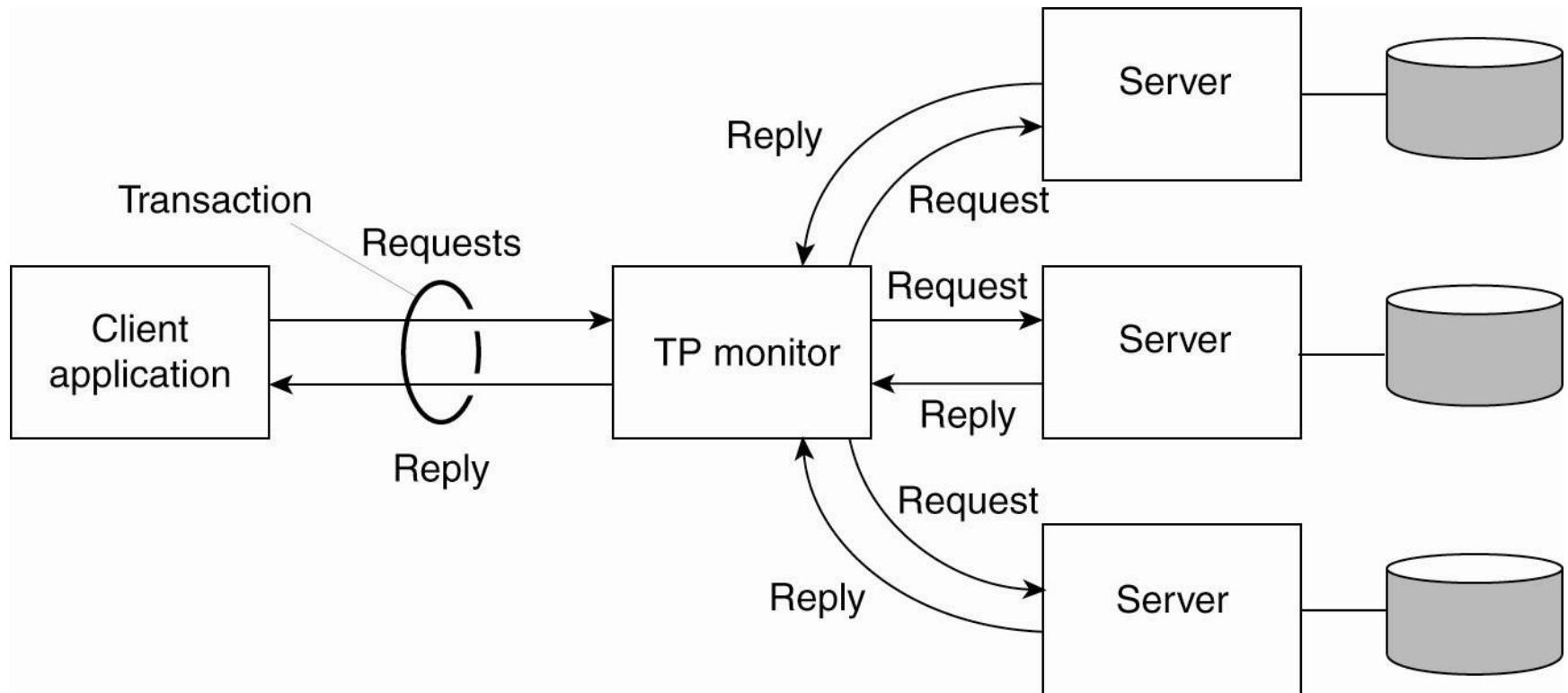


Figure 1-9. A nested transaction.

# Transaction Processing Systems (4)



# What is a *process*?

- ▶ A process is the basic engine of work on a computer system. A process is a computer program that is being executed.
- ▶ This is true in local systems with all processes running on a [single] central processing unit (CPU)
- ▶ It is also true in a distributed system, where processes may run on different, network-connected CPUs

- ▶ One program can run in many processes at the same time, such as when you start up multiple instances of a browser on your PC.
- ▶ The CPU of a computer system runs the instructions of a process one at a time, in order.
- ▶ Processes use resources, such as memory, disk space, printers, displays.

# Inter-process communication

- ▶ On a local system, processes communicate (share program variables) through shared locations in memory, using *Inter-Process Communication* (IPC).
- ▶ In distributed systems, when IPC is not available, processes must send *messages* over the network.

# Shared memory?

- ▶ The *perfect* communication solution. No need for this class.
- ▶ A very simple coordination mechanism between processes
- ▶ IPCs (Inter-Process Communications) are used to communicate between processes through shared memory. One writes, the other reads, then vice versa.
- ▶ In distributed systems because we do not have some emulation of shared memory we have to send messages over the network.

# Context switching

- ▶ On a *multi-tasking* computer, one *central processing unit* (CPU) is shared by all processes.
- ▶ [Example: in Windows, run *taskmgr*]
- ▶ Processes are continuously swapped in and out of the CPU so that every process gets a chance to run for a while. This is called *context switching*.
- ▶ The operating system process managing context switching may switch a process at any time.

# Atomic Action

- ▶ The unit cannot be broken down further. Derives from the idea of an atom, which at the time of discovery was believed to be the basic building block of the universe.
- ▶ For application systems, this means that any process will be allowed to complete an atomic section of code as though it were never swapped out of the CPU and superseded by another process, or interleaved with another process at the software level.

# The critical section coordination problem

- A. Look at semaphore (“gatekeeper”). If TRUE (busy) then return to A, else if FALSE (free) then just continue
  - B. Set the semaphore to TRUE
  - C. Complete the *Critical Section* (CS)
  - D. Set semaphore back to FALSE
- 
- 1. Process X completes to B, then is switched out
  - 2. Process Y completes through B and enters CS, then is switched out
  - 3. Process X continues after B, enters CS. Big trouble—two processes are in the CS at the same time!

# Test and Set Instruction –How it works:

- A. Look at semaphore. If TRUE (busy) then return to A else if FALSE AND SET TO TRUE (was free – now busy) then...
- B. Complete the *Critical Section*
- C. Set semaphore back to FALSE
  - 1. Process X completes to B, then is switched out
  - 2. Process Y loops to A continuously
  - 3. Process X switched back in, completes CS, and C
  - 4. Process Y completes CS and C— all is well.

# Monitors in D<sub>S</sub>es. Hey—where's my hardware support?

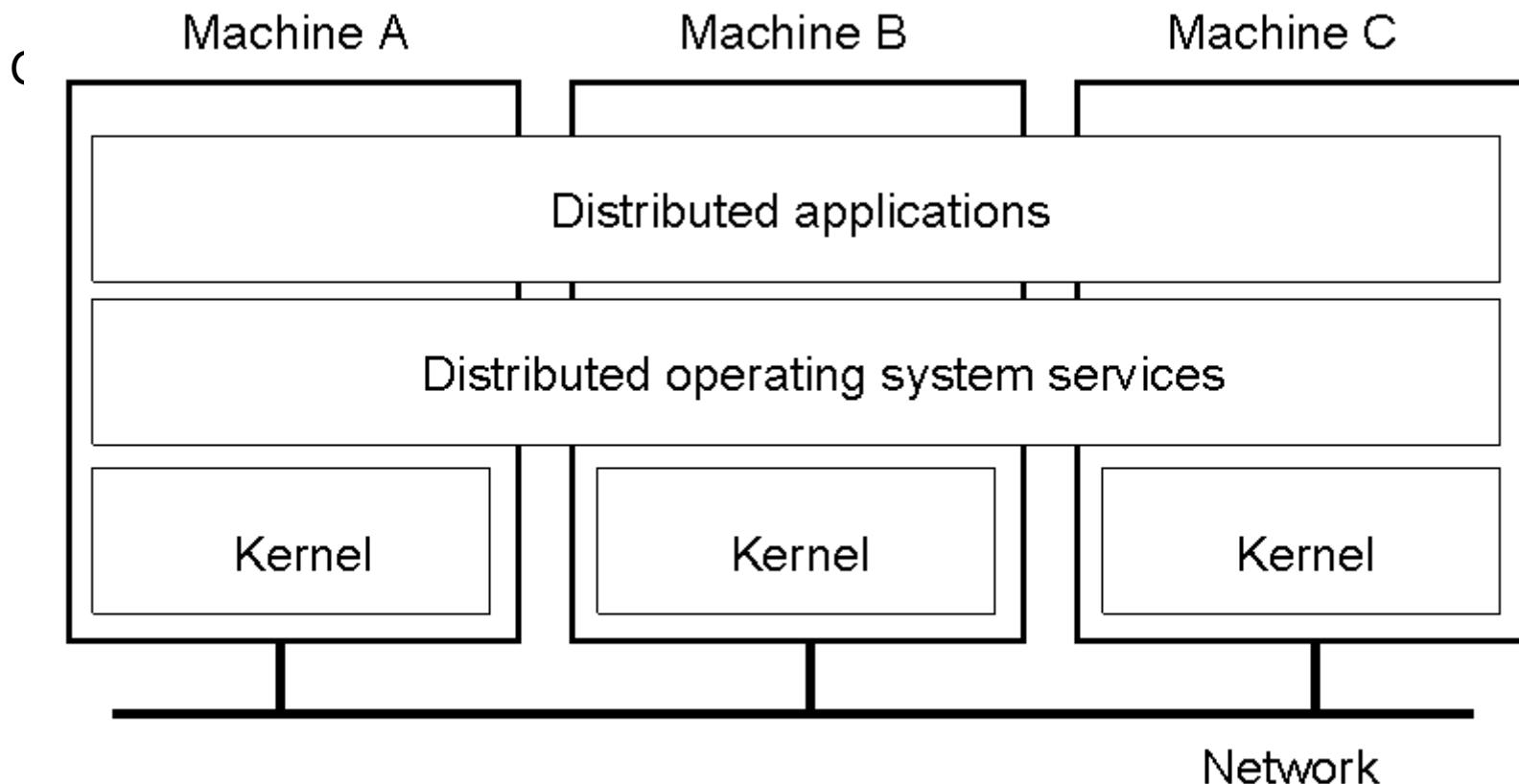
- ▶ Locked / shared resources, may be on a remote machine.
- ▶ They may involve processes on different machines with different memory spaces—**no *test and set!***
- ▶ This is relevant to transactions and distributed transactions.
- ▶ Usually requires a process to monitor the shared resource. Processes can fail.
- ▶ Dead processes locking a resource can be a big problem, especially on a remote machine.

# Universally Unique Identifiers (UUIDs / GUIDs)

- ▶ Some character string (or number) that no one else in the *universe* will ever use.
- ▶ URLs are UUIDs: One subnet, one machine, [one user], one directory, one file. This is a unique physical path on earth. XML uses this feature to create unique names.
- ▶ A very large random number will be pseudo-unique and often good enough. E.g., for a universe of a thousand processes, a random number between zero and a trillion will have a conflict only one time in a billion.

- ▶ A range of numbers can be unique, as long as they are divided up and each range is given to a unique process.
- ▶ We can also generate a *range* of numbers large enough for a one-to-one mapping of numbers and all possible current and future real memory locations in all computers on earth.

# Distributed Systems—where's my IPC?



# The perfect solution: distributed shared memory systems

- ▶ In theory we could build a universal shared memory system as middleware.
- ▶ All core memory is mapped to UUIDs which serve as the middleware memory addresses.
- ▶ Each machine has its own range of UUID memory addresses mapped to local memory.
- ▶ Any machine in the distributed system could uniquely address the memory space of any other machine without confusion or conflict.
- ▶ Distributed IPCs could now be used.

# But...

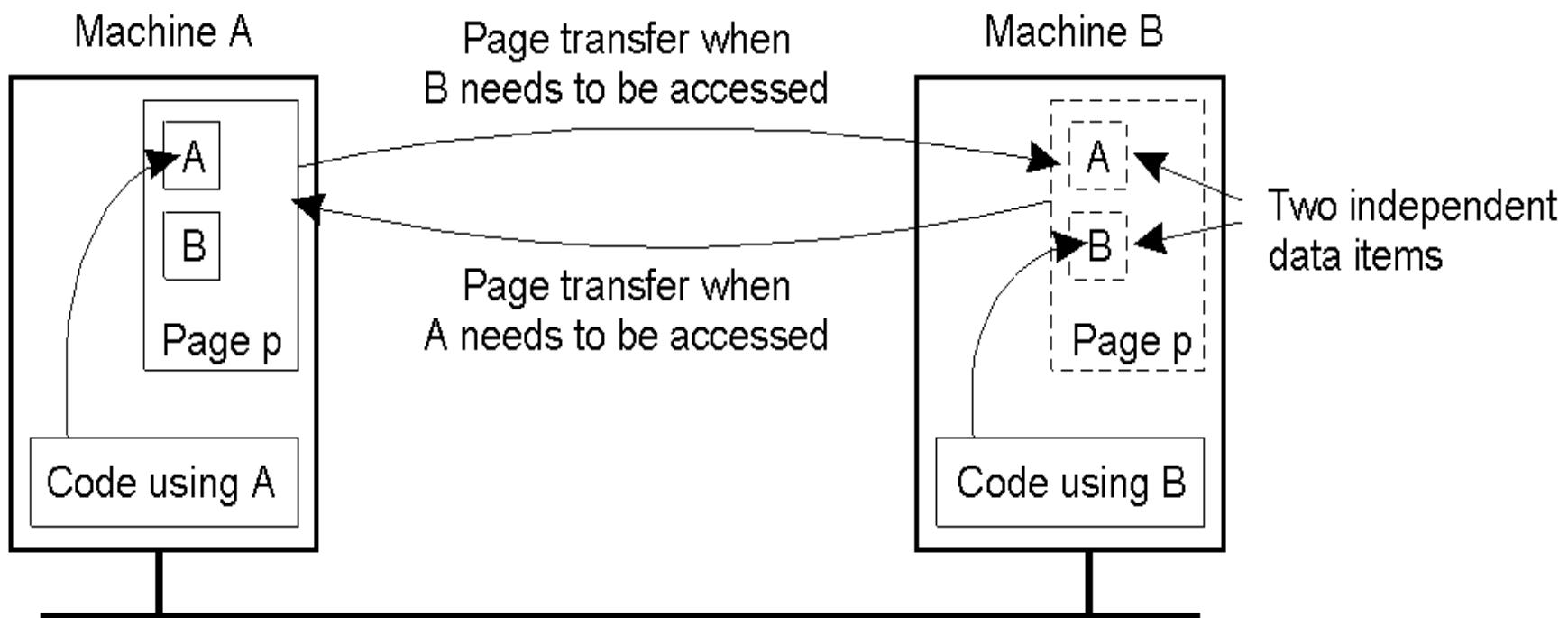
- ▶ WAY too slow and way too buggy
- ▶ Has to be implemented with message passing between processes “under the hood” anyway.
- ▶ Impossible to implement in an efficient way.
- ▶ Requires predicting how programs will run to make it work.

# Why study shared memory systems?

- ▶ Shared memory systems have not proven practical, because they are too devilish to implement (all message passing underneath)
- ▶ A shared-memory subsystem would solve many of the problems of DS coordination, including that of a global clock.
- ▶ Useful to consider when trying to understand DS architectures, and what they are trying to provide.

# Distributed Shared Memory Systems

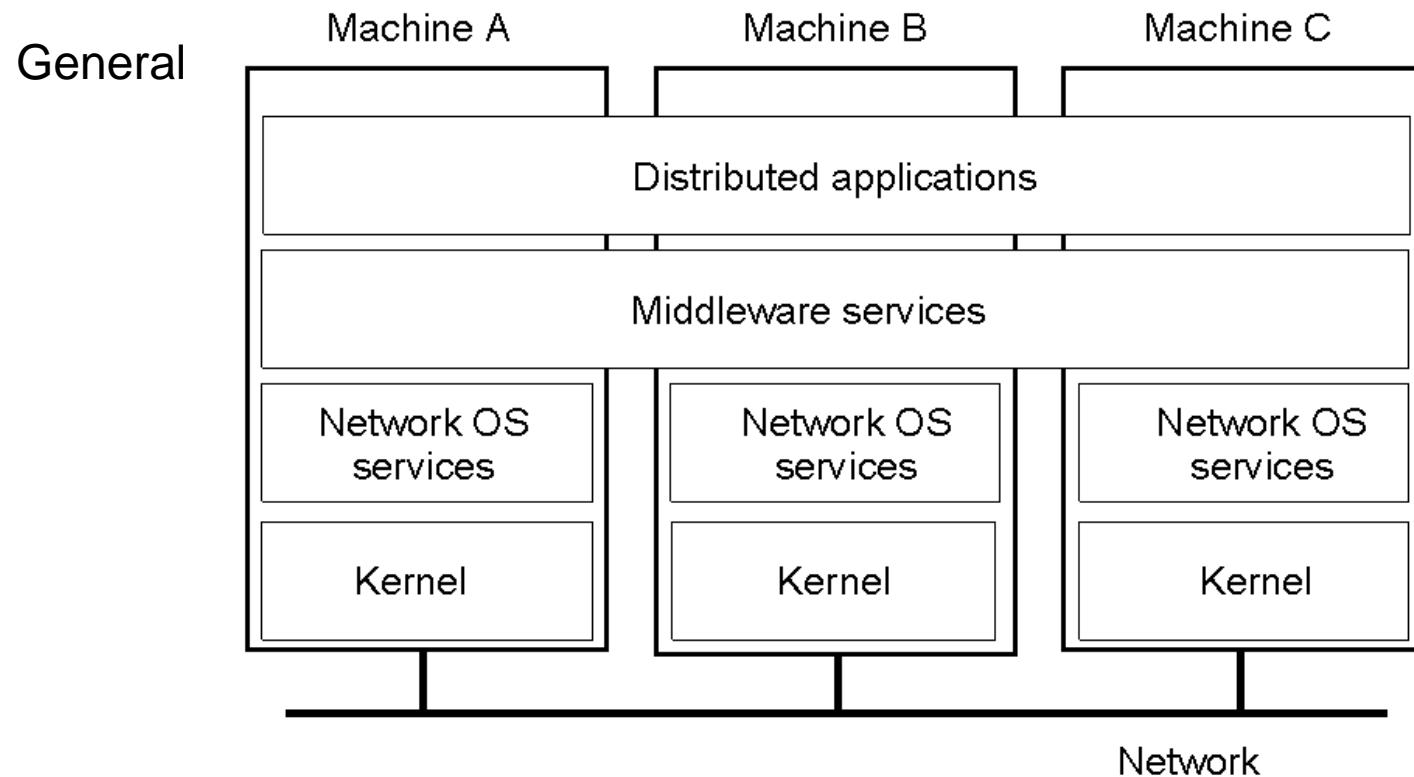
## – an example difficult problem



# What is *middleware*?

- ▶ Middleware sits between the application programs, and the operating system library calls and native instructions.
- ▶ Middleware presents a uniform interface for the application programmer, often (but not always) independent of the underlying operating system.
- ▶ Berkeley Sockets, Java RMI, CORBA, .NET, RPC, are examples of middleware

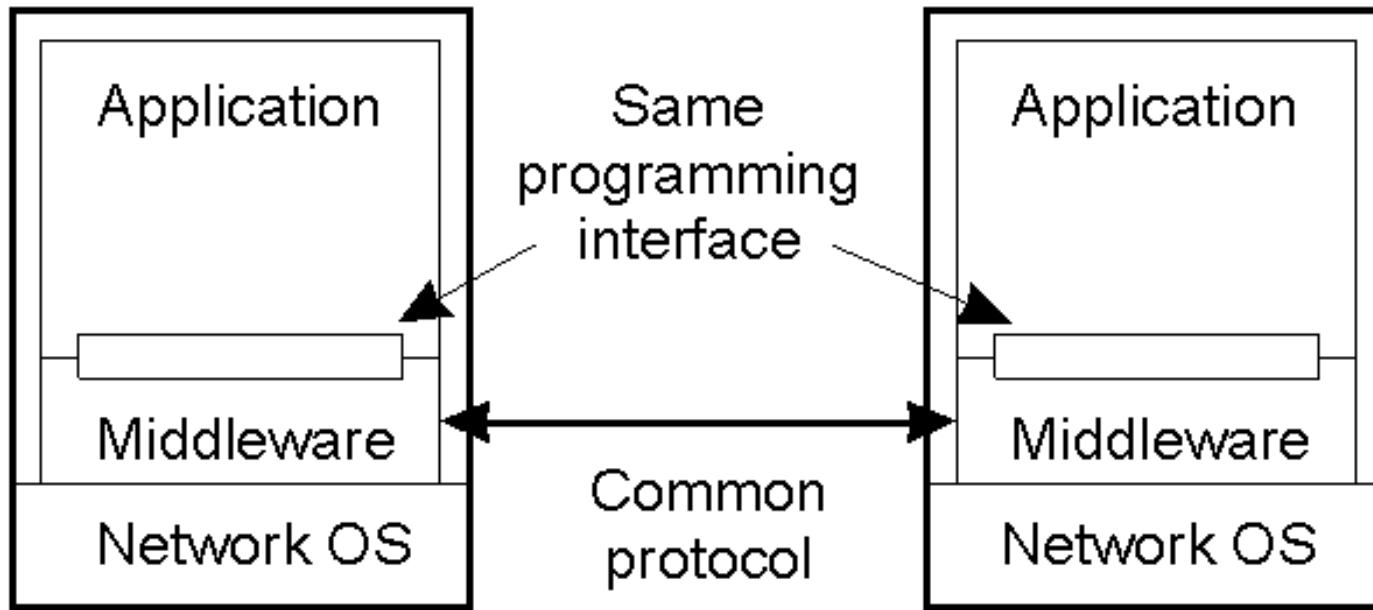
# Positioning Middleware



# Usefulness of open systems

- ▶ In *open* distributed systems, applications can use one protocol to communicate with other local and remote processes, independent of the operating system on which they are running.
- ▶ The underlying operating system calls, buffering, network protocols, etc. are hidden from the application programmer who works through an *interface*.

# Middleware and Openness



In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.

# Message protocol tradeoffs

- ▶ There is always a tradeoff between reliability and innate efficiency of the message passing protocol
- ▶ “Send and forget” is (a) the most efficient, and (b) the least reliable.
- ▶ “Guarantee delivery for every message,” is [possibly] the least efficient, and the most reliable.
- ▶ There are many intermediate compromises.

# Efficiency

- ▶ If you don't care about execution efficiency, then use the most reliable protocol (e.g., the web using TCP/IP) and emphasize programmer efficiency (e.g., XML).
- ▶ If you do care about execution efficiency, then implement only what you need, when you need it. But these will use more development time, and be less open solutions. Typically associated with UDP.

# *Messages*

- ▶ When the basic computing entity is a (possibly remote) set of processes, the essential mode of communication is through *messages* from one process to another, or from process to many others.
- ▶ But the sending of messages can be problematic, especially the sending of remote messages.

# Some problems with messages

- ▶ Receiver's receive buffer is full
- ▶ Sender can't wait, but the send buffer is full
- ▶ The message gets lost
- ▶ The acknowledgment (*ack*) gets lost
- ▶ The message is delayed.
- ▶ The messages arrive in the wrong order
- ▶ A process joins a group, but the broadcasting process doesn't know it
- ▶ Coordinating messages cross in transit.

# RPC / RMI / etc. → MOM

## Problems using send/receive

- ▶ Sender and receiver have to be active
- ▶ Sender and receiver have to know each other's address or *endpoint*
- ▶ Buffering concerns must always be considered.
- ▶ Push toward MOMs that handle some of these problems.
- ▶ Trade control and efficiency for convenience

# Coordination of Processes

- ▶ Messages are the coordination mechanism for cooperating processes.
- ▶ Some coordination problems occur over and over again in many types of distributed applications.
- ▶ Many of these problems have been formalized, and have formal solutions.
- ▶ Blocking, buffering, and the reliability of the message protocol are among the concerns.

# Producer / Consumer Coordination

Original Problem on single system using IPC:

- ▶ *Producer* writes until catches up with *Consumer* then blocks
- ▶ *Consumer* reads until catches up with *Producer* then blocks
- ▶ Assume a circular buffer.
- ▶ Concurrent code?

# Traditional producer / consumer solution with circular buffer and shared memory pointers via IPC

Writer

```
Shared buffer* ReadPtr, WritePtr;

while (TRUE) {

    // Wait for reader:
    while (WritePtr == ReadPtr) ;

    buffer[WritePtr] = produce();
    WritePtr++;
    if (WritePtr == BUF_END) { // loop back
        WritePtr = BUF_START;
    }
}
```

Reader

```
Shared buffer* ReadPtr, WritePtr;

while (TRUE) {

    // Wait for writer:
    while (ReadPtr == WritePtr);
    consume(buffer[ReadPtr]);
    ReadPtr++;
    if (ReadPtr == BUF_END) { // loop back
        ReadPtr = BUF_START;
    }
}
```

**ReadPtr and WritePtr are shared memory via IPC**

# Producer / Consumer Coordination in a distributed system.

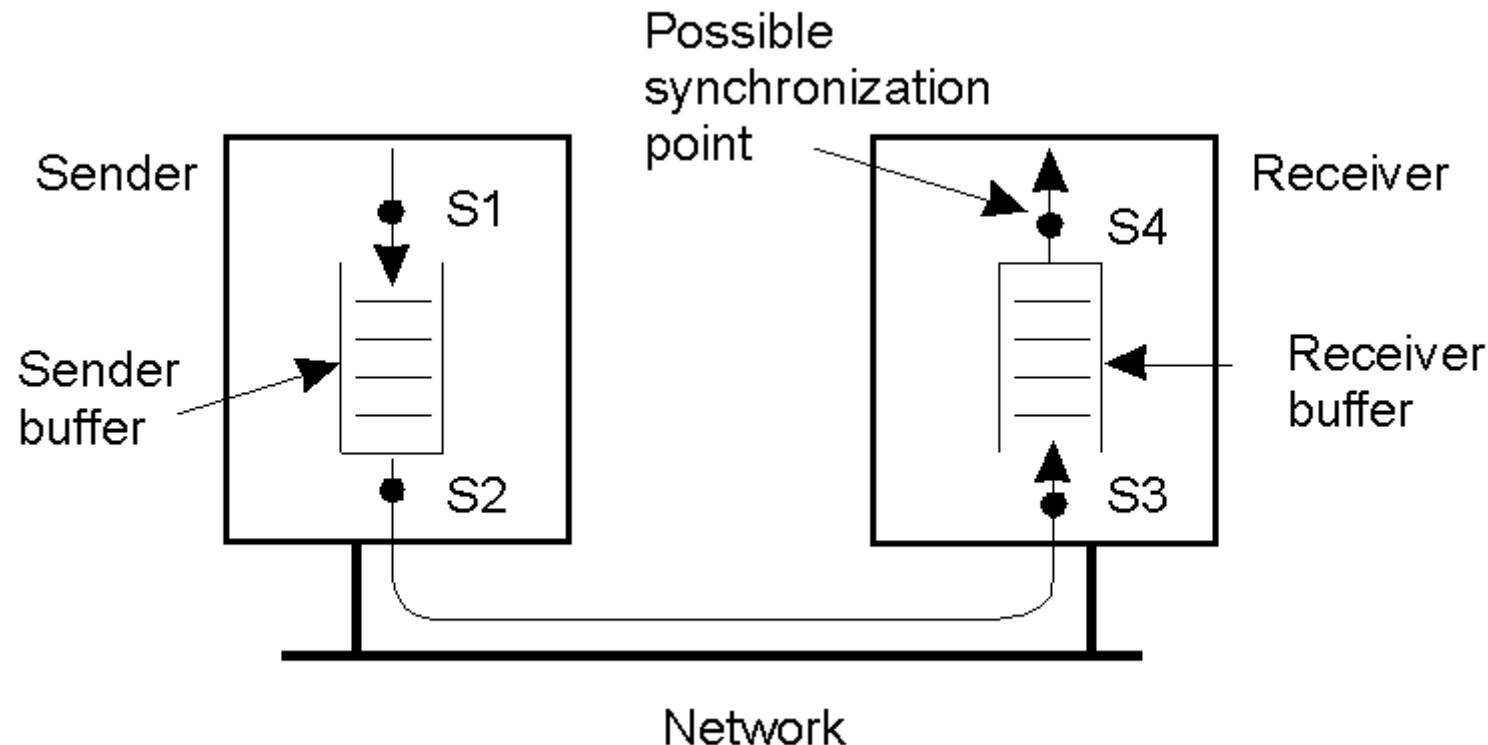
- ▶ Problem is that *ReadPtr* and *WritePtr* do not live in shared memory so cannot be set/read by the other system.
- ▶ Producer (on one system) doesn't know when to stop producing. Consumer (on a different system) doesn't know when to stop reading.
- ▶ So where do you coordinate your reading and writing?

# Coordinating for Distributed System:

- Block sender until send buffer not full
- Block sender until local subsystem sends the message
- Block sender until remote subsystem receives the message and sends ack.
- Block sender until consumer application receives message and sends ack.
- Block the sender until the application actually starts processing the request and sends the ack.
- Etc.

# Process coordination w/ msgs

AI



# Asynchronous calls—Many styles!

- ▶ There is *one* kind of blocking (synchronous) call: send the message, then block waiting until you get a return message.
- ▶ There are *many* kinds of asynchronous calls.
  - Minimally, send and forget... move on immediately.
  - Maximally, wait until the server application actually starts work on the request.
  - Every variation in between.

# Asynchronous calls—complex

- ▶ In addition to all the complexities of determining when you release the caller to go about its work...
- ▶ When the result is ready on the server, you must now interrupt work back on the client to receive it.
- ▶ Typically the client must maintain a “results server” to receive the result into a buffer. The client has to check periodically and branch to handle it.

# False assumptions made by first time Distributed Systems developers:

- ▶ 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.

# Transparency—programmers vs. users

- ▶ As programmers/designers we love to think about computer code and quirky design issues. We want to see everything. But users don't.
- ▶ Example:
  - There are lots of interesting IT problems and algorithms to solve in the software that runs a tin can manufacturing company.
  - Users just want to make the cans.
- ▶ This is the motivation behind transparency in distributed systems. Think of “transparent” as meaning “the user can't see it.”

# Transparency—in a perfect world

- ▶ Access – local and remote access looks the same
- ▶ Location – users do not need to bother where a resource is (or do they?)
- ▶ Concurrency – should not be concerned who else needs the shared resource. Record locking for update?
- ▶ Replication – duplicate data to increase performance or reliability, but hide management of it. (But be careful because this does not match the semantics of the real world!)

- ▶ Failure – conceal failures from users. (But sometimes users need to know what is going on!)
- ▶ Mobility – processes, hardware (esp. mobile), users, clients, services, servers all migrate without concern about updating users.
- ▶ Performance – reconfigure, add hardware, without notification
- ▶ Scale up – Grow without changing interfaces

# Transparency not always good

- ▶ Sometimes users need knowledge about underlying difficulties so they can make intelligent decisions.
- ▶ Case 1: *Do users care about bandwidth details?*
  - Your LINE or SKYPE video call keeps getting dropped because of a low bandwidth condition
  - You switch to voice only

## ► Case 2: *Do users care about location of the software?*

- You manage truck routes for San Francisco with software that uses the local weather as input.
- ...but you travel to Amsterdam for meetings. The software uses the local Amsterdam weather to route trucks in San Francisco.

# Electronic Health Care Systems (1)

Questions to be addressed for health care systems:

- ▶ Where and how should monitored data be stored?
- ▶ How can we prevent loss of crucial data?
- ▶ What infrastructure is needed to generate and propagate alerts?
- ▶ How can physicians provide online feedback?
- ▶ How can extreme robustness of the monitoring system be realized?
- ▶ What are the security issues and how can the proper policies be enforced?

# Electronic Health Care Systems (2)

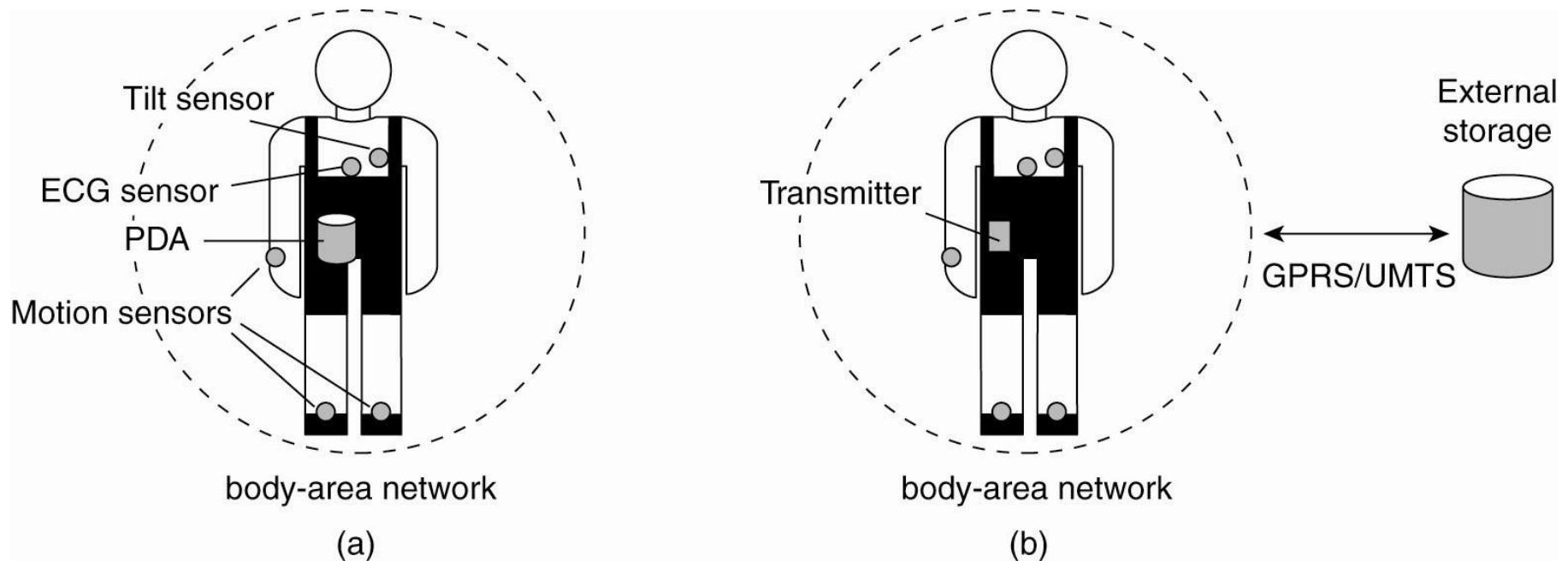


Figure 1-12. Monitoring a person in a pervasive electronic health care system,  
using (a) a local hub or  
(b) a continuous wireless connection.

# Sensor Networks (1)

Questions concerning sensor networks:

- ▶ How do we (dynamically) set up an efficient tree in a sensor network?
- ▶ How does aggregation of results take place? Can it be controlled?
- ▶ What happens when network links fail?

# Sensor Networks (2)

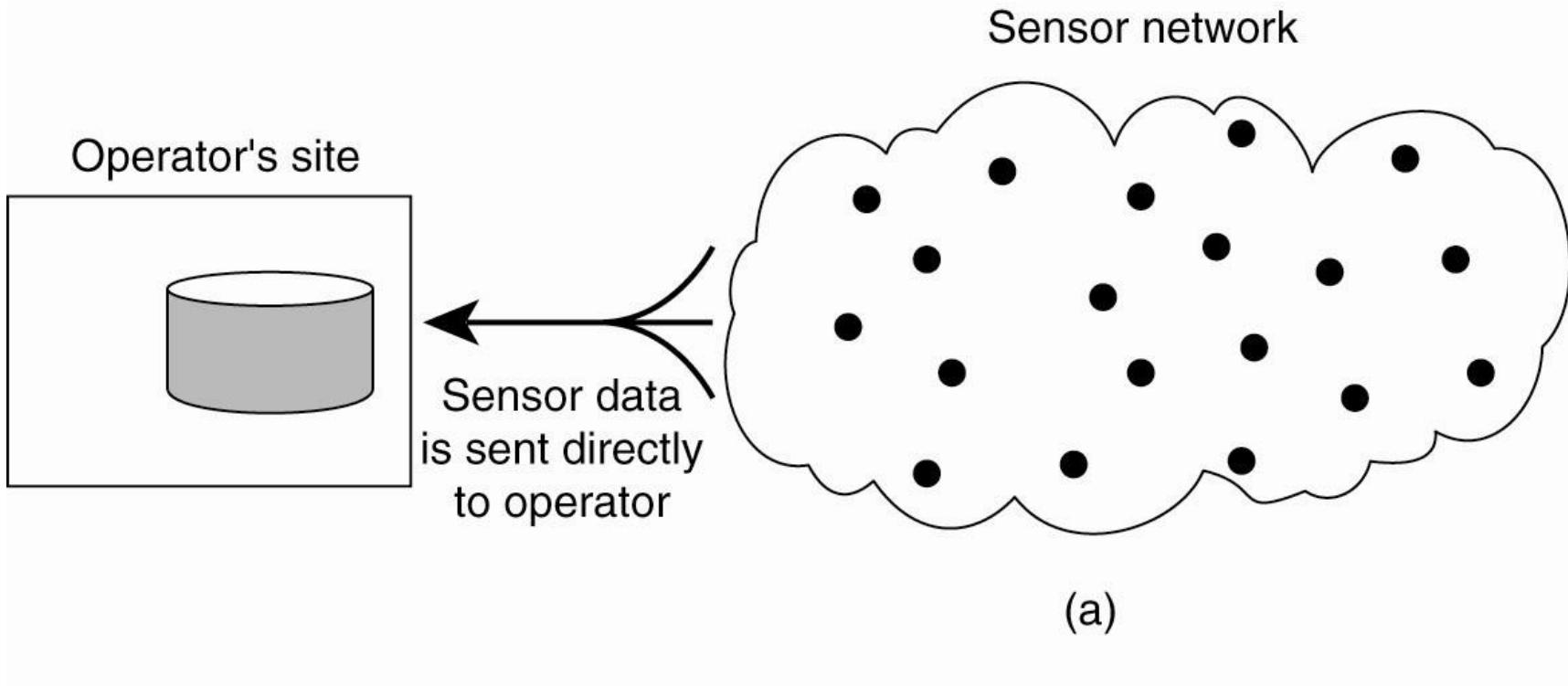


Figure 1-13. Organizing a sensor network database, while storing and processing data (a) only at the operator's site or ...

# Sensor Networks (3)

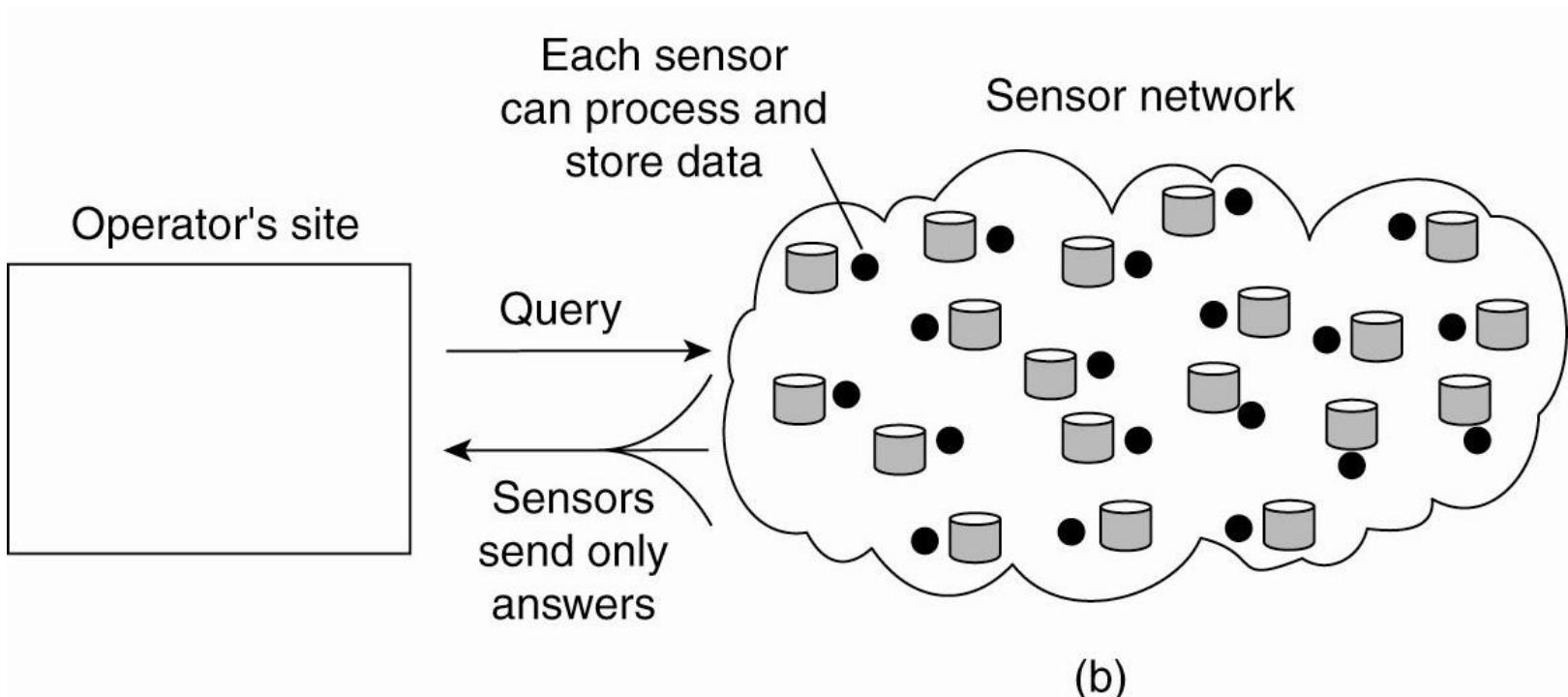


Figure 1-13. Organizing a sensor network database, while storing and processing data . . . or (b) only at the sensors.

# Scaling Techniques

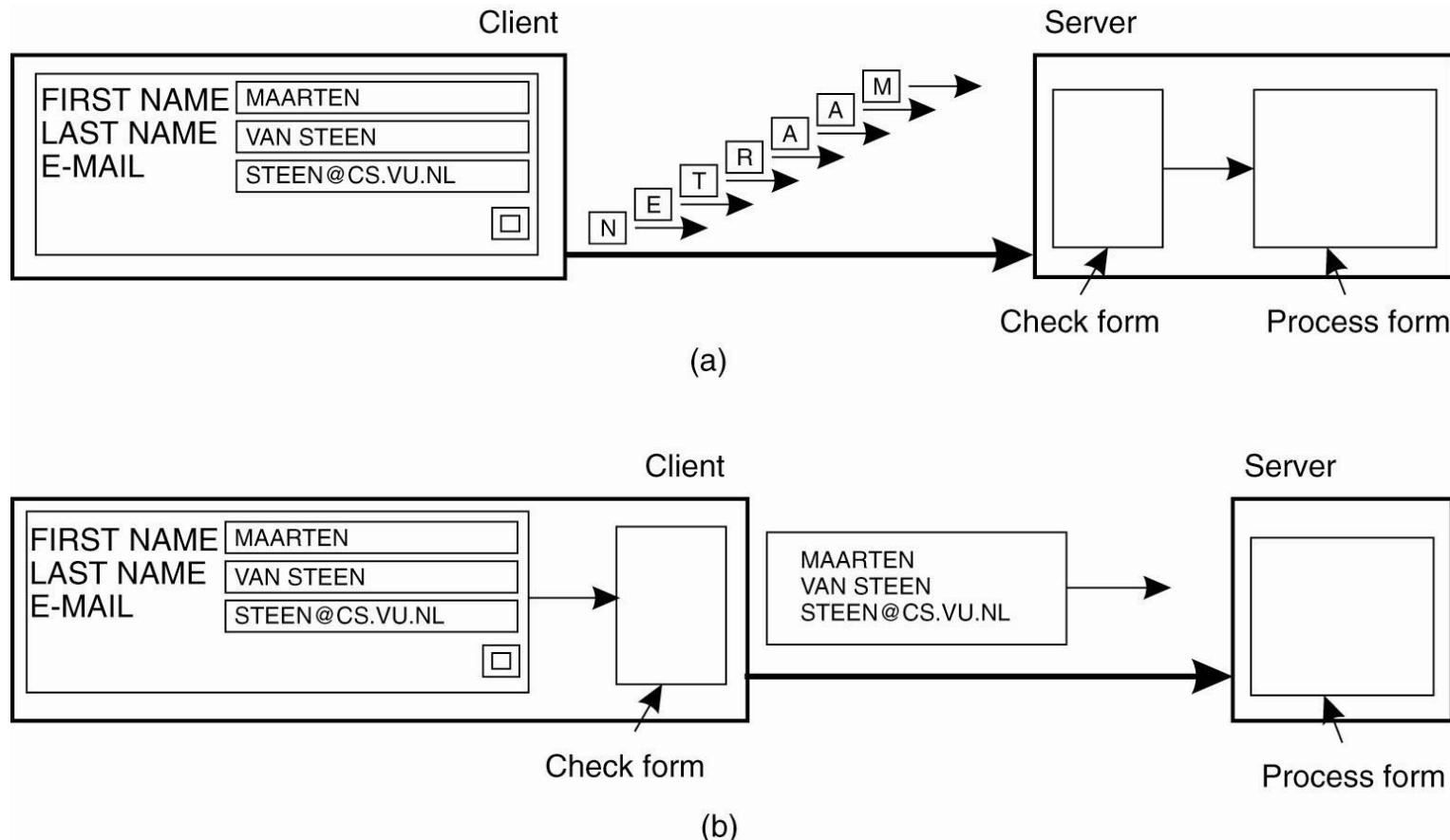


Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

# Example: Security tradeoffs thick/thin client

- ▶ Thin: More packets, more exposure and more chance for profiling sender/receiver patterns.
- ▶ Thick: more code exposed on the client, more chance for out-of-date, insecure client code.
- ▶ We will revisit thick and thin clients later.

# Not Used

# Clients and servers

- ▶ Used to cover a broad range of actual designs and activities
- ▶ General: *invoke an operation* means to make a request from one computer for service which is performed on another computer.
- ▶ Two processes connected over network via message passing.
- ▶ Web clients (browser) and server are ex.

# Components of client server

- ▶ Clients and servers are *processes* (a program instance in execution)
- ▶ Server machines run at least some servers and clients machines run at least some clients. Many have both.
- ▶ A server process may also be a client process to some other server process
- ▶ Communication via messages
- ▶ Services (access, update, manage resources) present interfaces and work through servers.

# Brief tour of the web

- ▶ There was internet before the web!
- ▶ Weak design from CS perspective: bad protocol (http ON TCP/IP?!), not efficient, not secure
- ▶ Why did it succeed?
- ▶ Open system
- ▶ SIMPLE! EASY TO USE!
- ▶ Filled huge need to interactively connect data stores to client users via formatted text, with INPUT boxes.
- ▶ Hyper-linked documents and servers

# Brief tour / web

- ▶ High school kid with an afternoon free can now allow shoes in Kansas to be sold in Tokyo.
- ▶ Gopher system: just document retrieval, always
- ▶ HTTP connectionless, built on top of TCP connection-maintaining...?!
- ▶ Each request a new process – secure, simple, wasteful but who cares?
- ▶ CGI: Stdin, stdout, stderr, Environment vars.
- ▶ URL: global IDs for any kind of content

# Web is OPEN

- ▶ Protocol (http), markup conventions (ascii html), program interface (cgi) are all simple and well defined: any hardware, any OpSys, any language, any editor.
- ▶ Almost any content can be shared: MIME typing, distribute *handlers* for any type of data.

# HTML

- ▶ Just another markup language
- ▶ Used to just tell students to learn in on their own the first week.
- ▶ Write your own once or twice! Use an editor and put some content on the web

# URL

- ▶ Uniform [Universal] Resource Locator [Identifier]
- ▶ Scheme-or-protocol://server id[:port]/  
*application-specific-string*
- ▶ Scheme is OPEN. Typical http ftp mailto
- ▶ Server-id[:port] are bound to internet, DNS or IP
- ▶ Remaining string has conventions, but are under  
*full program control of server process.*

# URL

- ▶ URLs can also be used as universally unique identifiers (UUIDs) because they map to exactly ONE physical location on earth, and there are conventions about who controls that location.
- ▶ <http://condor.depaul.edu/~elliott/435/abc.html>
- ▶ 140.192.175.100 is the physical network card
- ▶ /users/faculty/elliott/public\_html/435/abc.html is the physical file on disk.
- ▶ Controlled by Elliott.

# HTTP dynamic pages

- ▶ *Application-specific-string* is interpreted by the server application listening at the port
- ▶ Pages can have embedded forms that send query strings and A/V (attribute / value) pairs as part of the request string. *By convention* these A/V pairs are stored in the string following the “/”

# Common Gateway Interface

- ▶ What made the web work in the first place
- ▶ Description of simple interface to any program that runs natively in a typical OpSys shell
- ▶ Shell programs, C programs, Perl programs, (but not java programs!) all have input and output and read from the environment.

# Example CGI input form

- ▶ <html>
- ▶ <head><TITLE> DS420 Sample Form for AddNum </TITLE></head>
- ▶ <BODY>
- ▶ <H1> Addnum </H1>
  
- ▶ <FORM method="GET" action="http://localhost:2566/cgi/addnums.fake-cgi">
  
- ▶ Enter your name and two numbers:
  
- ▶ <INPUT TYPE="text" NAME="person" size=20 value="YourName"><P>
  
- ▶ <INPUT TYPE="text" NAME="num1" size=5 value="4"> <br>
- ▶ <INPUT TYPE="text" NAME="num2" size=5 value="5"> <br>
  
- ▶ <INPUT TYPE="submit" VALUE="Submit Numbers">
  
- ▶ </FORM> </BODY></html>

# CGI conventions

- ▶ Stdin, stdout, std err – streams to any executable program which is running as forked subprocess to the calling (server) process. Get redirected back to the server.
- ▶ Print to the console. Read from the console. Print error message to the console.

# CGI conventions

- ▶ Environment variables, including argv[] are r/w by server and r/w by cgi program
- ▶ Arge[]

# Environment vars

- ▶ WINDOWS: c:\>env
  - PROMPT=\$P\$G
  - CLIENTNAME=Console
  - COMPUTERNAME=ELLIOTTPC
  - EM\_PARENT\_PROCESS\_ID=2876
  - HOMEDRIVE=C:
  - HOMEPATH=\Documents and Settings\elliott.CSTCIS [...]
- ▶ UNIX: Elliott2> env
  - LOGNAME=elliott2
  - MACHTYPE=i386
  - VENDOR=intel
  - OSTYPE=linux [...]

# Arguments

- ▶ *Print MyFile.txt –orientation landscape*
- ▶ Runs print.exe somewhere
- ▶ Argv[0] points to the string “*print.txt*”
- ▶ Argv[1] points to “*MyFile.txt*”
- ▶ Argv[2] points to “*–orientation*”
- ▶ Argv[3] points to “*landscape*”

# Fork() off CGI process

- ▶ fork() - system call which creates a new process which is identical to the current process.
- ▶ Returns 0 to the newly created child process
- ▶ Returns child's process id [PID] to the parent (original) process.

# Fork() off CGI process

- ▶ PID = fork();
- ▶ If (PID == 0) then ...
  - [CGI program invocation code]
  - Set up stream management for stdin, stdout, stderr
  - set up environment vars
  - exec(*cgi-program.exe [args]*)
- ▶ Else...
  - server management code for subprocesses using the PID value

# Openness

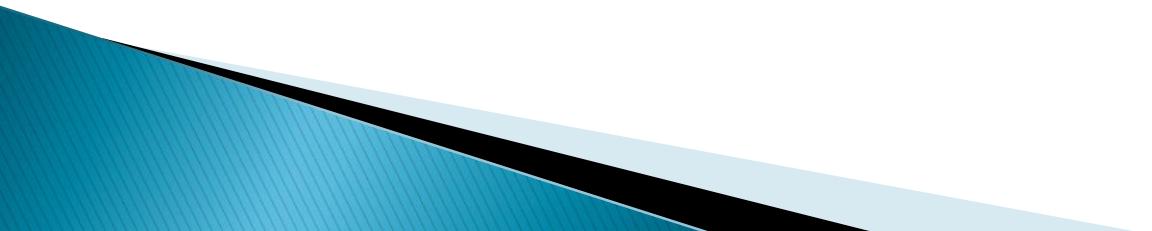
- ▶ Drives the web
  - ▶ Publish interfaces as de facto standards
  - ▶ Standards really usually driven by industry
  - ▶ Industry has its own agenda – publish the interfaces? Diverse developers interoperate.
  - ▶ RFCs are requests for comments from the community at large, and allow a very democratic participation in development of standards.
  - ▶ New services, and extensions of old, are easy.
  - ▶ The rise of *Middleware*

# Security

- ▶ Its all on the network lines now
- ▶ World-wide network of evil reaching into your pocket.
- ▶ Security has been part of DS from its inception.
- ▶ No more PW protected file system and address space for processes.
- ▶ Attacks on infrastructure: DOS, IP spoofing, certificate irregularities, cache poison, bad record management, etc.

# Blank





**DISTRIBUTED SYSTEMS**  
**Principles and Paradigms**

Second Edition

ANDREW S. TANENBAUM  
MAARTEN VAN STEEN

**Chapter 2**  
**ARCHITECTURES**  
**Elliott mods in Cambria**

# Architectural Styles (1)

Important styles of architecture for distributed systems

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

# Architectural Styles (2)

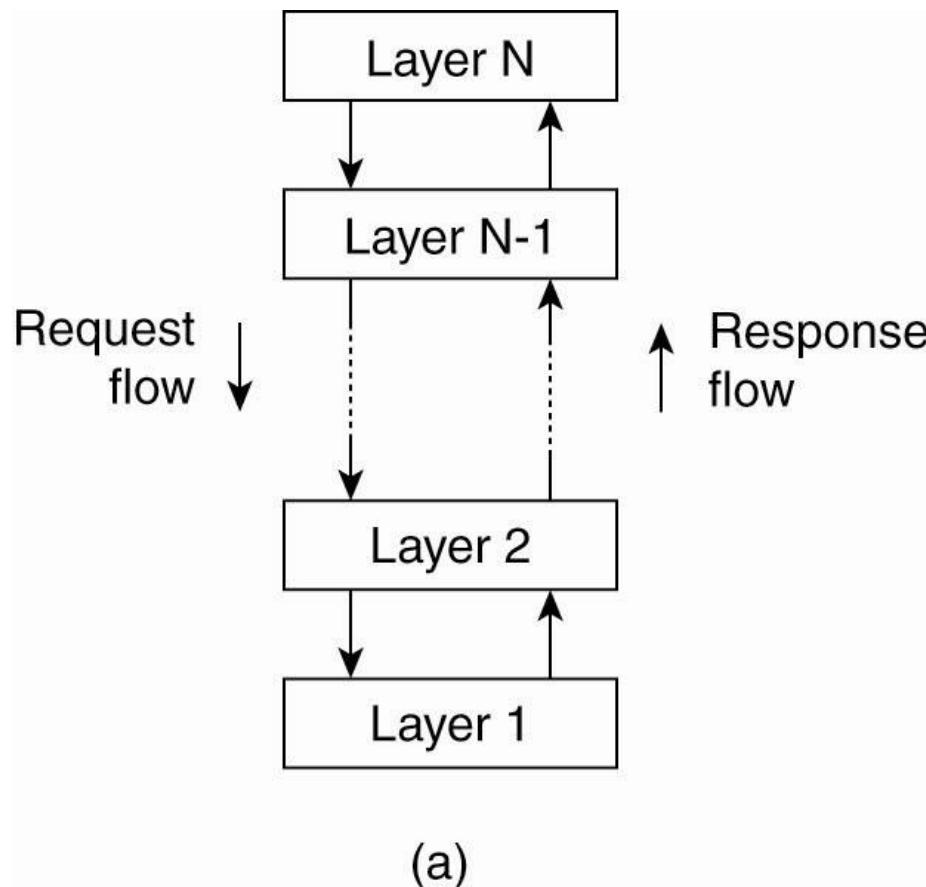
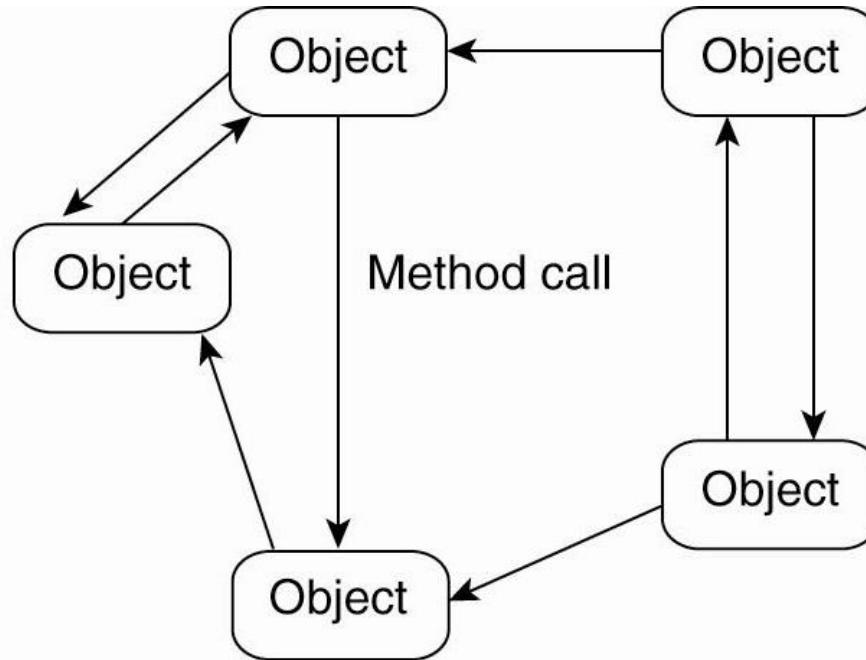


Figure 2-1. The (a) layered architectural style and ...

# Architectural Styles (3)



(b)

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

# Architectural Styles (4)

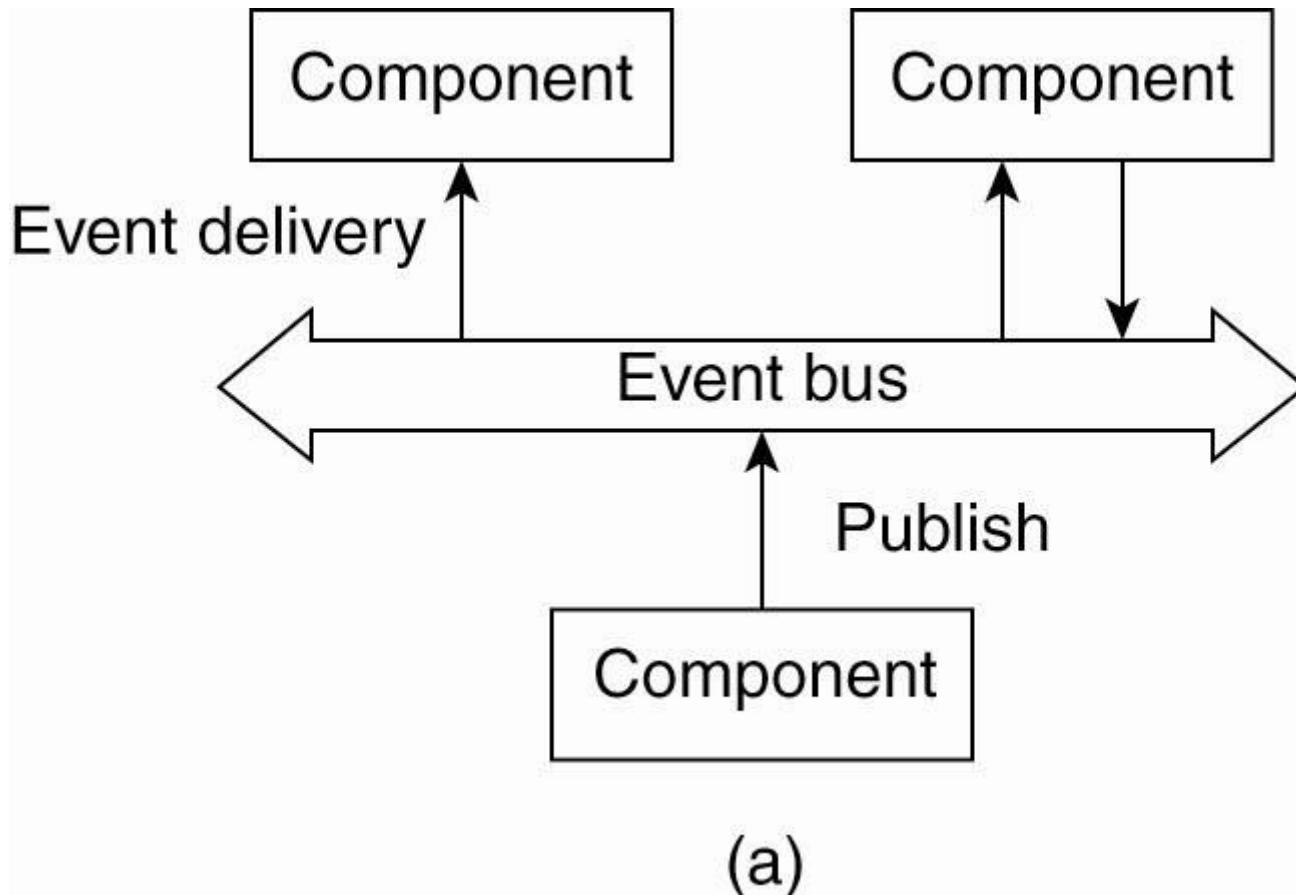


Figure 2-2. (a) The event-based architectural style and ...

# Publish / subscribe

Publish to, and subscribe from, the event bus

- Processes *publish* events, and the middleware arranges that only those processes that *subscribed* to that type of event receive them
- Processes are *referentially decoupled* which means that they do not need to know about each other.

# Architectural Styles (5)

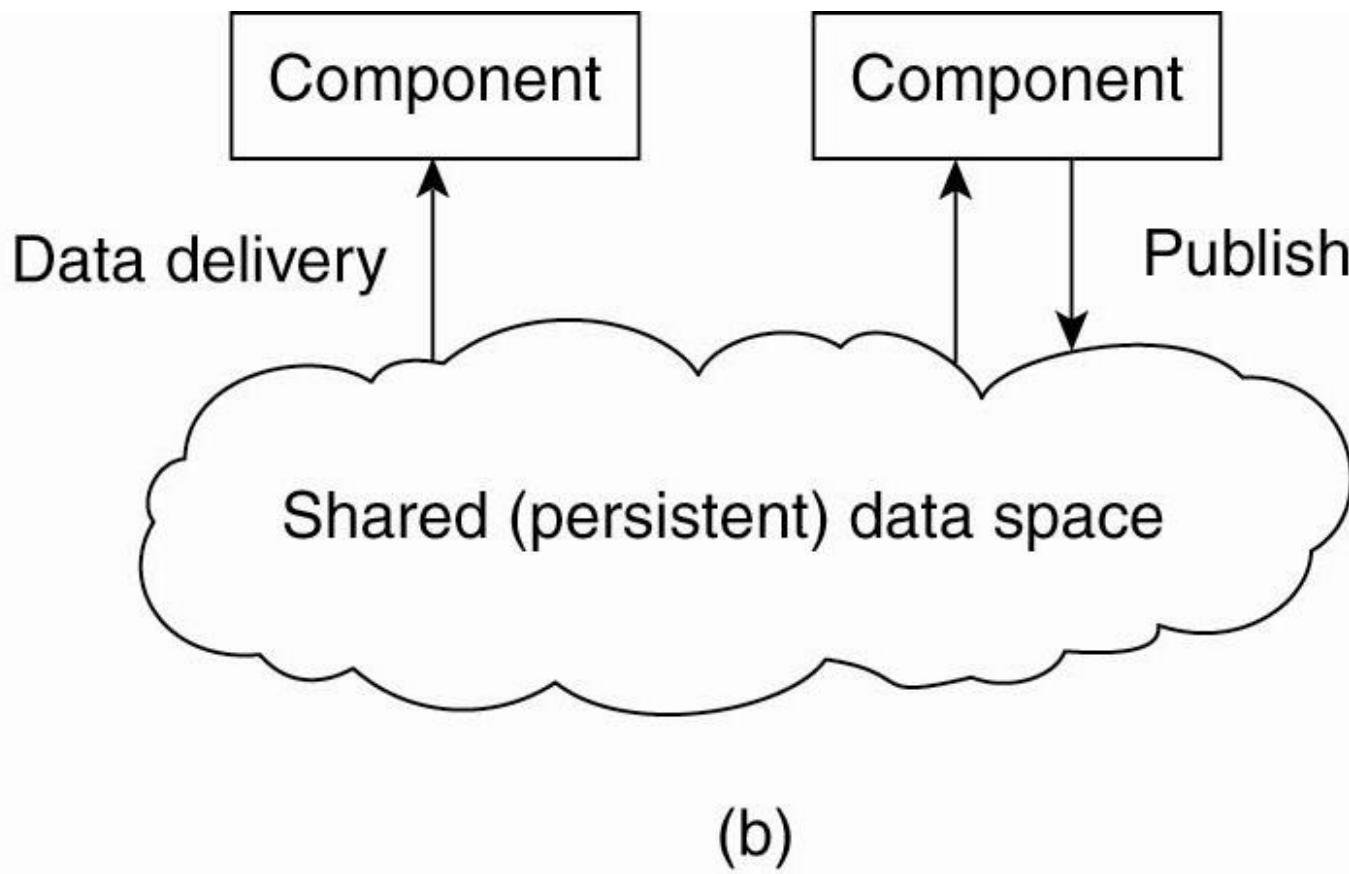


Figure 2-2. (b) The shared data-space architectural style.

# Publish / subscribe across time

Publish to, and subscribe from, the persistent data space

- Processes *publish* events, and the middleware arranges that only those processes that *subscribed* to that type of event receive them – but the processes can be running, or not, at any time
- Processes are *time decoupled* which means that the events (and data) must be stored in some persistent form.

# Data-centric intelligent DB

Can create intelligent databases using logic:

- Modus ponens:  $A \rightarrow B$  AND  $A$ , THEN  $B$
- Resolution:  $X \vee Y$  AND  $\text{not}(Y) \vee Z$  THEN  $X \text{ or } Z$

Forward chaining database:

- Publish A // a fact
- Publish  $A \rightarrow B$  [ $\text{not}(A) \vee B$ ] // an implication

# Retrieve $B$ from intelligent DB

Use simple external code on top of the database:

Select  $*B*$  (pattern match) from *Implications* yields

- “not( $A$ )  $\vee$   $B$ ”
- [...] (all other implications with  $B$  in them)

Using: “not( $A$ )  $\vee$   $B$ ”

- Search for  $A$ ;
- If  $A$  is true, then report that  $B$  is true.

# Using *resolution* logic...

Forward chaining database:

- Publish  $X \vee Y$
- Publish  $\text{not}(Y) \vee Z$

Search for proof of  $X \vee Z$ :

- Retrieve terms with  $X$  or with  $Z$  in them.
- Using resolution, combine:
  - $X \vee Y$
  - $\text{Not}(Y) \vee Z$
- Yields  $X \vee Z$

# Event Bus architectures.

Referential decoupling

Temporal decoupling if persistence is implemented.

Add to your toolbox!



# Centralized Architectures

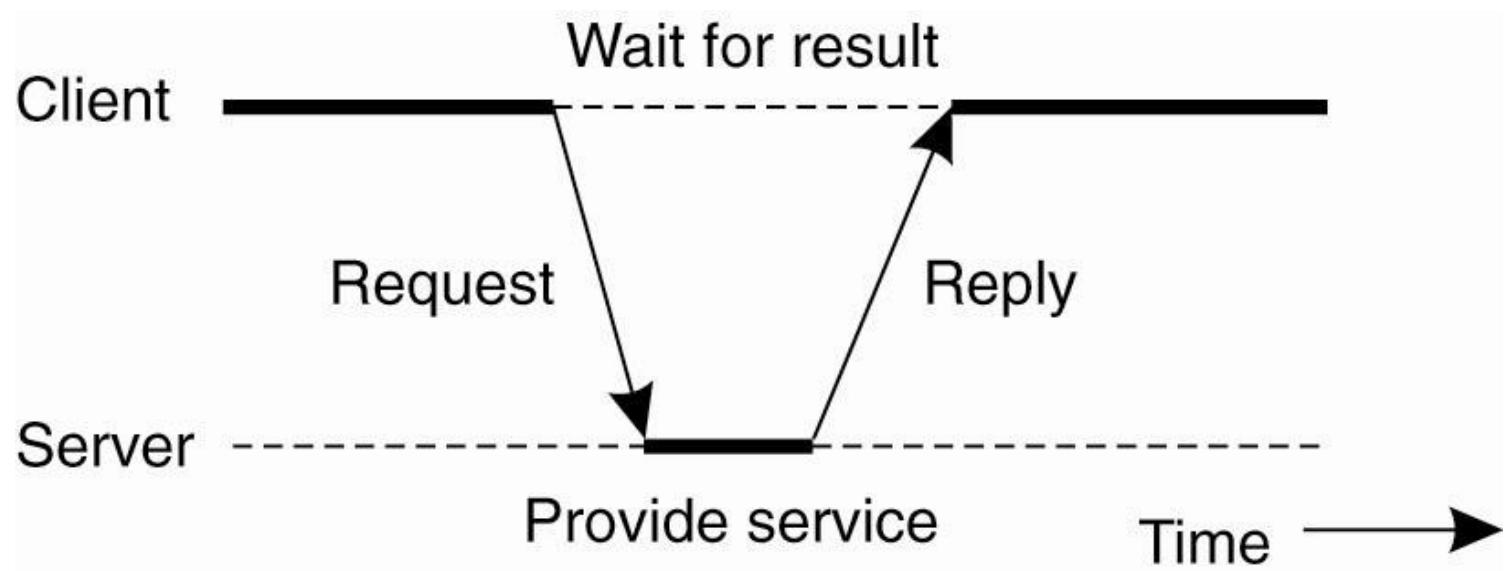


Figure 2-3. General interaction between a client and a server.

# Application Layering (1)

Recall previously mentioned layers of architectural style

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

# Application Layering (2)

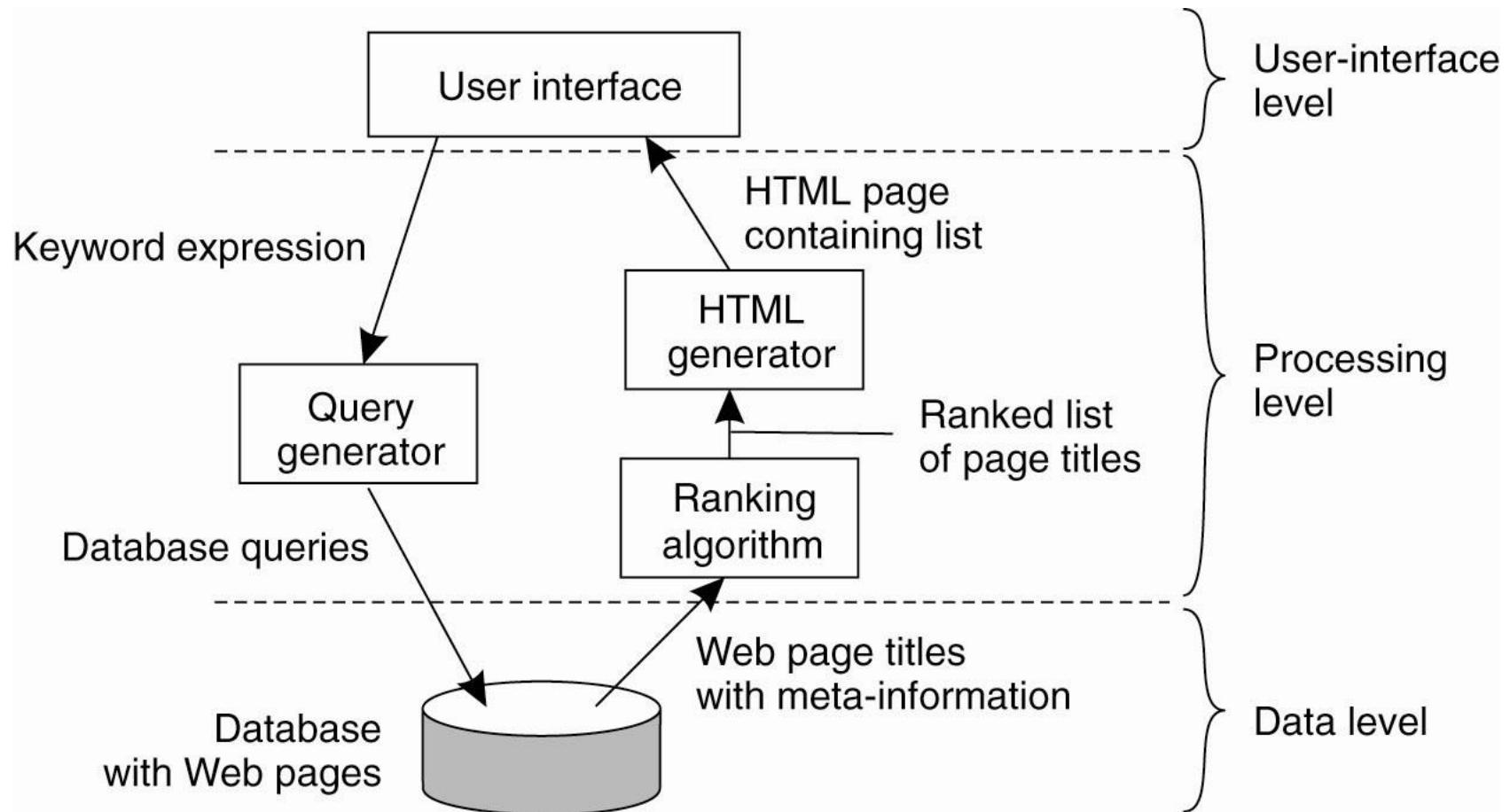


Figure 2-4. The simplified organization of an Internet search engine into three different layers.

# 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

# Scaling Techniques

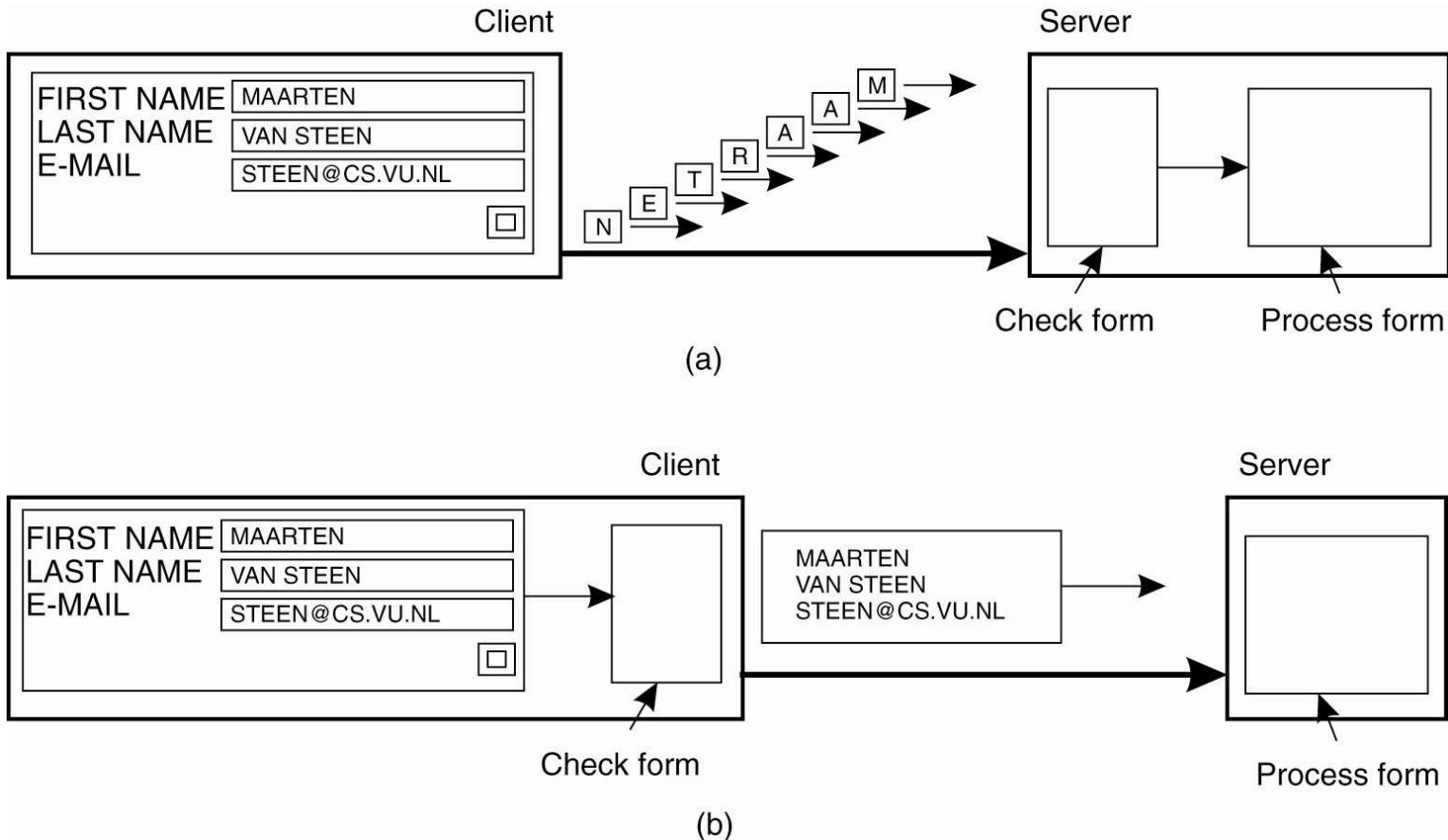


Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

# Example: Security tradeoffs thick/thin client

- Thin: More packets, more exposure and more chance for profiling sender/receiver patterns.
- Thick: more code exposed on the client, more chance for out-of-date, insecure client code.

# Multitiered Architectures (2)

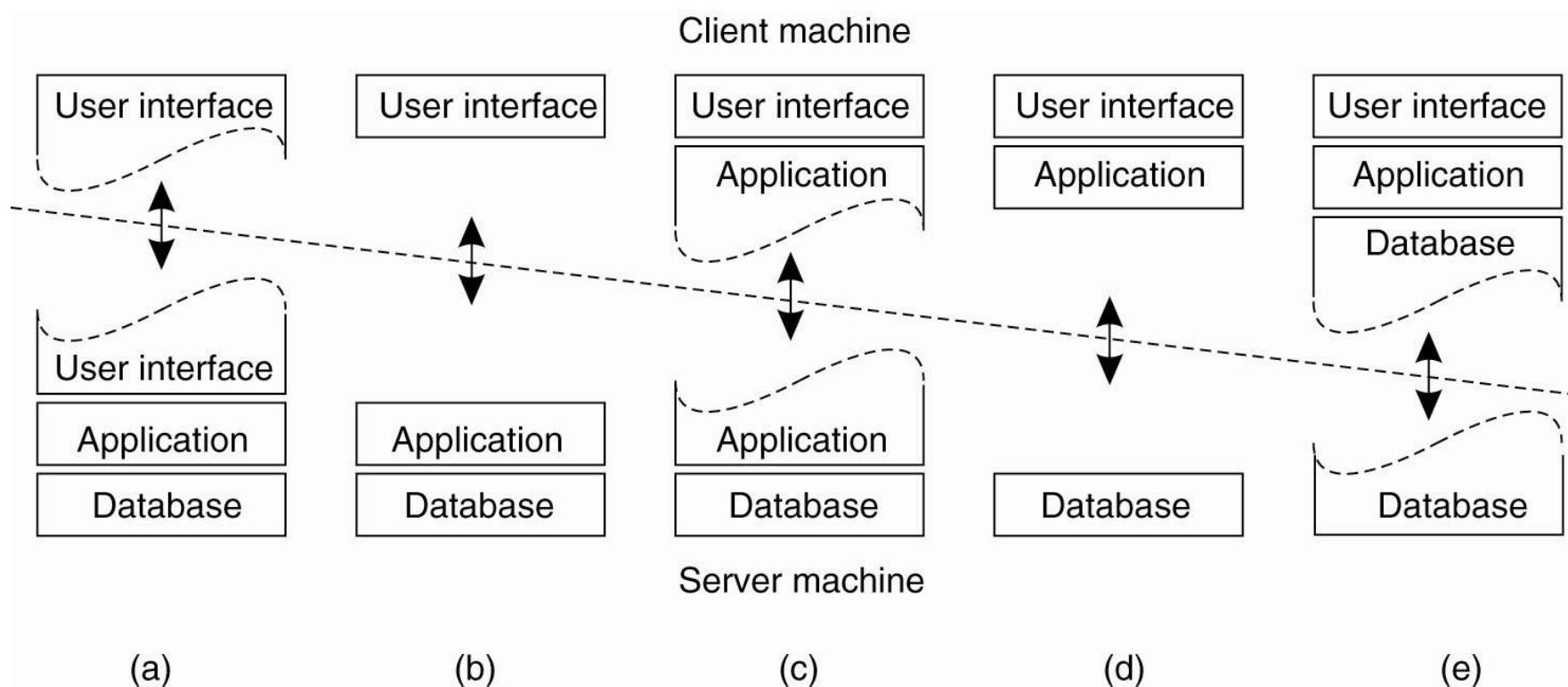


Figure 2-5. Alternative client-server organizations (a)–(e).

# Heavy Client

## Pro:

- Less network traffic
- Quick interactive response
- Handle many exceptions locally and gracefully
- Possibly fewer security issues to address
- Offload computational and data burden to client
- Offload data security problem to client

## Con:

- **Offline/push update nightmare!** ← !!!!!!!!!!!!!!!
- Code lives in many places at once, versions out of sync.
- More difficult to administer
- Possibly harder to code because of split application.
- System errors must be handled by the client

# Heavy Server

## Pro:

- Do not have to distribute updates!!!!)
- Simple to code because everything is in one place.
- Simple to maintain and monitor
- May have increased security from code managed on server
- Handle exceptions in one place

## Con:

- Lots of (inefficient) traffic over the network
- Slow interactive client response.
- Increased computational and data burden on server
- May have to ship large amounts of data from client
- Security challenges for sensitive client data(!)

# Thick Client / Thin Client

Add your expert knowledge of the pros and cons of thick and thin clients to your toolbox:



# Multitiered Architectures (3)

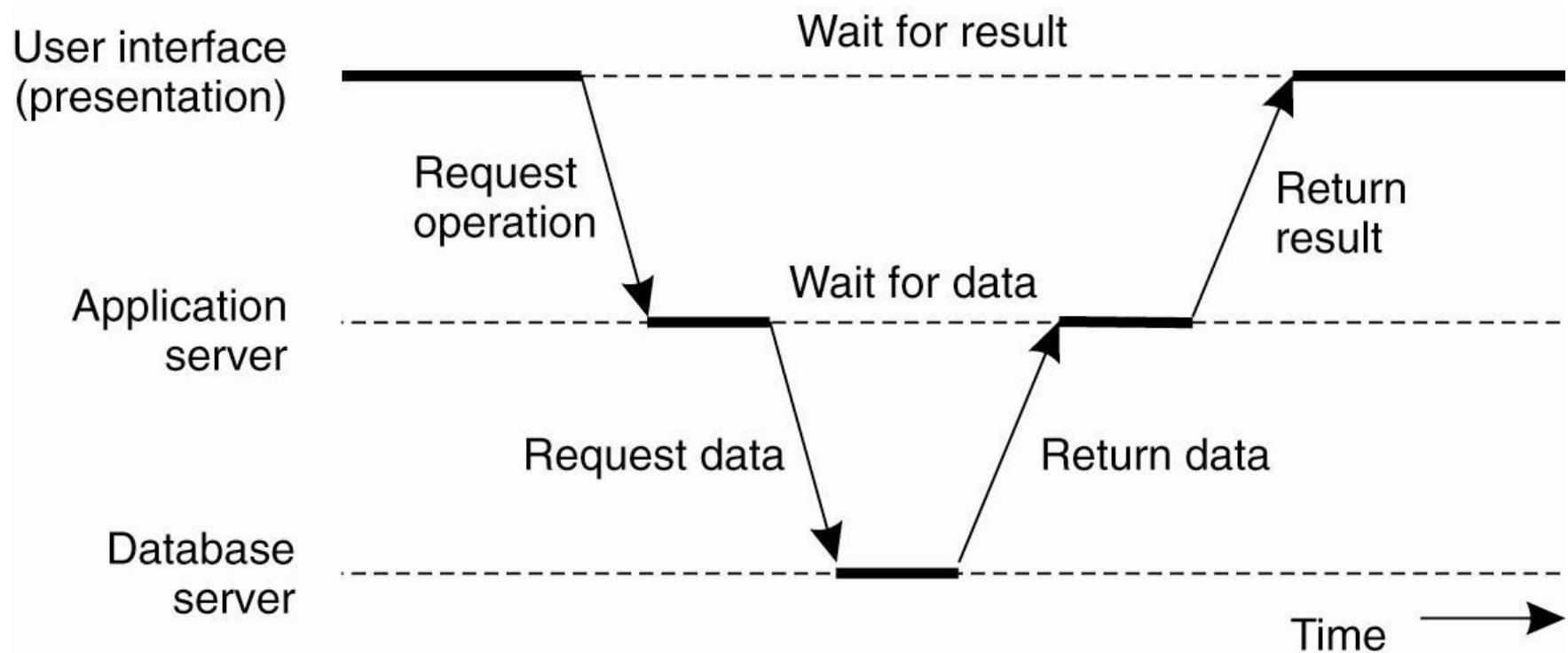


Figure 2-6. An example of a server acting as client.

# Different types of operations

When messages can be lost, there is a big difference between *idempotent* operations, and those that can only be executed one time.

“Add \$50,000 to Smith’s account” (only once)

“Return the current balance in Smith’s account”  
(idempotent, can repeat many times if thought lost)

# Structured Peer-to-Peer Architectures (1)

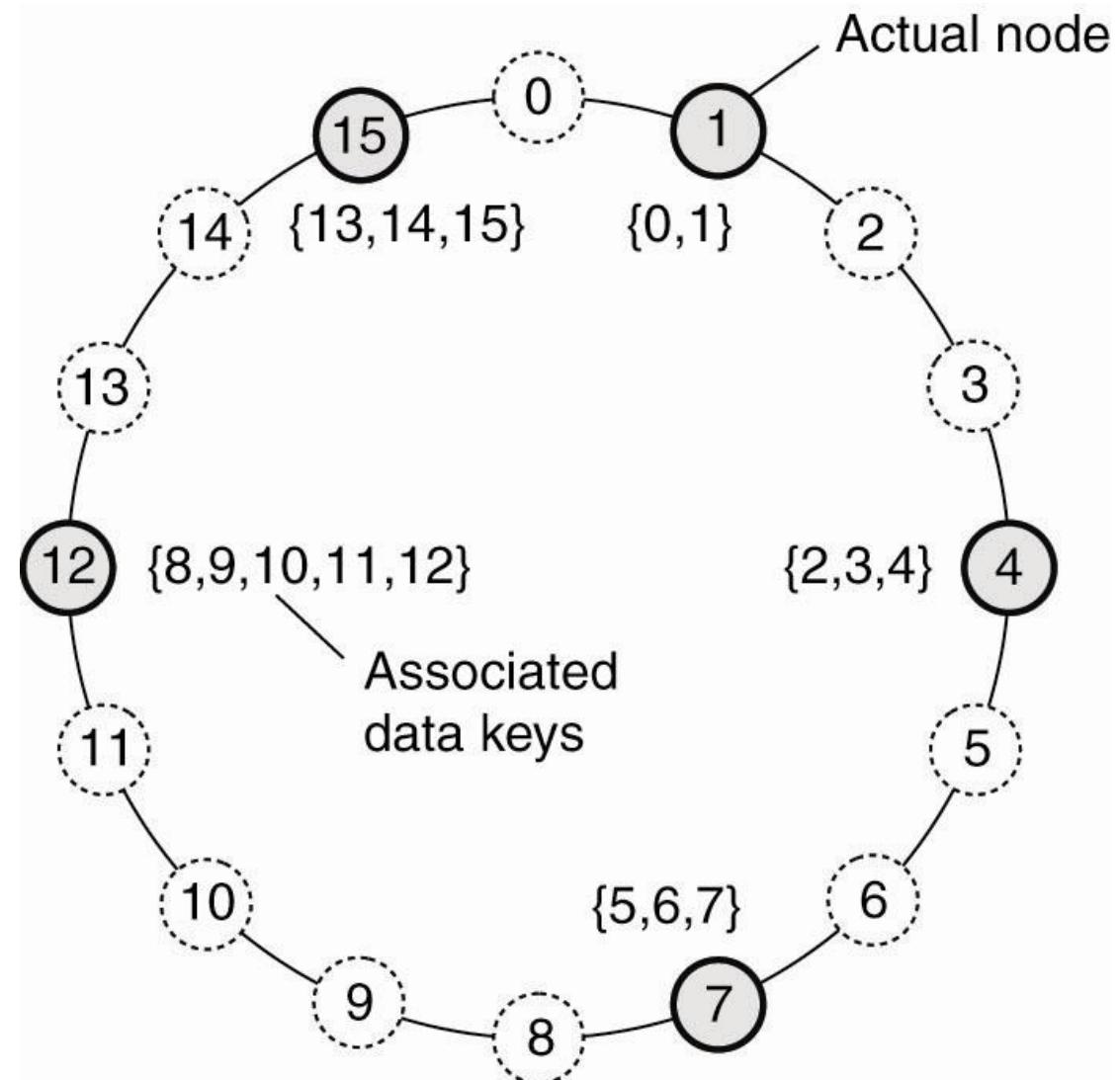


Figure 2-7. The mapping of data items onto nodes in Chord.

# Adding/removing a node from DHT

To add a node to a DHT (Distributed Hash Table)

- Use a RNG (random number generator) to produce some number in the correct key range—Chance of conflict is low
- Look up the next highest numbered node, make successor
- Insert new node in between predecessor and successor. Move data/processing code from successor as needed.
- Removal of node is the reverse, including moving data/processing code

# DHT dynamics

What makes DHT work:

- Deterministic scheme to map key of data item to identifier of some node to store it.
- Look up data by same key – get to node that has stored it
- Nodes can join and quit at will
- Logical organization independent of physical network
- General purpose: can support many kinds of applications.

# Rapid communication between nodes

Communication in DHT is *not* around the ring...

- Nodes maintain overlay network
- Actual location time is  $O(\log(N))$  where N is the number nodes.

For our examples we will use the simpler “around the ring” description because it is easier to visualize

# Distributed Hash Tables (DHT)

Dynamic distributed collaboration

Extremely powerful and flexible

Add these devices to your toolbox:



# Bit Torrent

- Primarily peer-to-peer with some centralized components.
- Both structured and unstructured.
  - Structured: .torrent files/trackers → centralized
  - Unstructured: regular nodes → de-centralized
- Nodes (computers) exist in *swarms*. For any particular file, at least one set of nodes in the swarm must have a complete copy of the file, and typically this is on a single node to start.

# Torrent Trackers

- .torrent files (fixed location) point to *trackers*
- Trackers (servers) for  $F$  keep lists of active nodes with file  $F$
- Download chunks of  $F$  from various nodes.
- Nodes upload and download the file simultaneously
- Two nodes can even trade pieces of  $F$

# Leechers and Seeders

- *Seeders* remain connected to the swarm even when they are not downloading from it
- *Leechers* don't allow upload, then just want to download. In general this is locally optimal.
- So, for these systems, reward for *good behavior* must be built into the design.

# Trackerless Torrents

- Often implemented as a DHT. Sometimes can also work alongside a traditional tracker system as a backup.
- Nodes discover one another and share information about the torrent.
- No central server.

# Bit Torrent

- Both structured and unstructured.
  - Structured: .torrent files/trackers → centralized
  - Unstructured: regular nodes → de-centralized
- .torrent files (fixed location) point to *trackers*
- Trackers (servers) for  $F$  keep lists of active nodes with file  $F$
- Download chunks of  $F$  from various nodes.
- Good behavior enforced: upload and download rates are considered
- Two nodes can even trade pieces of  $F$

# Bit torrent

Add this powerful and efficient file distribution design to your toolbox:



**DISTRIBUTED SYSTEMS**  
**Principles and Paradigms**

Second Edition  
ANDREW S. TANENBAUM  
MAARTEN VAN STEEN

**Chapter 3**  
**Processes**

# Thread Usage in Nondistributed Systems

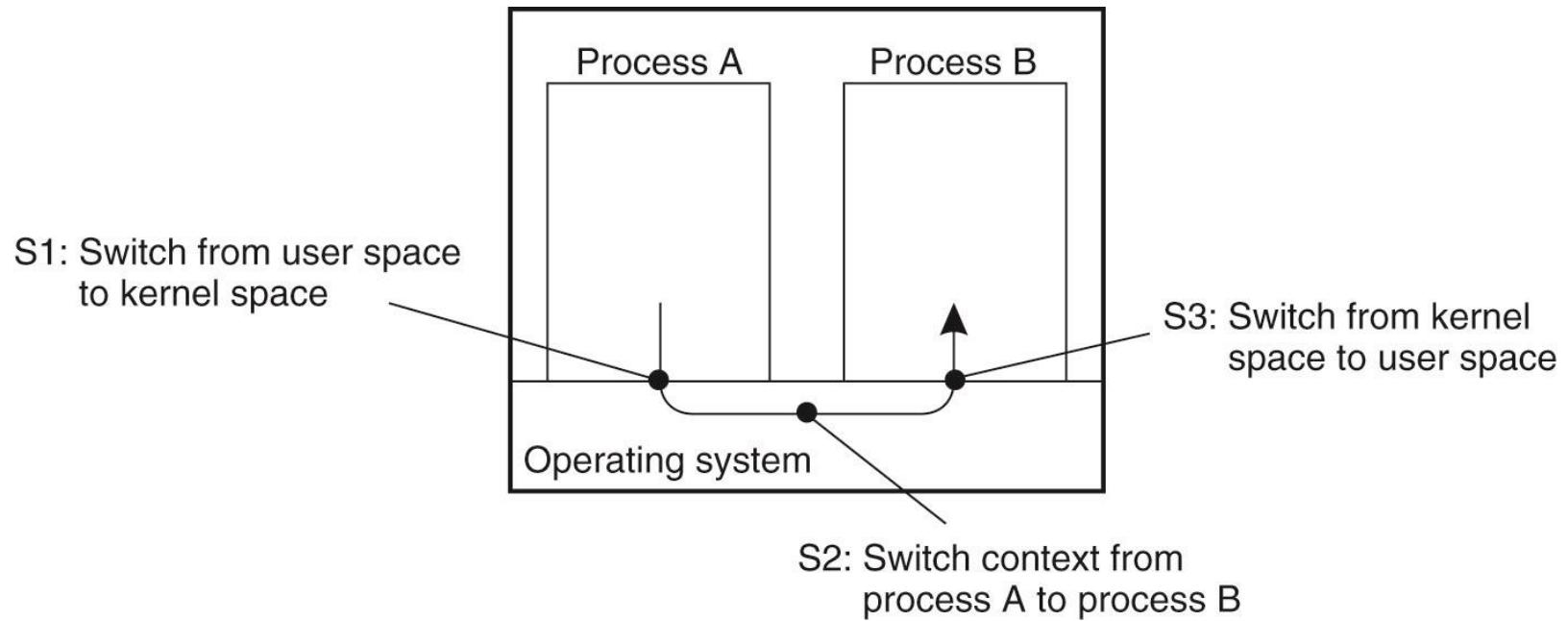


Figure 3-1. Context switching as the result of IPC.

# Thread Implementation

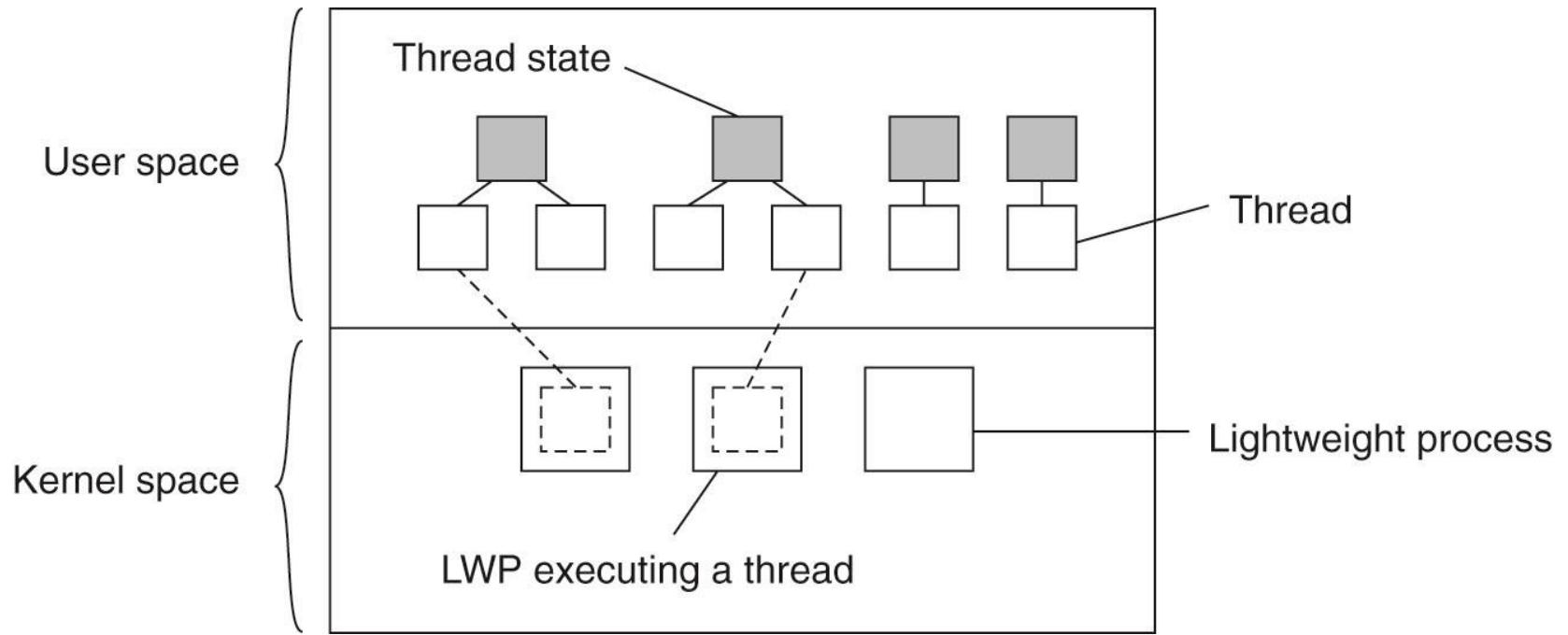


Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

# Multithreaded Servers (1)

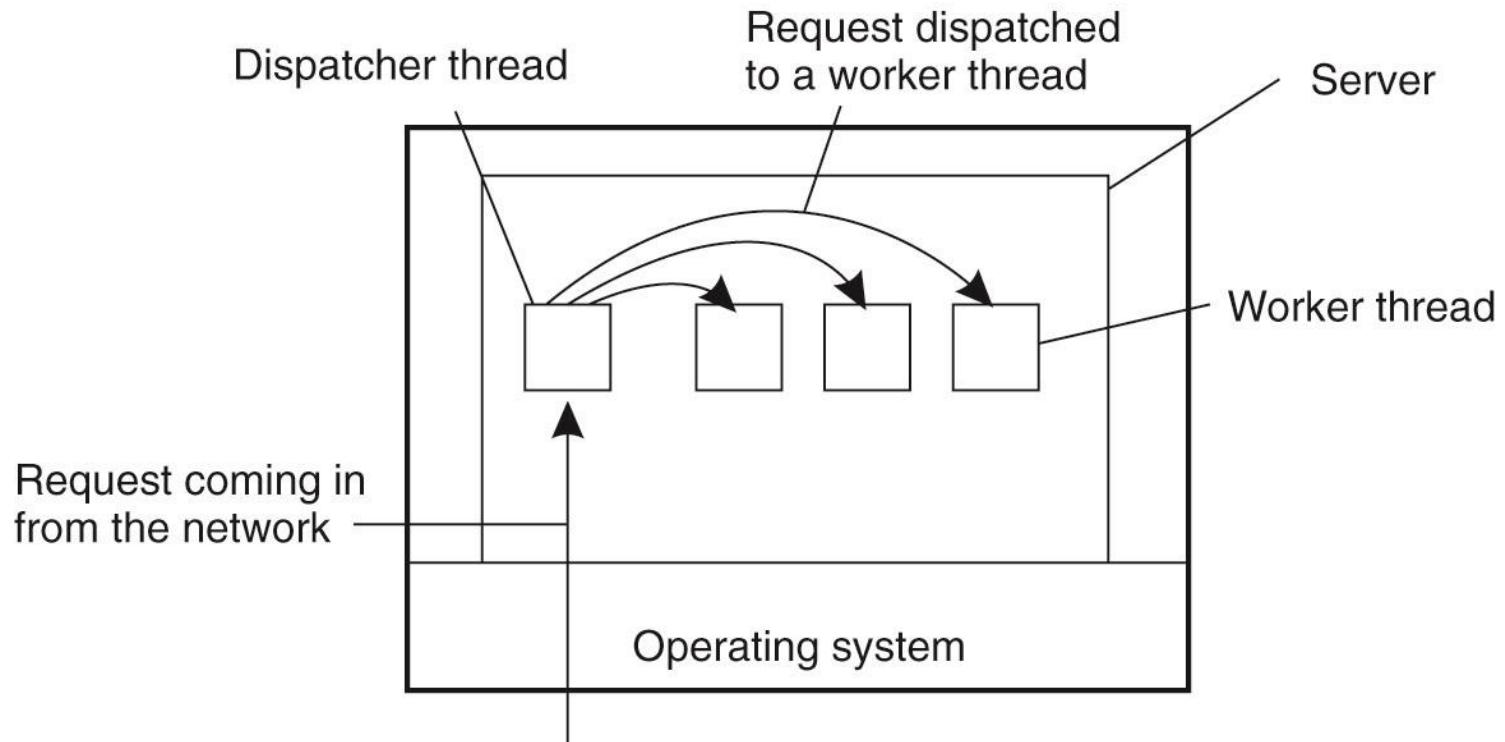


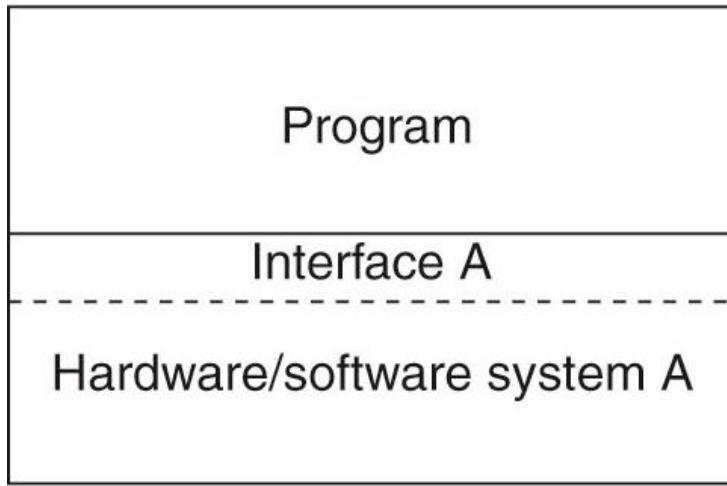
Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

# Publish / subscribe across time

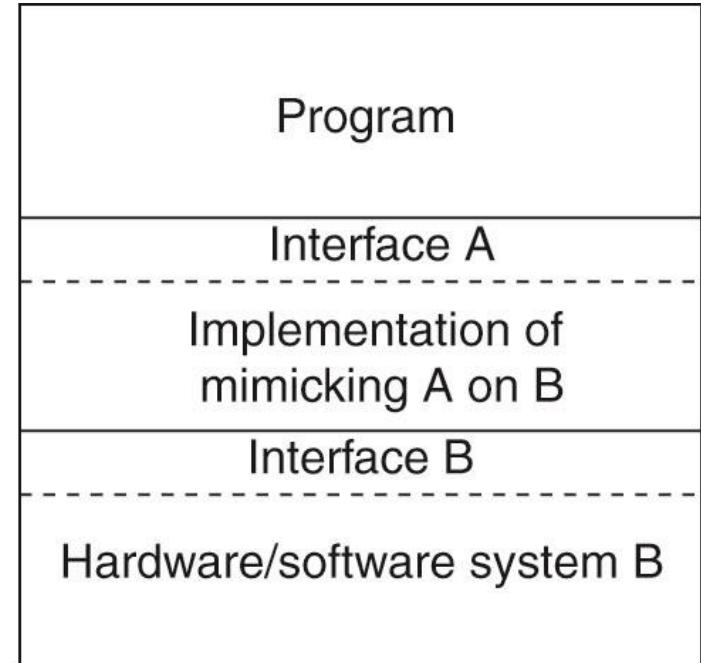
Publish to, and subscribe from, the persistent data space

- Processes *publish* events, and the middleware arranges that only those processes that *subscribed* to that type of event receive them – but the processes can be running, or not, at any time
- Processes are *time decoupled* which means that the events (and data) must be stored in some persistent form.

# The Role of Virtualization in Distributed Systems



(a)



(b)

Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

# Architectures of Virtual Machines (1)

Interfaces at different levels:

- [A] An interface between the hardware and software consisting of machine instructions:
  - that can be invoked by any program.
  - that can be invoked only by privileged programs, such as an operating system.

# Architectures of Virtual Machines (2)

- [B] An interface consisting of *system calls* as offered by an operating system.
- [C] An interface consisting of library calls
  - generally forming what is known as an application programming interface (API).
  - In many cases, the aforementioned system calls are hidden by an API.

# Architectures of Virtual Machines (3)

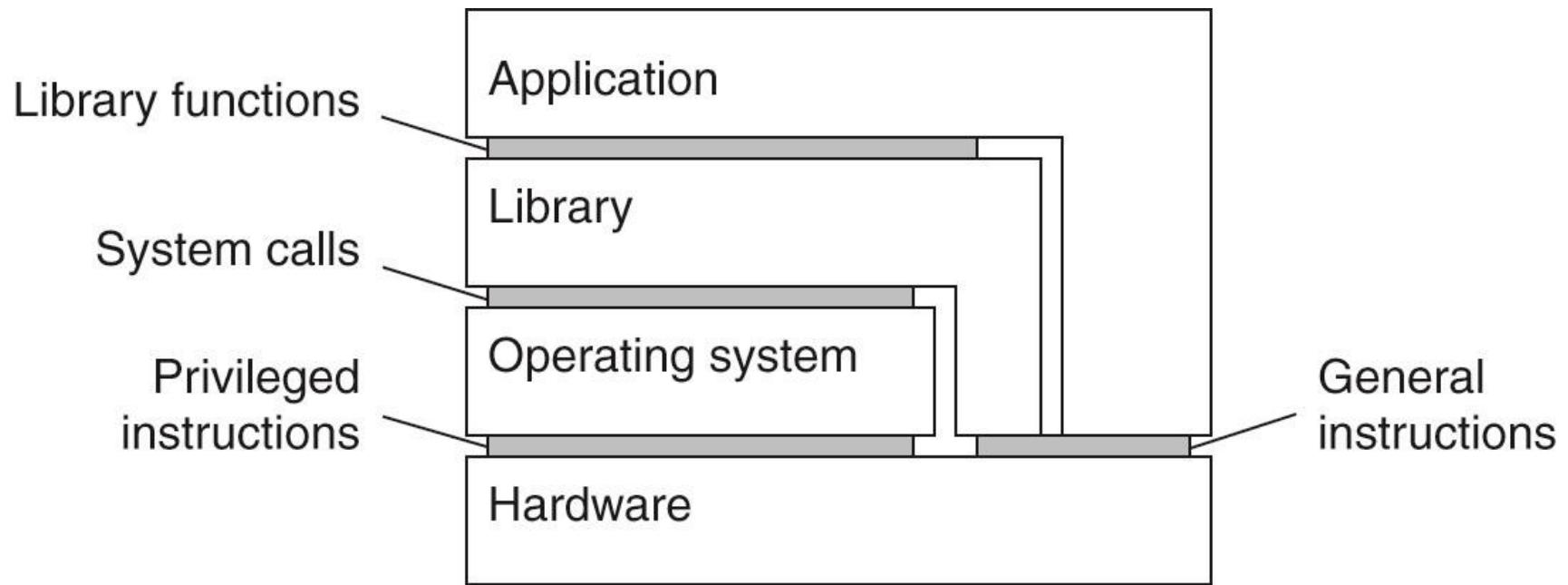


Figure 3-6. Various interfaces offered by computer systems.

# Architectures of Virtual Machines (4)

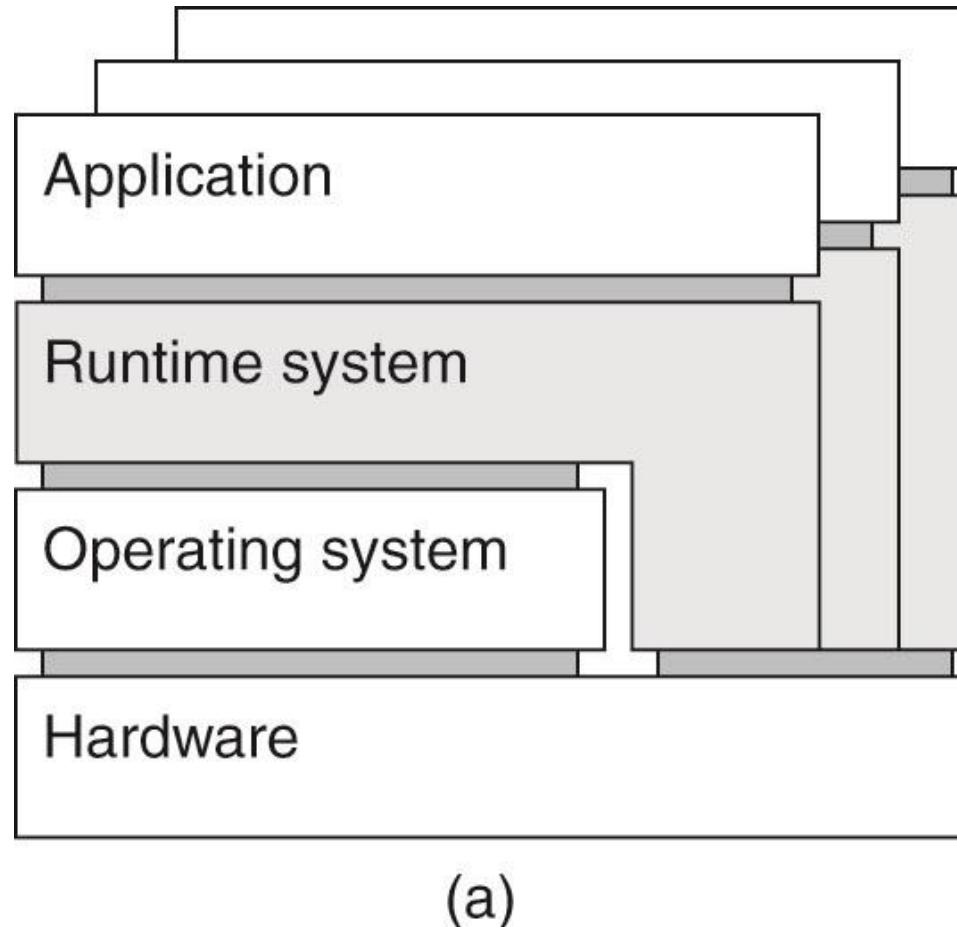


Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations.

# Architectures of Virtual Machines (5)

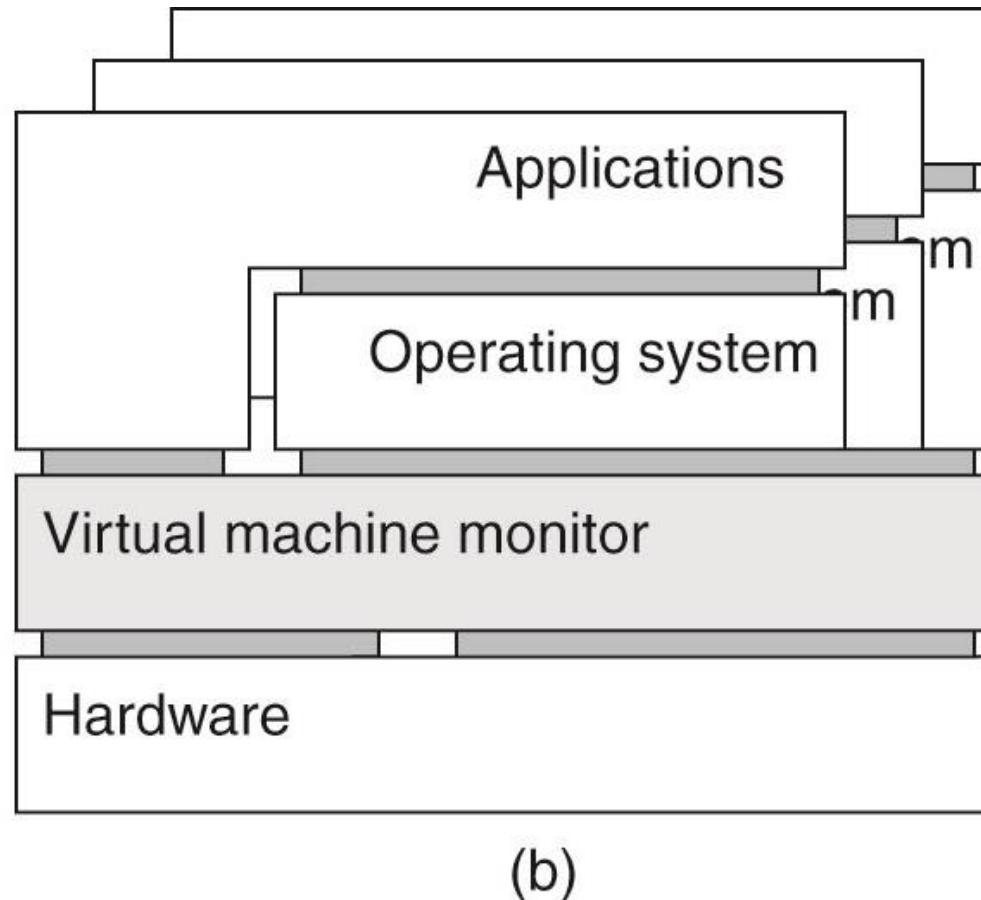


Figure 3-7. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

# Networked User Interfaces (1)

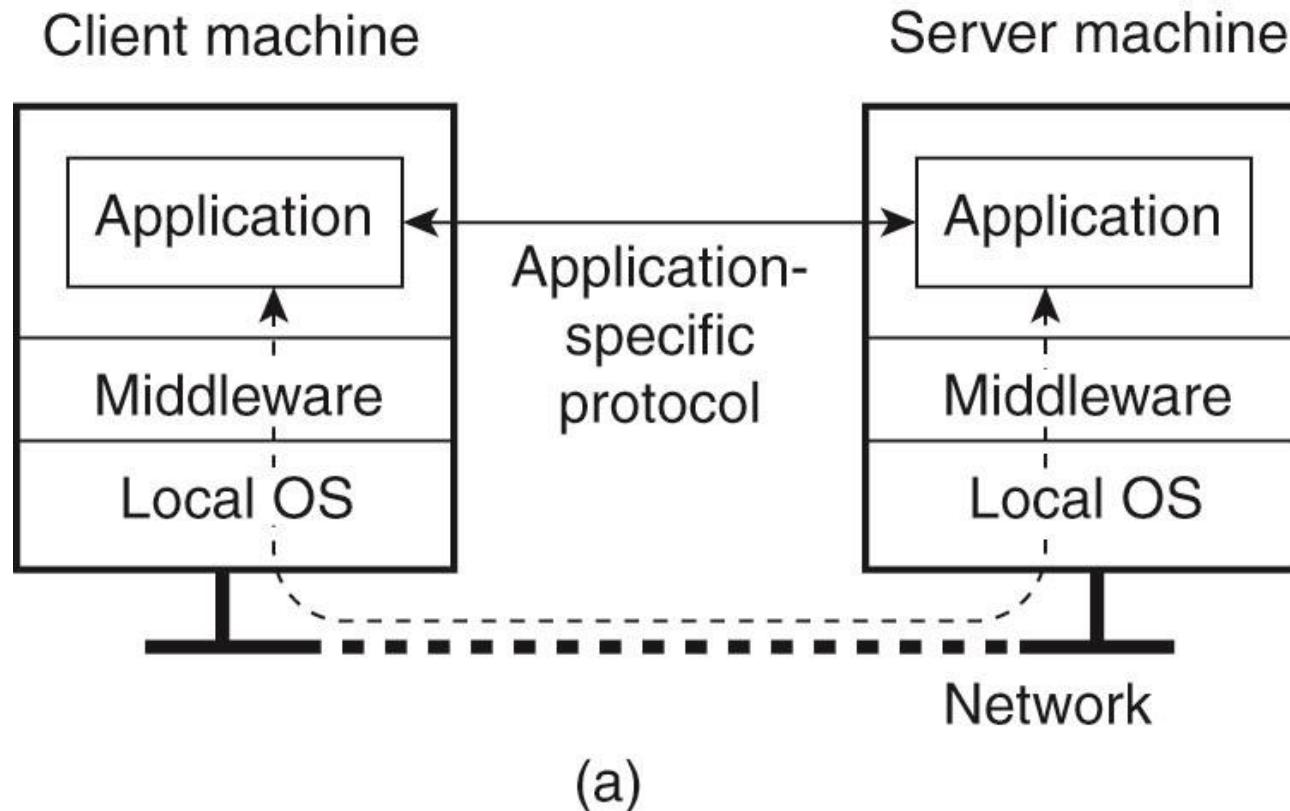


Figure 3-8. (a) A networked application with its own protocol.

# Networked User Interfaces (2)

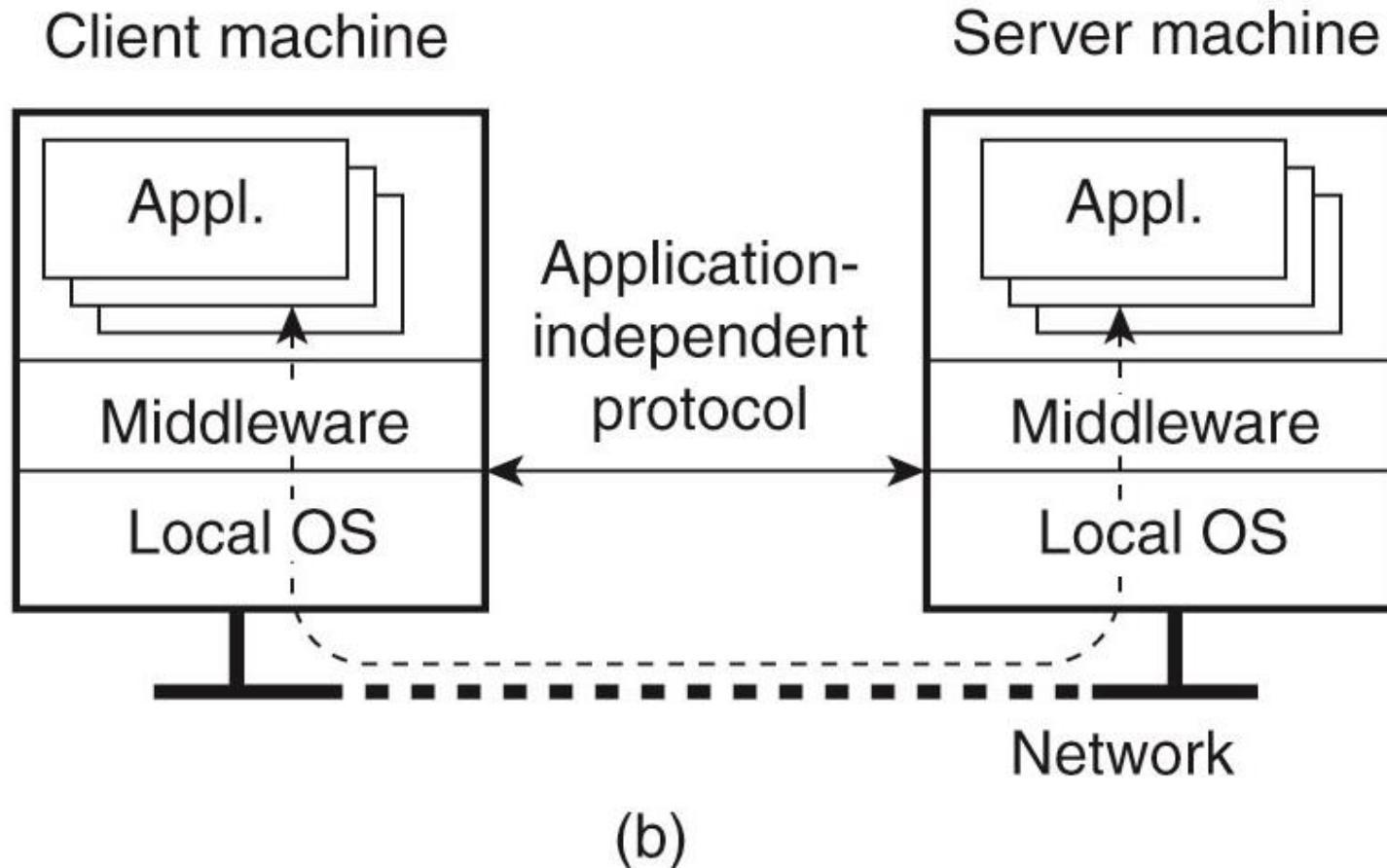


Figure 3-8. (b) A general solution to allow access to remote applications.

# Example: The XWindow System

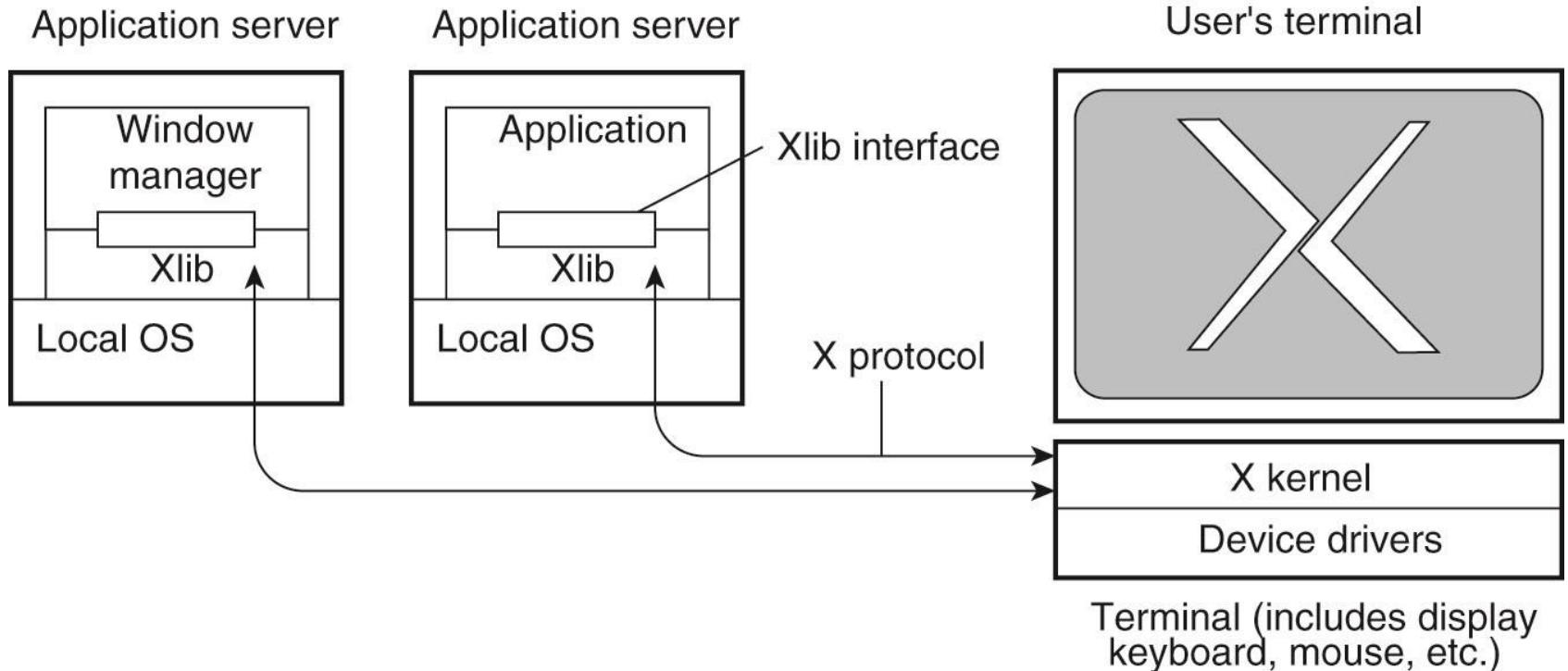


Figure 3-9. The basic organization of the XWindow System.

# Client-Side Software for *Distribution Transparency*

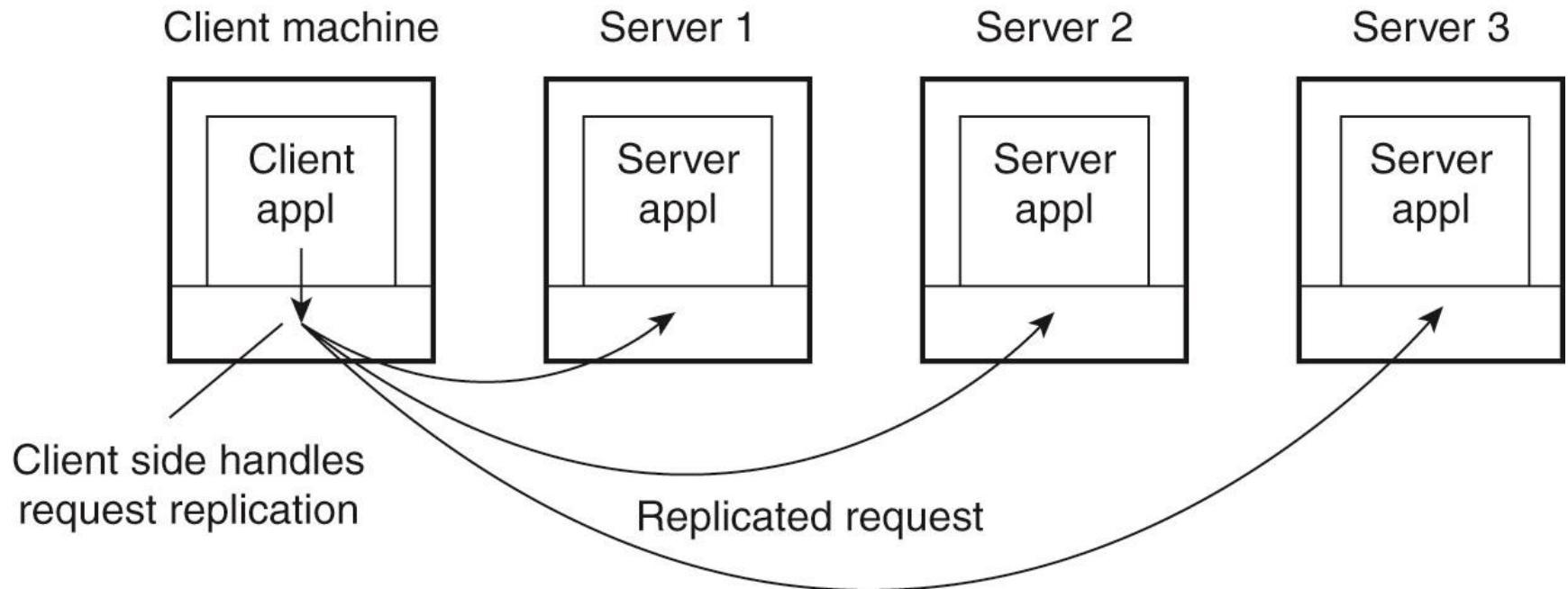


Figure 3-10. Transparent replication of a server using a client-side solution.

# Daemons

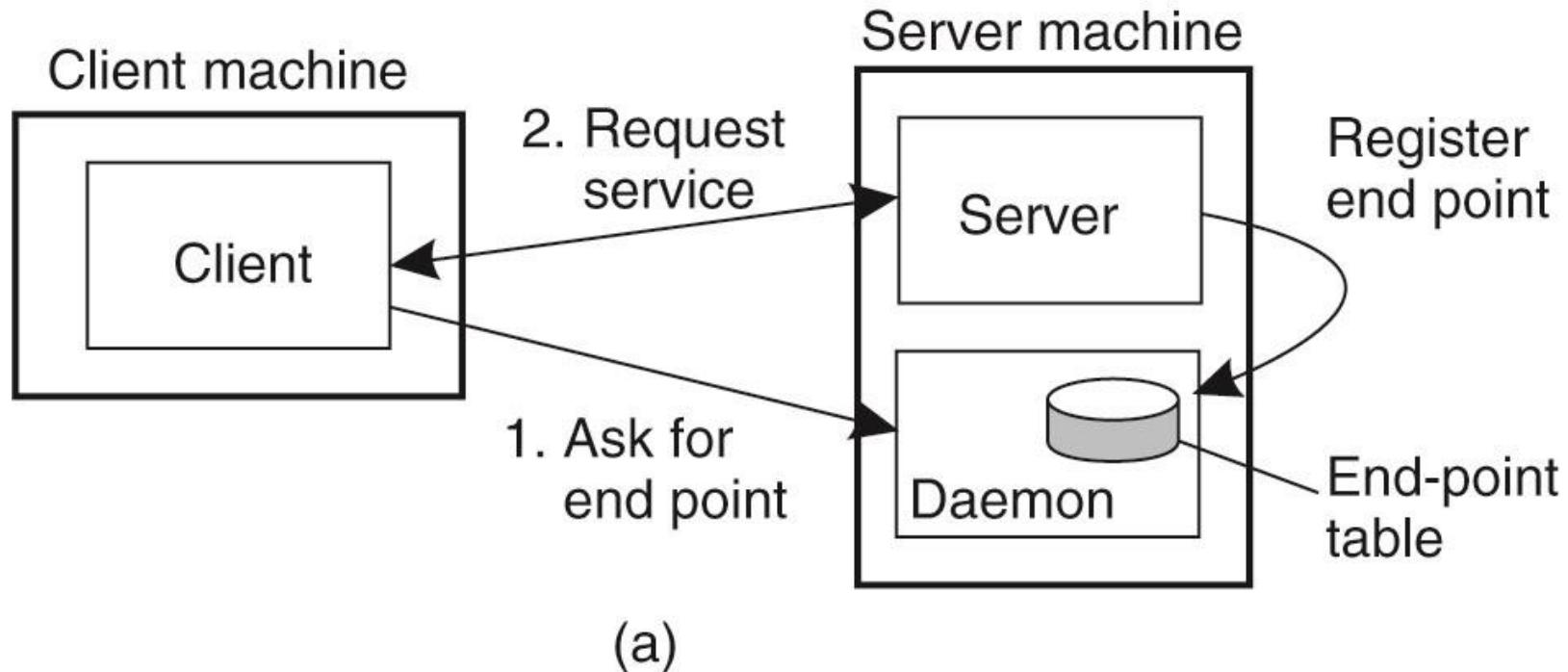


Figure 3-11. (a) Client-to-server binding using a *daemon*.

# Superservers

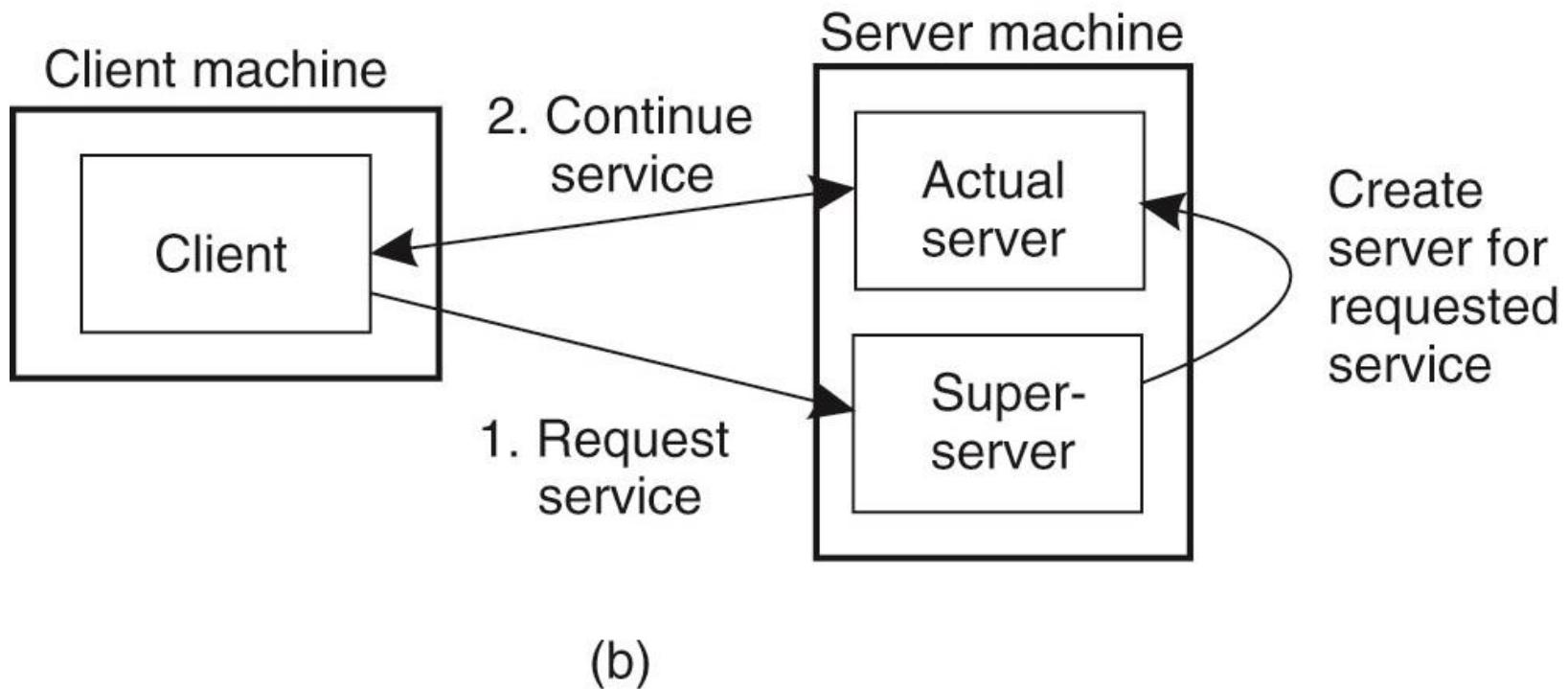


Figure 3-11. (b) Client-to-server binding using a superserver.

# Server Clusters (1)

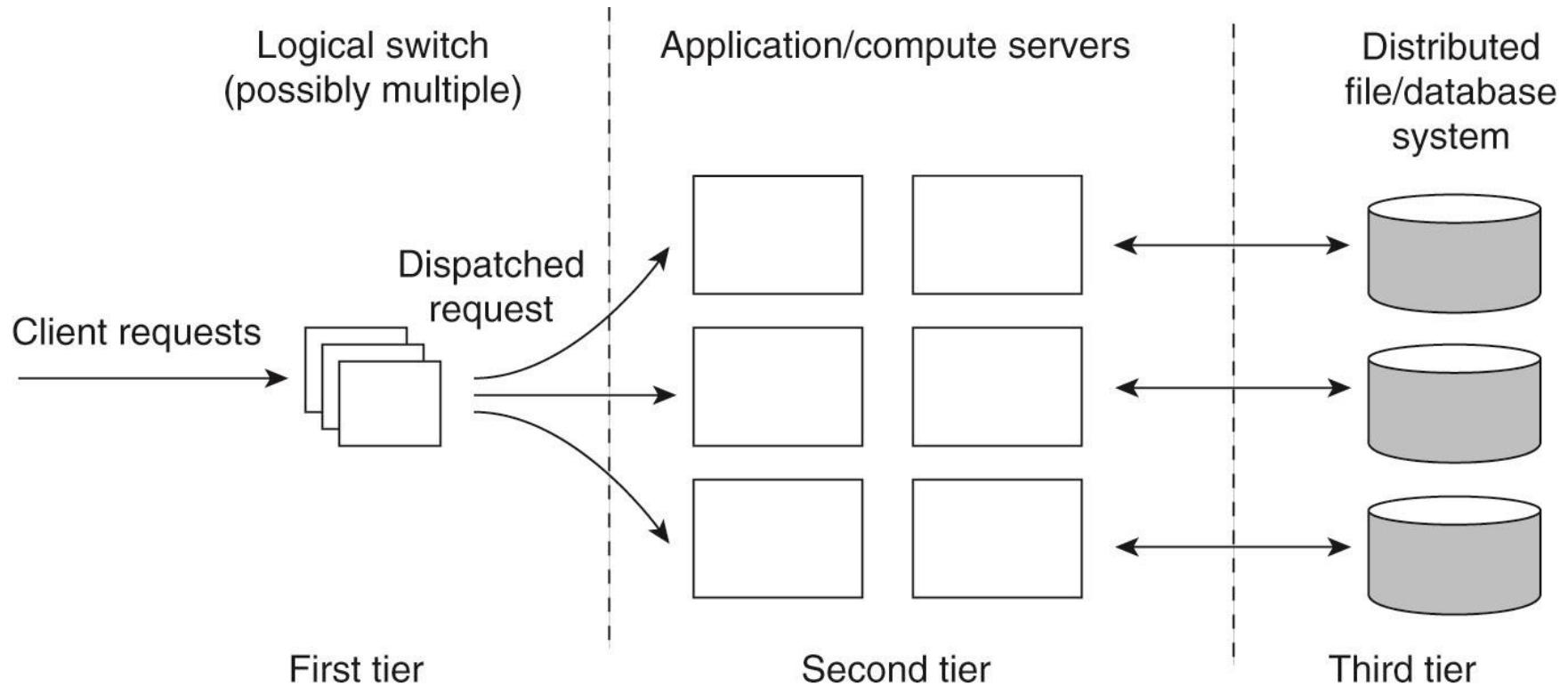


Figure 3-12. The general organization of a *three-tiered* server cluster.

# Server Clusters (2)

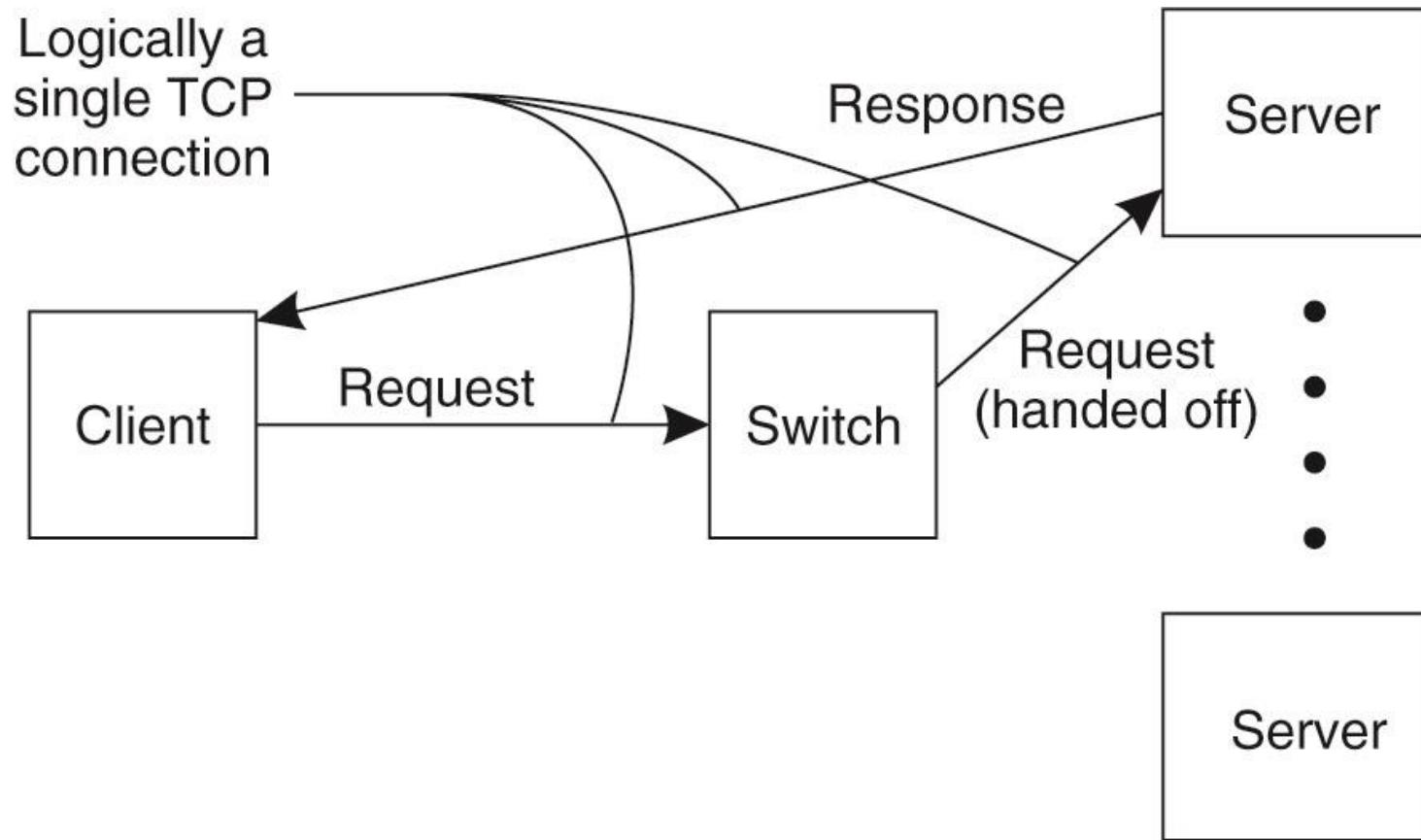


Figure 3-13. The principle of TCP handoff.

# Reasons for Migrating Code

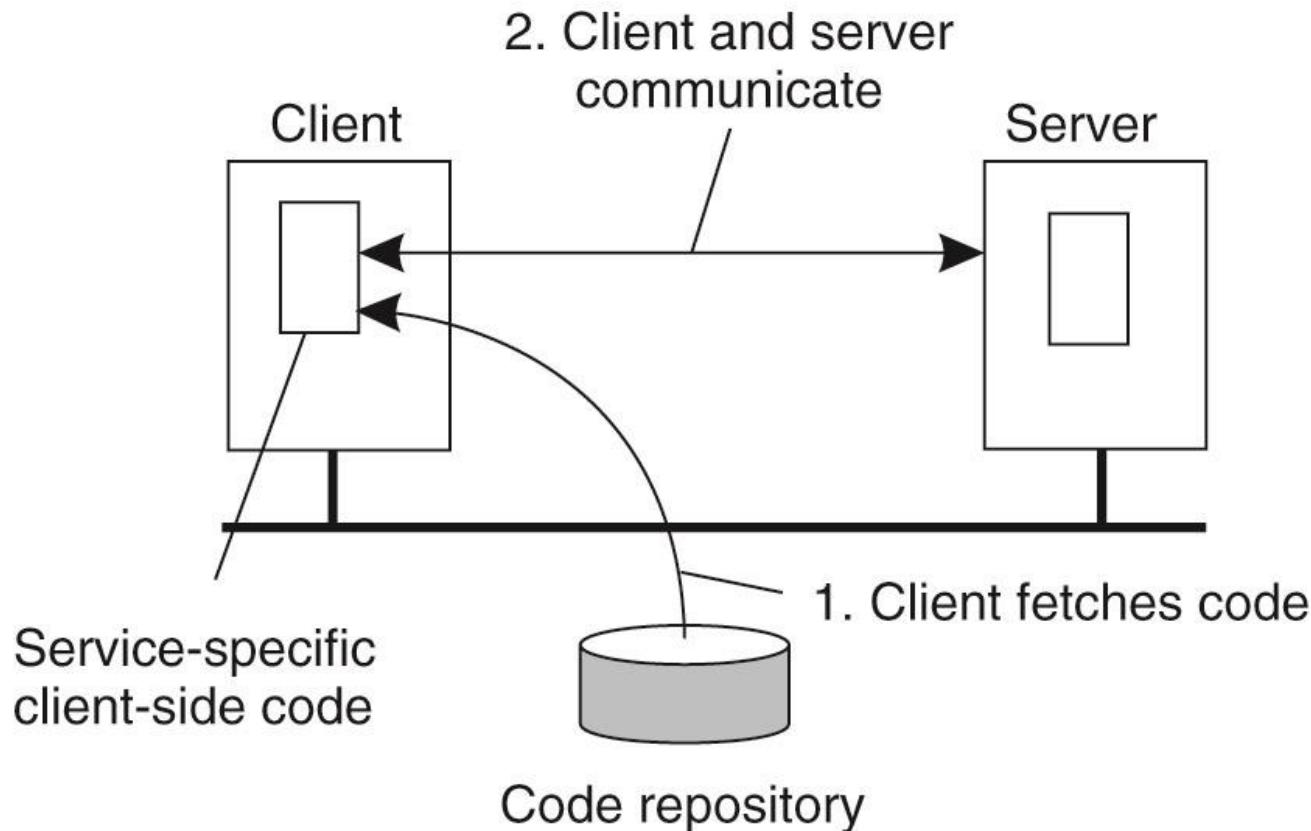


Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

# Models for Code Migration

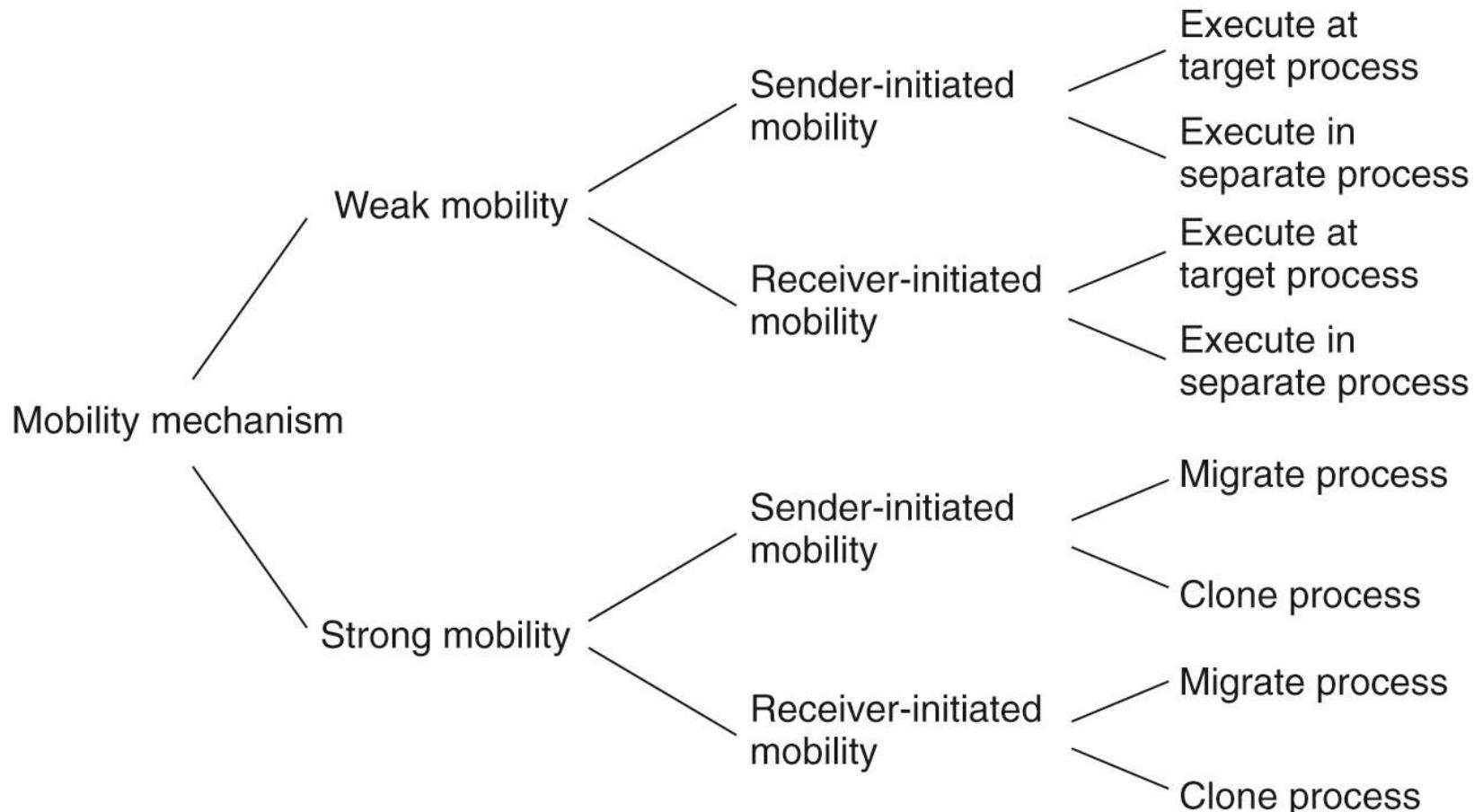


Figure 3-18. Alternatives for code migration.

# Migration: *process-to-resource* binding

- **By identifier** – Strongest – URL, internet address. The absolute name. Thus the absolute thing.
- **By value** – such as standard libraries. Copy of the library is fine.
- **By type** – weakest – reference to “some terminal” or “some printer”

# Migration – Can resource be moved?

- **Unattached** resource—such as data file specific to the application (so just move it)
- **Fastened** resource—could be moved theoretically, but not practical. Complete web site, local database used by others.
- **Fixed** resource – cannot be moved. E.g., real IP address / end point.

# Migration and Local Resources

## Resource-to-machine binding

		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV,GR)	GR (or CP)	GR
	By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR Establish a global systemwide reference

MV Move the resource

CP Copy the value of the resource

RB Rebind process to locally-available resource

Figure 3-19. Actions to be taken with respect to the references to local resources when migrating code to another machine.

# Migration in Heterogeneous Systems

Three ways to handle migration (which can be combined)

- **Pushing** memory pages to the new machine and resending the ones that are later modified during the migration process.
- **Stopping** the current virtual machine; migrate memory, and start the new virtual machine.
- Letting the new virtual machine **pull** in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.

# Migration in Heterogeneous Systems

For large-scale system design these could be important

- The possibility of migrating code, including in real time
- The concepts of resource types and process-to-resource binding types for processes that migrate—how do you design for these?
- The compromises of Push, Pull and Stop of a process when migrating in real time.

So add the general concepts to your toolbox!



# Figure Slides for Chapter 4: Networking and Internetworking



DISTRIBUTED SYSTEMS  
CONCEPTS AND DESIGN

George Coulouris  
Jean Dollimore  
Tim Kindberg



*From Coulouris, Dollimore and Kindberg*  
**Distributed Systems:**  
**Concepts and Design**

Edition 4, © Pearson Education 2005

# CDK Slides

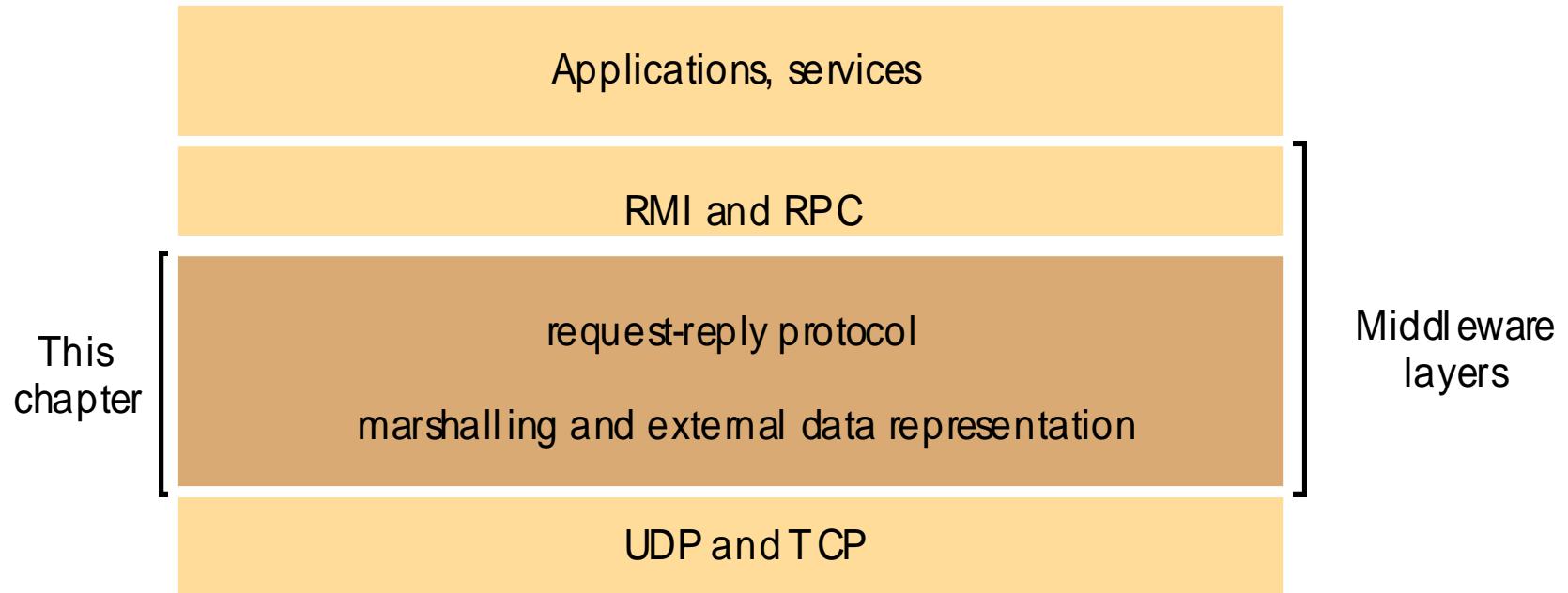
- Slides prepared by...
  - Professor Clark Elliott
  - DePaul University
- Some slides may be courtesy of Mark Goetsch
  - DePaul University

# Overview

- Introduction to interprocess communication
- API for internet protocols
- External data representation and marshalling.
- Client/server communication. Request/reply protocols.
- Group communication

# Figure 4.1

## Middleware layers



# API for Internet Protocol

- From the programmer's point of view:
- UDP – pass a single *message* from one process to another. *Datagrams*
- TCP – create a two-way stream between processes. Foundation for producer/consumer communication.

# Synchronous vs. Asynchronous Calls

- Fundamental to understanding DSes.
- Synchronous calls are like procedure calls in a program, caller waits for the return.
- Asynchronous (that is, not-synchronous) calls are entirely different and require a whole new programming paradigm. Many variations. All release the caller at some point, so there is less waiting. All require re-establishing contact with the caller.

# Synchronous calls

- Caller initiates contact with the receiver, sends request/call/data and *waits* until the receiver returns a response, then continues.
- Caller is *blocked* and can do nothing until the return.
  - Easy to code and to understand
  - Only one style
  - Inefficient
  - Caller may *hang* on bad receiver

# Asynchronous Calls

- Caller sends request/call/data and then proceeds to useful local work *without waiting*.
- Caller must then be *interruptable*, that is, drop everything and process the return, when the response comes. This is complicated to program.
  - More efficient
  - More complex
  - Many variations, depending where you synchronize
  - Caller will not hang

# Write your own async. call

- Because of the complexity of responding to an asynchronous response, this is left to the application programmer.
- Multiple threads in one process allow a blocking call -- where one thread waits for the response -- to effect asynchronous calls.
- Threads are synchronized as needed.
- Imagine *INET* client as asynchronous

# Sending

- Usually use *get-next-available-port()* call to find a spare port and send with that.
- Windows: Netstat

# Binding to a server, variations

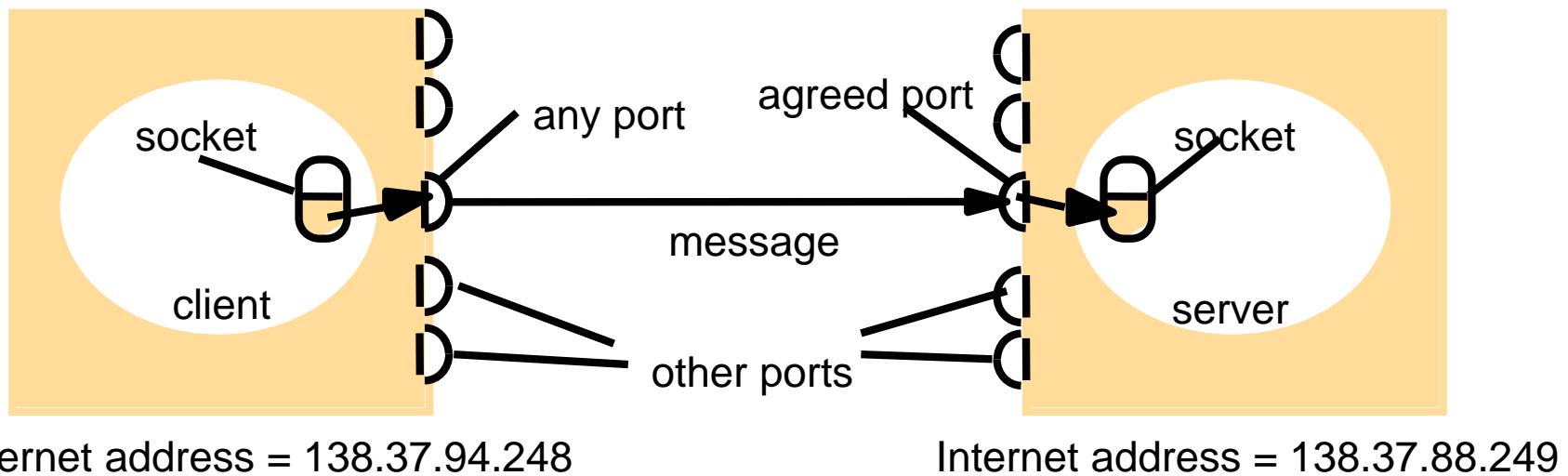
- Always use a fixed port (e.g. http)
- Client refers to service by name and a *name server* or *super server* translates the name into a port number at run-time.
- Some operating systems (Mach) do this translation at a low level. Some opsys (V system) address processes.
- Allows servers to migrate as needed.

# Ports

- Can be fixed, e.g. http, ftp, etc.
- Only loosely coupled to processes, so multiple ports in and out can be used.
- Allows use of “throw away” ports as needed – e.g. client send in http.
- Easy to manage in application tables – just a number for an argument-slot in the API.

# Figure 4.2

## Sockets and ports



# UDP

- Universal Datagram Protocol
- No ack, no resend, no guarantee; cheap!
- Receive method returns address and port of sender, as well as the data.
- Receiver block, sender non-block is usual
- Datagrams arrive on receive port from any sender, in any order, is usual but...
- Can form connection between processes

# UDP “failures”

- No duplication
- Omissions can occur
- Datagrams can arrive out of order (and also mixed in with those of other conversations)
- Cheap because:
  - No *state* stored at source / destination
  - No extra messages transmitted (ack/syn)
  - Less latency b/c no setup

# UDP

- Cheap, fast, efficient
- May be all you need
- Can construct just the features you need
- Add to your toolbox!



# Hey CDK write comments in your code!

- m is message, comes from arg 0
- aSocket is opened to server
- aHost is server IP address
- ServerPort is arbitrary, why not from args?
- reply is datagram response
- If you play, do yourself a favor and put in messages about processing of requests.

## Figure 4.3

### UDP client sends a message to the server and gets a reply

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request = new DatagramPacket(m,  args[0].length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length());
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```

## Figure 4.4: UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                    request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }finally {if(aSocket != null) aSocket.close();}
    }
}
```

# ServerSocket

- ServerSocket (doorbell socket) blocks while listening for requests, result of request is a Socket, and ServerSocket goes back to listening.
- Queue size – drop simultaneous requests for a worker if full

# Socket

- Used by pair of processes with a connection.
- Construction creates a local socket and port, and also connects to remote
- Throws unknown host, I/O, exceptions
- Provides getInputStream and getOutputStream

Figure 4.5: TCP client makes connection to server, sends request and receives reply

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);           // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();
            System.out.println("Received: "+ data);
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }finally {if(s!=null) try {s.close();}catch (IOException
e){System.out.println("close:"+e.getMessage());}}
    }
}
```

## Figure 4.6 TCP server makes a connection for each client and then echoes the client's request

```
import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}

// this figure continues on the next slide
```

## Figure 4.6 continued

```
class Connection extends Thread {  
    DataInputStream in;  
    DataOutputStream out;  
    Socket clientSocket;  
    public Connection (Socket aClientSocket) {  
        try {  
            clientSocket = aClientSocket;  
            in = new DataInputStream( clientSocket.getInputStream());  
            out =new DataOutputStream( clientSocket.getOutputStream());  
            this.start();  
        } catch(IOException e) {System.out.println("Connection:"+e.getMessage());}  
    }  
    public void run(){  
        try {  
            // an echo server  
            String data = in.readUTF();  
            out.writeUTF(data);  
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());}  
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());}  
        } finally{ try {clientSocket.close();}catch (IOException e){/*close failed*/}}  
    }  
}
```

# Computer Architecture Definition for Distributed Systems

- The instruction set. Each instruction has a unique bit string in the *instruction register*
- The data arguments to the instructions, specified as bit strings in *CPU registers*.
- The semantics of the instructions of the instructions and data arguments.

# ADD example

- ADD R1 R2 R3 ← Assembly language:  
add the contents of Register 1 to Register  
2 and put the results in Register 3
- 0100 0001 0010 0011 ← bit strings in  
Instruction Register, Arg1, Arg2 and Arg3

# ADD example two

- Different chipsets have different architectures.
- ADD R1 R2 R3 ← Assembly language:  
add the contents of Register 1 to Register 2 and put the results in Register 3
- 00100 10000 01000 11000 ← bit strings in Instruction Register, Arg1, Arg2 and Arg3

# Marshaling and External Data Representation

- Program data is *random access*, stored in memory
- Messages are *serial*, sent bit-after-bit on transmission lines.
- Marshaling/Un-marshaling is the process of translating one into the other.
- The problem of fully automating this process is *generally unsolvable*, & *specifically hard*.

# Marshaling--may have to *translate* data

- Source and destination computers may have different architectures and so different data representations
- Lots of computer architectures:  
[http://en.wikipedia.org/wiki/Instruction\\_set#ISAs\\_implemented\\_in\\_hardware](http://en.wikipedia.org/wiki/Instruction_set#ISAs_implemented_in_hardware)
- 64-bit vs. 32 integers? Big-endian vs. little-endian?
- Floating point representations?
- Text. ASCII: 7 bits?, Unicode 16 bits?, EBCIDIC 8 bits.

# Marshaling translation choices

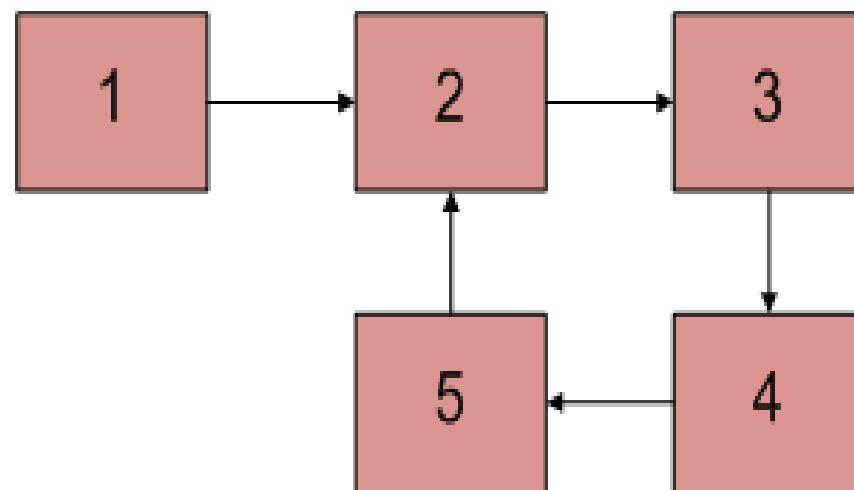
- Agree on external format and translate at both ends (e.g., CDR, XML)
- Sender translates for receiver (not common?)
- Sender sends agreed meta-information (or is implicit in application) and receiver translates
- Use virtual machines so no translation (Java)

# Why impossible to fully automate

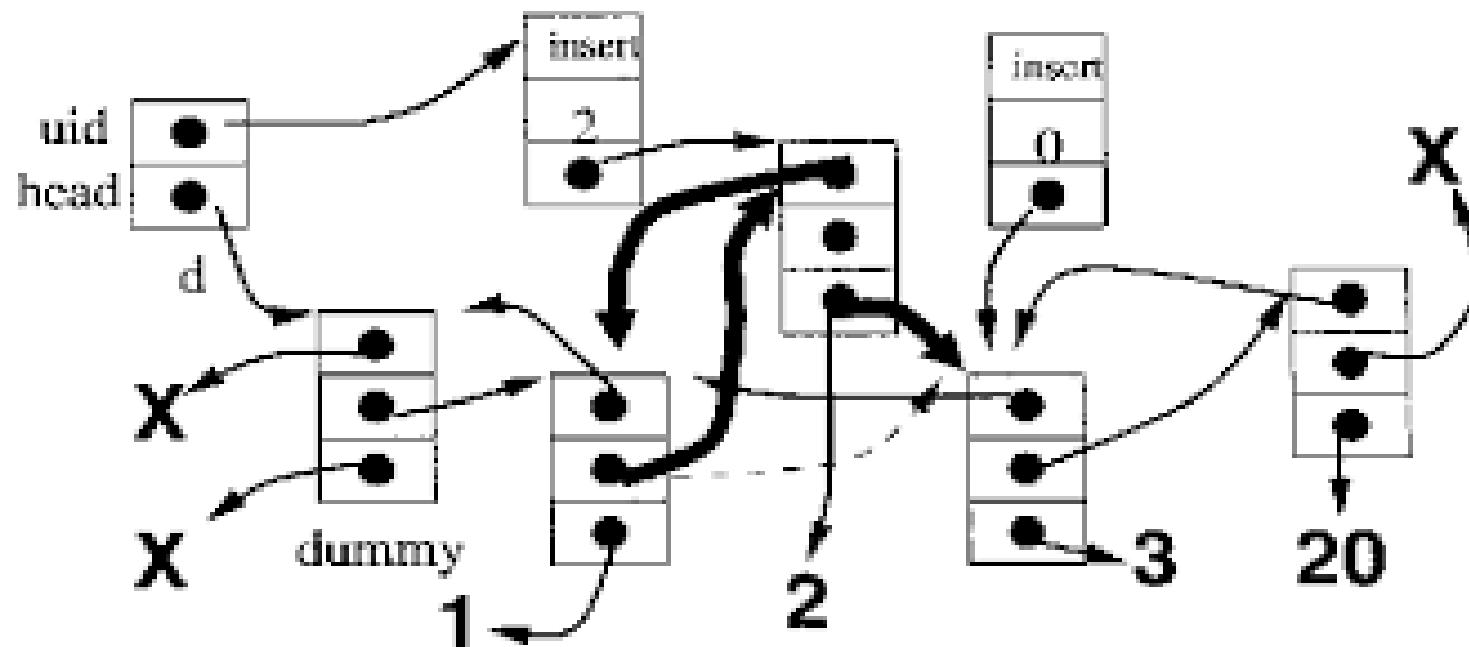
- Automated marshaling means that without knowing anything about the data, a lower-level middleware algorithm can marshal it.
- Dynamically allocated data objects:
  - What does Null pointer mean? Suppose application programmer uses 237 as the null pointer?
  - What about a random-access cyclically linked objects?

- The typical reference to a linked data structure is the address in memory (an unsigned integer—a *pointer*) of the start of the object, or the address of a *head pointer* that points to it.
- What about *dynamic* data structures that are changed by the application process while they are being marshaled?

# Some dynamic linked data structures

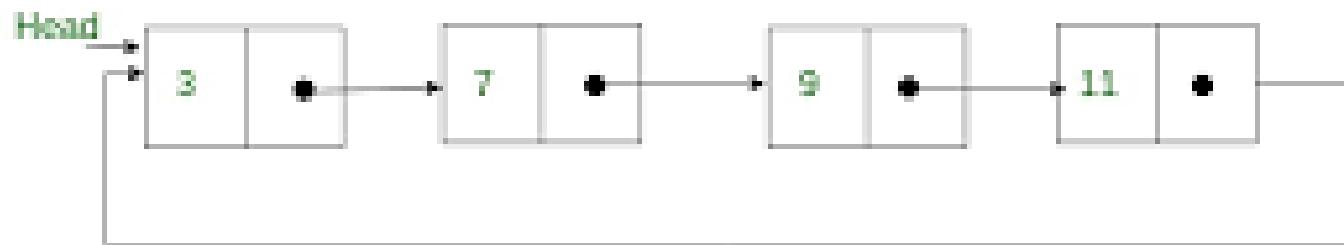


# Some dynamic linked data structures

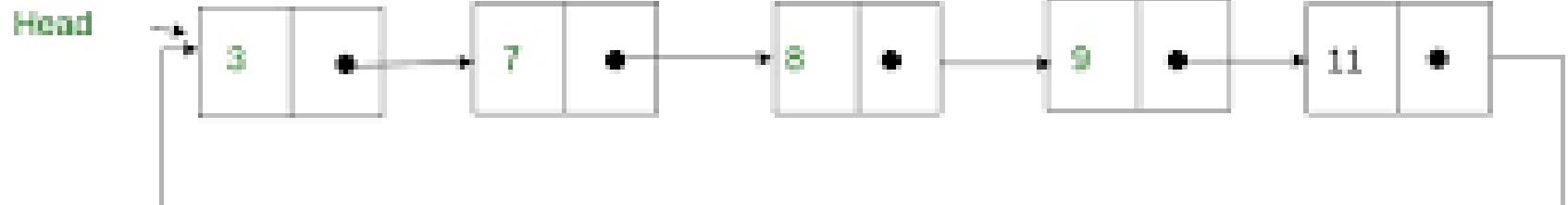


# Marshaling a dynamic cyclically linked list

- Marshaling process [M] starts marshaling linked object at node 3, completes to 9, App inserts node 8, M completes to 11, now what?



After inserting 8, the above CLL should be changed to the following



Now write the code to marshal this



# Introspection

- Sometimes objects have information about their class available. This has to be marshaled as well.
- Java supports introspection in its serialization
- CDR sends only the data and assumes knowledge of the objects.

# XML namespaces

- Information about contained objects, but may be in an external reference.
- XML namespaces sometimes use URLs as unique strings (UUIDs), and have nicknames within a document.
- Any named item in the document with the nickname thus made globally unique.
- E.g., “ID\_num” can appear in many documents, but is made unique by the docs namespace

# Serialization, “flattening”

- To store random access memory objects on disk, or to send them on network lines, the process is the same.
- Serialization is the process of creating a string of bits, a “series,” to be processed one after another, from a collection of bits that can all be immediately accessed in any order.
- Thus all messages, and serialized objects, can also be stored on disk.
- This has profound implications for the semantics of *time* and *canonical sinks* of data.

# Odd semantics for objects

- Not like the real world.
- The object can be written to disk, and then restored in 1,000 years.
- The object can be written to disk, and then restored in five new locations every month. Which is the canonical representation?
- An object can *leave* a group, *join* another group, but still be in the first group. It can return in a different state than the object that did not leave.
- An object can be *nowhere* but later restored.
- An object can be half in one location and half in another, during transmission.

# CORBA's CDR

- Common Object Request Broker Architecture, Common Data Representation
- External serial form written, and read, by many different, unlike, platforms.
- 15 primitive types, plus constructed types
- No type information. Sender and Receiver must have prior knowledge. Use IDL?

## Figure 4.7

### CORBA CDR for constructed types

---

<i>Type</i>	<i>Representation</i>
<i>sequence</i>	length (unsigned long) followed by elements in order
<i>string</i>	length (unsigned long) followed by characters in order (can also can have wide characters)
<i>array</i>	array elements in order (no length specified because it is fixed)
<i>struct</i>	in the order of declaration of the components
<i>enumerated</i>	unsigned long (the values are specified by the order declared)
<i>union</i>	type tag followed by the selected member

---

# Figure 4.8

## CORBA CDR message

<i>index in sequence of bytes</i>	← 4 bytes →	<i>notes on representation</i>
0–3	5	<i>length of string</i>
4–7	"Smit"	'Smith'
8–11	"h "	
12–15	6	<i>length of string</i>
16–19	"Lond"	'London'
20–23	"on "	
24–27	1934	<i>unsigned long</i>

The flattened form represents `Person` struct with value: {'Smith', 'London', 1934}

# Java serialization

- When objects refer to other necessary objects, they have to be included as well.
- Version checking is performed – often very important in DSes, when services and clients can go off line.
- Class information is written, followed by types, and names of instance variables.
- References are serialized as *handles* referring to an object within the serialized form. The object is written just once. Thereafter the object is not written, just the handle.

## Figure 4.9

### Indication of Java serialized form

<i>Serialized values</i>				<i>Explanation</i>
Person		8-byte version number	h0	<i>class name, version number</i>
3	int year	java.lang.String name:	java.lang.String place:	<i>number, type and name of instance variables</i>
1934	5 Smith	6 London	h1	<i>values of instance variables</i>

The true serialized form contains additional type markers; h0 and h1 are handles

# Java serialization

- Number of vars, followed by type of vars, followed by value of vars
- (Strings preceded by length)
- Write out the class information, serialize all other objects it references (unless in the JVM). *Handles* refer to the other objects in the serialized form. 1-1 correspondence between handles and objects.
- Recall: Objects written only once, then use handles.

# Java serialization

- To Serialize:
  - Create instance of ObjectOutputStream
  - Invoke writeObject method, passing our *person* object as the argument.
- To DeSerialize:
  - Open an ObjectInputStream on the stream
  - Use ReadObject method to reconstruct the object.

## Figure 4.10 XML definition of the Person structure

```
<person id="123456789">
    <name>Smith</name>
    <place>London</place>
    <year>1934</year>
    <!-- a comment -->
</person >
```

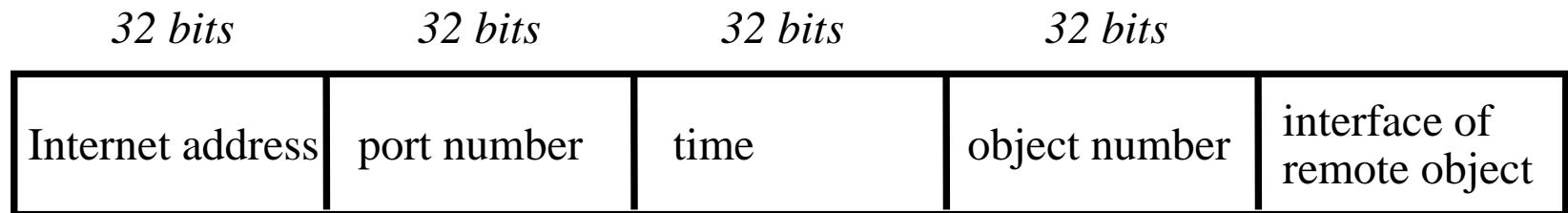
# Figure 4.11 Illustration of the use of a namespace in the *Person* structure

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk4.net/person">  
    <pers:name> Smith </pers:name>  
    <pers:place> London </pers:place >  
    <pers:year> 1934 </pers:year>  
</person>
```

# Figure 4.12 An XML schema for the Person structure

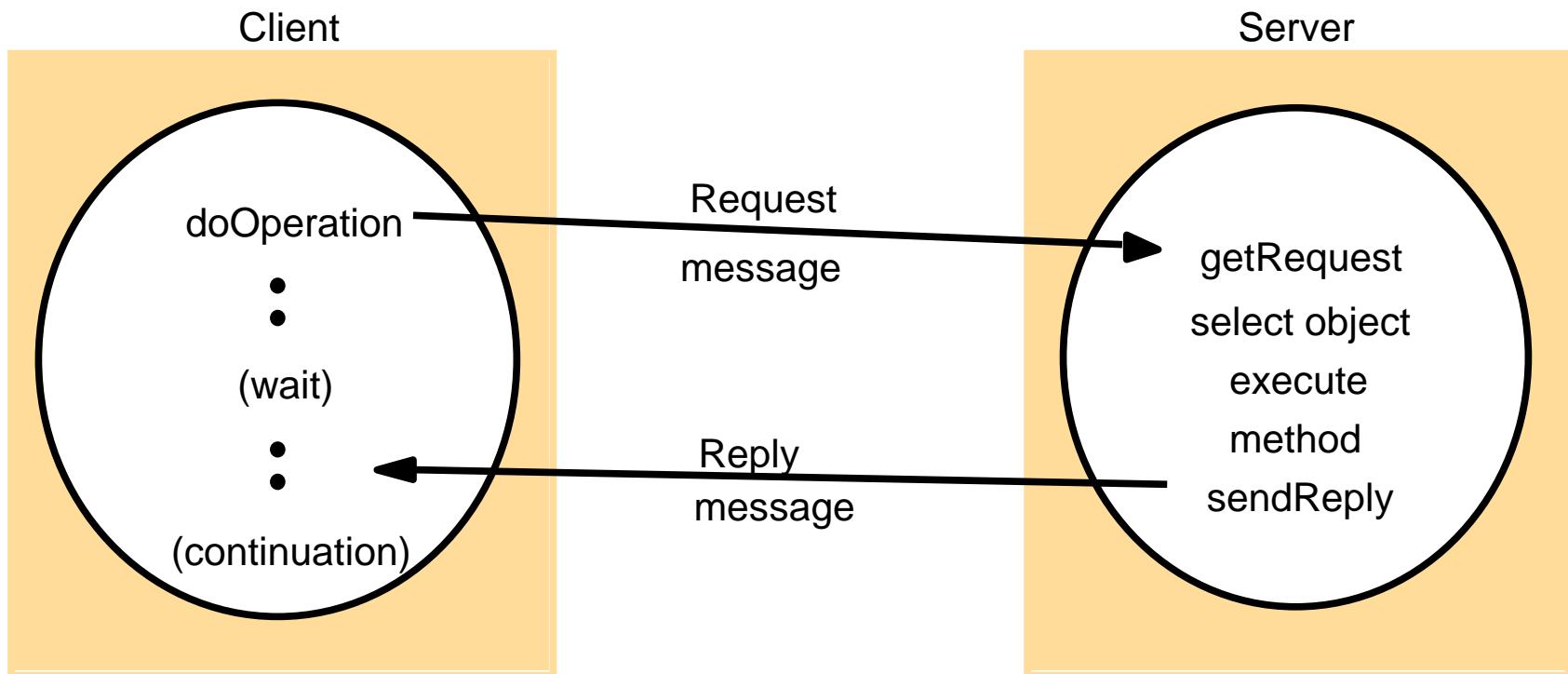
```
<xsd:schema xmlns:xsd = URL of XML schema definitions >
  <xsd:element name= "person" type = "personType" />
    <xsd:complexType name= "personType">
      <xsd:sequence>
        <xsd:element name = "name" type="xs:string"/>
        <xsd:element name = "place" type="xs:string"/>
        <xsd:element name = "year" type="xs:positiveInteger"/>
      </xsd:sequence>
      <xsd:attribute name= "id" type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

Figure 4.13  
Representation of a remote object reference



# Figure 4.14

## Request-reply communication



## Figure 4.15

# Operations of the request-reply protocol

*public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)*  
sends a request message to the remote object and returns the reply.  
The arguments specify the remote object, the method to be invoked and  
the arguments of that method.

*public byte[] getRequest ();*  
acquires a client request via the server port.

*public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);*  
sends the reply message reply to the client at its Internet address and port.

## Figure 4.16

# Request-reply message structure

messageType
requestId
objectReference
methodId
arguments

*int (0=Request, 1= Reply)*

*int*

*RemoteObjectRef*

*int or Method*

*array of bytes*

# Figure 4.17

## RPC exchange protocols

---

<i>Name</i>	<i>Messages sent by</i>		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Request</i>		
RR	<i>Request</i>	<i>Reply</i>	
RRA	<i>Request</i>	<i>Reply</i>	<i>Acknowledge reply</i>

---

# RPC request reply...

- *Timeouts* – server binding problem – resend from client?
  - Server discards duplicates? Request identifier needed.
- *Lost reply*: Suppose server has executed request, and sent reply, when duplicate shows up? Must re-execute, or store a *history*. Strictly *idempotent* operations? E.g. read or add to a set.

# RPC request reply...

- *History* – may be used when re-transmission is required.
- Contains message identifier, and endpoint of client to which it was sent.
- Can grow too large.
- If clients request only one operation at a time (e.g., synchronous request) can use following request as ack. of previous.

# RR-Ack

- History need only previous message.
- When client terminates, messages get saved. Need time-out, purge.
- With RRA, no history need be saved
- Acks imply ALL previous replies have been received, so the loss of an ack, followed by another RRA sequence, is automatically handled, and client continues without delay.

# Use TCP for RRA?

- If the request or return values are high, might want to use TCP instead of UDP which has a limit on packet size. TCP streams allow for any size object to be sent.

# Figure 4.18

## HTTP request message

<i>method</i>	<i>URL or pathname</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

# Figure 4.19

## HTTP reply message

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

## Figure 4.20

### Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
        }
    }
}
// this figure continued on the next slide
```

# Figure 4.20

## continued

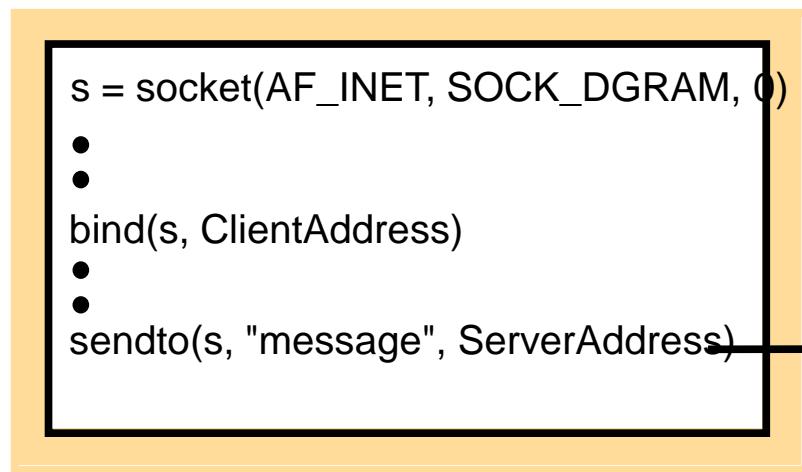
```
// get messages from others in group
    byte[] buffer = new byte[1000];
    for(int i=0; i< 3; i++) {
        DatagramPacket messageIn =
            new DatagramPacket(buffer, buffer.length);
        s.receive(messageIn);
        System.out.println("Received:" + new String(messageIn.getData()));
    }
    s.leaveGroup(group);
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());}
}finally {if(s != null) s.close();}
```

}

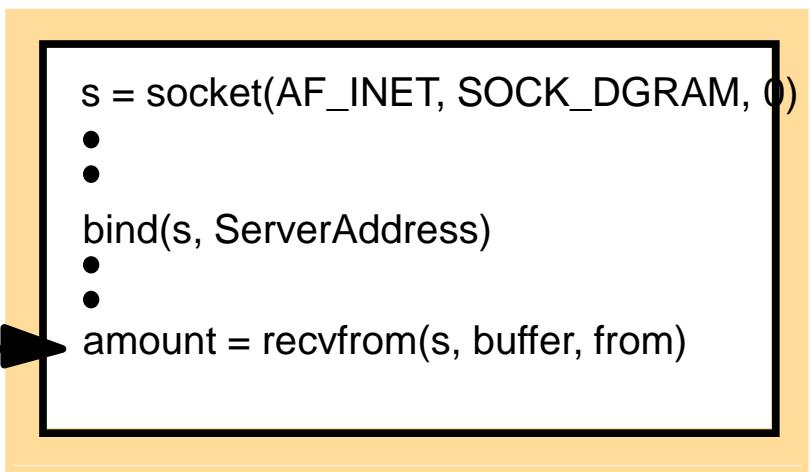
# Figure 4.21

## Sockets used for datagrams

Sending a message



Receiving a message



*ServerAddress* and *ClientAddress* are socket addresses

# Figure 4.22

## Sockets used for streams

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)  
•  
•  
connect(s, ServerAddress)  
•  
•  
write(s, "message", length)
```

Listening and accepting a connection

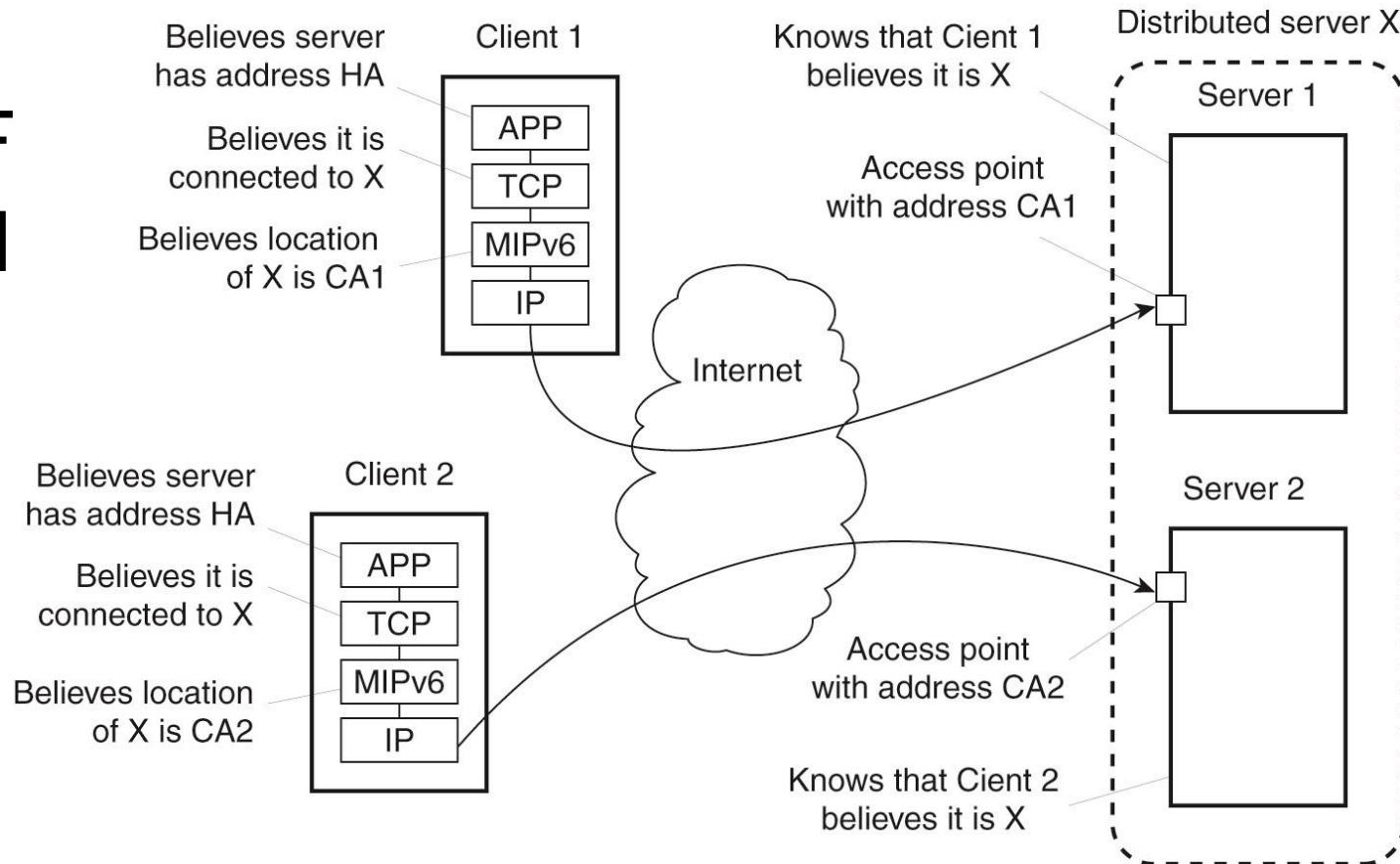
```
s = socket(AF_INET, SOCK_STREAM,0)  
•  
bind(s, ServerAddress);  
listen(s,5);  
•  
sNew = accept(s, ClientAddress);  
•  
n = read(sNew, buffer, amount)
```

*ServerAddress* and *ClientAddress* are socket addresses

# Cut slides:

# Distributed Servers

- F  
d



# Blank

**DISTRIBUTED SYSTEMS**  
**Principles and Paradigms**

Second Edition  
ANDREW S. TANENBAUM  
MAARTEN VAN STEEN

**Chapter 4**  
**Communication**

# Layered Protocols (1)

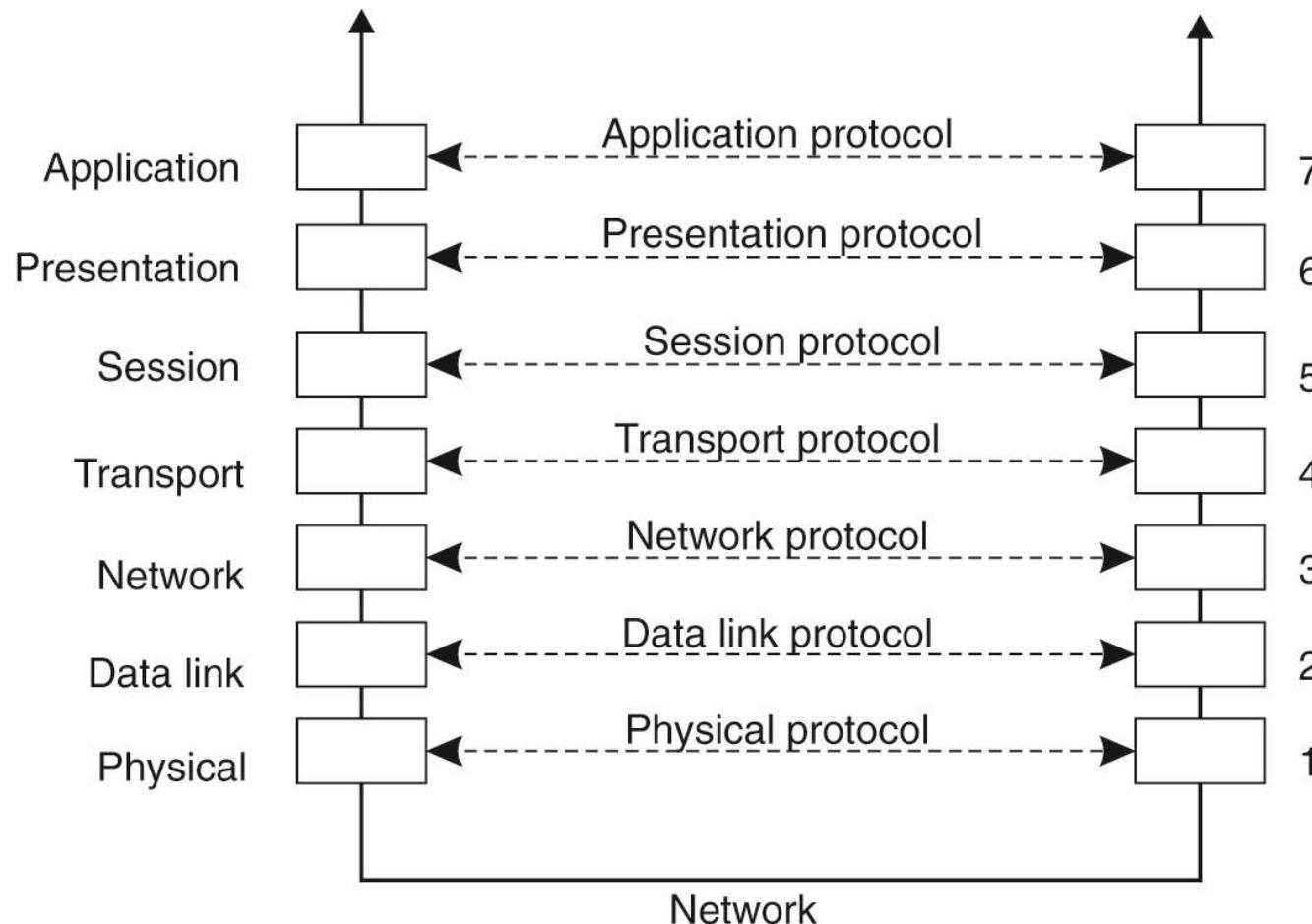


Figure 4-1. Layers, interfaces, and protocols in the OSI model.

# Layered Protocols (2)

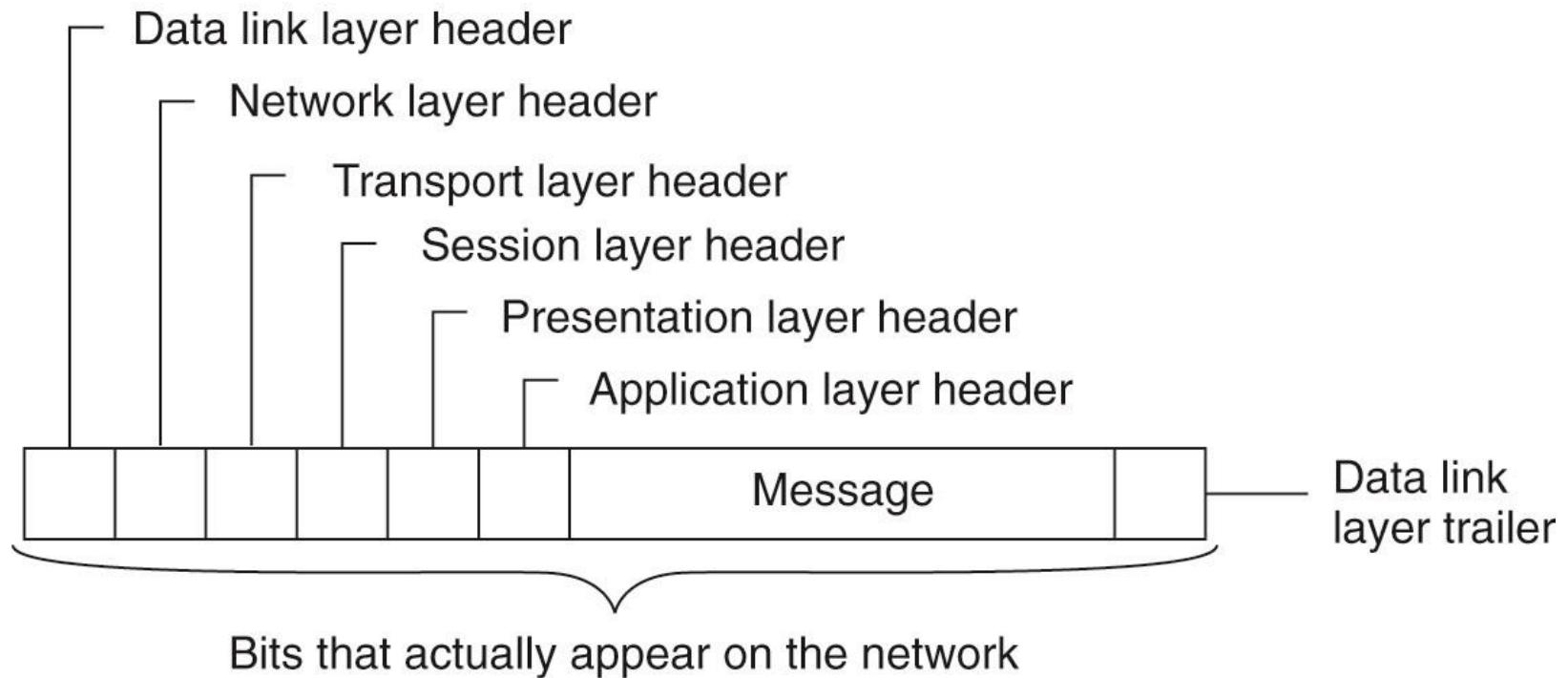


Figure 4-2. A typical message as it appears on the network.

# Middleware Protocols

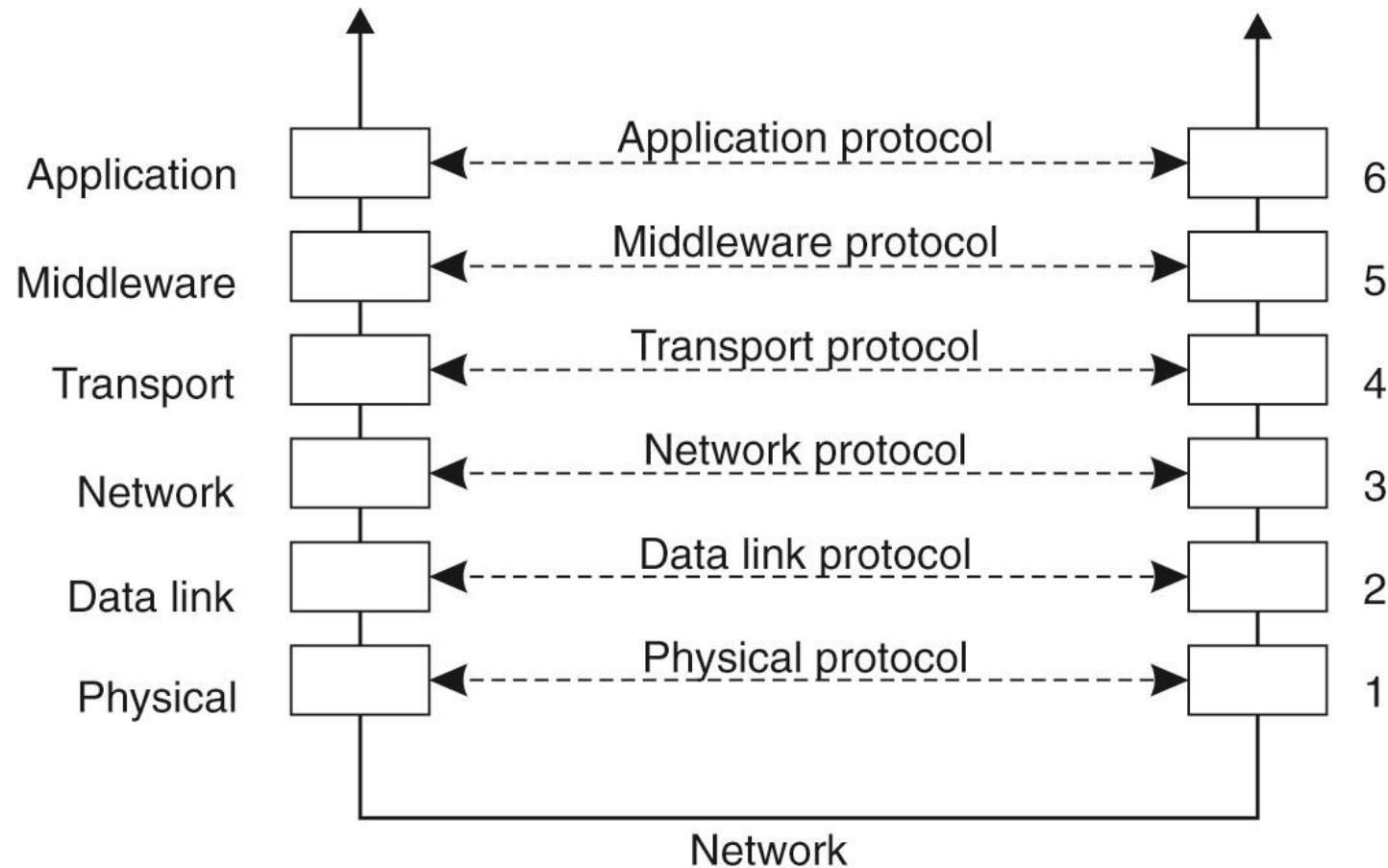
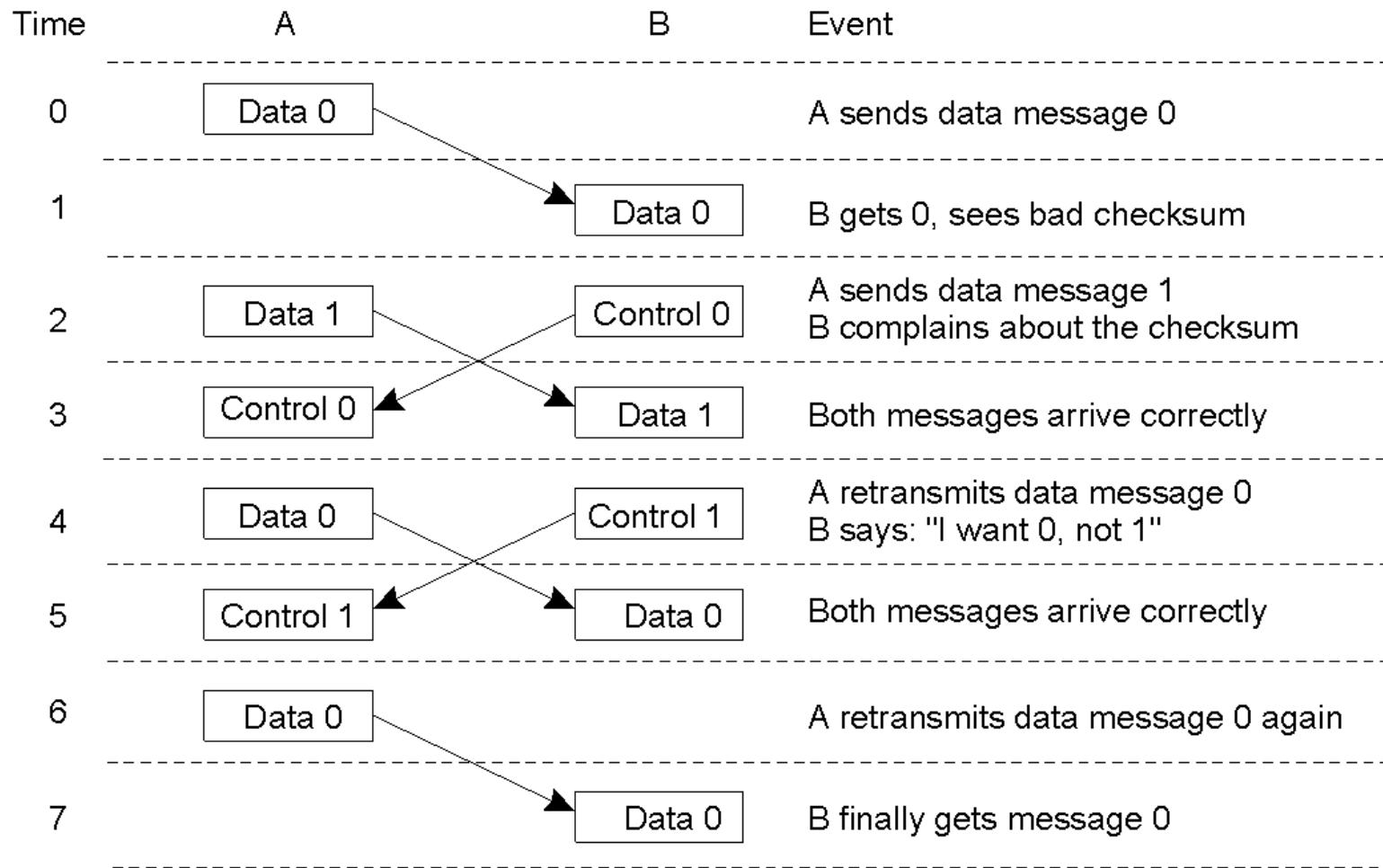


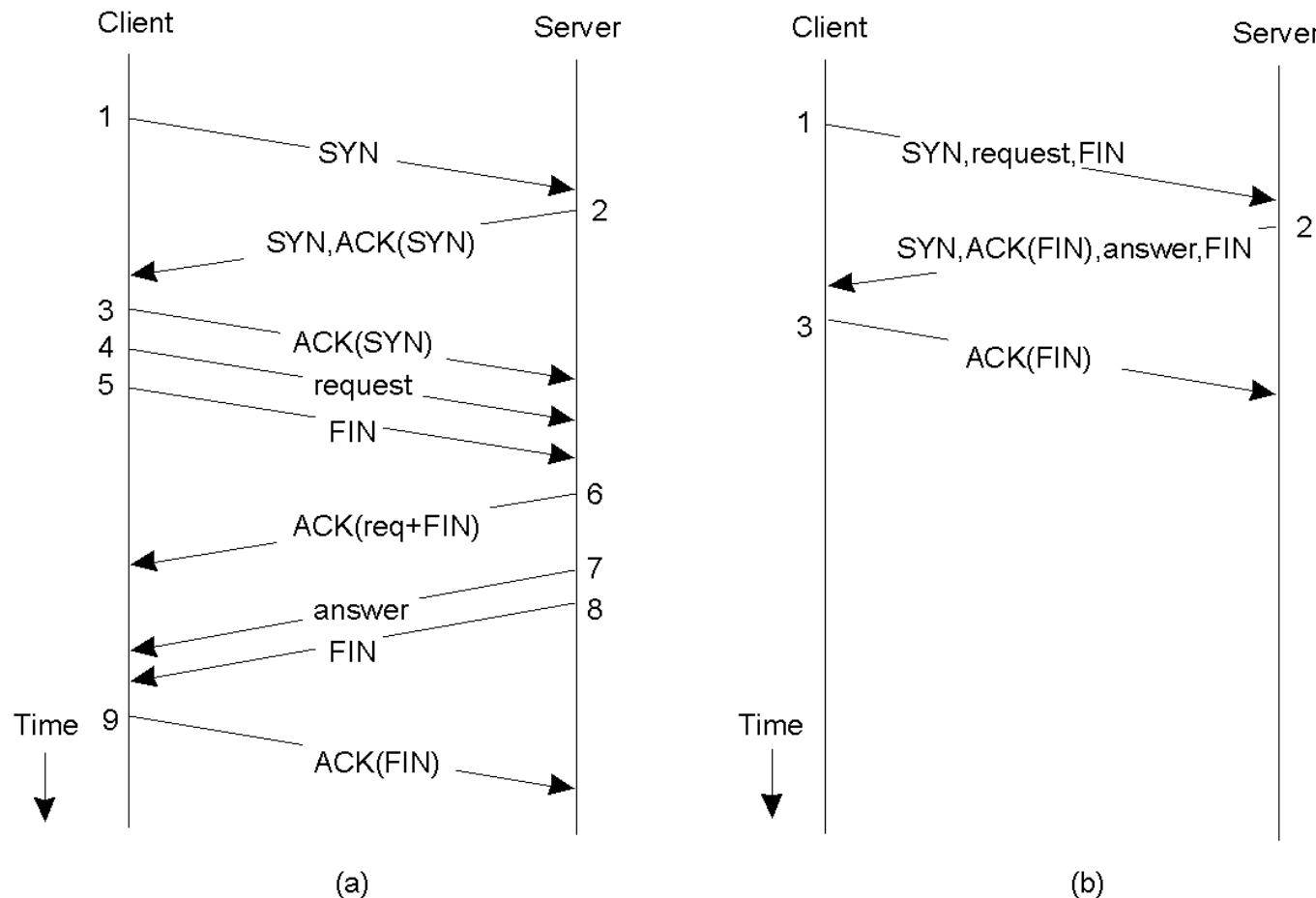
Figure 4-3. An adapted reference model for networked communication.

# Data Link Layer (ed. 1)



Discussion between a receiver and a sender in the data link layer.

# Client-Server TCP (ed. 1)



- a) Normal operation of TCP.
- b) Transactional TCP.

# Types of Communication

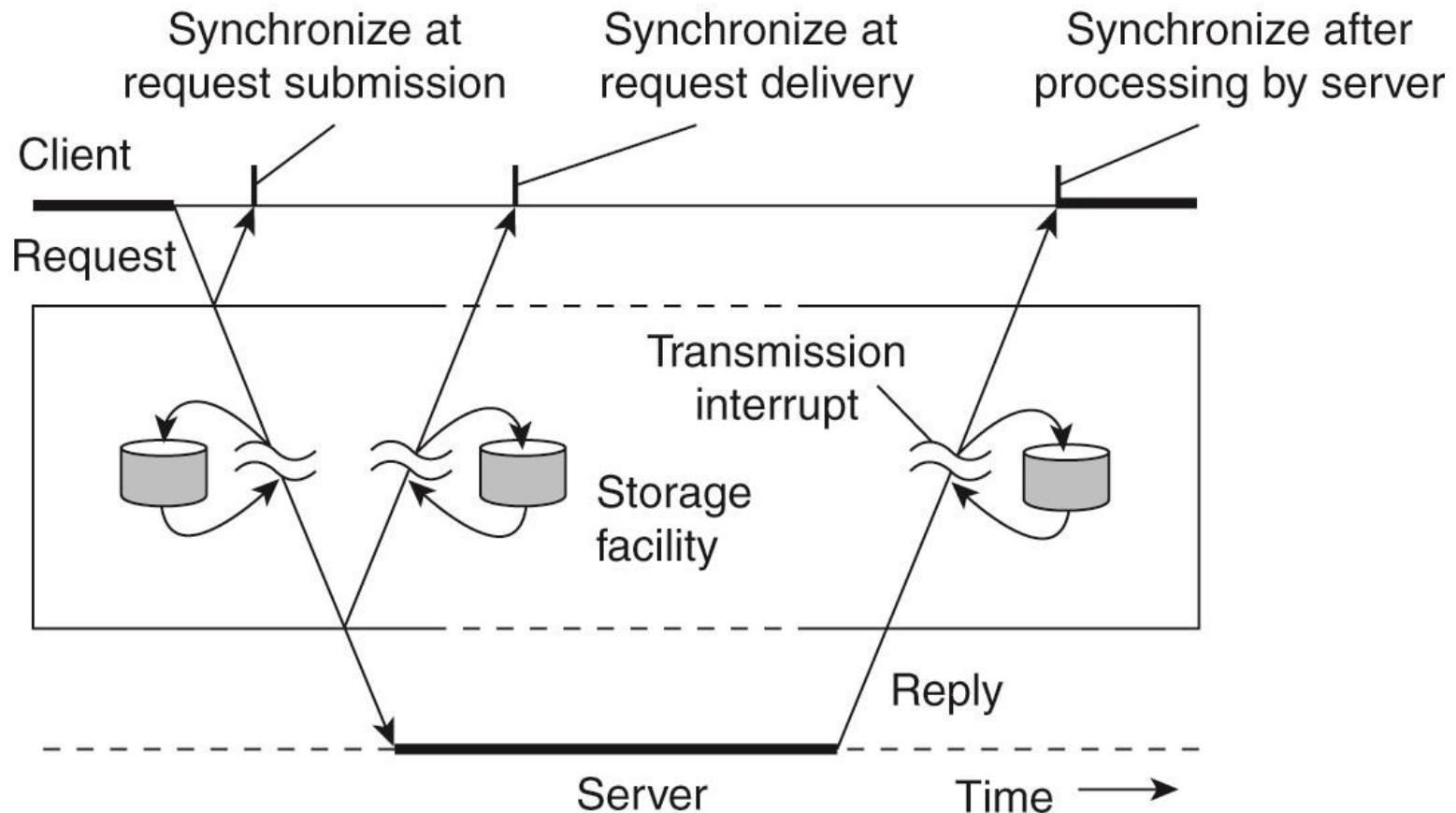


Figure 4-4. Viewing middleware as an intermediate (distributed) service in application-level communication.

# Conventional Procedure Call

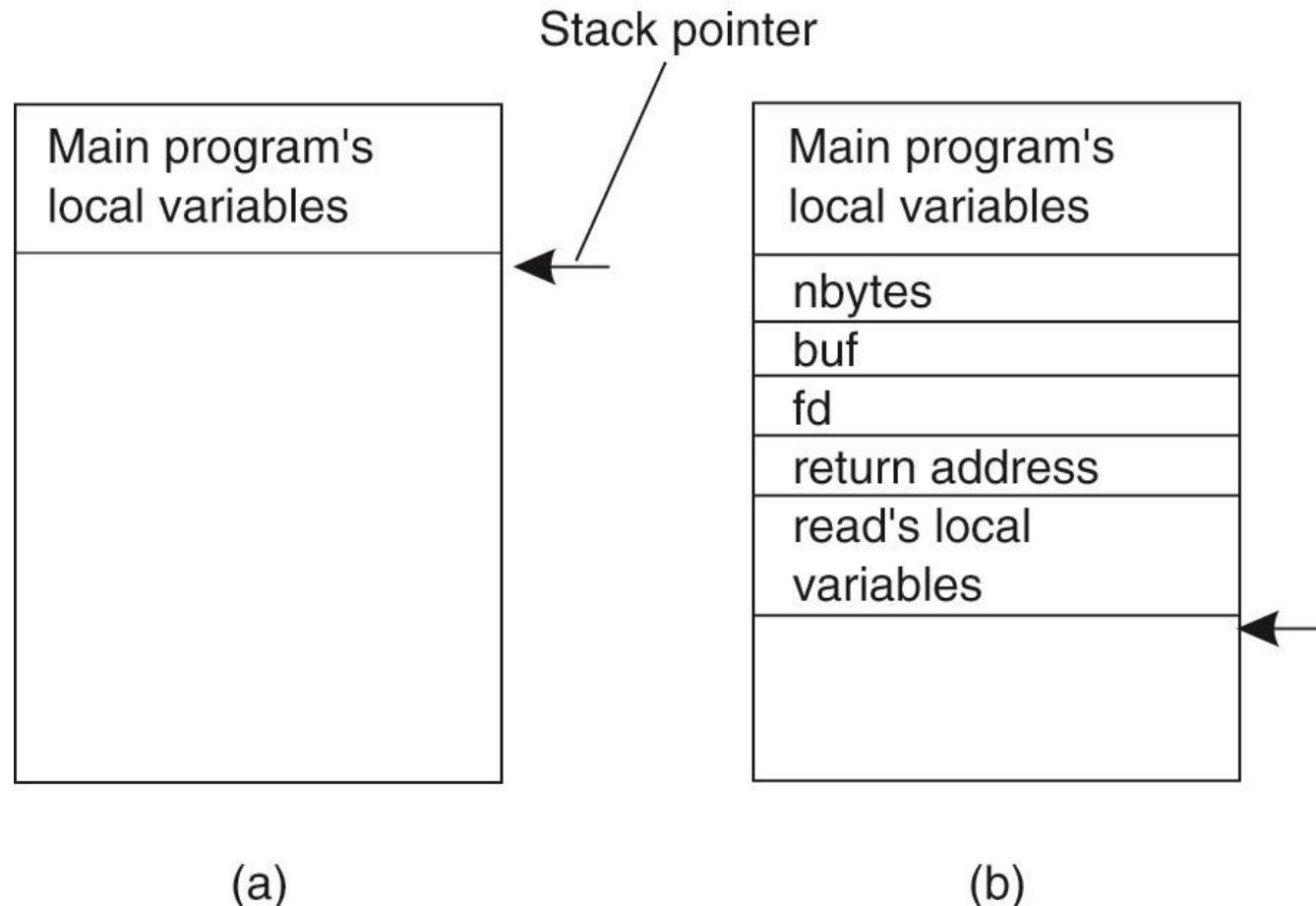


Figure 4-5. (a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

# Client and Server Stubs

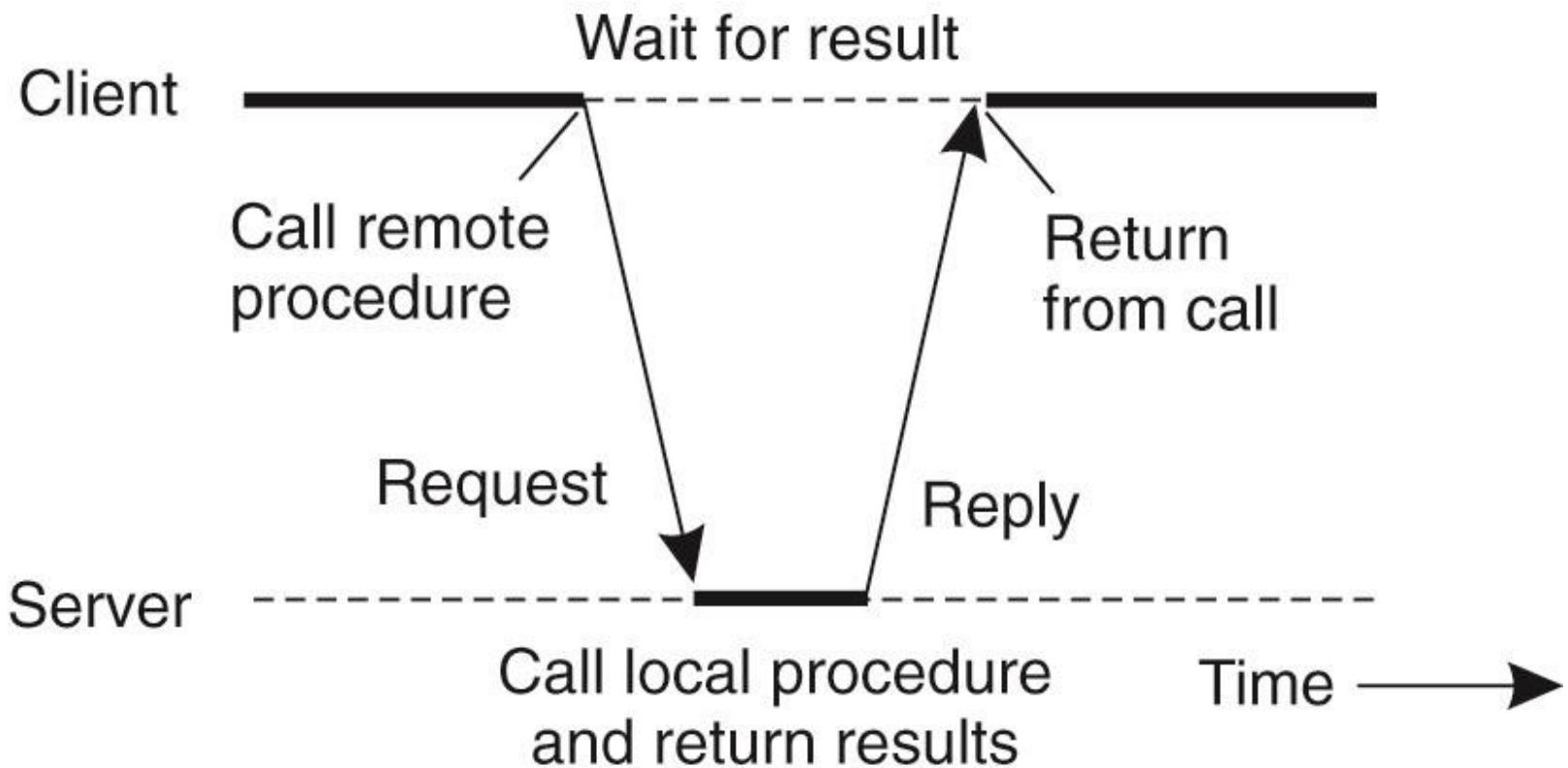


Figure 4-6. Principle of RPC between a client and server program.

# Steps of a Remote Procedure Call

1. Client stub builds message, calls local OS
2. Client's OS sends message to remote OS
3. Remote OS gives message to server stub
4. Server stub unpacks parameters, calls server
5. Server does work, returns result to the stub
6. Server stub packs it in message, calls local OS
7. Server's OS sends message to client's OS
8. Client's OS gives message to client stub
9. Stub unpacks result, returns to client

# Passing Value Parameters (1)

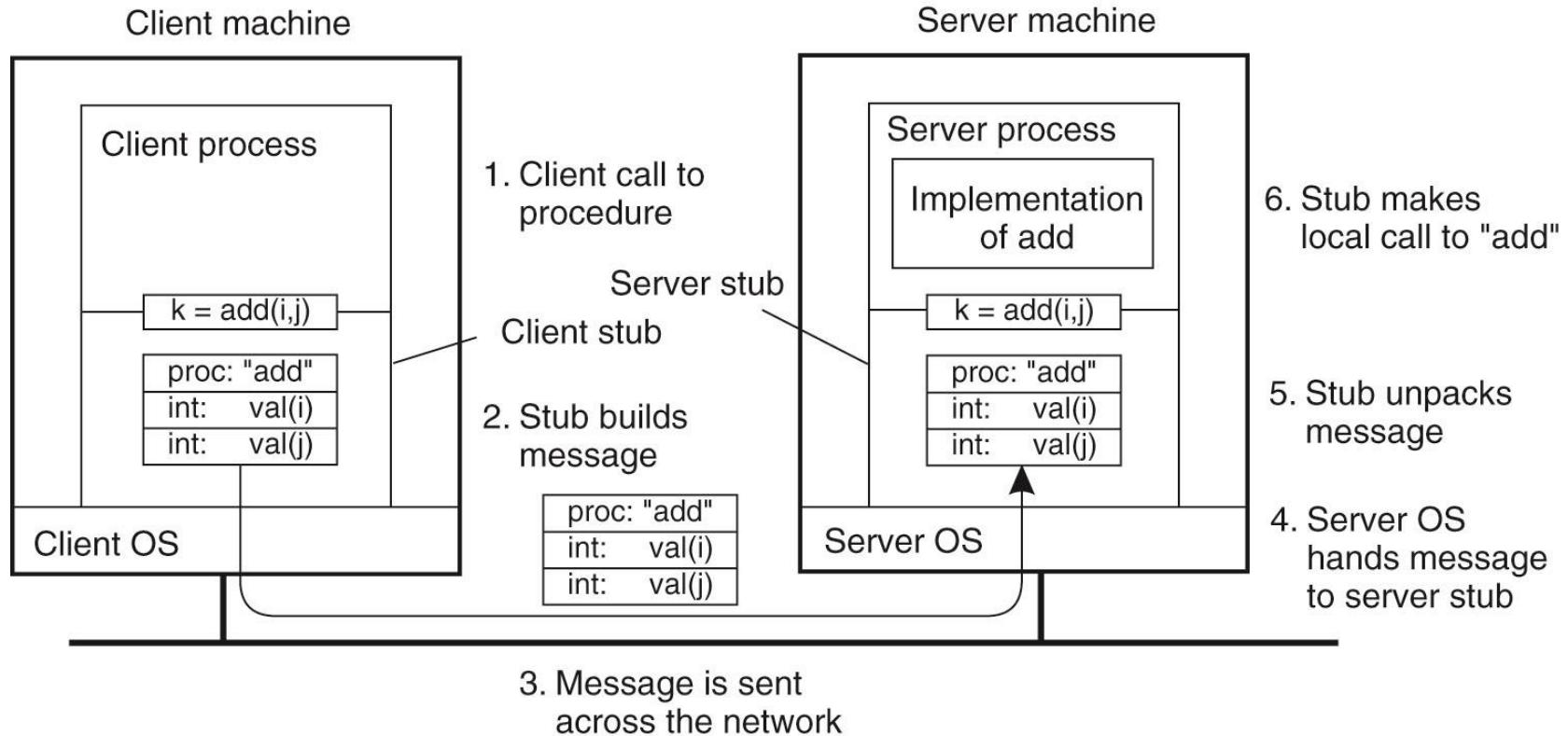


Figure 4-7. The steps involved in a doing a remote computation through RPC.

# Parameter Specification and Stub Generation

```
foobar( char x; float y; int z[5] )  
{  
    ....  
}
```

(a)

foobar's local variables	
	x
	y
	5
	z[0]
	z[1]
	z[2]
	z[3]
	z[4]

(b)

Figure 4-9. (a) A procedure. (b) The corresponding message.

# (A)synchronous RPC (1)

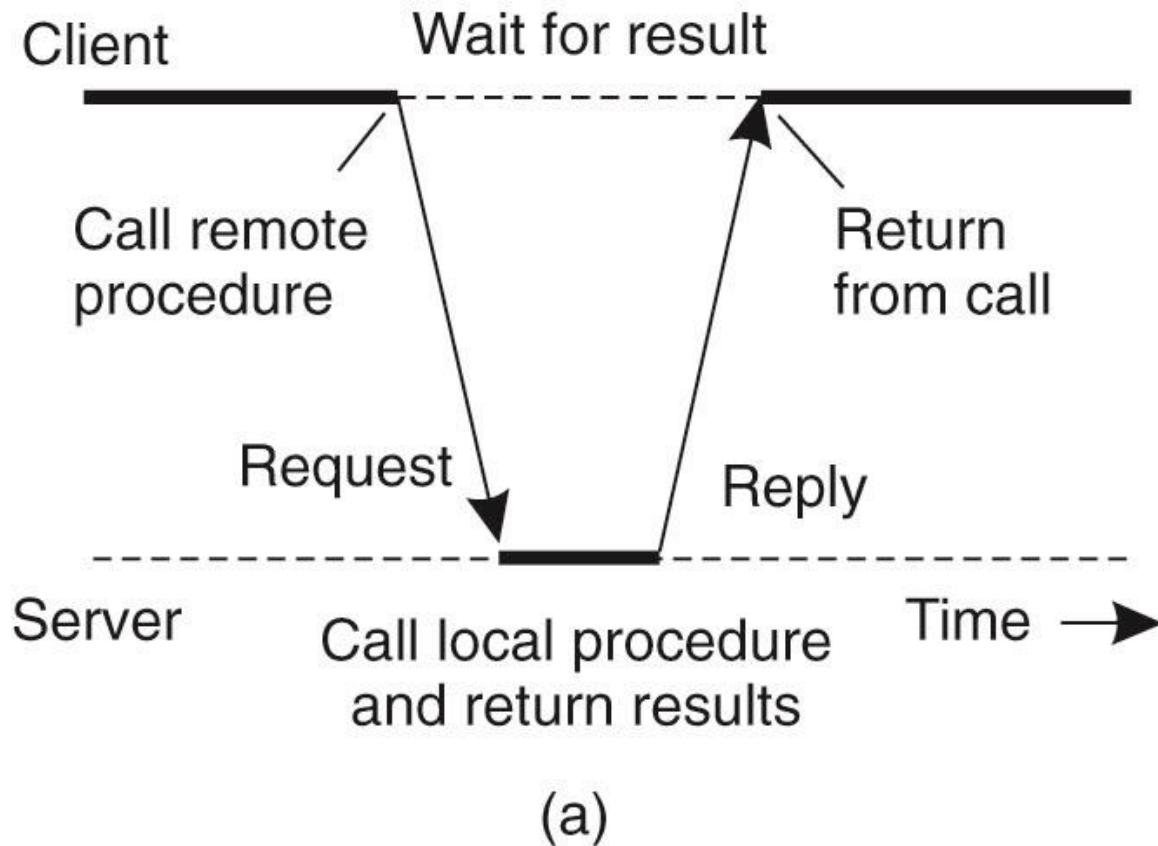
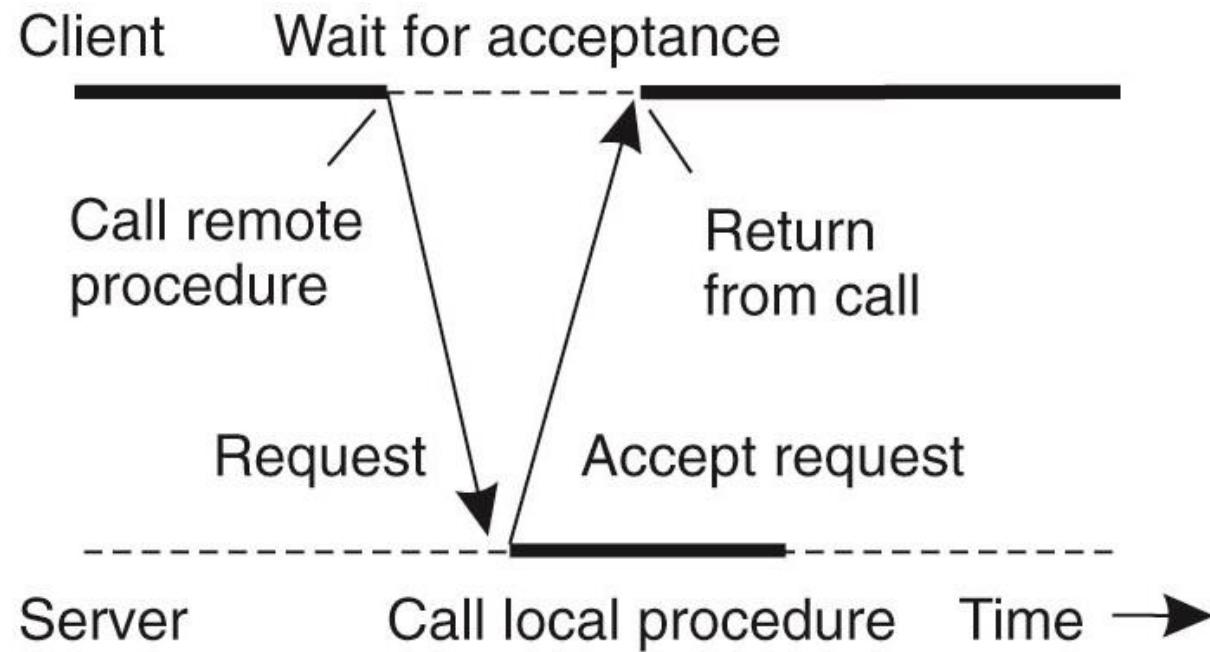


Figure 4-10. (a) The interaction between client and server in a traditional RPC.

# Asynchronous RPC (2)



(b)

Figure 4-10. (b) The interaction using asynchronous RPC.

# Asynchronous RPC (3)

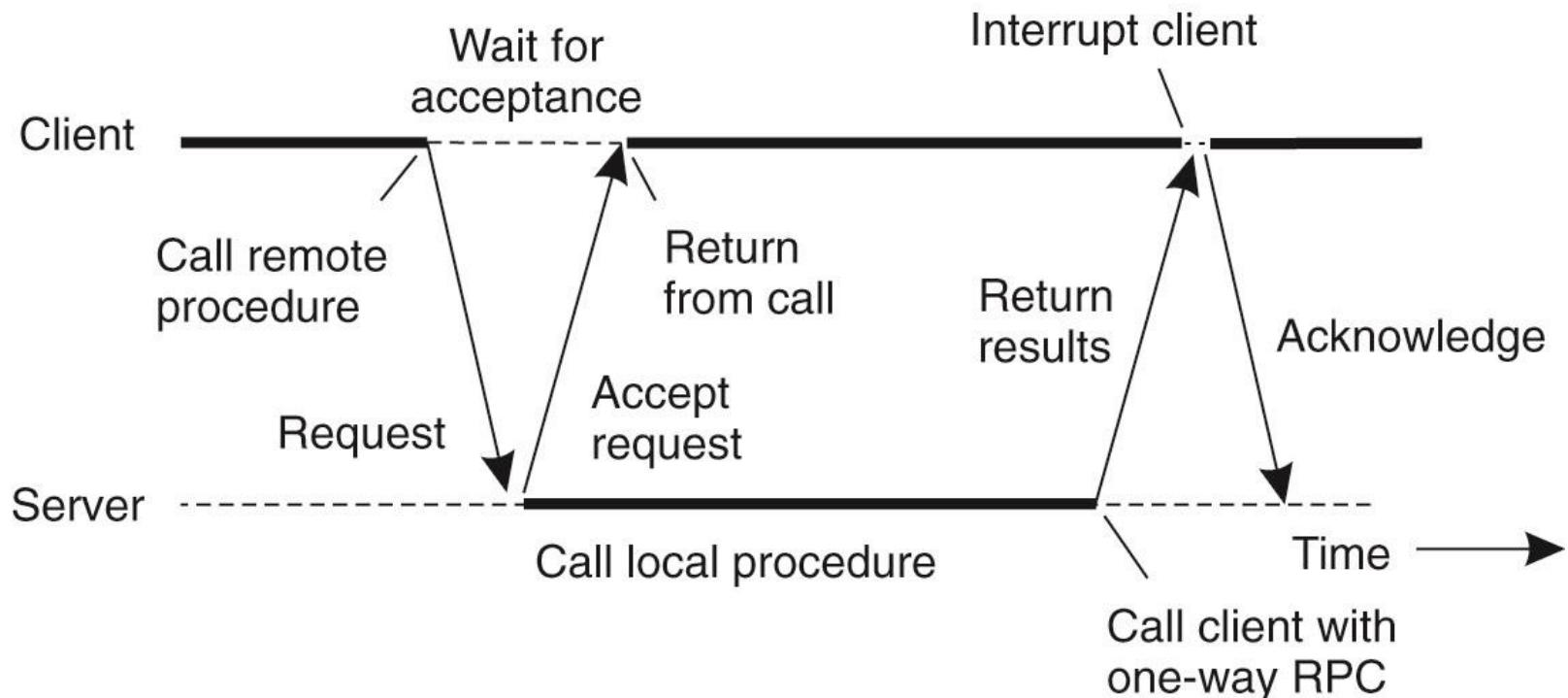


Figure 4-11. A client and server interacting through two asynchronous RPCs.

# Writing a Client and a Server (1)

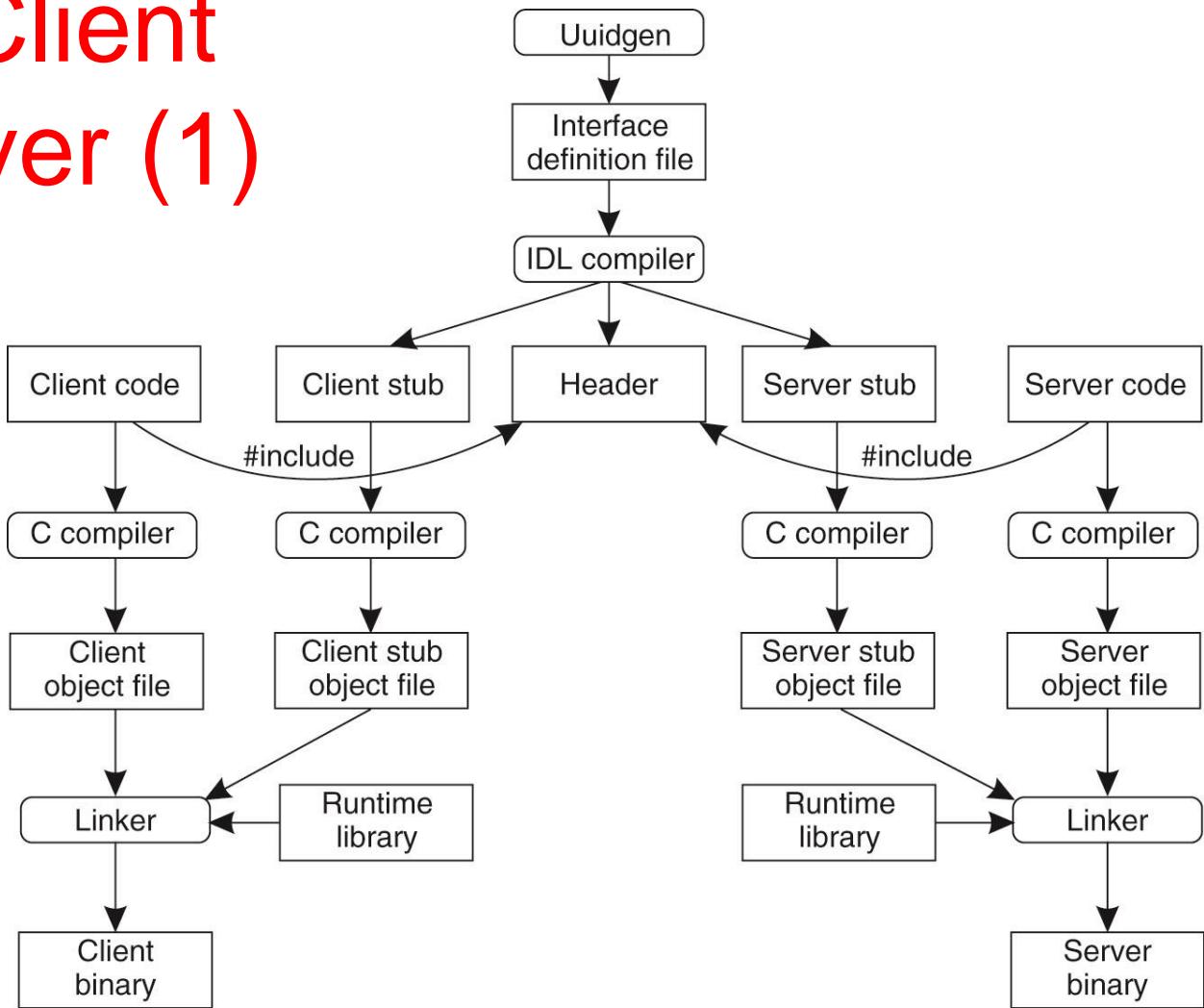


Figure 4-12. The steps in writing a client and a server in DCE RPC.

# Writing a Client and a Server (2)

Three files output by the IDL compiler:

- A header file (e.g., interface.h, in C terms).
- The client stub.
- The server stub.

# Binding a Client to a Server (1)

- Registration of a server makes it possible for a client to locate the server and bind to it.
- Server location is done in two steps:
  1. Locate the server's machine.
  2. Locate the server on that machine.

# Binding a Client to a Server (2)

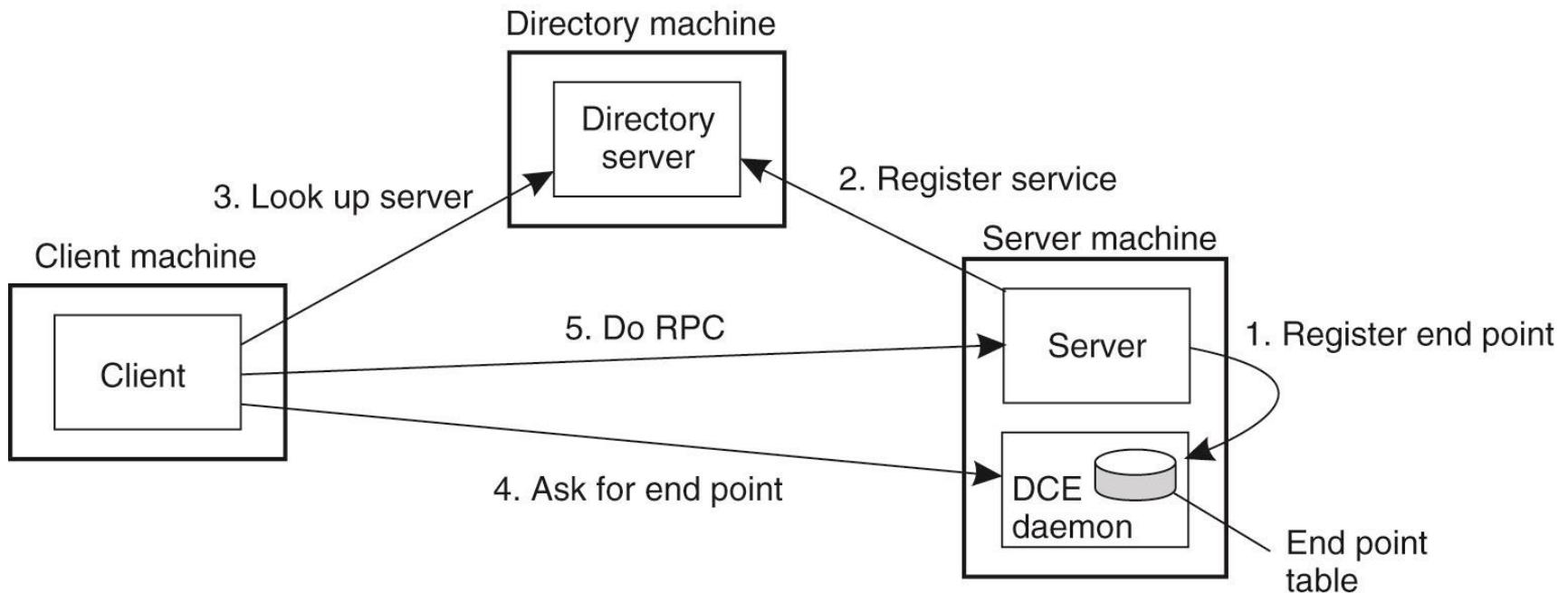
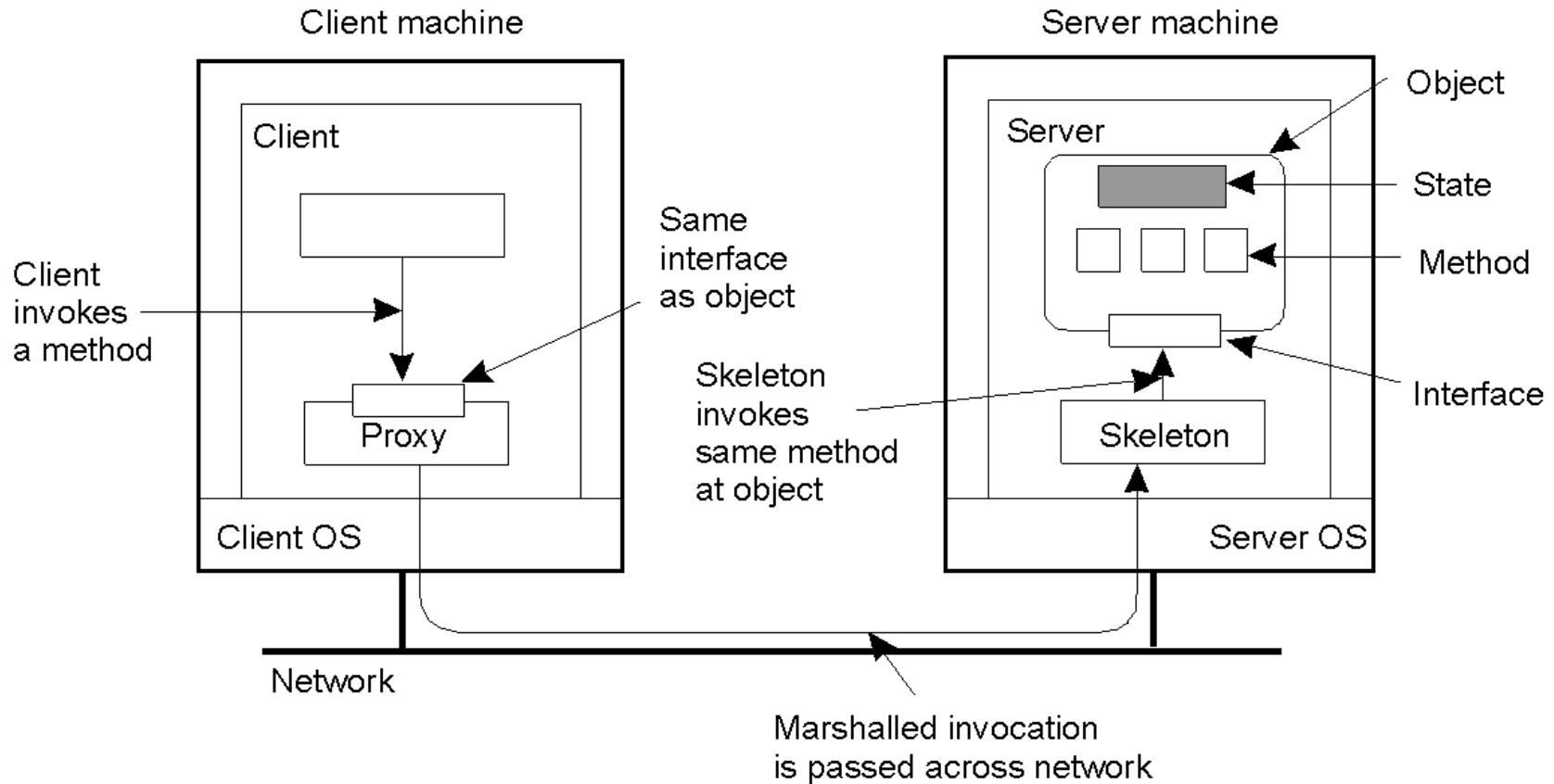


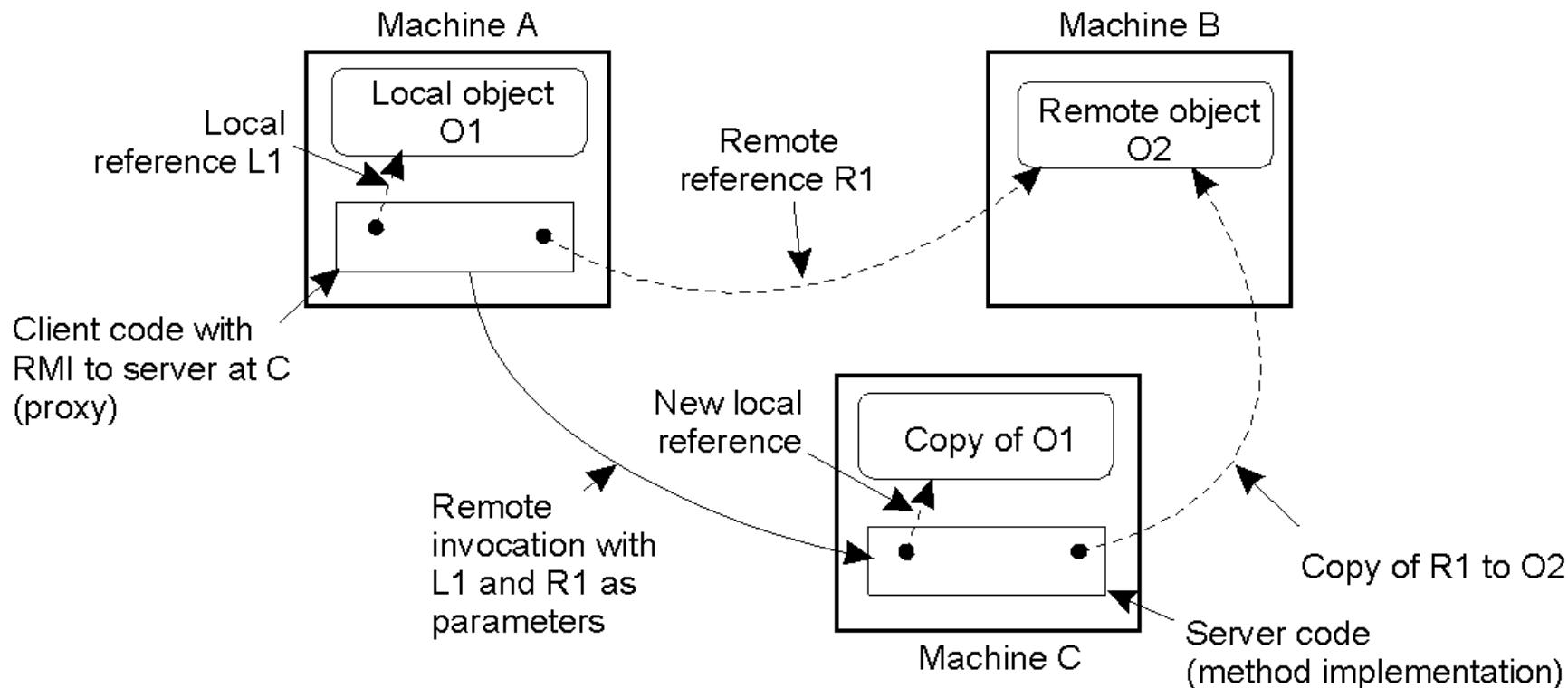
Figure 4-13. Client-to-server binding in DCE.

# Distributed Objects



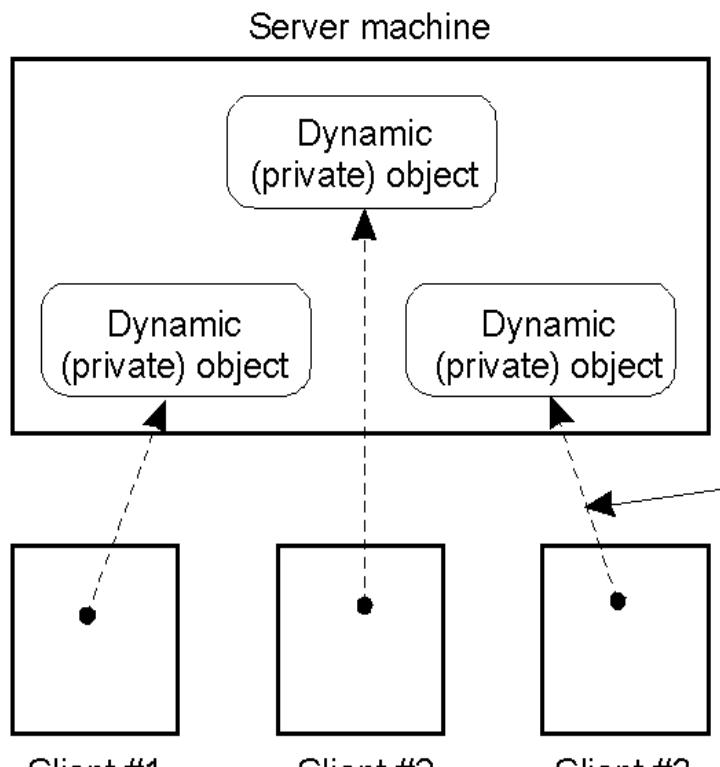
Common organization of a remote object with client-side proxy.

# Parameter Passing



The situation when passing an object by reference or by value.

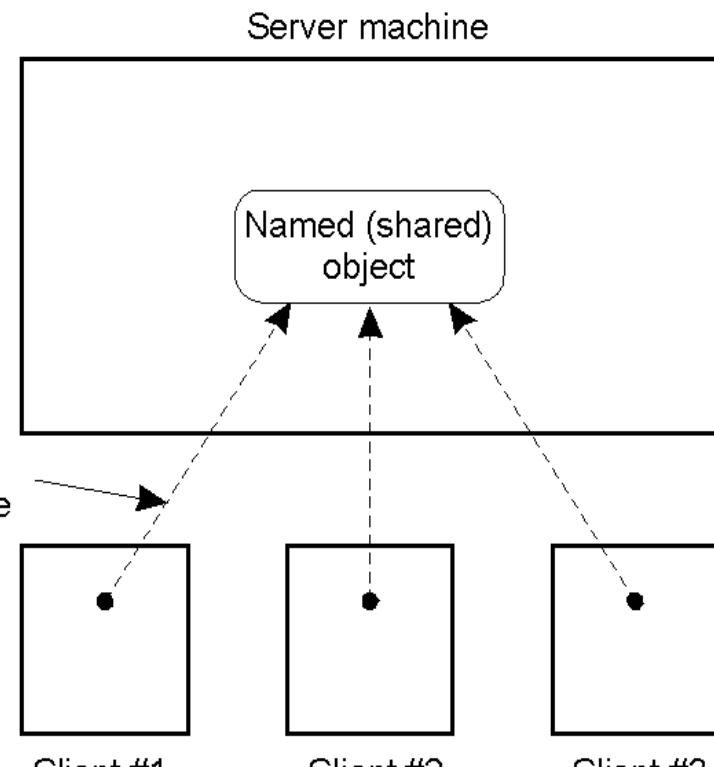
# The DCE Distributed-Object Model



(a)

2-19

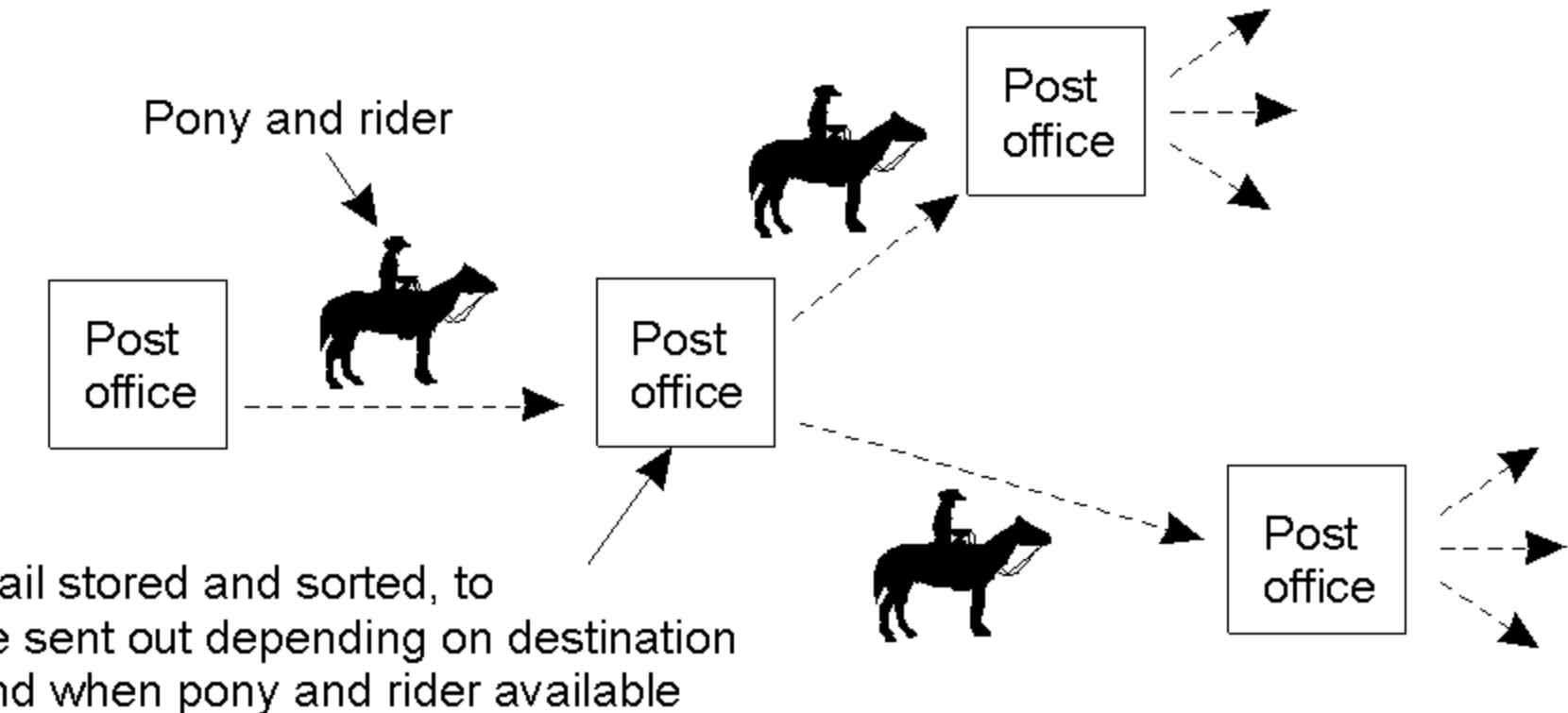
Remote  
reference



(b)

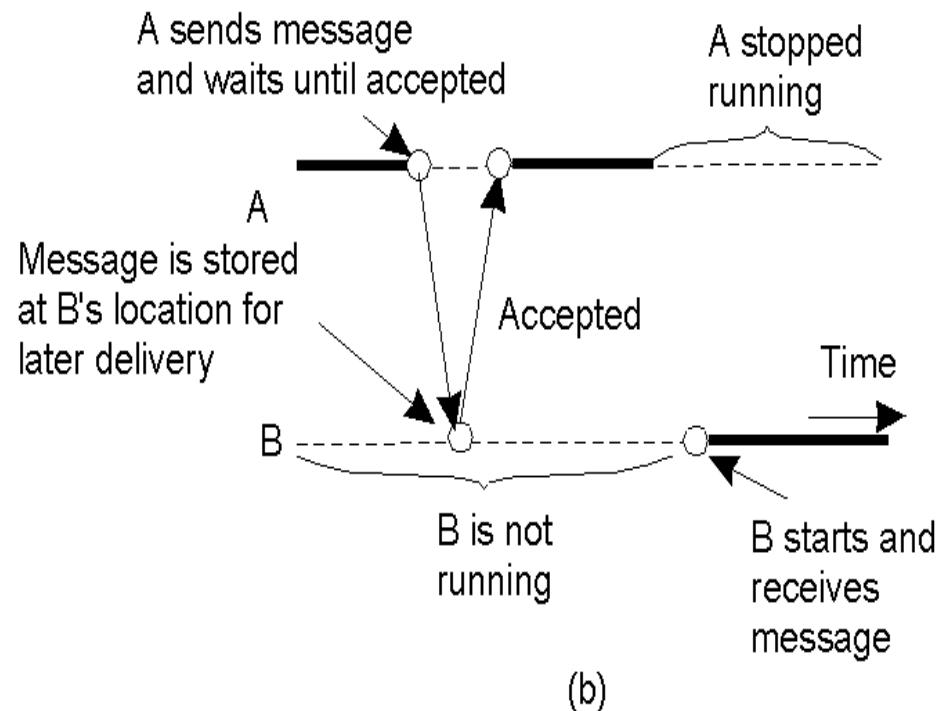
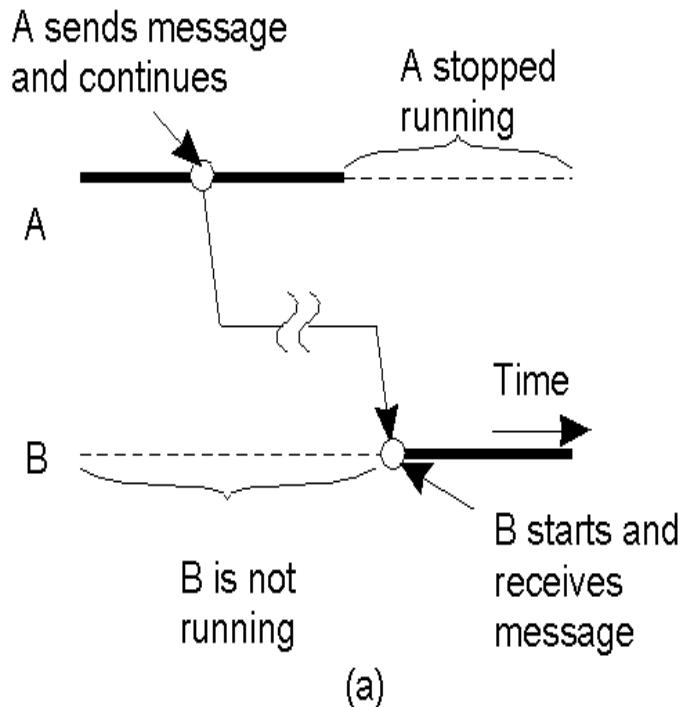
- a) Distributed dynamic objects in DCE.
- b) Distributed named objects

# Persistence and Synchronicity in Communication (2)



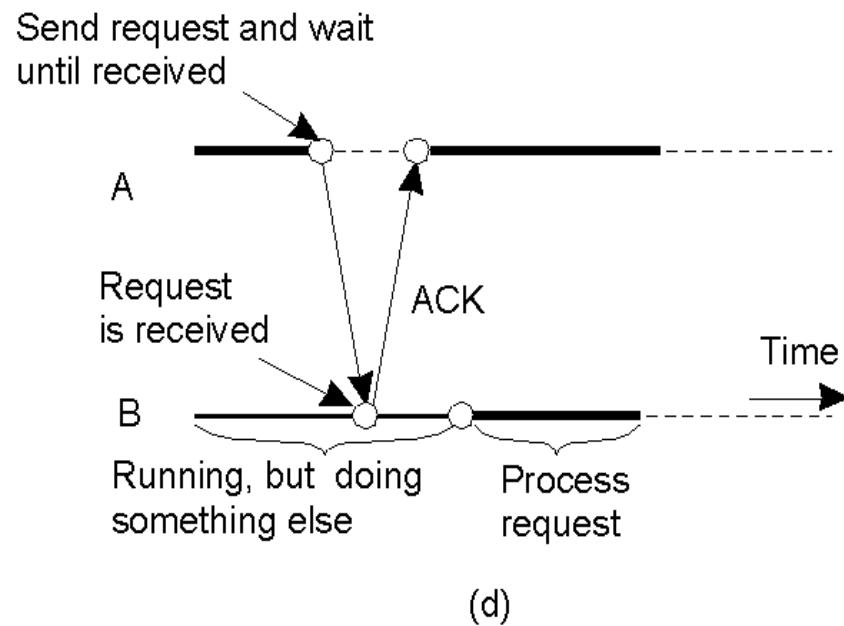
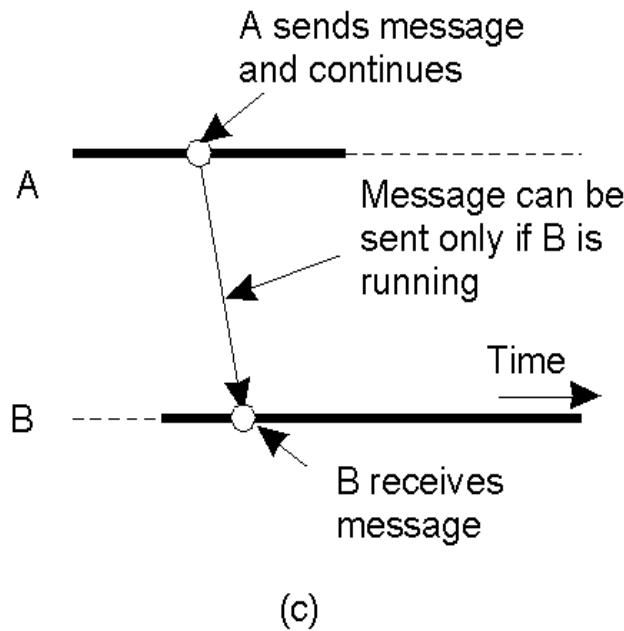
Persistent communication of letters back in the days of the Pony Express.

# Persistence and Synchronicity in Communication (3)



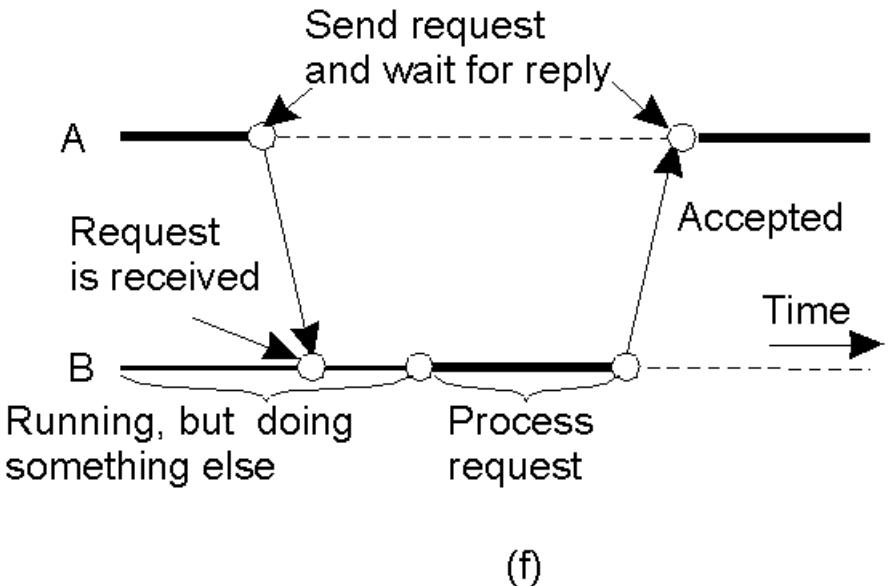
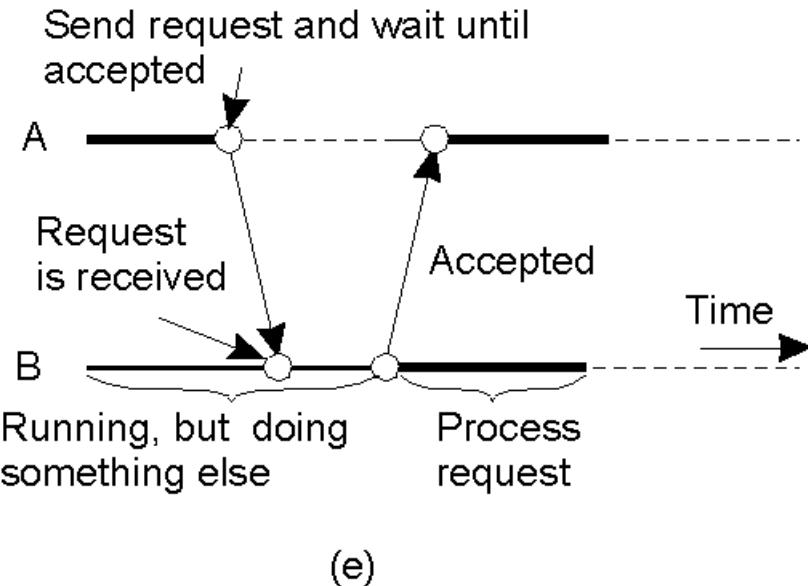
- a) Persistent asynchronous communication
- b) Persistent synchronous communication

# Persistence and Synchronicity in Communication (4)



- c) Transient asynchronous communication
- d) Receipt-based transient synchronous communication

# Persistence and Synchronicity in Communication (5)



- e) Delivery-based transient synchronous communication at message delivery
- f) Response-based transient synchronous communication

# Berkeley Sockets

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

Figure 4-14. The socket primitives for TCP/IP.

# Data Stream

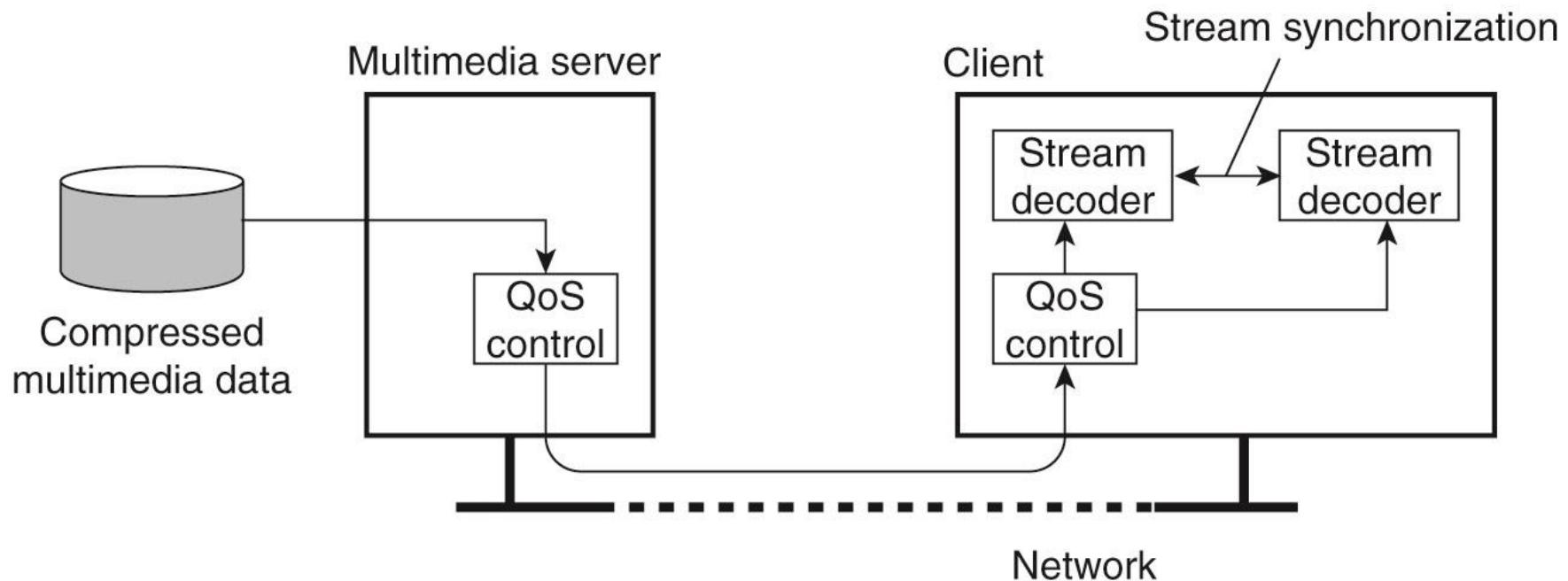


Figure 4-26. A general architecture for streaming stored multimedia data over a network.

# Data stream

- A sequence of data units
- Can be used for sending *discrete* items such as with our JokeServers
- Can be *continuous*, such as for audio or video

# Asynchronous Transmission Mode

- The data items are sent one after the other in sequence. No other timing constraints constraints.

# Synchronous Transmission Mode

- Maximum end-to-end delay
- No constraint on arriving early.
- Example is sensor data that is aggregated at intervals—must arrive in time, but no penalty if it arrives early.

# Isochronous Transmission Mode

- Data units are transferred *on time*
- Cannot arrive late, and cannot arrive early
- Early arrival can cause problems with buffer space.
- Requires bounds on *jitter*—the variation in the transfer time between units.
- Continuous data transfer using isochronous transfer mode is often simply referred to as *streaming data*

# Complex vs. Simple Streams

Simple:

- Sequenced data, homogenous

Complex

- Two or more substreams that are related.
- Stereo signal – two independent streams, but must be synchronized
- Movies – Video image, up to six channels of audio (or more), subtitles in different languages, etc.

# Continuous vs. discrete

Properties for Quality of Service:

- Discrete, static media:
  - Text, still images, executables, objects
  - How to degrade signal? E.g. compromise between delay, or loss of some data?
- Continuous media: sound files, movie files.
  - Temporal relationship is fundamental
  - How to degrade the signal? E.g., compromise between delay, or poor quality?

# Properties of streaming data

- Stored data streamed from files
- Live data streamed from a real-time source
- “Loop” delayed live streaming data
- Compressed data—how much compression?
- Secure data—how much overhead?  
authentication model at playback?

# Audio timing is critical

- Audio image is ephemeral and humans are very sensitive to it. By contrast the visual image *tends* to be stable.
- External *clocks* for high-quality recordings can cost tens of thousands of dollars, working at billionths of a second
- Audio *imaging* is also highly sensitive to timing issues, affecting *meaning* in the sound.
- The difference between a \$30,000 CD player and a \$100 player usually is *jitter*

# Modern sound

- Up to 10.1 channels of sound
- All must be carefully synchronized.
- Clock timing, *jitter*, and synchronization are all crucial to good sound.
- Sound can degrade in many ways:
- Hard to understand spoken dialogue (often *much* worse than analogue)
- *Harsh* sounding, or unpleasant, without quite realizing it. Affects body state.
- *Fatiguing* to listen for long periods.

# Many channels

- Many surround sound channels
- Video
- Picture-in-picture track for commentary
- Multiple subtitles
- Alternate audio for director commentary, actor commentary, different languages...

# Network level support – DiffServ on IP

- Differentiated services, 6 bit header
- Classify and mark packets by up to 64 ( $2^6$ ) categories. But typical is only 4:
  - Default per-hop behavior (PHB): Best effort, depends on current circumstances
  - Expedited forwarding: Low loss, low delay, low jitter, no more than 30% total traffic
  - Voice Admit: EF, but limits number to avoid conflicts
  - Assured forwarding: assured up to a saturation level, then not

# Streams and Quality of Service

Properties for Quality of Service:

- The required bit rate at which data should be transported.
- The maximum delay until a session has been set up
- The maximum end-to-end delay .
- The maximum delay variance, or jitter.
- The maximum round-trip delay.

# Enforcing QoS (1)

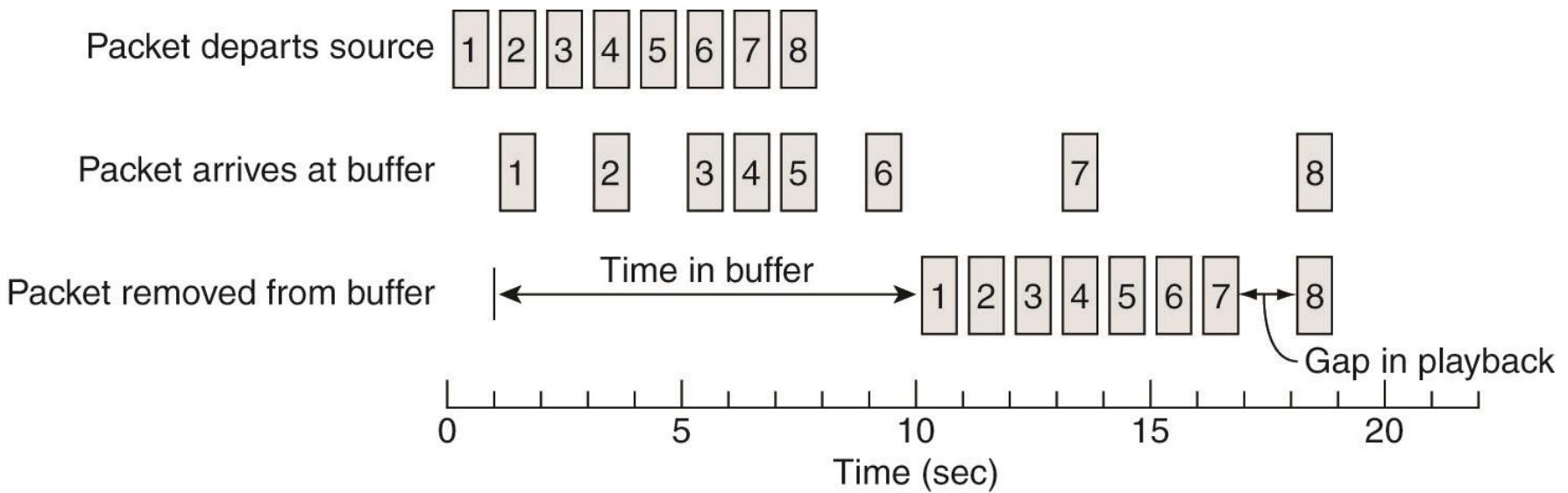


Figure 4-27. Using a buffer to reduce jitter.

# Bigger Buffer....

To reduce jitter:

- Constant *average delivery* is timely
- The bigger the buffer, the better extremes are handled.
- More delay in startup to fill the buffer part way.
- Always a tradeoff, and prediction is important:
  - E.g. – how much do you fill the buffer at startup?

# Enforcing QoS (2)

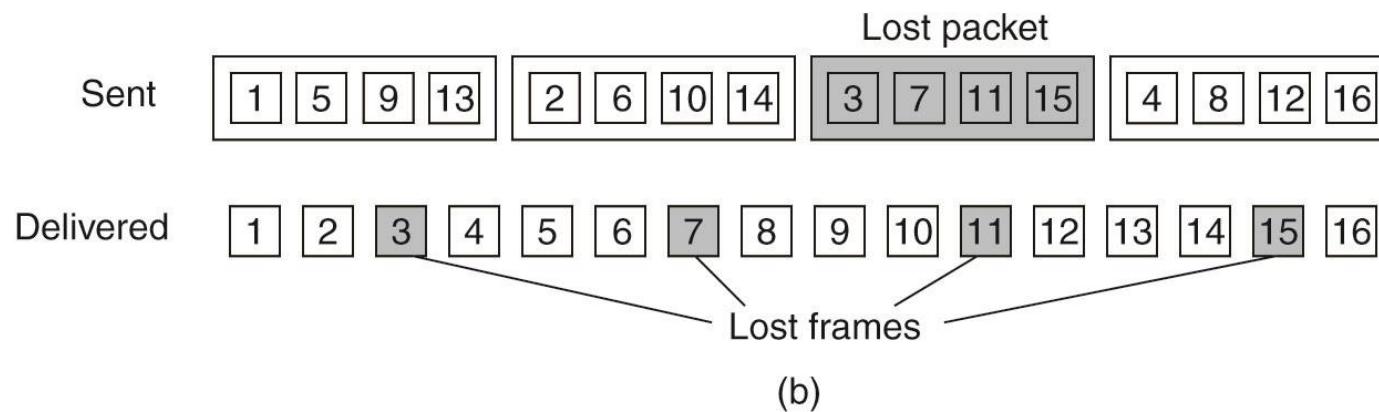
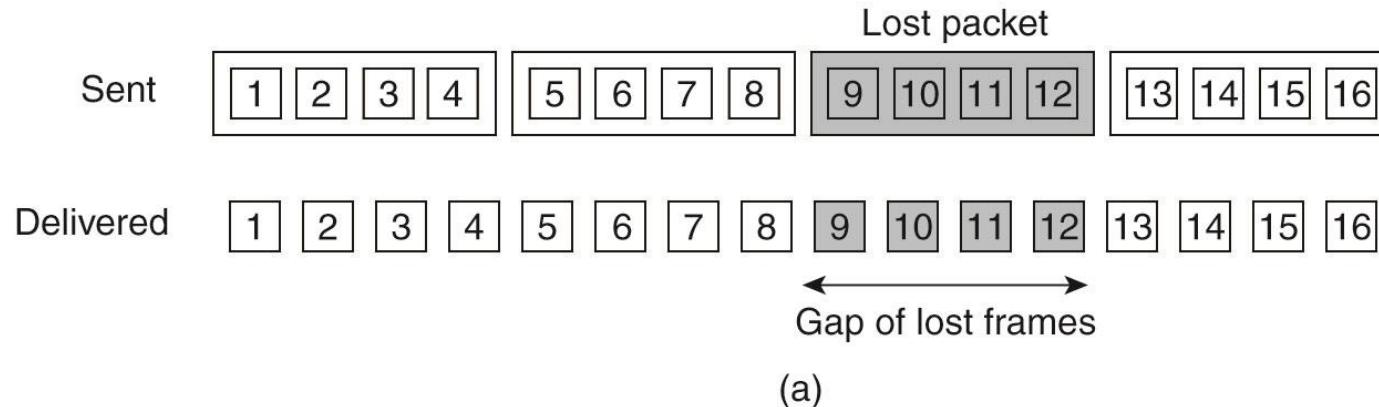


Figure 4-28. The effect of packet loss in (a) non interleaved transmission and (b) interleaved transmission.

# Synchronization Mechanisms (1)

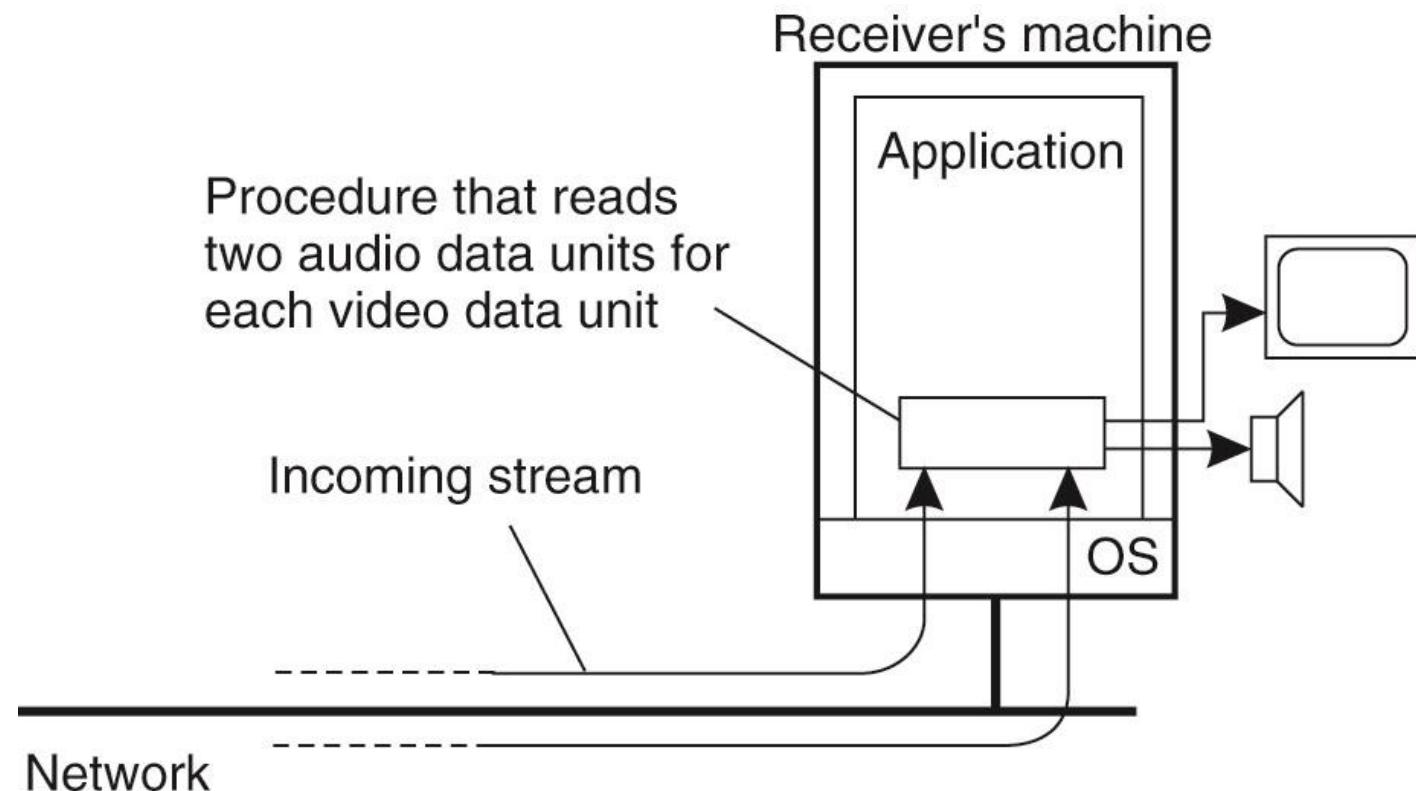


Figure 4-29. The principle of explicit synchronization on the level data units.

# Synchronization Mechanisms (2)

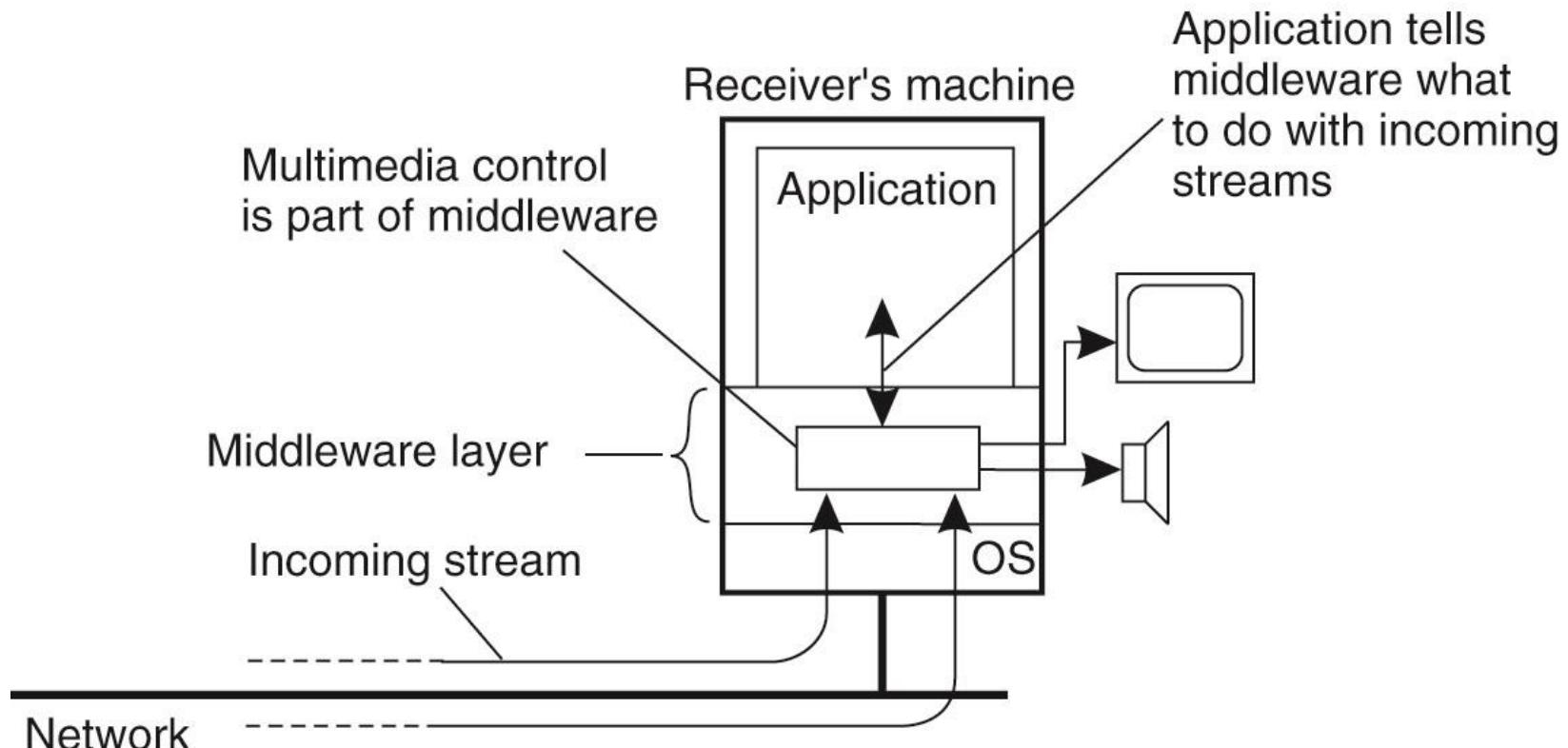


Figure 4-30. The principle of synchronization as supported by high-level interfaces.

# Multiplexing

- As synchronization gets more complex, there is more need for real-time distribution of synchronization specifications
- Multiplex all the data streams into one, and send the sync info too.
- Motion Picture Experts Group (MPEG) standards
- Can perform sync at sender if multiplexed, when receiver decodes, is already sync'd

# Structured Peer-to-Peer Architectures (1)

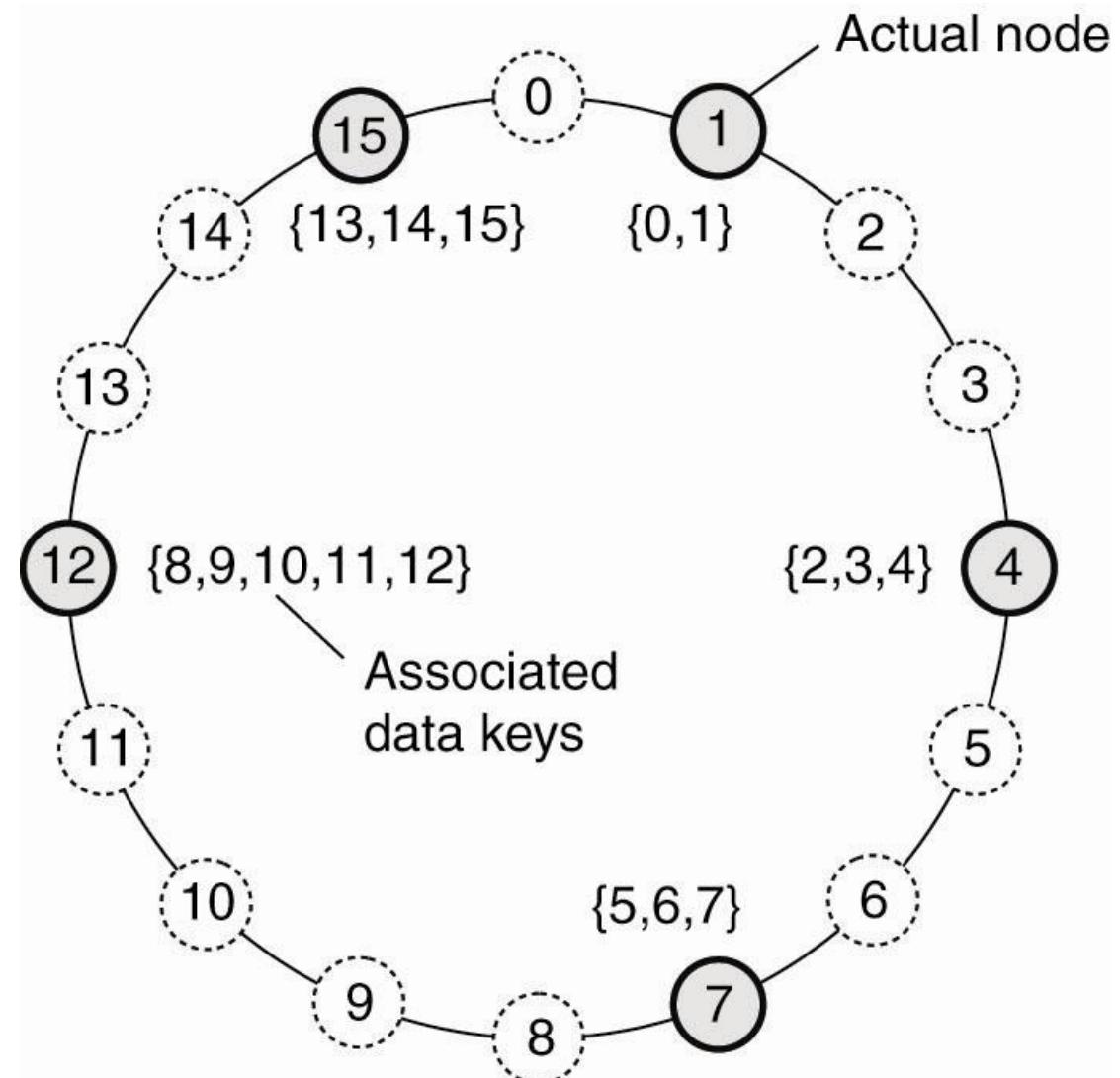


Figure 2-7. The mapping of data items onto nodes in Chord.

# Application-level Multicast

- Can cross router and network boundaries
- May not be as efficient as network-level multicast
- Must construct an *overlay-network*
- Organized as a *tree* – one path to each node
- ...or as a *mesh-network* – multiple paths between pairs of nodes. This is more robust when a node fails – alternate paths.

# MulticastTree from DHT (Chord)

- Any DHT node can accept Mcast commands
- Nodes can come and go from the DHT at will
- Nodes can join up with and drop from the Mcast group at will.
- Mcast nodes can be DHT members, or linked from DHT members.

- Create(M\_ID)
- Join (M\_ID, process)
- Leave (M\_ID, process)
- Mcast(M\_ID, message)
- Destroy(M\_ID)

- Generate Multicast ID  $M$ , a random value—this is the NAME of the multicast group.  $M$  is handled by the appropriate DHT node.
- Send build msg  $DHT(M)$  to any DHT node, this gets passed along to the right DHT node which becomes the root of the multicast group where messages are processed, archived, forwarded to the group, etc.

- To join an Mcast group, any node marks itself as being a member of the group for  $M$  then, if it is not a DHT node, it passes the request to a DHT node.
- Any DHT node can accept a request (including from itself) to join an Mcast group, which is passed up to the Mcast root for  $M$ .

- If an intermediary DHT node has not seen a request before, it becomes the *parent* of the new node in the multicast tree. It marks itself as a *forwarder*.
- Then it forwards a request, along the DHT path toward the root. Forwarders also need parents. (It can, itself, join the group later by simply changing its own status from forwarder to member.)

- If the DHT node has *already* seen a request for *any node* to join the multicast group  $M$ , it adds the previous node as its own *child*, and does not forward the request up toward the root (because it has done so previously).

- Nodes wishing to join the multicast group are theoretically independent of the DHT. In this way, a DHT node may have many Mcast children for the group  $M$ , one of which might be in the DHT

- Three types of DHT nodes:
  - Not (yet) participating
  - Pure forwarders
  - Forwarder-members.
- To send a multicast message, just issue...

# Sending Multicast messages

- $M\_Cast(M, \text{ message})$
- These are passed up to the root for the Mcast group identified by  $M$
- The root then processes the request (censors / archives / etc.) and then issues the multicast.
- $M\_Cast$  group members get the messages, forwarders and members forward them.

- To change from forwarder to member is easy. Just flip a bit.
- To change from member to forwarder is easy. Just flip the bit back.

# Multiple groups

- A DHT can support as many Mcast groups as there are ID numbers in the DHT hash universe. Each Mcast group is distinct. E.g., a four-bit hash identifier for the DHT allows for 16 Mcast groups.
- To add another Mcast group, the procedure is the same. Each DHT keeps an *array* or *list* of forwarding pointers, with one element for each Mcast group

# Flexibility

- Nodes can leave the Mcast group at will: DHT members become forwarders. External nodes are dropped as children, or else stay as children but stop receiving messages, if they will join again later.
- Nodes can join and leave the DHT in the usual way by splitting and combining Mcast responsibilities, moving them as necessary.

# Overlays

- In the following slide, there is a *logical* path from *Node B* to *Node D* but this is implemented via the intermediary *physical* path from *Node A* to *Node C*.
- A packet that is multicast from *Node A* (A to B, A to C, B to D) might end up being sent twice over the physical path from A to C.

# Overlay Construction

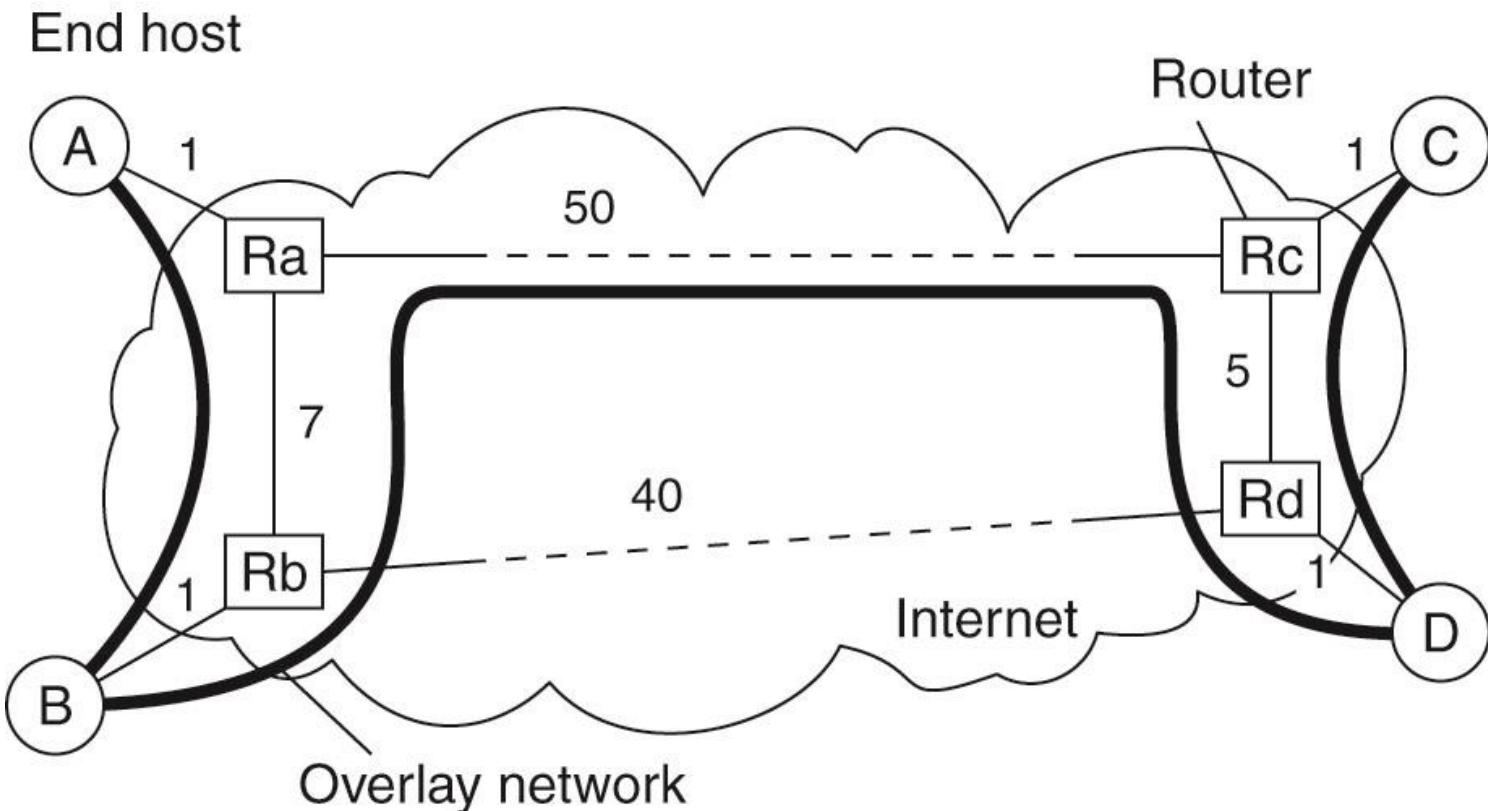


Figure 4-31. The relation between links in an overlay and actual network-level routes.

# Gossip-based dissemination – Epidemic protocols

- Simple algorithm
- Robust
- Needs only local information
- A simplifying assumption is that sending is initiated by a single node, thus avoiding multiple-write problems.
- Nodes are *infected* (Seen the msg), or *susceptible* (have not seen the msg).
- A node is *removed* if not willing to infect others.

# Information Dissemination Models (1)

- Pick another node at random
- Anti-entropy propagation model
  - Node P picks another node Q at random
  - Subsequently exchanges updates with Q
- Approaches to exchanging updates
  - P only pushes its own updates to Q
  - P only pulls in new updates from Q
  - P and Q send updates to each other

# Anti-entropy propagation model

- Some node P in a network picks another node Q at random.
- Then exchange updates with Q.

# Push-only

- P pushes its own update to Q
- A-> B, C, D, E, F
- D-> G, H, I, J
- J-> K, L, M, N, O, P
- How long to infect from A to P?
- Bad choice for rapid updates.
  - Only infected nodes can propagate
  - Probability of selecting an already infected node increases as the message spreads.

# Pull only

- P only pulls a new update from Q
- Works better when many are already infected
- Triggered by susceptible nodes, works better to spread to them

# Push-pull

- Works best overall, though the others will work as well, and are simpler.
- Depends on speed of spread requirements and efficiency needs
- $O(\log(N))$  “rounds” where a round is *all* nodes have chosen to exchange updates with some node, and  $N$  is number of nodes. So scales up.

# Gossiping / rumor spreading

- Keep telling the “hot news” until it appears that some number [“ $k$ ”] of nodes have already heard the news, then get tired and give up. Approximate this with  $1/k$  probability of giving up.

# Gossiping / rumor spreading

- Sometimes pure gossiping will leave some fraction of the nodes still not infected. Why keep gossiping if everyone you call already knows the gossip?
- Combining gossip with anti-entropy can solve this problem.
- The interaction between processes in gossip-based dissemination is small, so it tends to scale well.
- *Directional gossiping* favors nodes with fewer network connections, considering them bridges to other parts of the graph.

# Information Dissemination Models (2)

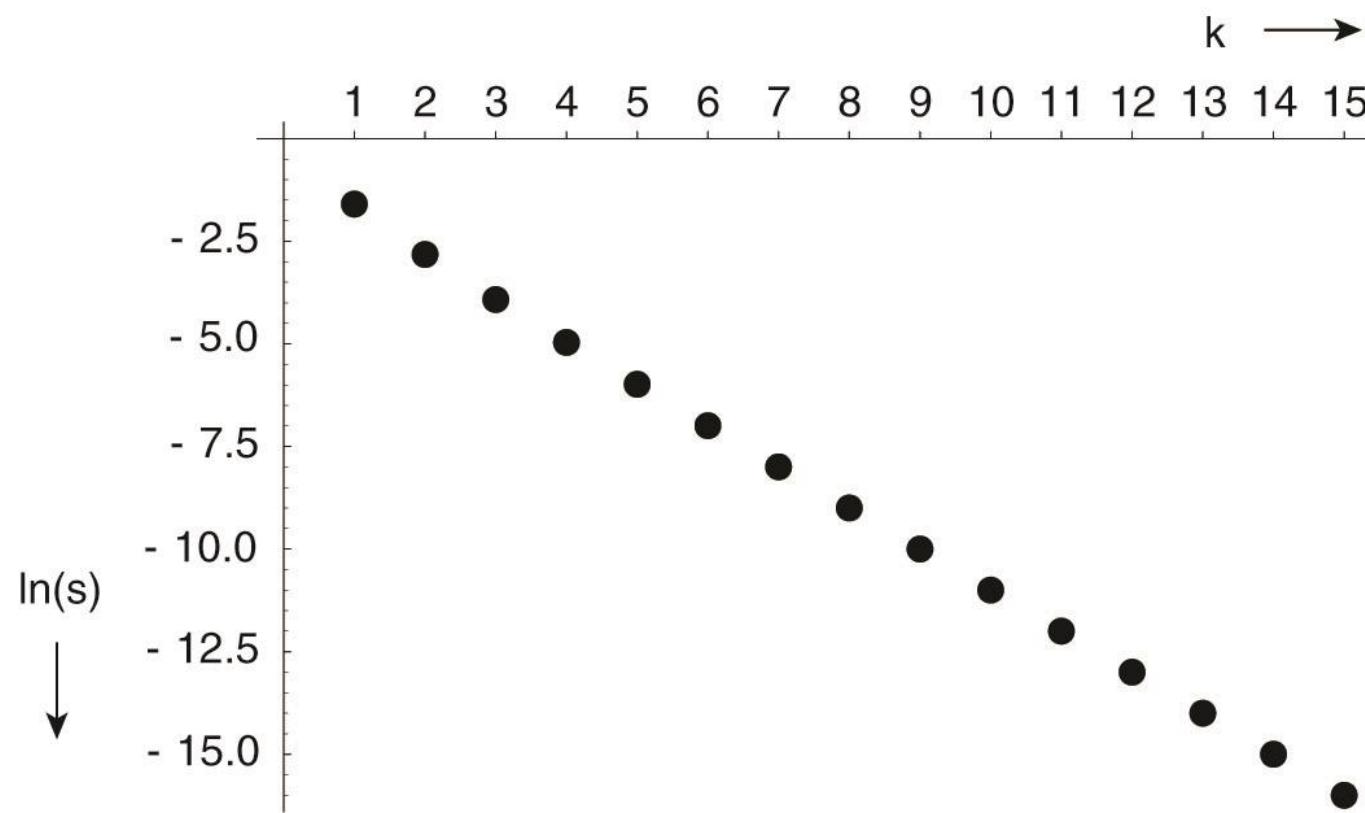


Figure 4-32. The relation between the fraction  $s$  of update-ignorant nodes and the parameter  $k$  in pure gossiping. The graph displays  $\ln(s)$  as a function of  $k$ .

# Removing data in epidemics

- The problem is that if all traces of the data have been removed, then later-arriving echoes of updates may unwittingly restore it.
- Spread *death-certificates* instead. These mark data as deleted.
- May wish to expire the death-certificates for cleanup.
- Some nodes keep old certificates around so that if there is a late-arriving update, the death-certificate can be re-broadcast

# Calculate the average of the network using an epidemic

- For nodes  $j$  and  $i$  that trade values  $X$  then set  $X-i$  and  $X-j$  to be the value  $(X-i + X-j) / 2$ . [The average – just store this somewhere, this is the new value for  $X$  for the purpose of the calculation]
- Over time all the nodes will have the average stored locally, and something close to it as time progresses

# Calculate the size of the network using an epidemic

- Some node  $q$  sets its  $X-q = 1$ .
- All other nodes  $n$  set their  $X-n = 0$ .
- For nodes  $j$  and  $i$  that trade values  $X$  then set  $X-i$  and  $X-j$  to be the value  $(X-i + X-j) / 2$ . [The average – just store this somewhere, this is the new value for  $X$  for the purpose of the calculation]
- Over time all the nodes  $n$  will have a value from which they can calculate the size of the network as  $1 / X-n$ .

DISTRIBUTED SYSTEMS  
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM  
MAARTEN VAN STEEN

Chapter 6  
Synchronization  
(Elliott additions)

# Clock Synchronization

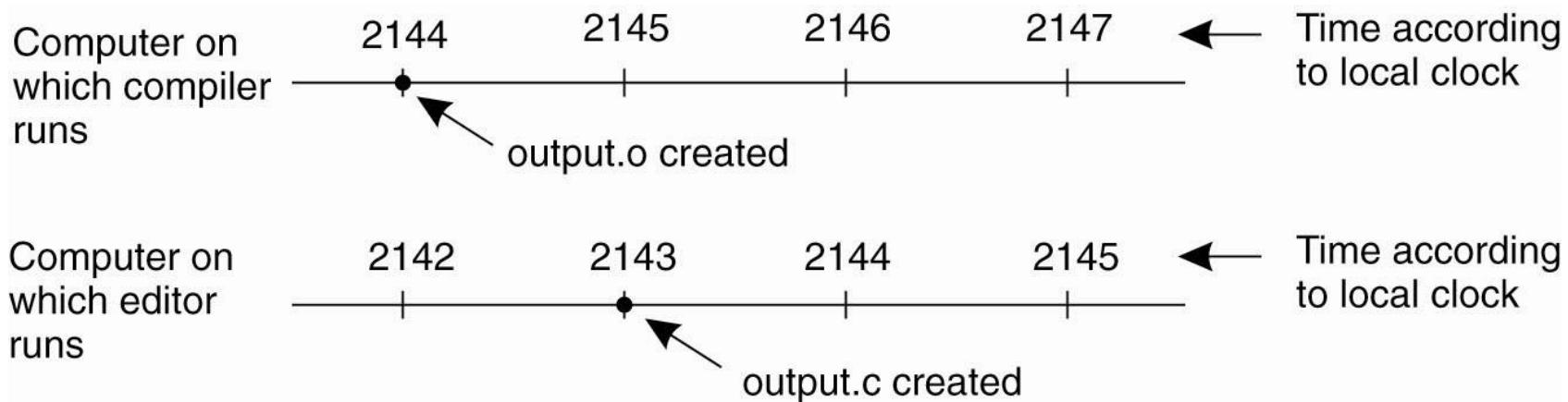


Figure 6-1. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

# Physical Clocks (1)

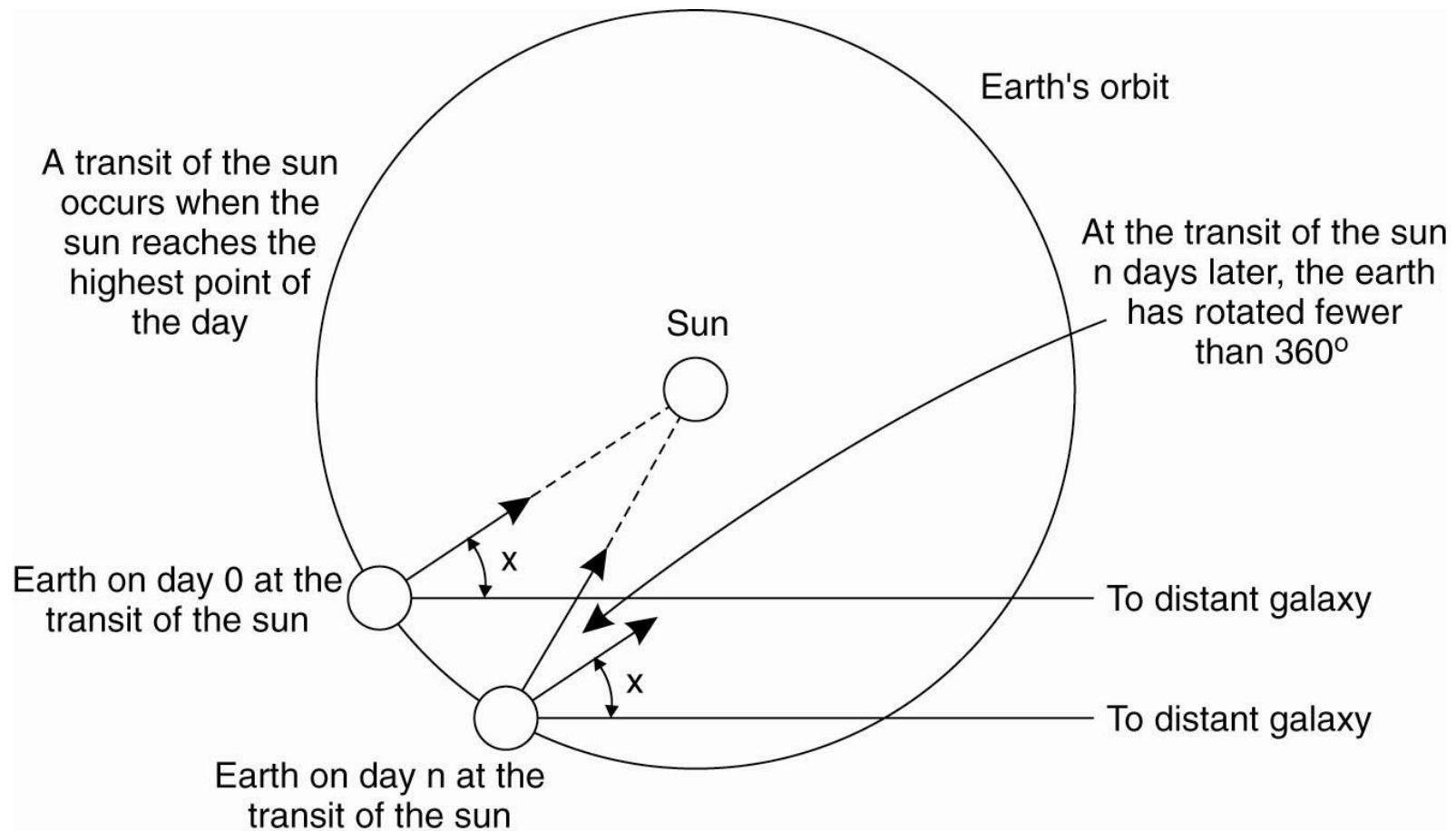


Figure 6-2. Computation of the mean solar day.

- Figuring a standard measure is hard.
- (3600x24) Solar second is  $1/86400^{\text{th}}$  of a solar day.
- But earth slowing down, and also wobbles.
- 300 million years ago we had 400 days.
- Temporary fluctuations, probably due to molten iron core.

- Mean solar day takes an average.
- *Longitude* by Dava Sobel—what time is it?
- Cesium 133 atom, 9,192,631,770 transitions in a second.  
(1948)
- TAI International atomic time, cesium 133 counter since midnight 1958-01-01.
- But 86,400 is now ~3msec less than a solar day, because slowing of Earth's spin!

- Julian calendar used from 46 B.C., but miscalculated the solar year by 11 minutes.
- Politics! 1582 Pope Gregory removed 10 days from the Julian calendar. Rent/interest vs. month's pay. September 2 became September 14 in 1752 in England leading to unrest.
- “Gregorian Calendar” is still off by 26 seconds.

- BIH (Bureau international de l'Heure) collects reports of Cesium clocks makes and average.
- Cesium time is just a count since 1958
- Leap seconds introduced.
- Universal Coordinated Time UTC

# Better clocks

- Cesium-133 under microwave bombardment yields accuracy of about 1 second drift in 200 million years—not good enough.
- Optical clocks (under development) are 100 times more accurate.

# Physical Clocks (2)

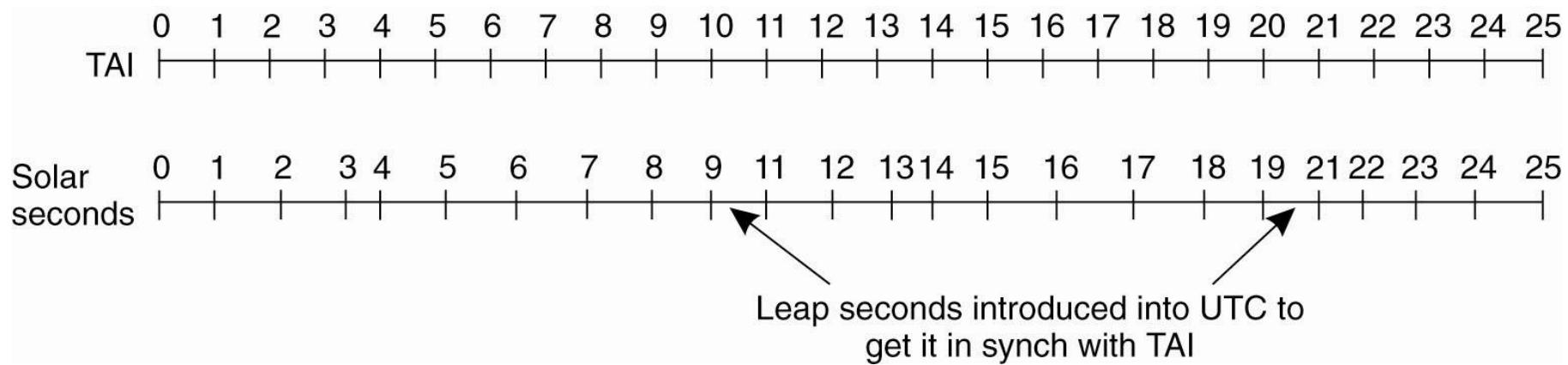


Figure 6-3. TAI (Int. Atomic Time) seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

# Global Positioning System (1)

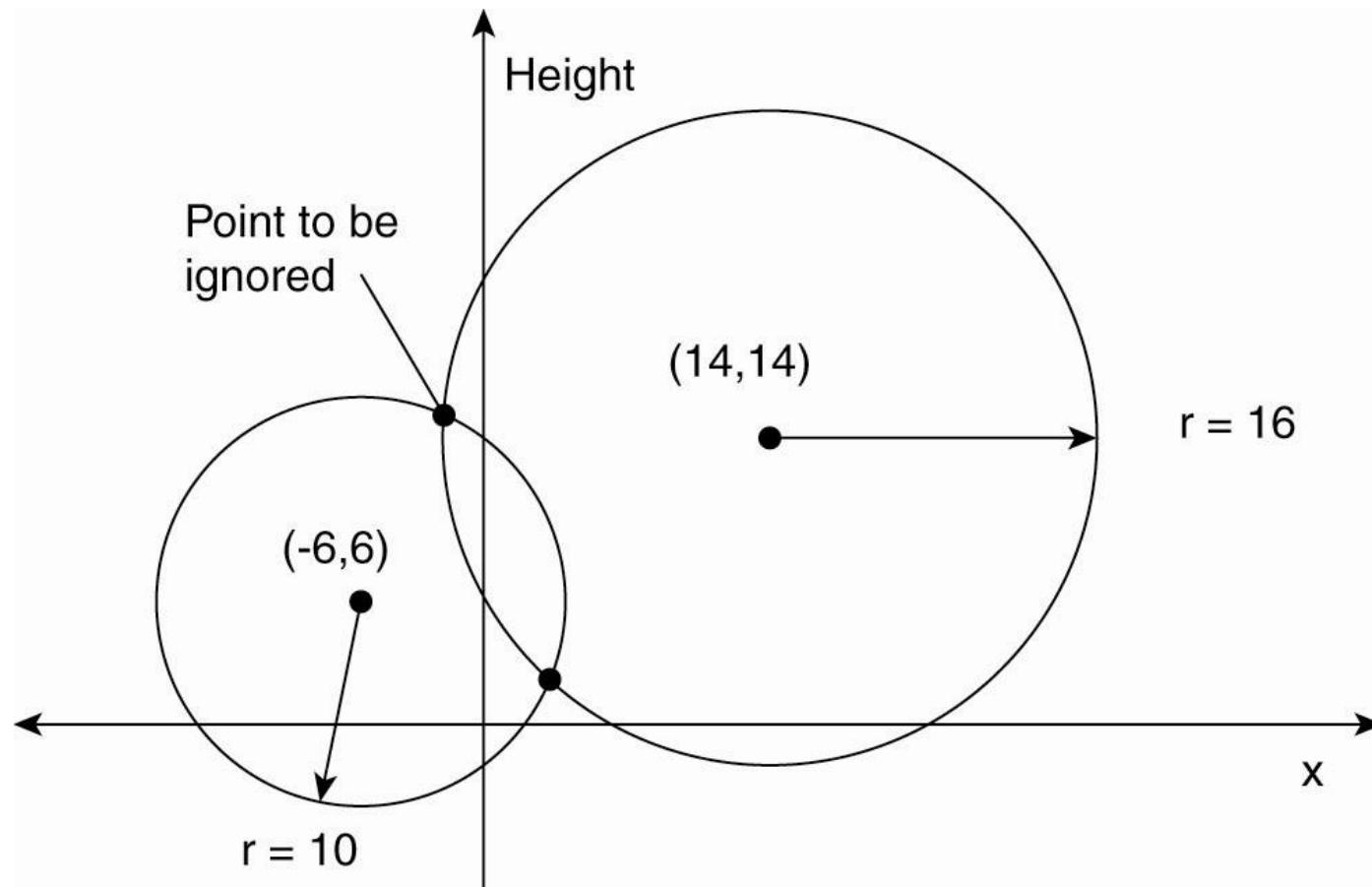


Figure 6-4. Computing a position in a two-dimensional space.

# GPS – 2 dimensional case

If you know the location of the satellite and you know what time you've received its signal, then you know how far away you are from it (the radius).

Using two radii you know where you are on the plane (ignoring the upper intersection).

GPS signals also give us the *time*

# Global Positioning System (2)

Real world facts that complicate GPS

1. It takes a while before data on a satellite's position reaches the receiver.
2. The receiver's clock is generally not in sync with that of a satellite.

# Clock Synchronization Algorithms

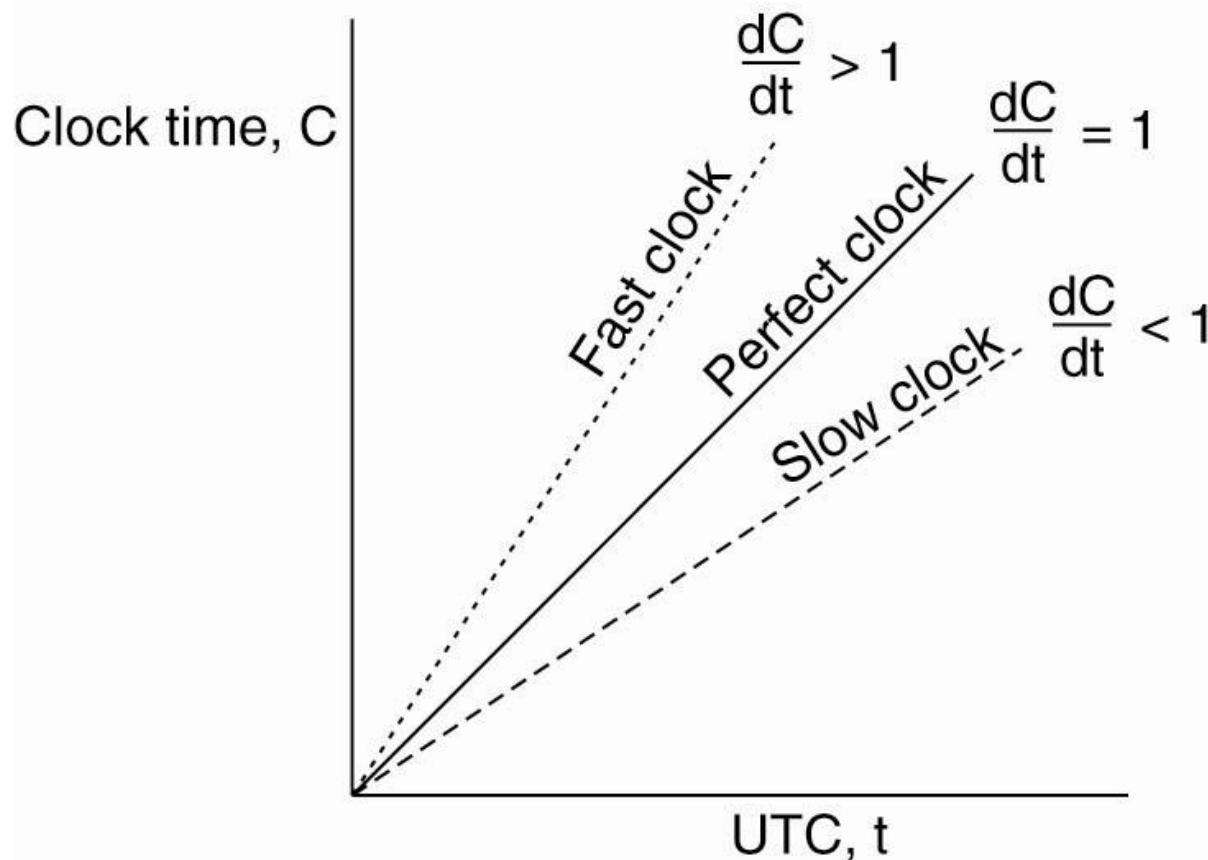


Figure 6-5. The relation between clock time and UTC when clocks tick at different rates.

# Network Time Protocol

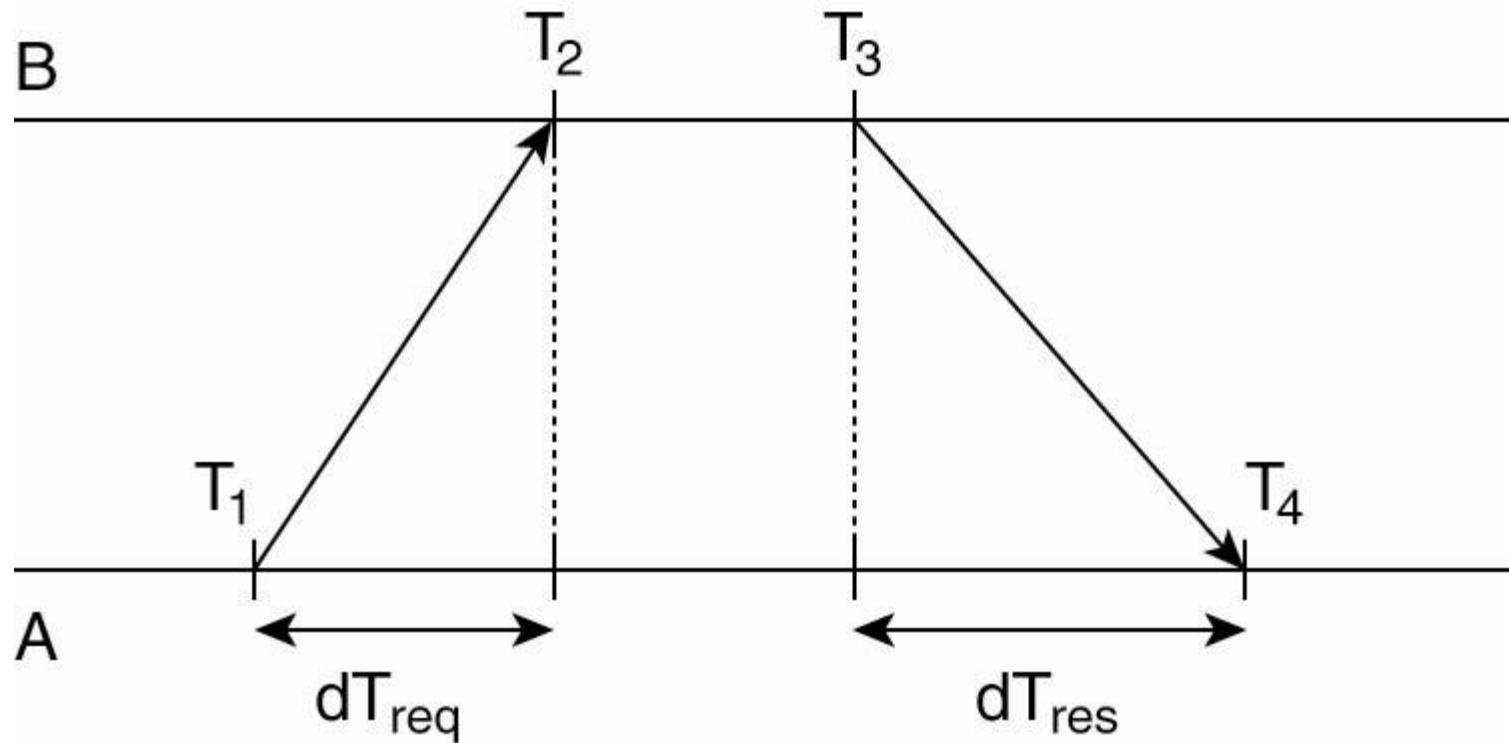


Figure 6-6. Getting the current time from a time server.

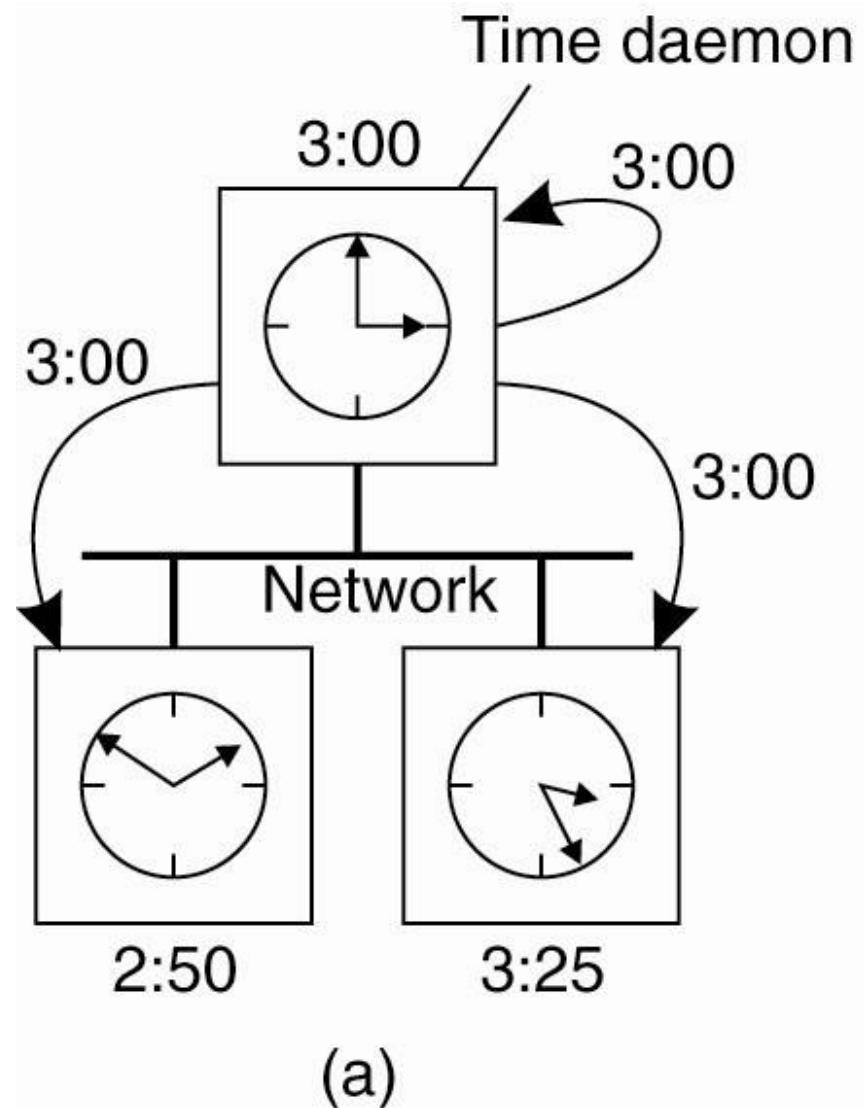
- **Approximately...**
  - Know time of request, and of receipt of response.
  - Know processing time at TimeServer B.
  - Know difference of two = sum of delays
  - Average of the two network delays should be added to time at send

# Berkeley Algorithm

- When no machine has an accurate clock, can use agreement.
- Set the clock manually, from time to time based on an external source

# The Berkeley Algorithm (1)

Figure 6-7. (a) The time daemon asks all the other machines for their clock values.



# The Berkeley Algorithm (2)

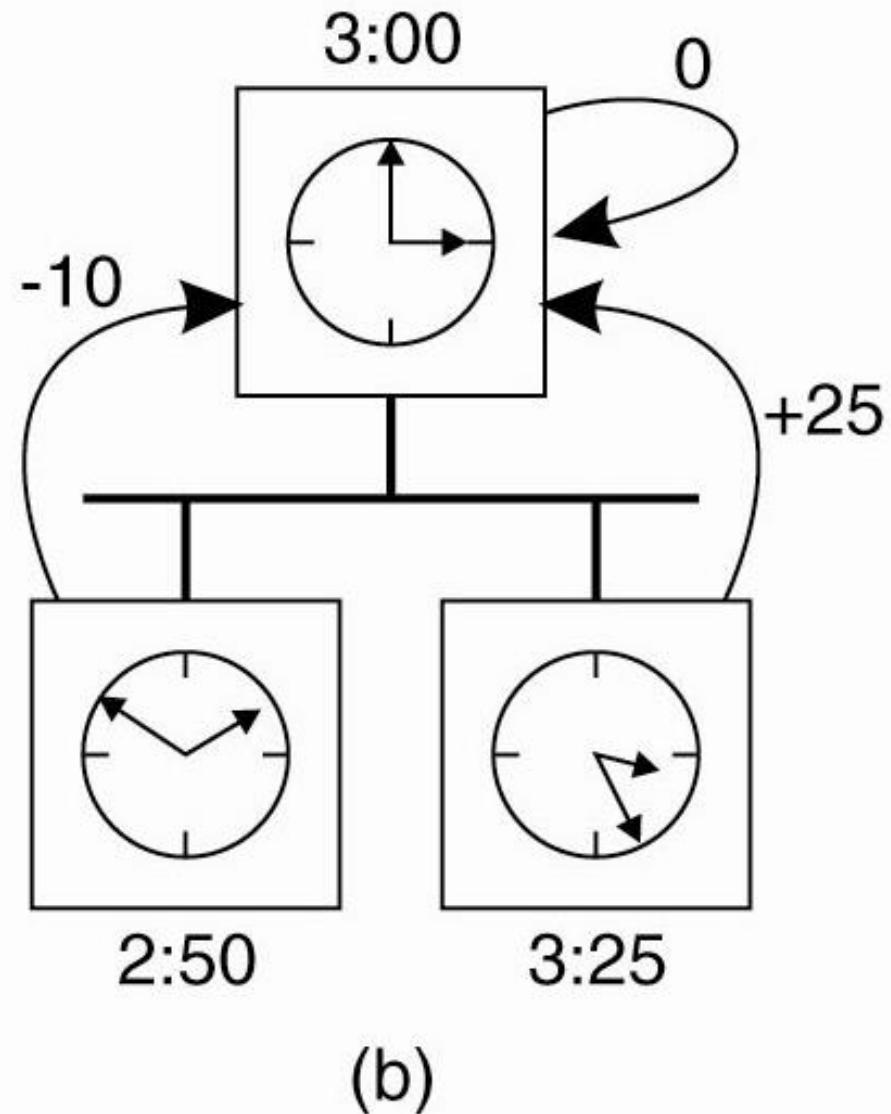
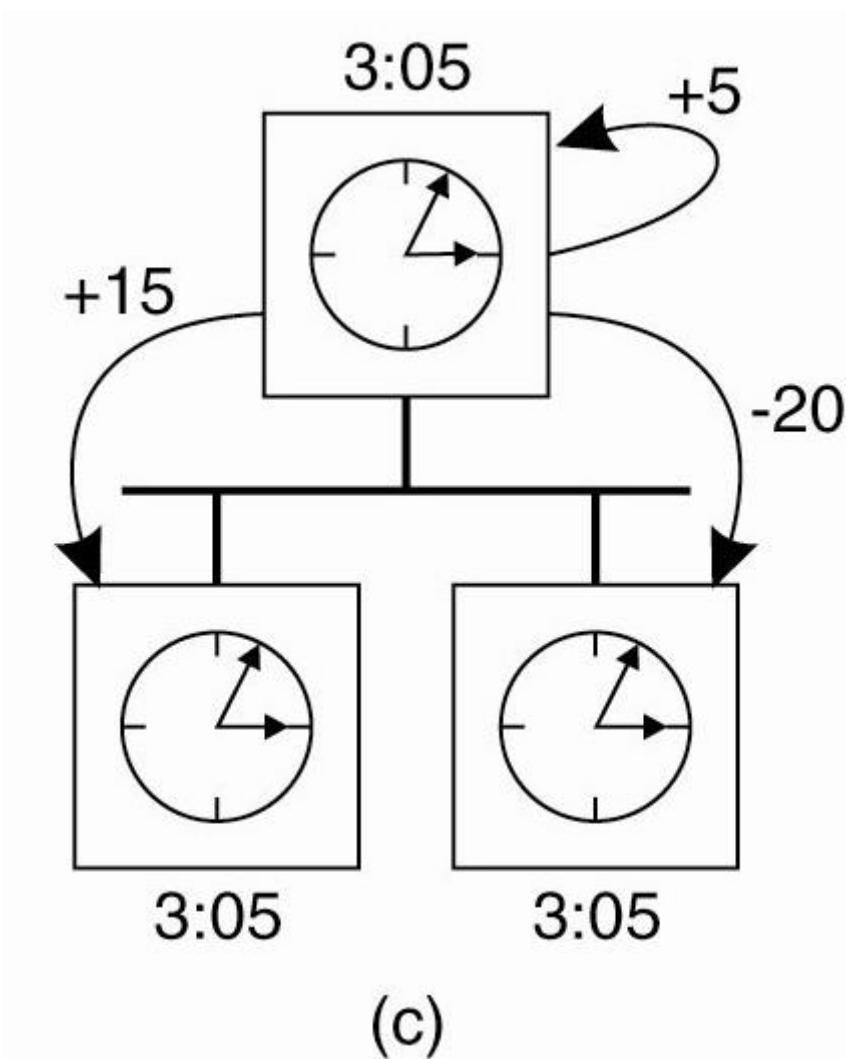


Figure 6-7.  
(b) The machines answer.

# The Berkeley Algorithm (3)

Figure 6-7. (c) The time daemon tells everyone how to adjust their clock.



- Adjusting clocks can be problematic:
- Time cannot run backwards
- Some clocks are known to be more accurate than others.
- Changes in time have to take place gradually on a system, because in practice systems use time-outs to simulate a *synchronous* system

# Example

- Failsafe at missile silo: if you don't get a reset from headquarters every hour then send the missiles.
- At headquarters: Every hour send a reset to the missile silo.
- Clock coordination message to headquarters at 2:59: Set the clock ahead to 3:05
- [...]!

# RBS – wireless sensor networks

Reference Broadcast Synchronization

Internally synchronize the clocks – no attempt at UTC

Only the receiver synchronizes

In sensor networks constructing the time message and accessing the network are constant, so not part of the critical path that determines variations in propagation.

No timestamp. Just mark when the reference message is received. Can then trade with other nodes to determine offset. Keeping a record allows for linear regression to account for drift.

# Clock Synchronization in Wireless Networks (1)

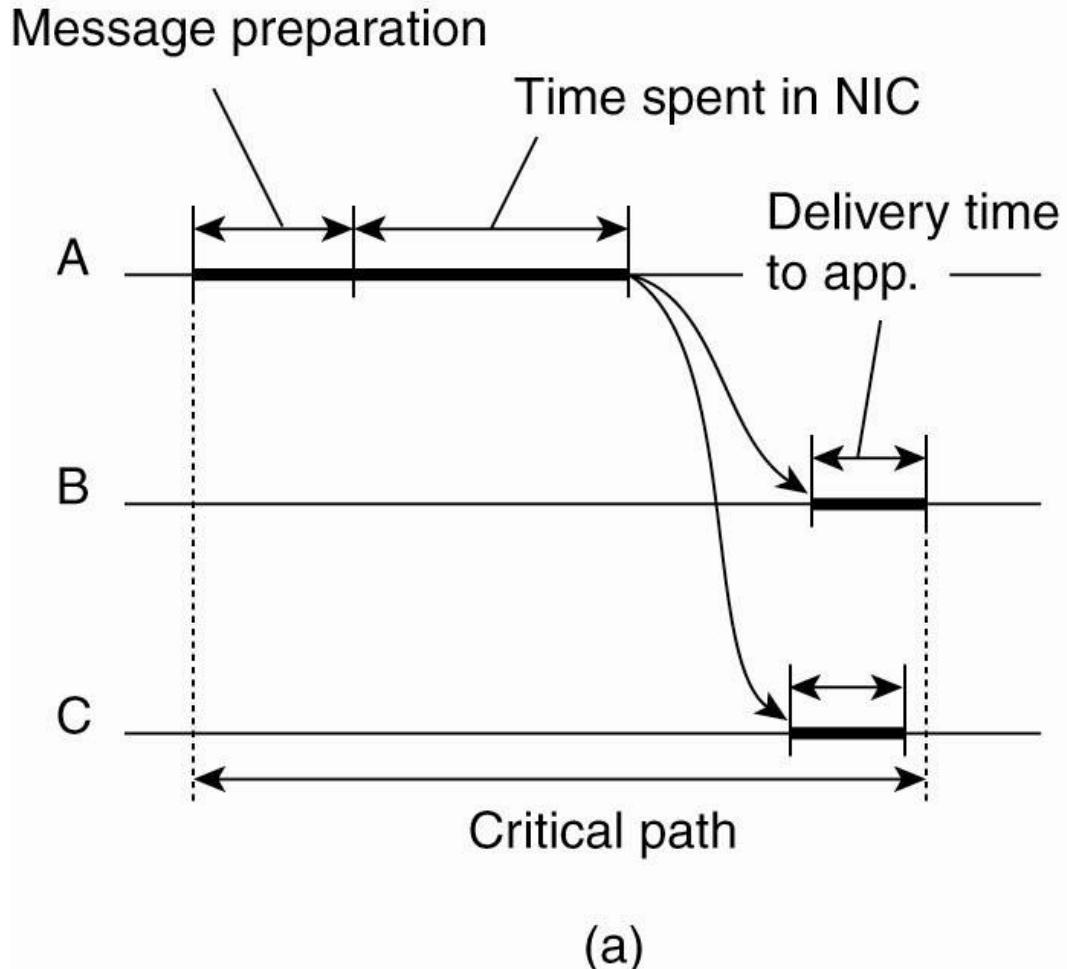


Figure 6-8. (a) The usual critical path in determining network delays.

# Clock Synchronization in Wireless Networks (2)

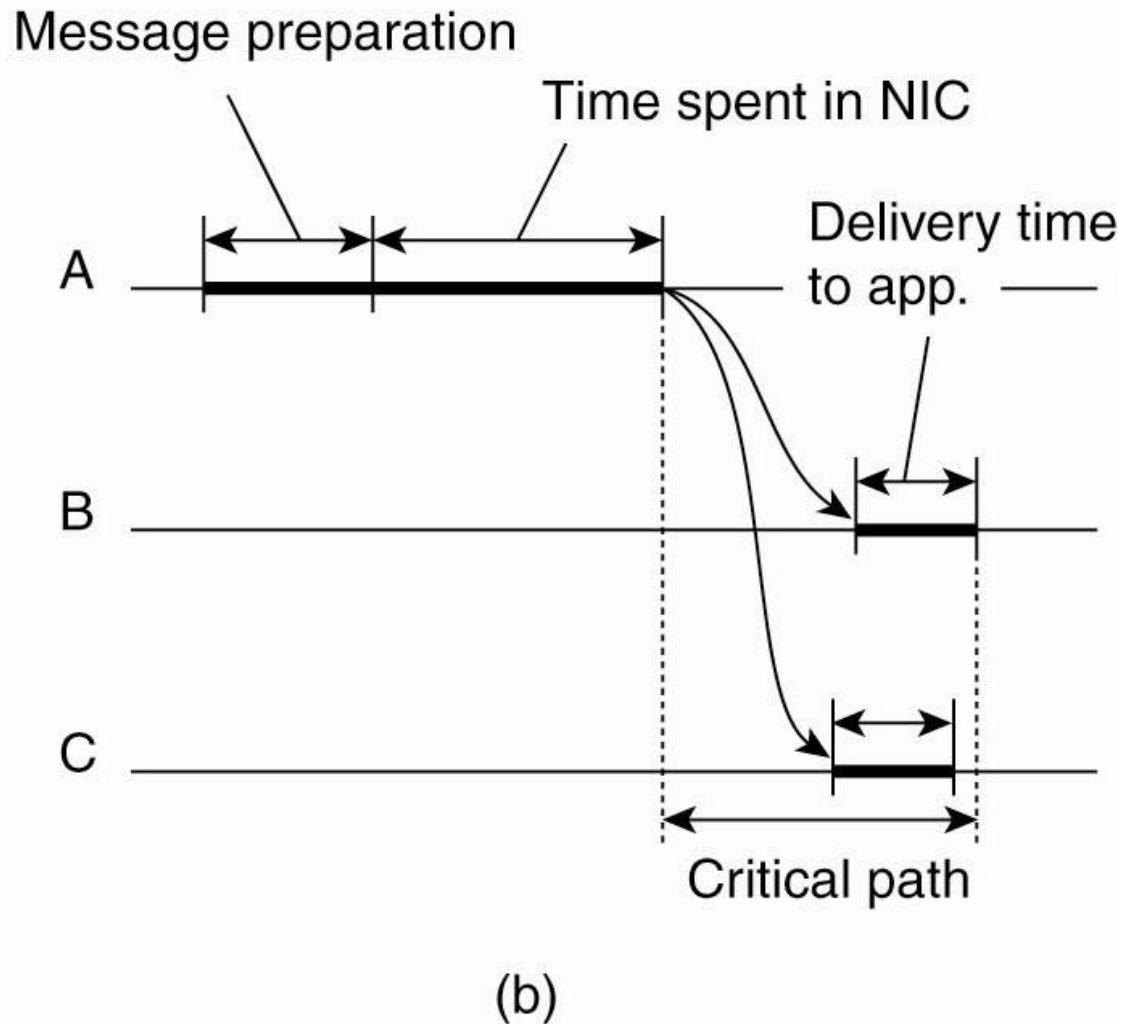


Figure 6-8. (b) The critical path in the case of RBS.

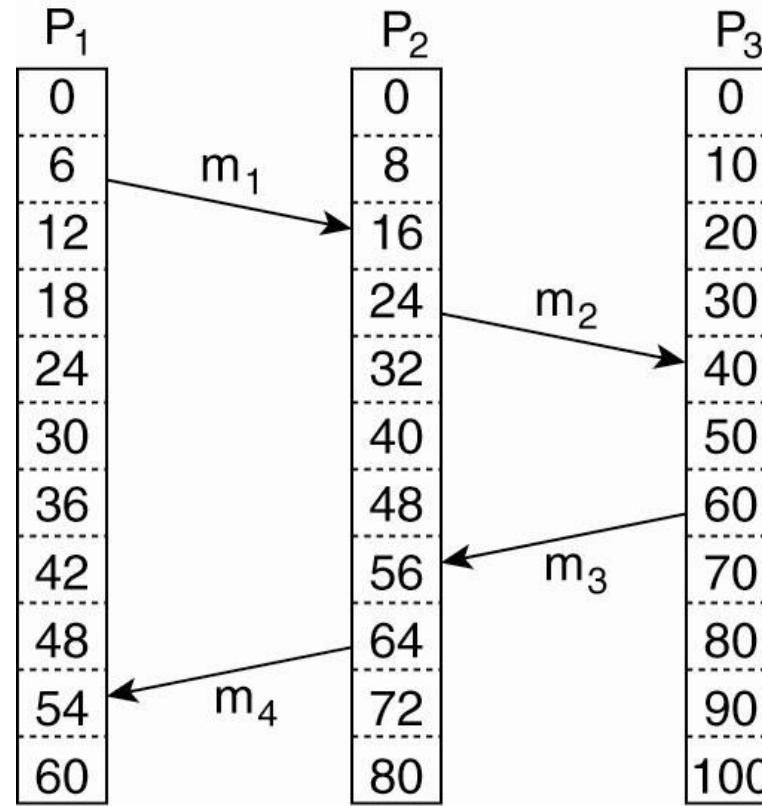
# Lamport's Logical Clocks (1)

The "happens-before" relation  $\rightarrow$  can be observed directly in two situations:

- If  $a$  and  $b$  are events in the same process, and  $a$  occurs before  $b$ , then  $a \rightarrow b$  is true.
- If  $a$  is the event of a message being sent by one process, and  $b$  is the event of the message being received by another process, then  $a \rightarrow b$  is true.
- Transitive:  $a \rightarrow b, b \rightarrow c \implies a \rightarrow c$

- If events  $x$ ,  $y$ , happen in different processes that do not exchange messages (incl. 3<sup>rd</sup> party transmission) then neither  $x \rightarrow y$ , or  $y \rightarrow x$ , are true. Instead are said to be *concurrent*.
- $C$  is clock value,  $C(x)$  is the clock value when  $x$  occurred, as agreed by all processes.
- Time only moves forward by incrementing the clock.

# Lamport's Logical Clocks (2)



(a)

Figure 6-9. (a) Three processes, each with its own clock.  
The clocks run at different rates.

# Lamport's Logical Clocks (3)

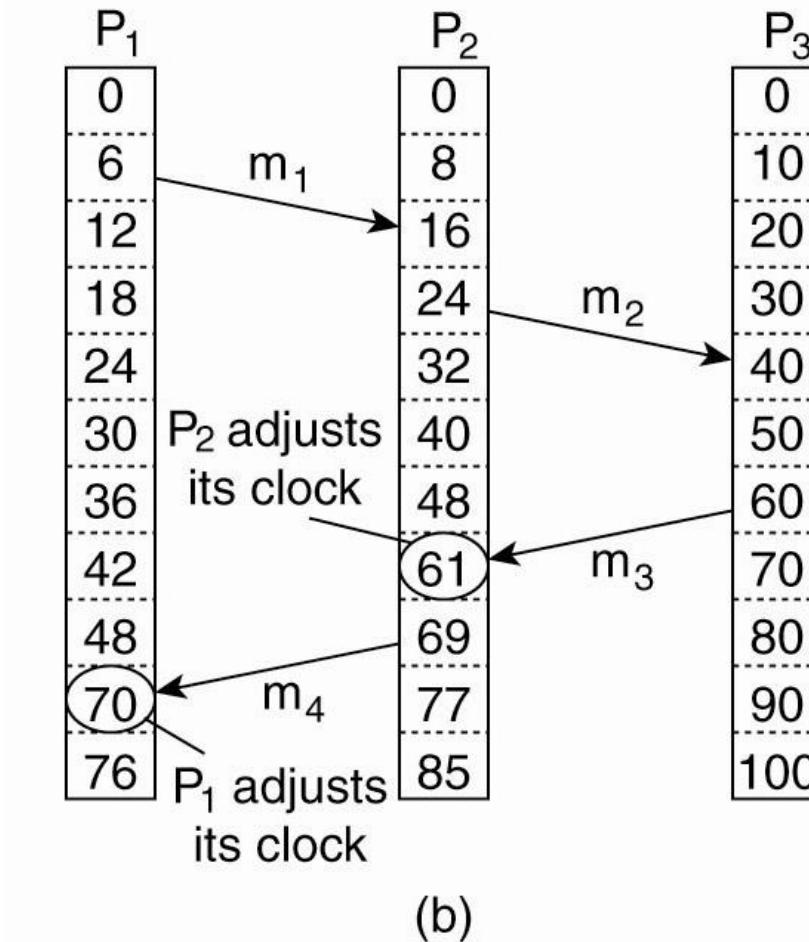


Figure 6-9. (b) Lamport's algorithm corrects the clocks.

# Lamport's Logical Clocks (4)

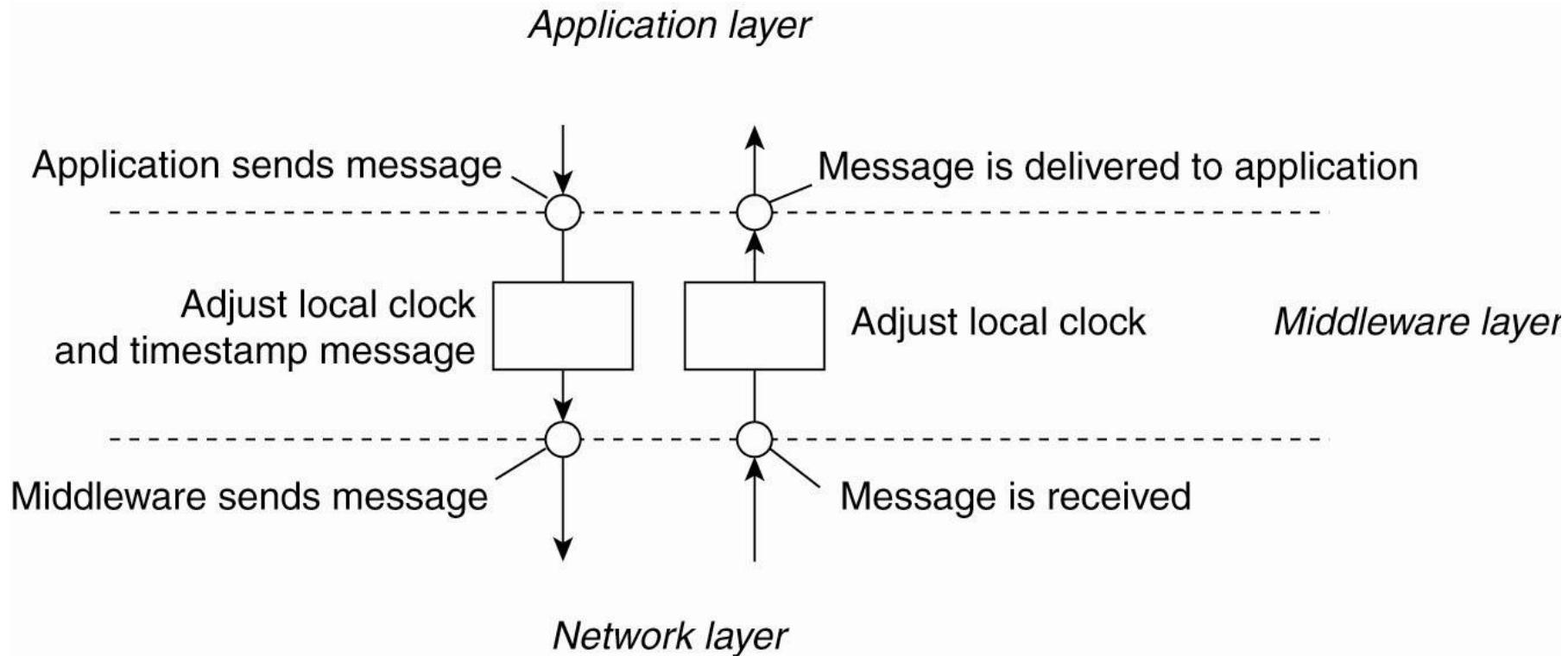


Figure 6-10. The positioning of Lamport's logical clocks in distributed systems.

# Lamport's Logical Clocks (5)

Updating counter  $C_i$  for process  $P_i$

1. Before executing an event  $P_i$  executes  
 $C_i \leftarrow C_i + 1$ .
2. When process  $P_i$  sends a message  $m$  to  $P_j$ , it sets  $m$ 's timestamp  $ts(m)$  equal to  $C_i$  after having executed the previous step.
3. Upon the receipt of a message  $m$ , process  $P_j$  adjusts its own local counter as  
 $C_j \leftarrow \max\{C_j, ts(m)\}$ , after which it then executes the first step and delivers the message to the application.

- Can add the process number on any particular machine after a decimal point, to guarantee unique events and time ordering.

# Example: Totally Ordered Multicasting

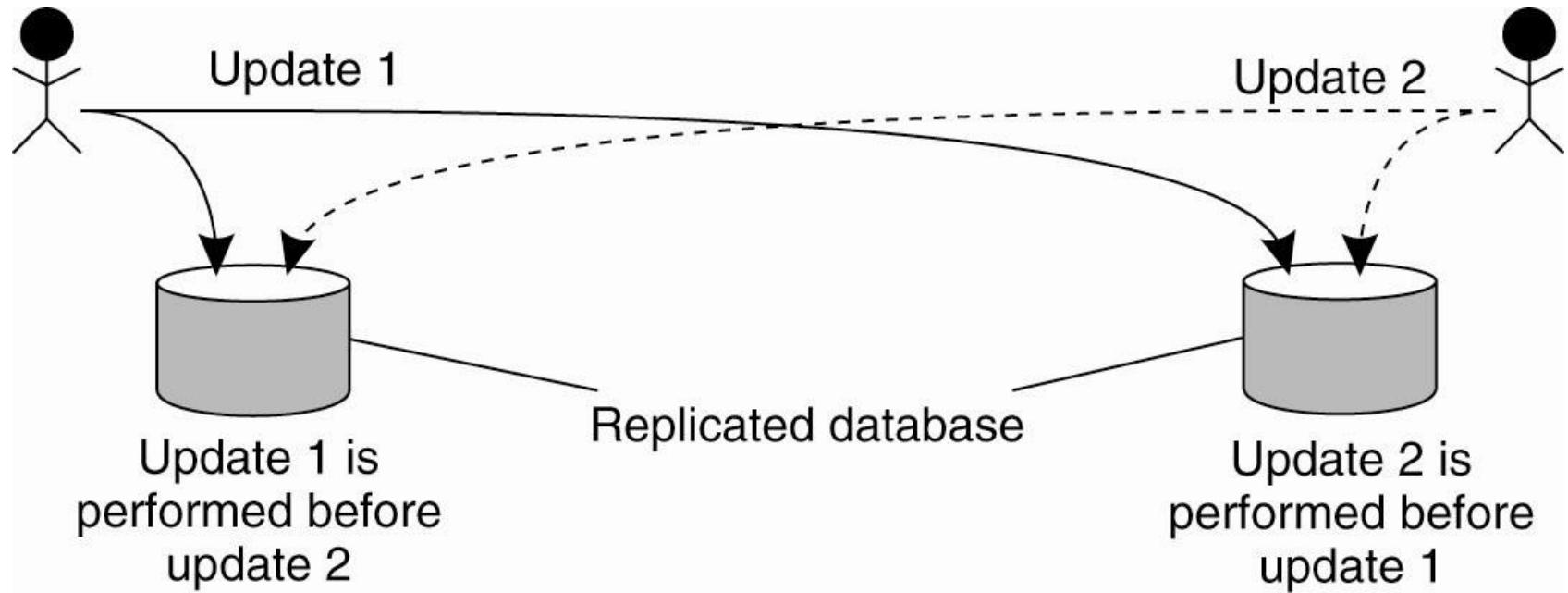


Figure 6-11. Updating a replicated database and leaving it in an inconsistent state.

# Vector Clocks (1)

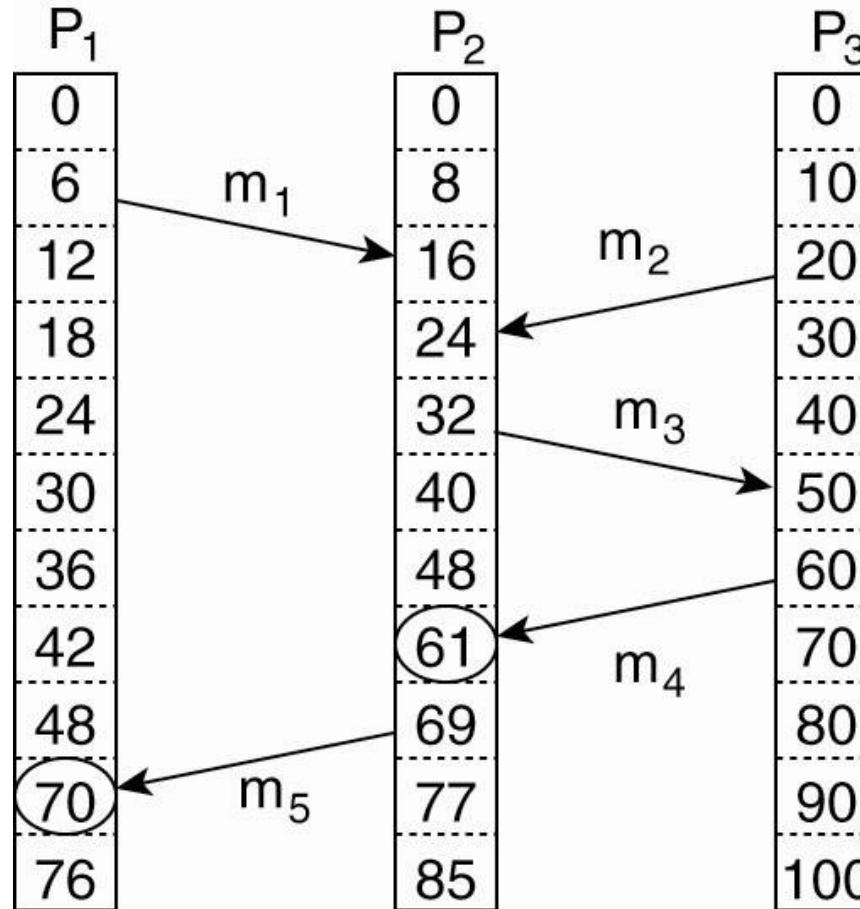


Figure 6-12. Concurrent message transmission using logical clocks.

# Vector Clocks (2)

Vector clocks are constructed by letting each process  $P_i$  maintain a vector  $VC_i$  with the following two properties:

1.  $VC_i[i]$  is the number of events that have occurred so far at  $P_i$ . In other words,  $VC_i[i]$  is the local logical clock at process  $P_i$ .
2. If  $VC_i[j] = k$  then  $P_i$  knows that  $k$  events have occurred at  $P_j$ . It is thus  $P_i$ 's knowledge of the local time at  $P_j$ .

# Vector Clocks (3)

Steps carried out to accomplish property 2 of previous slide:

1. Before executing an event,  $P_i$  executes  
$$VC_i[i] \leftarrow VC_i[i] + 1.$$
2. When process  $P_i$  sends a message  $m$  to  $P_j$ , it sets  $m$ 's (vector) timestamp  $ts(m)$  equal to  $VC_i$  after having executed the previous step. It then sends the *entire vector*,  $VC_i$ , along with  $m$  to the receiving process. (The book is not very clear on this point.)

# Vector Clocks (4)

1. Upon the receipt of a message  $m$  (and the entire vector) from a sender, process  $P_j$  adjusts its own vector by setting  $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$  for each  $k$ , after which it executes the first step and delivers the message to the application.
2. In this way, the maximum amount of time information about all processes in the system are transferred with each message.

# Vector clock example, enforcing causal communication

- *Causally-ordered communication* is weaker than totally ordered communication, but might be enough.
- In the next example, vector-entry time stamps are only incremented when a message is sent.
- Vector adjustment is as usual.

# Enforcing Causal Communication

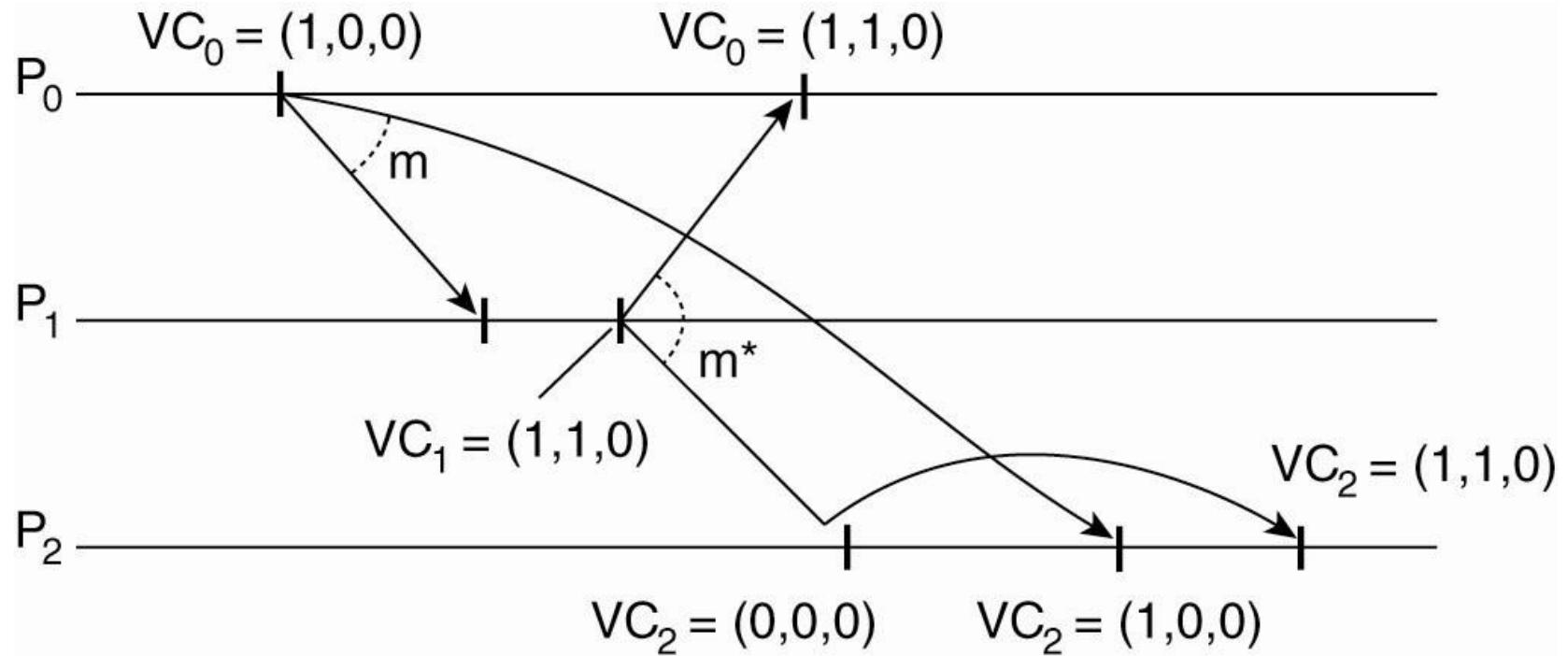


Figure 6-13. Enforcing causal communication.

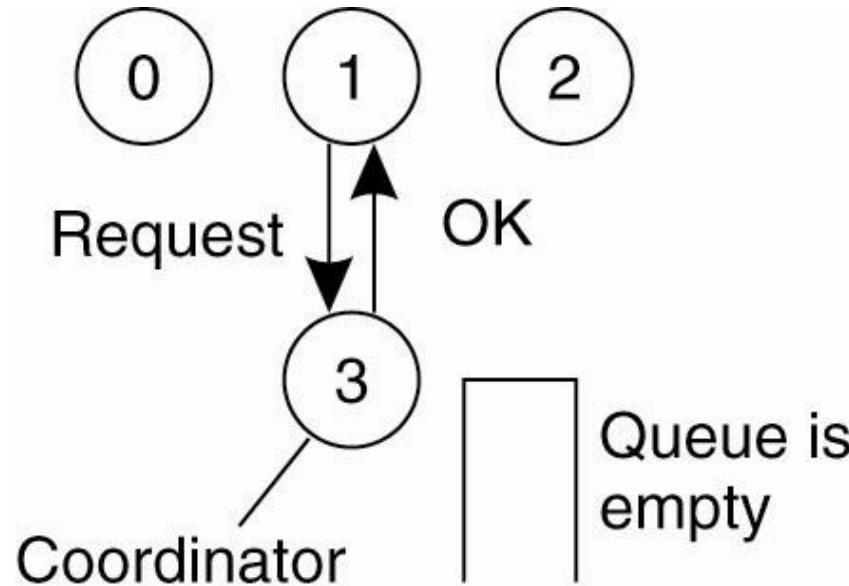
# Problems with middleware ordering of messages

- Middleware does not know what a message contains. E.g., two msgs from the same proc may not be related but are treated as causal. Less efficient.
- Not all actual causality may be captured. E.g., external causation (say by a phone call) will not be captured.
- Refer again to Saltzer (1984) *end-to-end argument* in system's design.
- But, places non-content burden on the developer, which we like to avoid.

- But, places non-content burden on the developer, which we like to avoid.

# Mutual Exclusion

## A Centralized Algorithm (1)

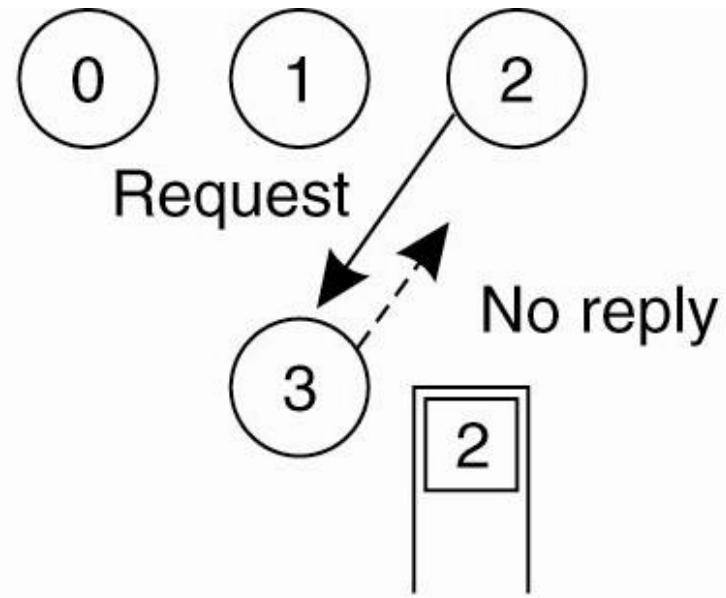


(a)

Figure 6-14. (a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

# Mutual Exclusion

## A Centralized Algorithm (2)

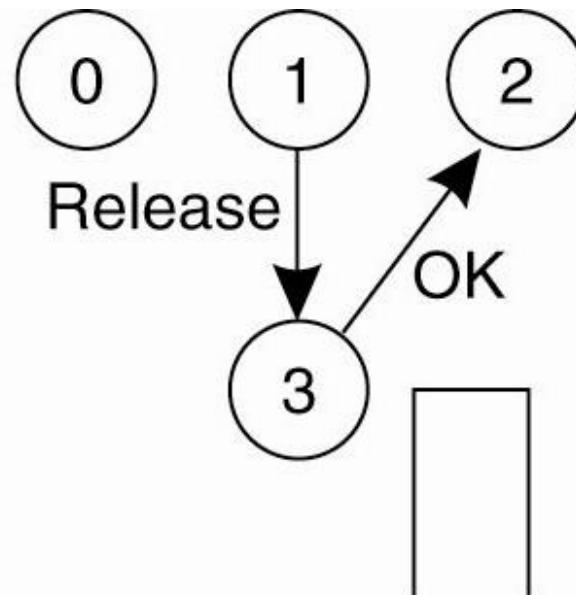


(b)

Figure 6-14. (b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

# Mutual Exclusion

## A Centralized Algorithm (3)



(c)

Figure 6-14. (c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

# A Distributed Algorithm (1)

Three different cases:

1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

# A Distributed Algorithm (2)

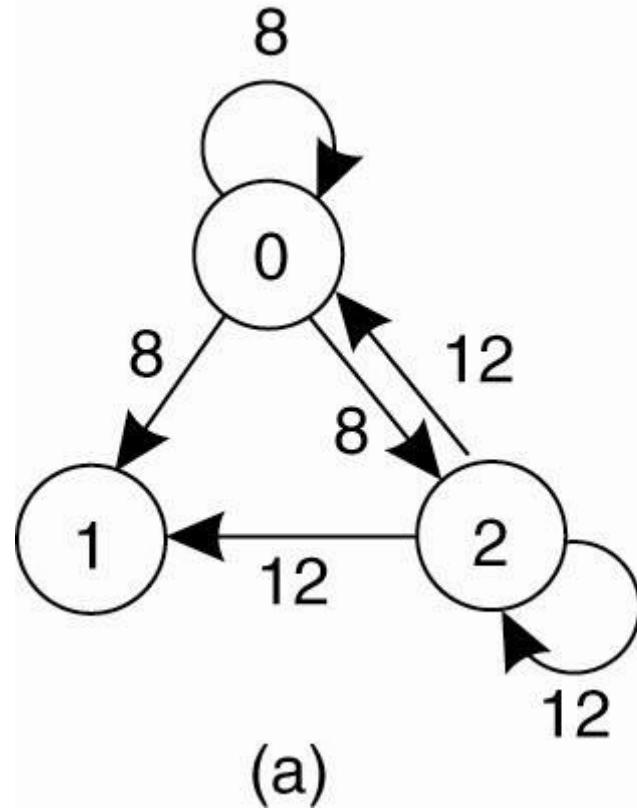


Figure 6-15. (a) Two processes want to access a shared resource at the same moment.

# A Distributed Algorithm (3)

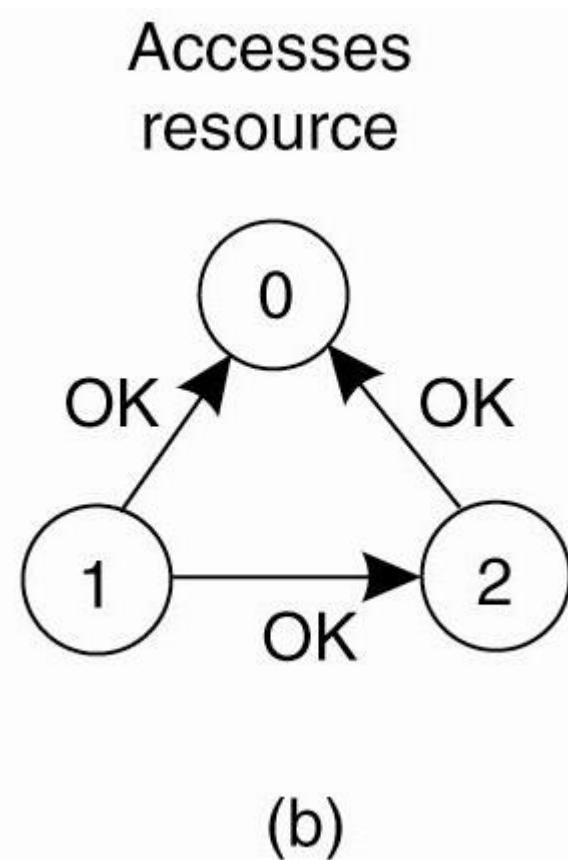
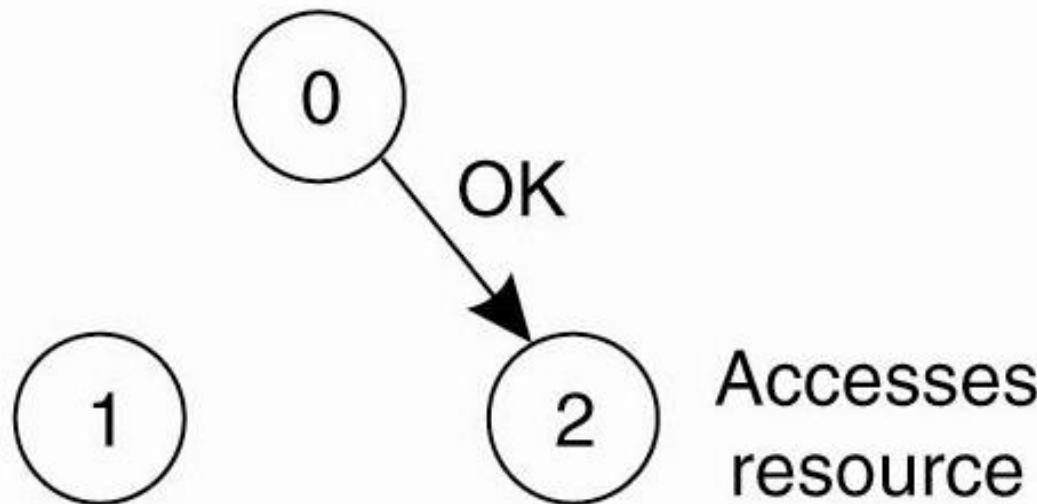


Figure 6-15. (b) Process 0 has the lowest timestamp, so it wins.

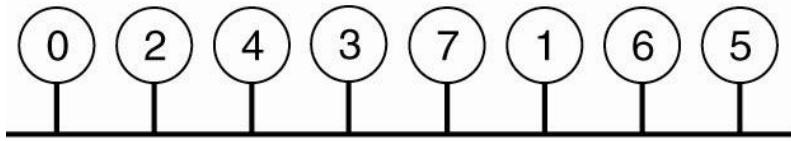
# A Distributed Algorithm (4)



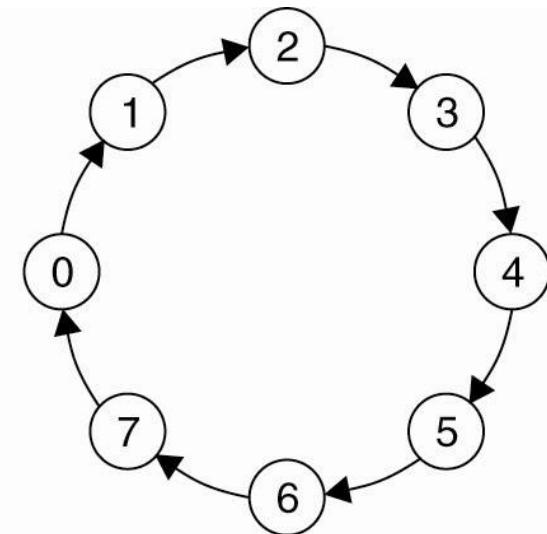
(c)

Figure 6-15. (c) When process 0 is done, it sends an OK also, so 2 can now go ahead.

# A Token Ring Algorithm



(a)



(b)

Figure 6-16. (a) An unordered group of processes on a network.  
(b) A logical ring constructed in software.

# Election Algorithms

## The Bully Algorithm

1.  $P$  sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds,  $P$  wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over.  $P$ 's job is done.

# The Bully Algorithm (1)

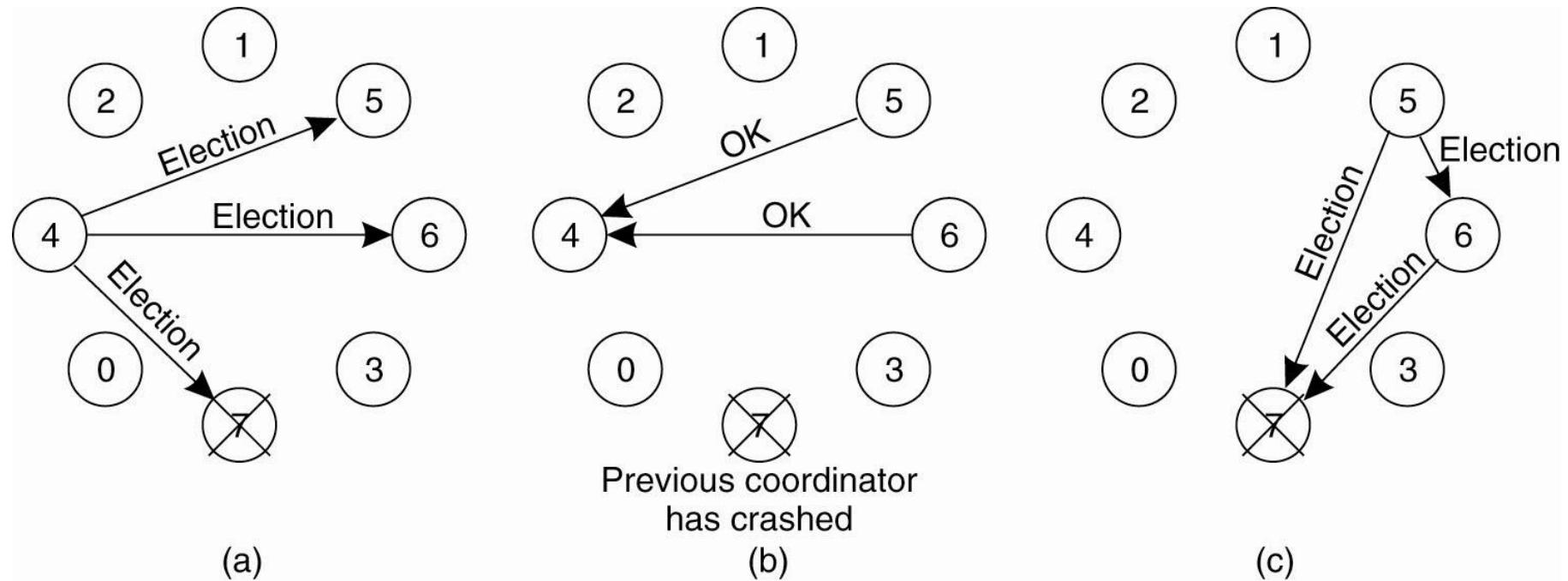


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.

# The Bully Algorithm (2)

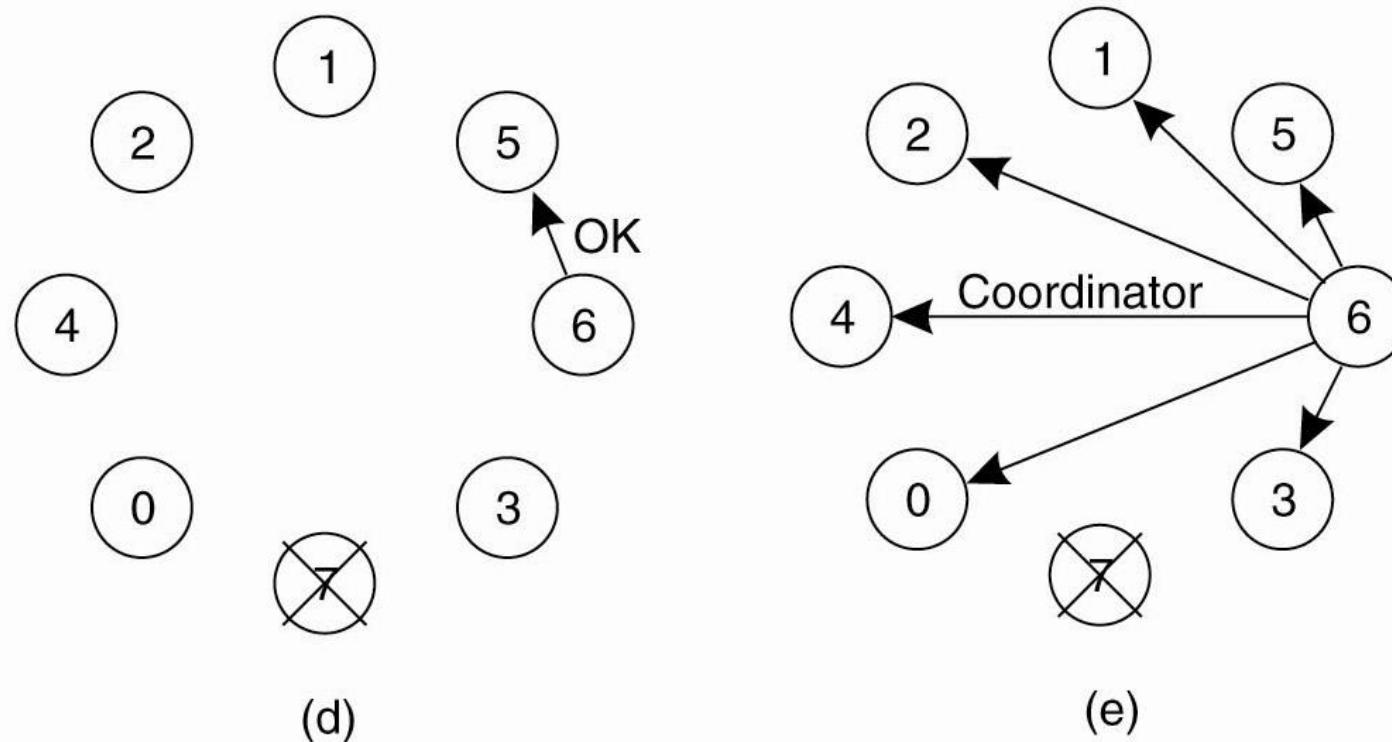


Figure 6-20. The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

# Non-token Ring Election

1. Processes know of their successor(s) in ring
2. Proc notes failure – sends election msg with own proc # to successor, or to succ of succ, etc.
3. Each proc adds its proc # to the list in elec msg
4. Message finally gets around ring to start
5. Select new coordinator (highest proc #?), send around the ring with new list of members
6. When coordinator msg returns, all are informed
7. If two sets of messages sent, highest coordinator wins. Presumably, lower coordinator is removed by any node.

# A Ring Algorithm

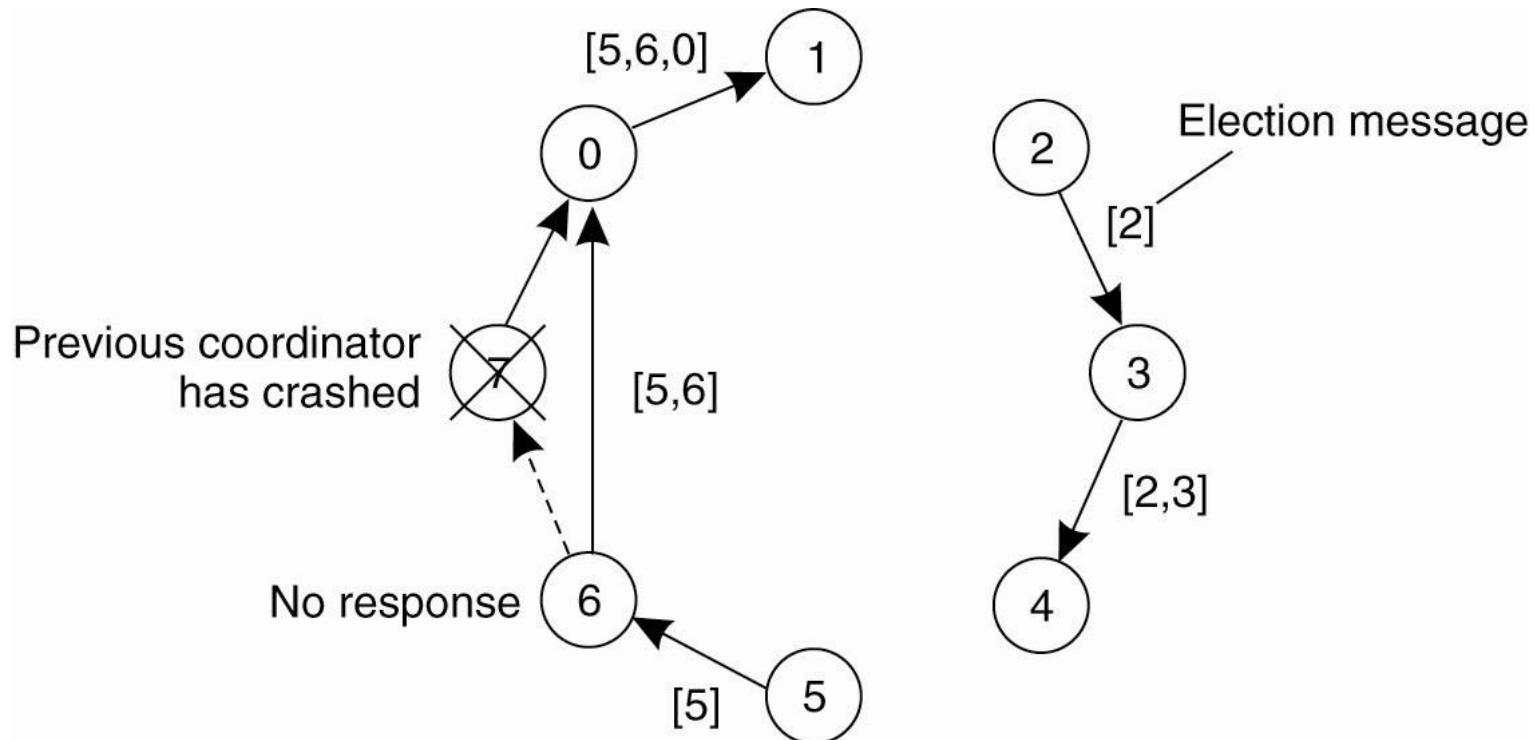


Figure 6-21. Election algorithm using a ring.

# Wireless Election

1. Cannot assume full network connectivity, stable connections, stable topology (mobile?)
2. Node and network failure, partitioning?
3. Need to select BEST (best connected / geography / battery) node for coordinator
4. Vasudevan et al. (2004): source node sends election msg to all nodes it can reach
5. Receiving node notes parent (sender) and sends to all nodes IT can reach, minus parent node
6. If node received msg from node other than its parent, simply send ack, then ignore.

# Wireless Election

1. Each node waits for ack from all its neighbors, then sends back ack to parent. Nodes that have a parent, send back immediate ack. If all acks then node is a leaf node in the network. Leaf nodes respond quickly and send information about qualities as a coordinator (e.g., battery life)
2. When a node finally sends message to its parent it also sends info about the best available coordinator candidate. This percolates back up to the original source, which then broadcasts a msg about who is the new coordinator.

# Elections in Wireless Environments (1)

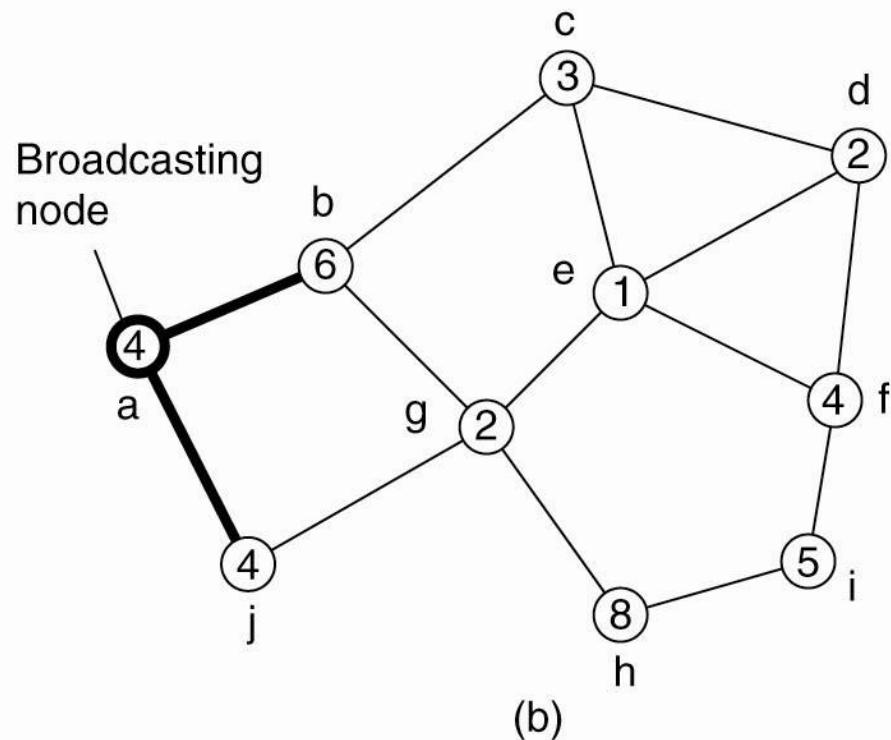
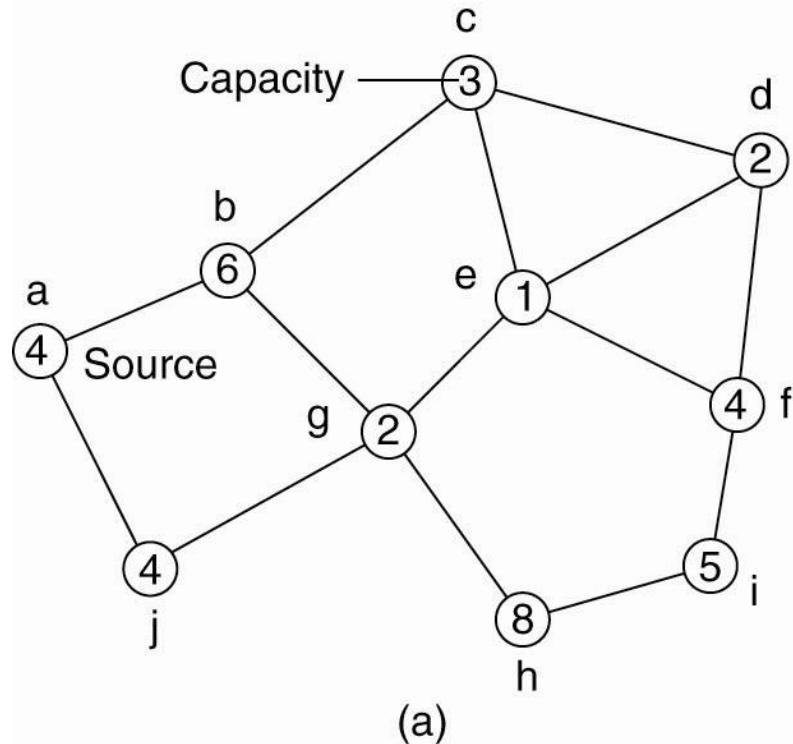


Figure 6-22. Election algorithm in a wireless network, with node *a* as the source. (a) Initial network. (b)–(e) The build-tree phase

# Elections in Wireless Environments (2)

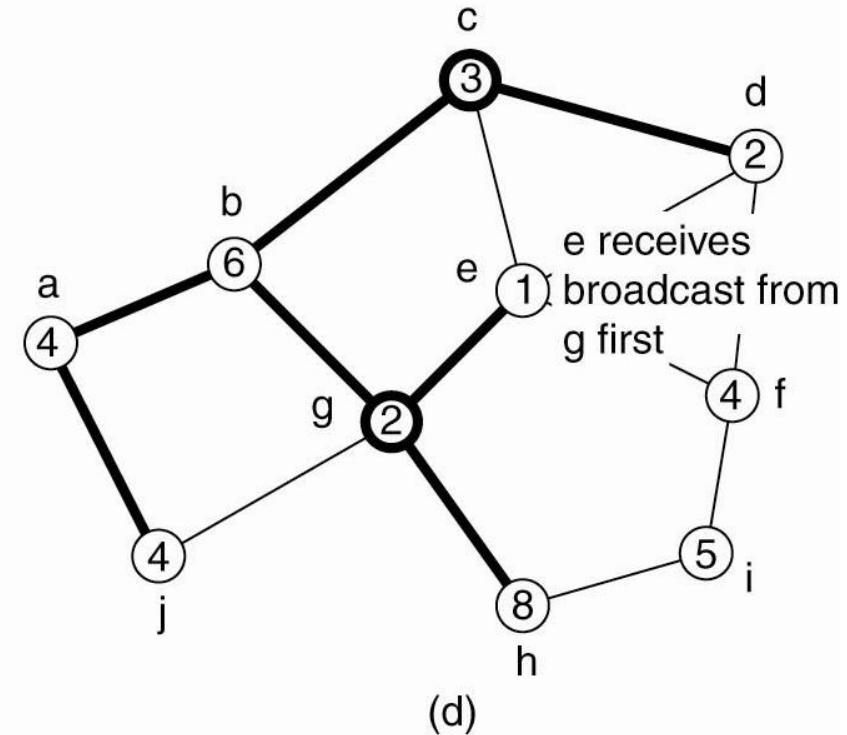
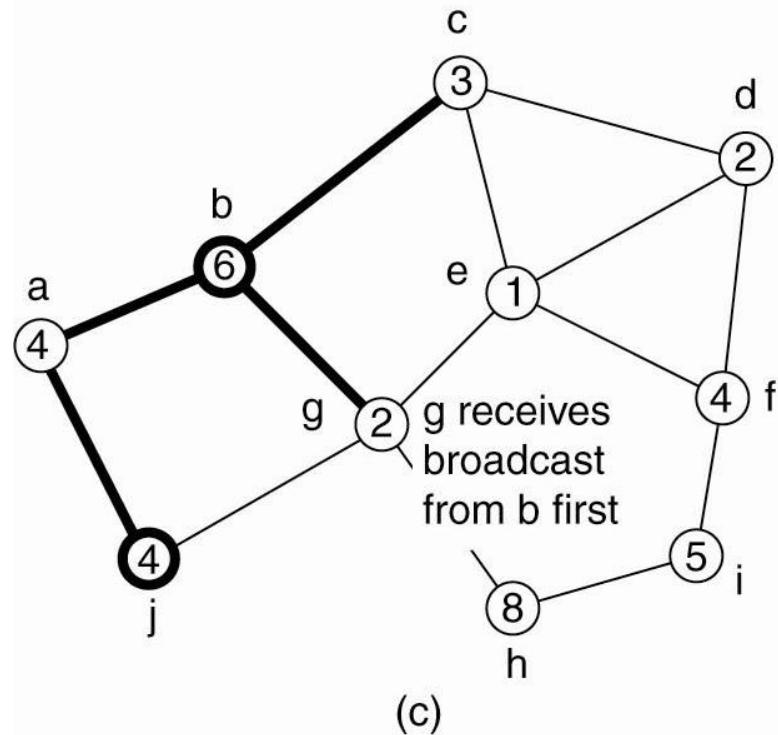
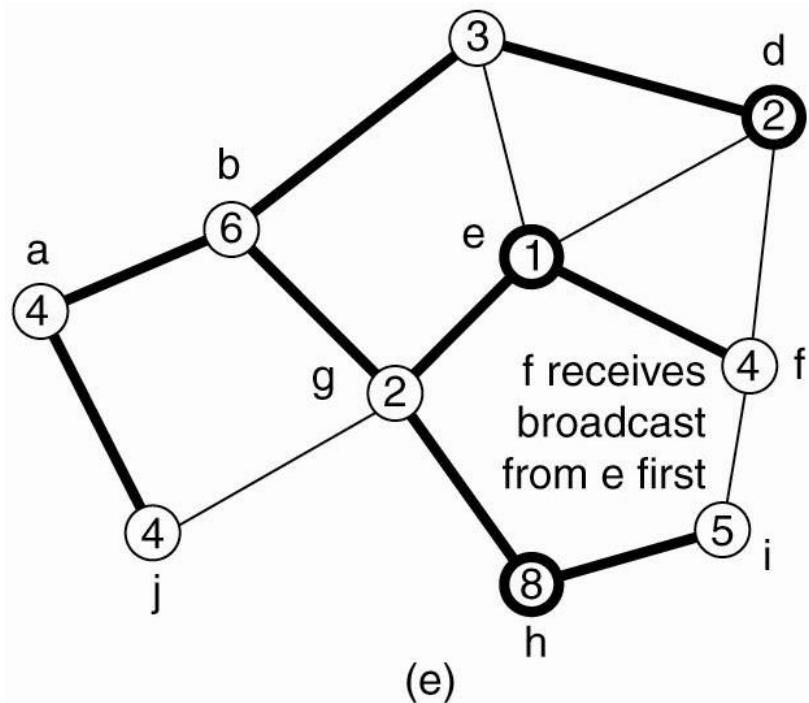
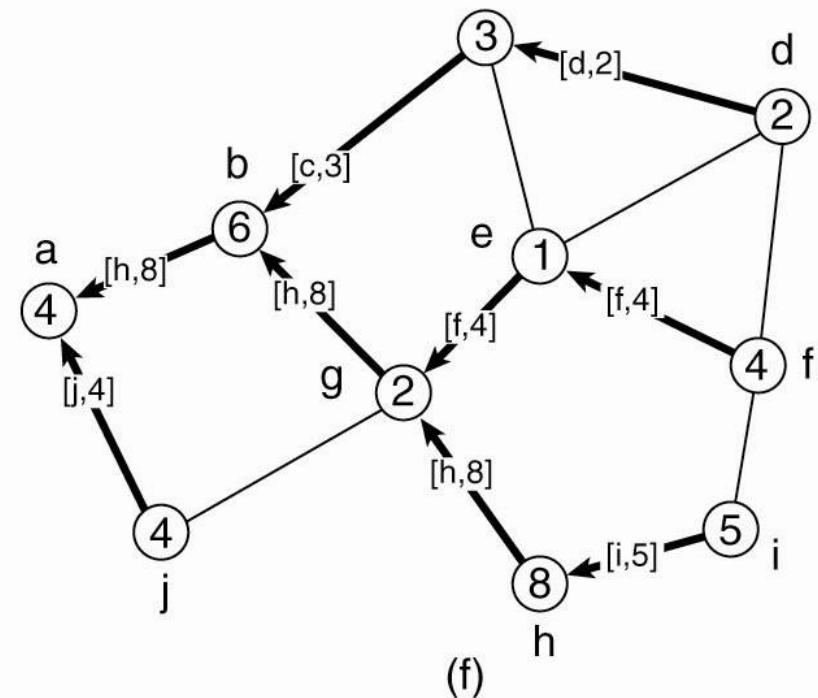


Figure 6-22. Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase

# Elections in Wireless Environments (3)



(e)



(f)

Figure 6-22. (e) The build-tree phase.  
(f) Reporting of best node to source.

# Elections in Large-Scale Systems (1)

## Requirements for superpeer selection:

1. Normal nodes should have low-latency access to superpeers.
2. Superpeers should be evenly distributed across the overlay network.
3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
4. Each superpeer should not need to serve more than a fixed number of normal nodes.

# SuperPeer selection with DHT

- DHT addressing is via m-bit space. Just using the leftmost [3] bits:
- When node-ID AND 1110000... not-equal zero then you are a SuperPeer
- Selects 1 in 8 as a SuperPeer, but can also select 1 in 4, 16, 32...

# Spreading SuperPeers in an unstructured network

- Each node with a token sends repulsion messages.
- Each node with a token that receives too much repulsive force, sends the token to a neighbor in the opposite direction.
- Can have local maxima problem

- Each node with a token sends repulsion messages.
- Each node with a token that receives too much repulsive force, sends the token to a neighbor in the opposite direction.
- Can have local maxima problem

# Elections in Large-Scale Systems (2)

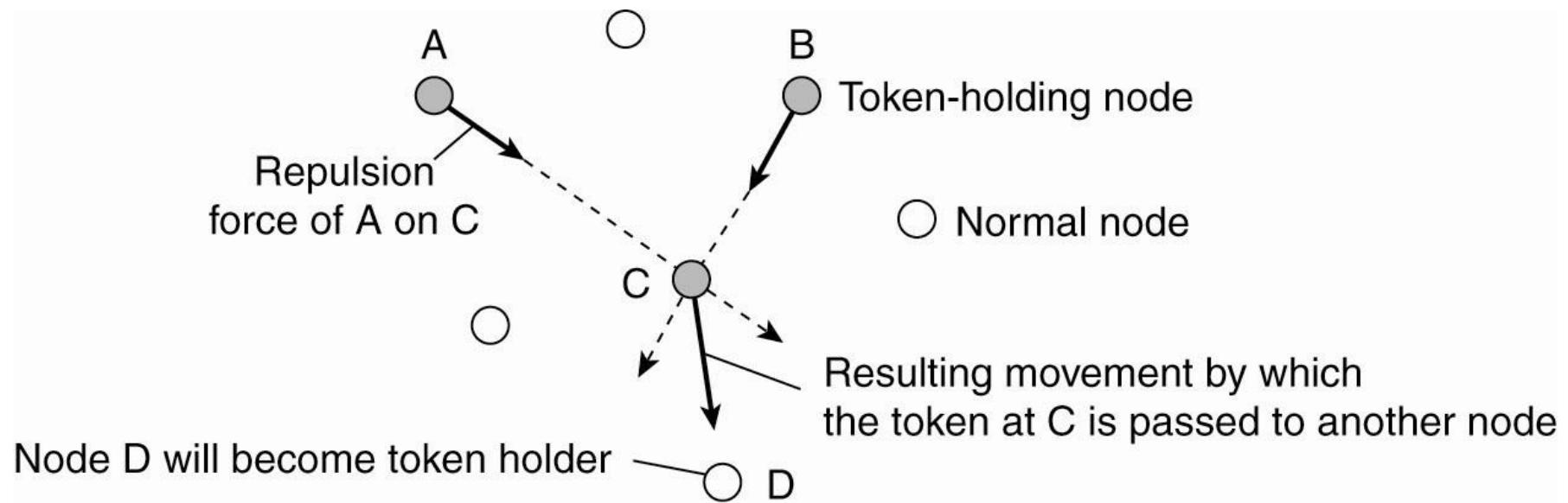


Figure 6-23. Moving tokens in a two-dimensional space using repulsion forces.

