ECE151 - Lecture 1

Chapter 1

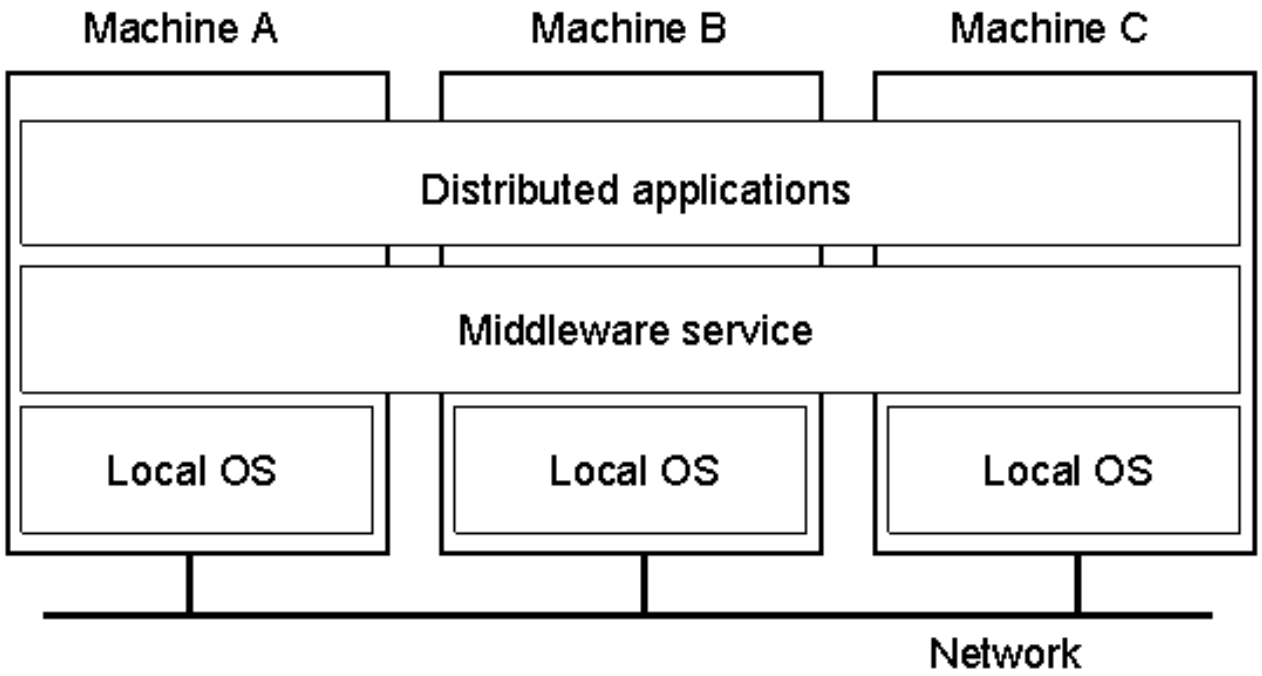
Introduction

Definition of a Distributed System (1)

A **Distributed System** is:

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

Definition of a Distributed System (2)



A distributed system organized as middleware.
Note that the middleware layer extends over multiple machines.

Goals of Distributed Systems

- Connecting resources and users
- Distribution transparency
- Openness
- Scalability

Transparency in a Distributed System

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may be moved to another location while in use
Relocation	Hide that a resource may be moved to another location
Replication	Hide that a resource may be available on several distinct computers
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

Different forms of transparency in a distributed system.

Degree of Transparency

Observation: Aiming at full distribution transparency may be too much:

- Users may be located in different continents; distribution is apparent and not something you want to hide

- Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  - You cannot distinguish a slow computer from a failing one
  - You can never be sure that a server actually performed an operation before a crash

- Full transparency will cost performance, exposing distribution of the system
  - Keeping Web caches *exactly* up-to-date with the master copy
  - Immediately flushing write operations to disk for fault tolerance

Openness of Distributed Systems

Open distributed system: Be able to interact with services from other open systems, irrespective of the underlying environment:

- Systems should conform to well-defined **interfaces**
- Systems should support **portability** of applications
- Systems should easily **interoperate**

Achieving openness: At least make the distributed system independent from **heterogeneity** of the underlying environment:

- Hardware
- Platforms
- Languages

Policies versus Mechanisms

Implementing openness: Requires support for different **policies** specified by applications and users:

- What level of consistency do we require for client cached data?
- Which operations do we allow downloaded code to perform?
- Which QoS requirements do we adjust in the face of varying bandwidth?
- What level of secrecy do we require for communication?

Implementing openness: Ideally, distributed operating systems and middleware provide only **mechanisms**:

- Allow (dynamic) setting of caching policies, preferably per cachable item
- Support different levels of trust for mobile code
- Provide adjustable QoS parameters per data stream
- Offer different encryption algorithms

Scalability Problems

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

Examples of scalability limitations.

Scale in Distributed Systems

Observation: Many developers of modern distributed system easily use the adjective "scalable" without making clear *why* their system actually scales.

Scalability: At least three components:

Number of users and/or processes **(size scalability)**

Maximum distance between nodes **(geographical scalability)**

Number of administrative domains **(administrative scalability)**

Most systems account only, to a certain extent, for size scalability. The (non)solution: powerful servers.

Today, the challenge lies in geographical and administrative scalability.

Techniques for Scaling

Distribution: Partition data and computations across multiple machines:

- Move computations to clients (Java applets)
- Decentralized naming services (DNS)
- Decentralized information systems (WWW)

Replication: Make copies of data available at different machines:

- Replicated file servers (mainly for fault tolerance)
- Replicated databases
- Mirrored Web sites
- Large-scale distributed shared memory systems

Caching: Allow client processes to access local copies:

- Web caches (browser/Web proxy)
- File caching (at server and client)

Scaling – The Problem

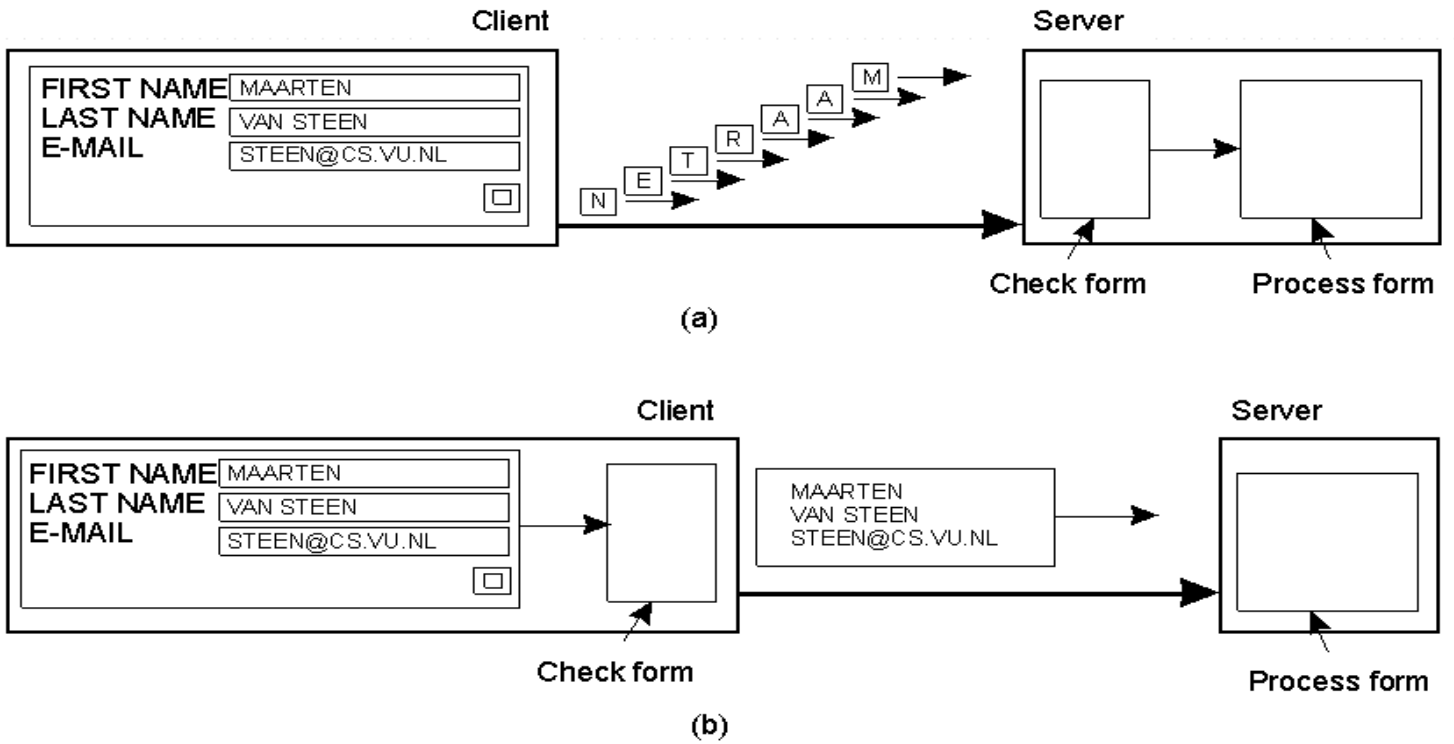
Observation: Applying scaling techniques is easy,
except for one thing:

- *Having multiple copies leads to **inconsistencies**:*
  modifying one copy makes that copy different from the rest.
- *Always keeping copies consistent and in a general way requires **global synchronization** on each modification.*
- *Global synchronization precludes large-scale solutions.*

Observation: If we can tolerate inconsistencies,
we may reduce the need for global synchronization.

Observation: Tolerating inconsistencies is application dependent.

Scaling Techniques (1)

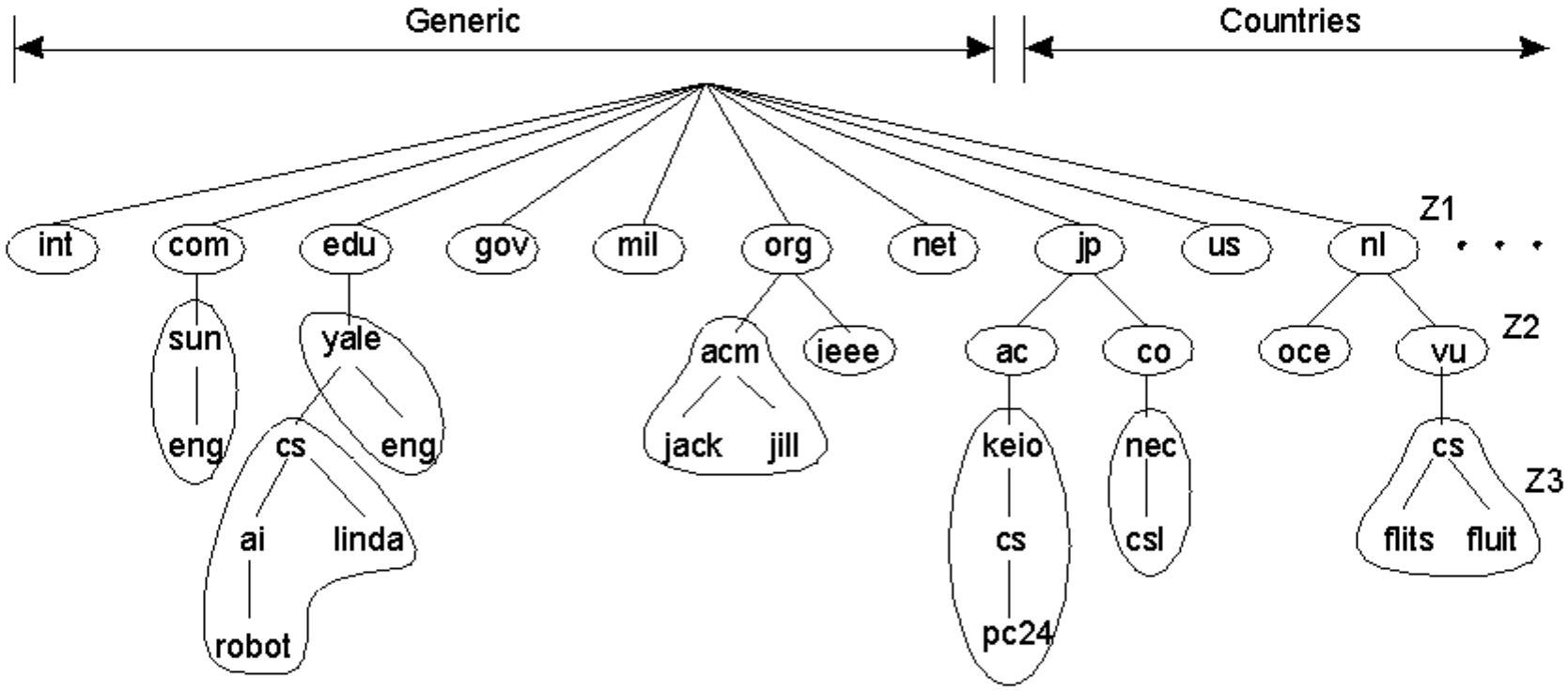


The difference between letting:

a) a server or

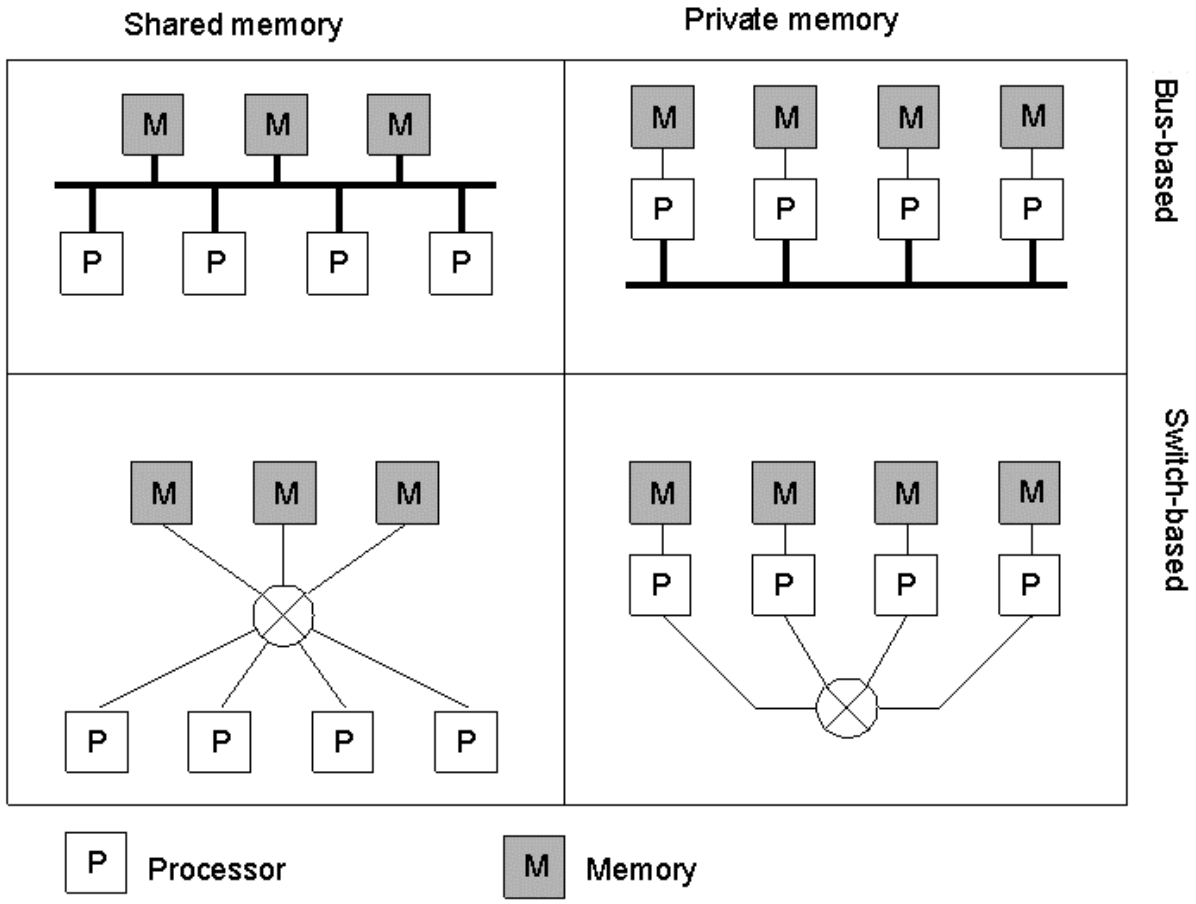
b) a client check forms as they are being filled

Scaling Techniques (2)



An example of dividing the DNS name space into zones.

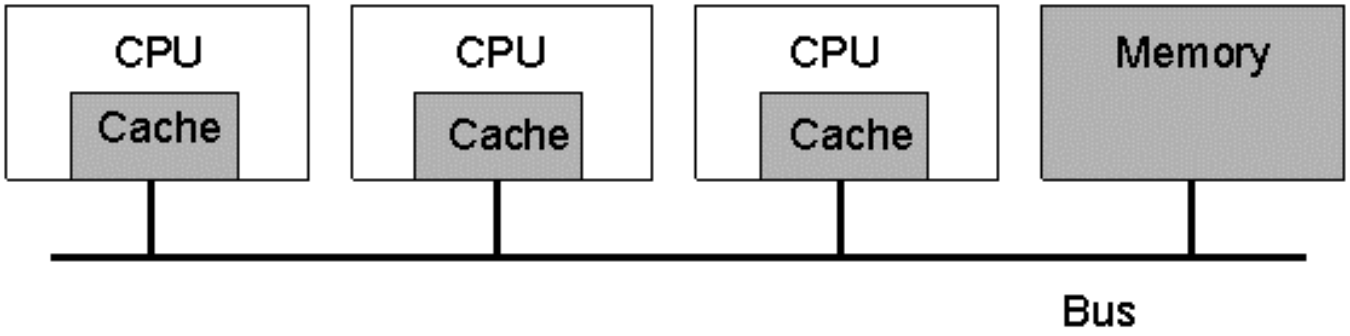
Hardware Concepts



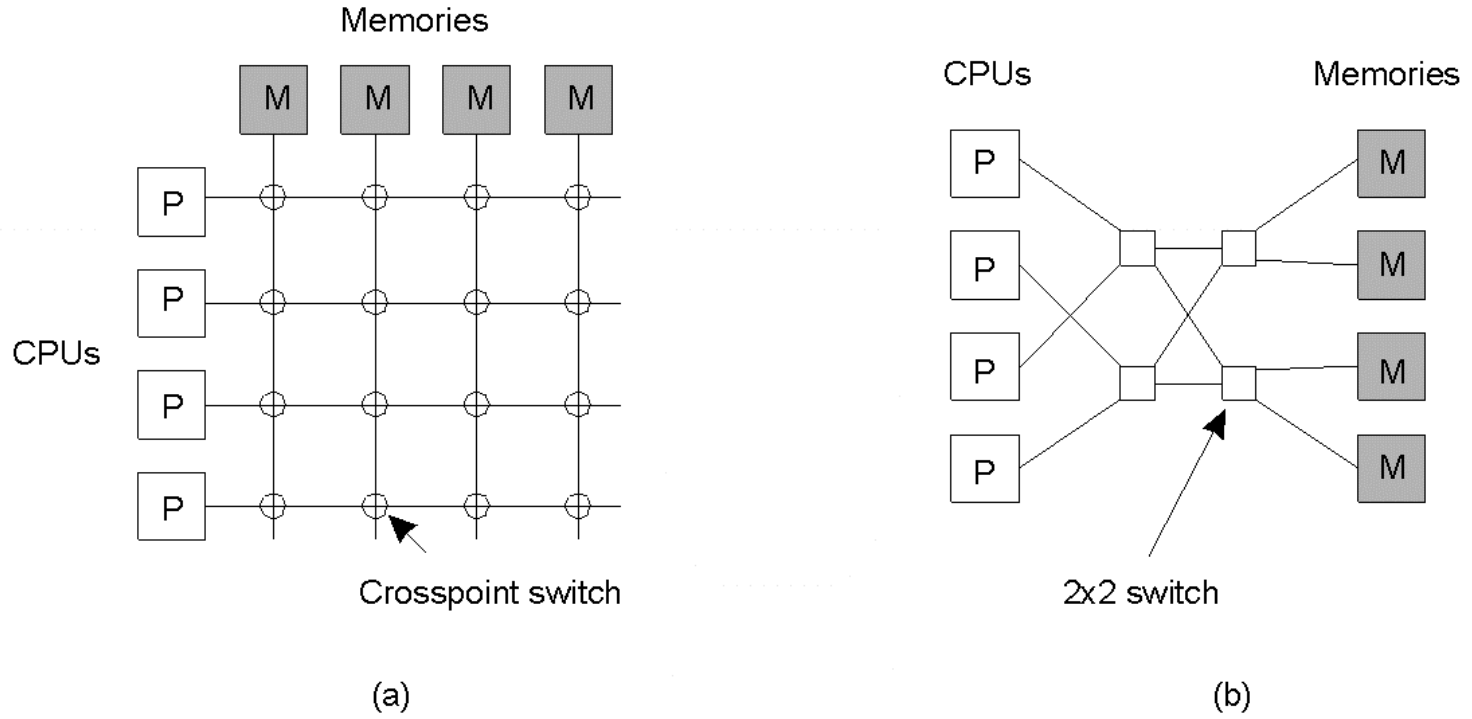
Different basic organizations and memories in distributed computer systems

Multiprocessors (1)

A bus-based multiprocessor.

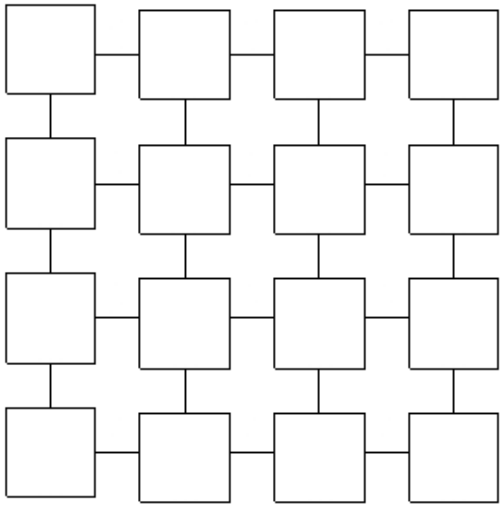


Multiprocessors (2)

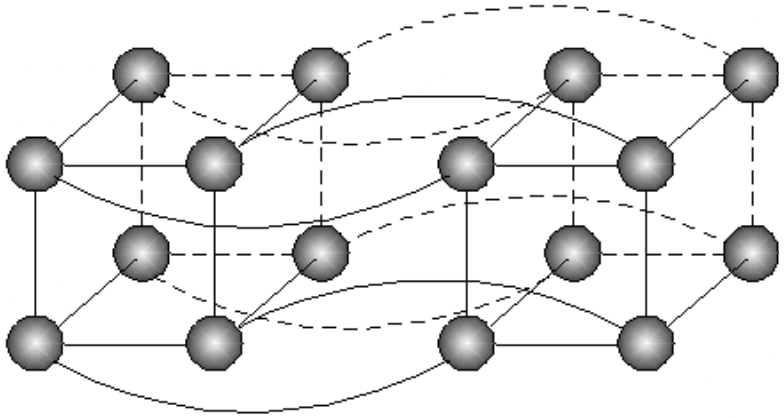


Homogeneous Multicomputer Systems

a) Grid



(a)



(b)

Networks of Computers

High degree of node heterogeneity:

- High-performance parallel systems (multiprocessors as well as multicomputers)
- High-end PCs and workstations (servers)
- Simple network computers (offer users only network access)
- Mobile computers (palmtops, laptops)
- Multimedia workstations

High degree of network heterogeneity:

- Local-area gigabit networks
- Wireless connections
- Long-haul, high-latency POTS connections
- Wide-area switched megabit connections

Observation: Ideally, a distributed system hides these differences

Software Concepts

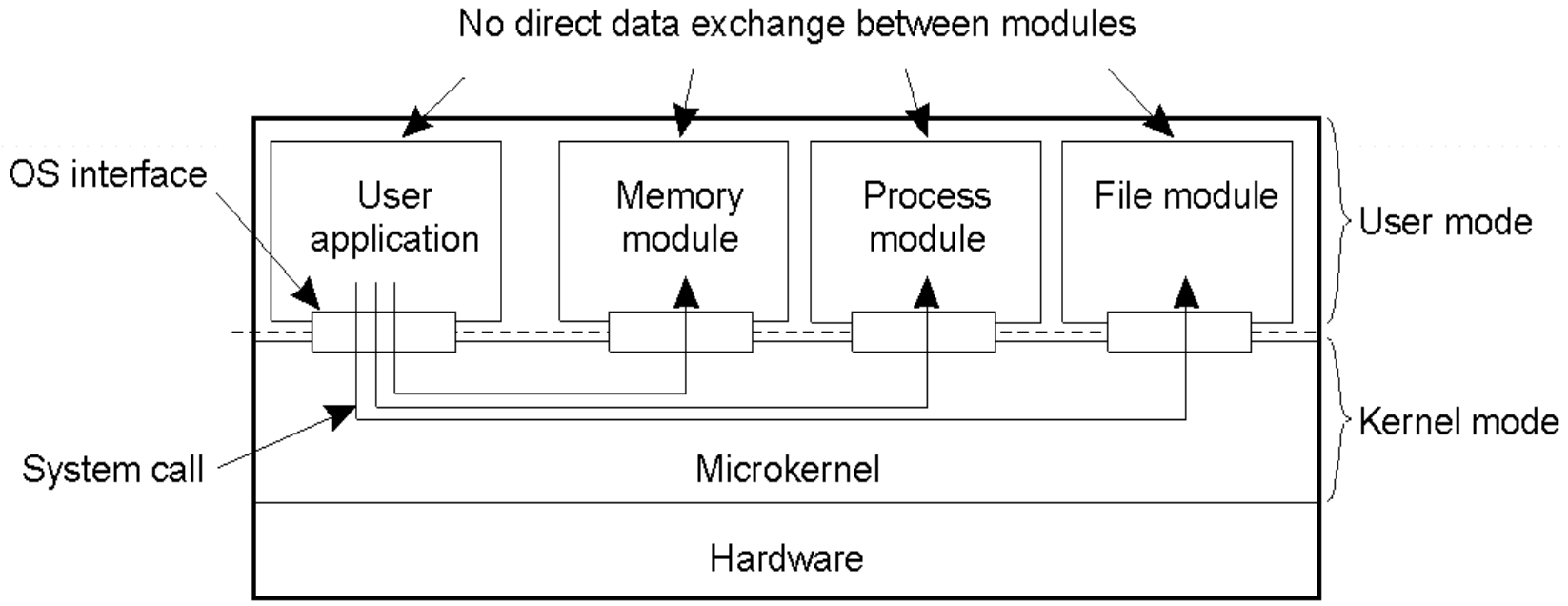
System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

An overview between

- DOS (Distributed Operating Systems)
- NOS (Network Operating Systems)
- Middleware

Uniprocessor Operating Systems

Separating applications from operating system code through a microkernel.



Multiprocessor Operating Systems (1)

A monitor to protect an integer against concurrent access.

```
monitor Counter {
private:
 int count = 0;
public:
  int value() { return count;}
  void incr () { count = count + 1;}
  void decr() { count = count – 1;}
}
```

Multiprocessor Operating Systems (2)

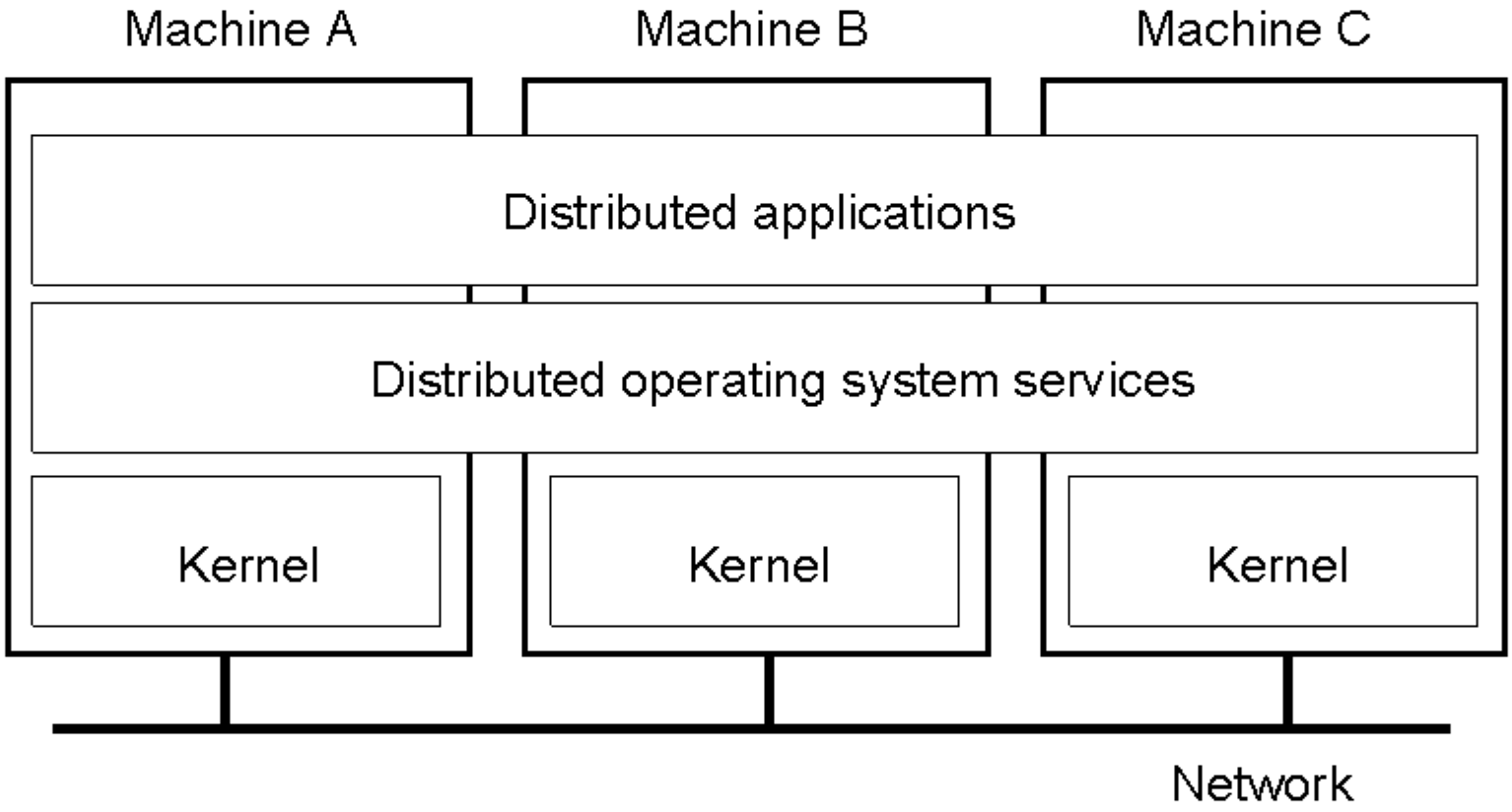
A monitor to protect an integer against concurrent access, but blocking a process.

```
monitor Counter {
private:
  int count = 0;
  int blocked_procs = 0;
  condition unblocked;
public:
  int value () { return count;}
  void incr () {
       if (blocked_procs == 0)
         count = count + 1;
       else
         signal (unblocked);
}
```

```
void decr() {
if (count ==0) {
  blocked_procs = blocked_procs + 1;
  wait (unblocked);
  blocked_procs = blocked_procs – 1;
   }
  else
   count = count – 1;
 }
}
```

Multicomputer Operating Systems (1)

General structure of a multicomputer operating system



Multicomputer Operating System

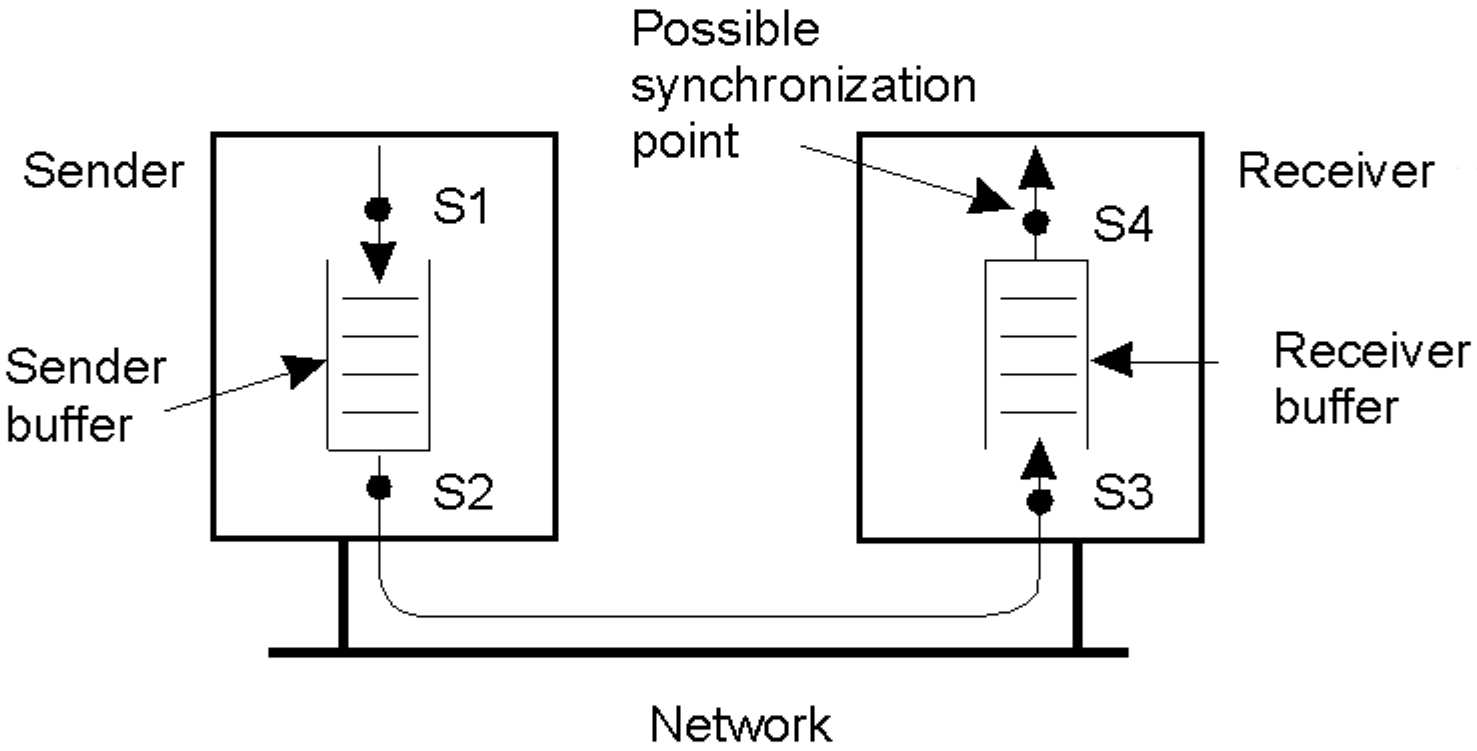
Harder than traditional (multiprocessor) OS:

Because memory is not shared, emphasis shifts to processor communication by message passing:

- Often no simple global communication:
  - Only bus-based multicomputers provide hardware broadcasting
  - Efficient broadcasting may require network interface programming techniques
- No simple systemwide synchronization mechanisms
- Virtual (distributed) shared memory requires OS to maintain global memory map in software
- Inherent distributed resource management: no central point where allocation decisions can be made

Multicomputer Operating Systems (2)

Alternatives for blocking and buffering in message passing.



Multicomputer Operating Systems (3)

Synchronization point	**Send buffer**	**Reliable comm. guaranteed?**
Block sender until buffer not full	Yes	Not necessary
Block sender until message sent	No	Not necessary
Block sender until message received	No	Necessary
Block sender until message delivered	No	Necessary

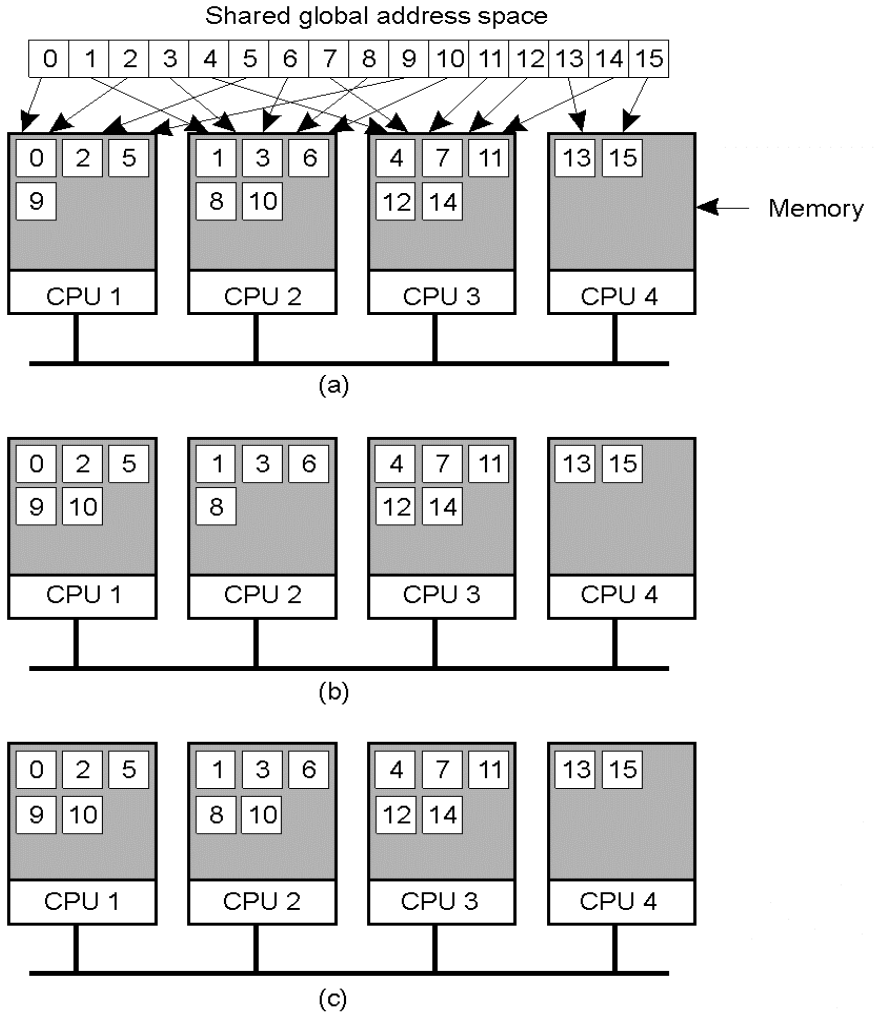
Relation between blocking, buffering, and reliable communications.

Distributed Shared Memory Systems (1)

a) Pages of address space distributed among four machines

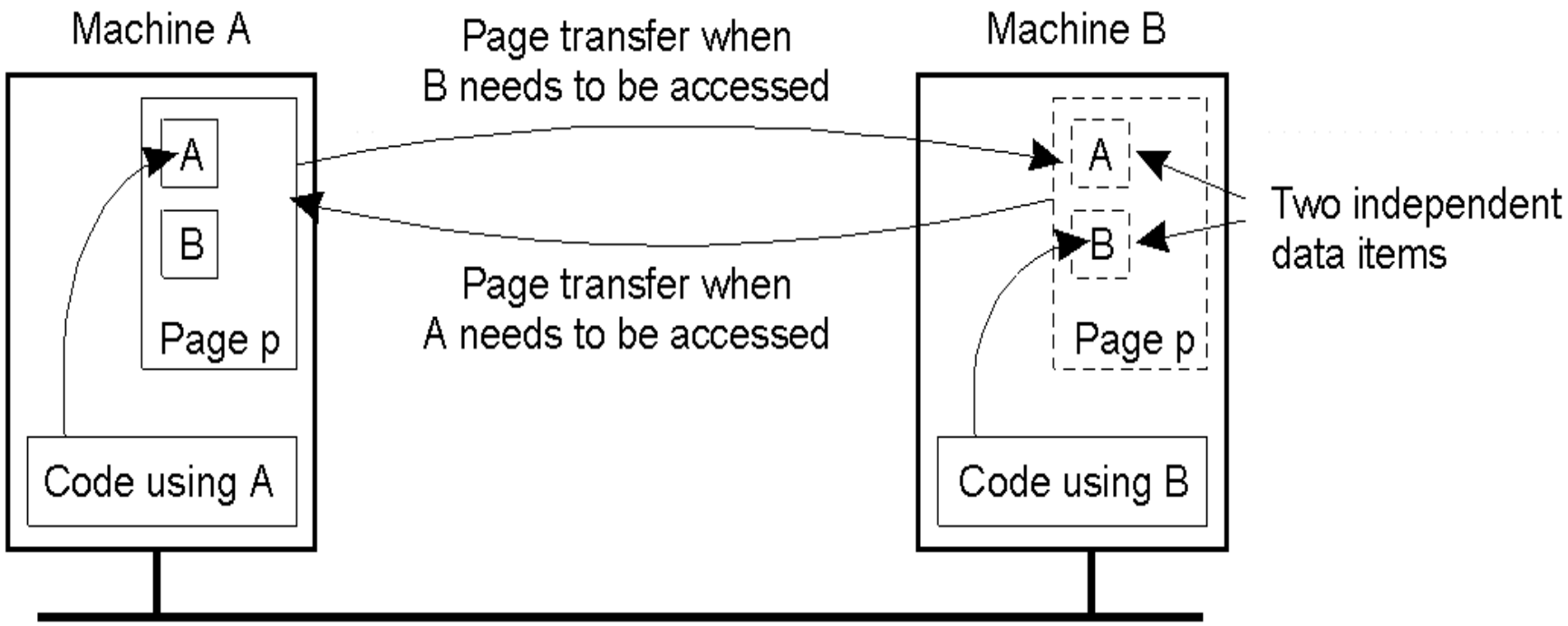
b) Situation after CPU 1 references page 10

c) Situation if page 10 is read only and replication is used



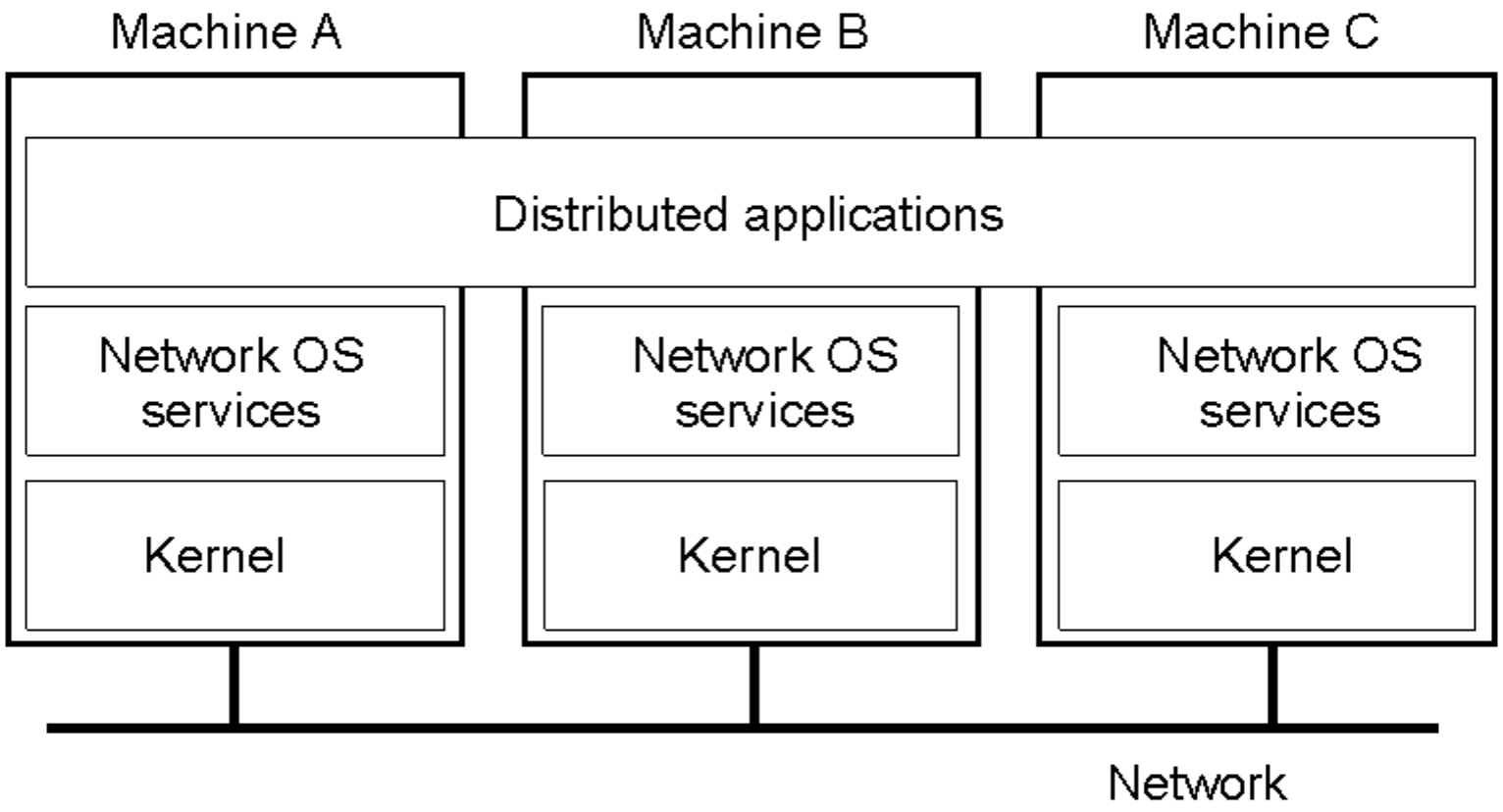
Distributed Shared Memory Systems (2)

False sharing of a page between two independent processes.



Network Operating System (1)

General structure of a network operating system.



Network Operating System

Some characteristics:

Each computer has its own operating system with networking facilities

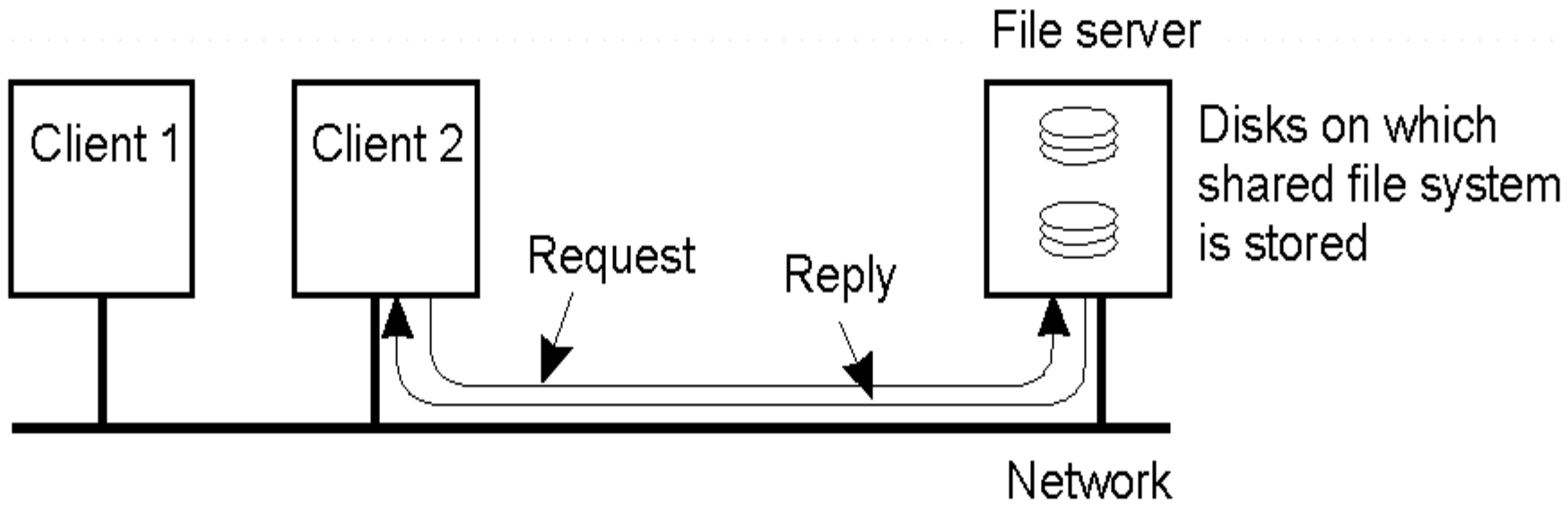
Computers work independently (i.e., they may even have different operating systems)

Services are tied to individual nodes (ftp, telnet, WWW)

Highly file oriented (basically, processors share *only* files)

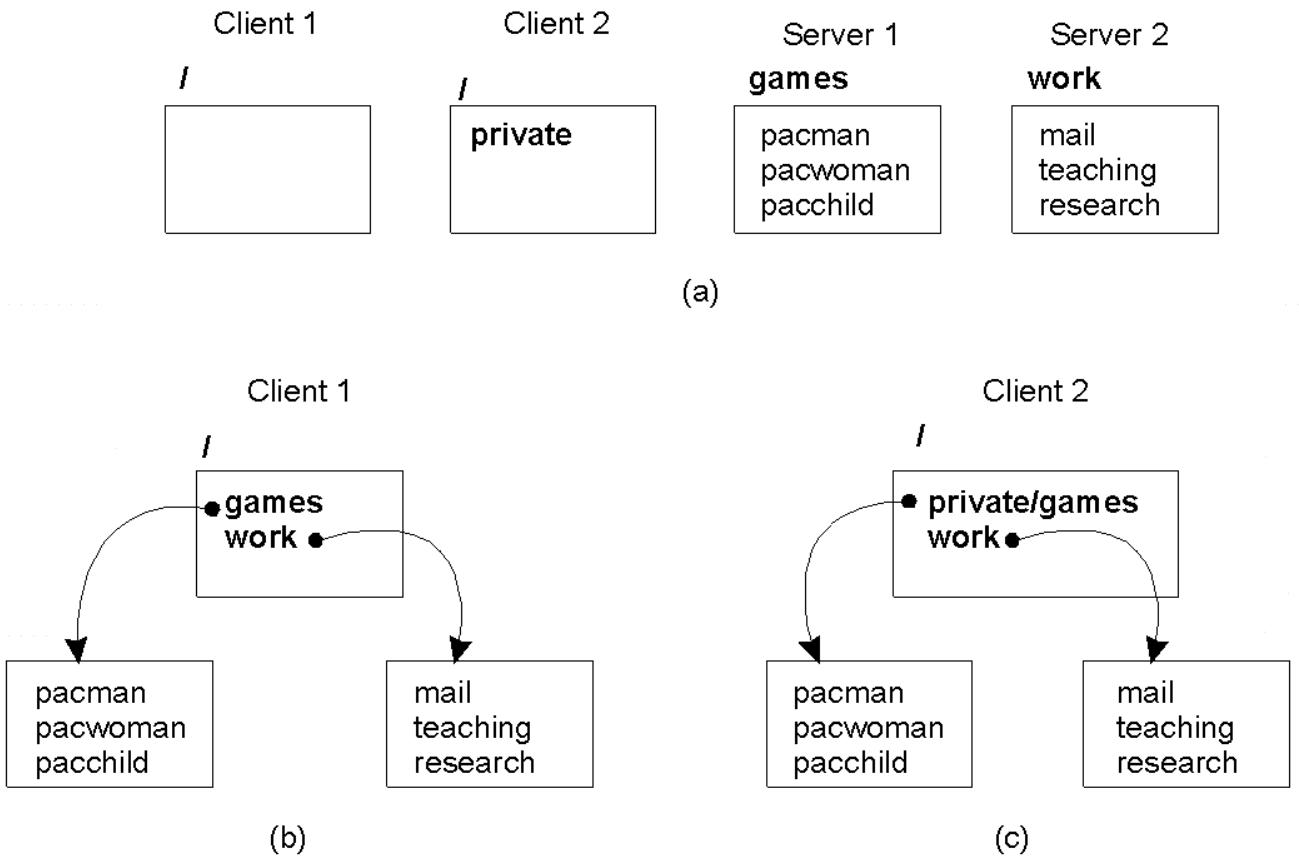
Network Operating System (2)

Two clients and a server in a network operating system.



Network Operating System (3)

Different clients may mount the servers in different places.



Need for Middleware

Motivation: Too many networked applications were hard or difficult to integrate:

Departments are running different NOSs

Integration and interoperability only at level of primitive NOS services

Need for federated information systems:

– Combining different databases, but providing a single view to applications

– Setting up enterprise-wide Internet services, making use of existing information systems

– Allow transactions across different databases

– Allow extensibility for future services (e.g., mobility, teleworking, collaborative applications)

Constraint: use the existing operating systems, and treat them as the underlying environment (they provided the basic functionality anyway)

Middleware Services

Communication services: Abandon primitive socket based message passing in favor of:

Procedure calls across networks
Remote-object method invocation
Message-queuing systems
Advanced communication streams
Event notification service

Information system services: Services that help manage data in a distributed system:

Large-scale, systemwide naming services
Advanced directory services (search engines)
Location services for tracking mobile objects
Persistent storage facilities
Data caching and replication

Middleware Services

Control services: Services giving applications control over when, where, and how they access data:

- Distributed transaction processing
- Code migration

Security services: Services for secure processing and communication:

- Authentication and authorization services
- Simple encryption services
- Auditing service

Distributed System Middleware

Some characteristics:

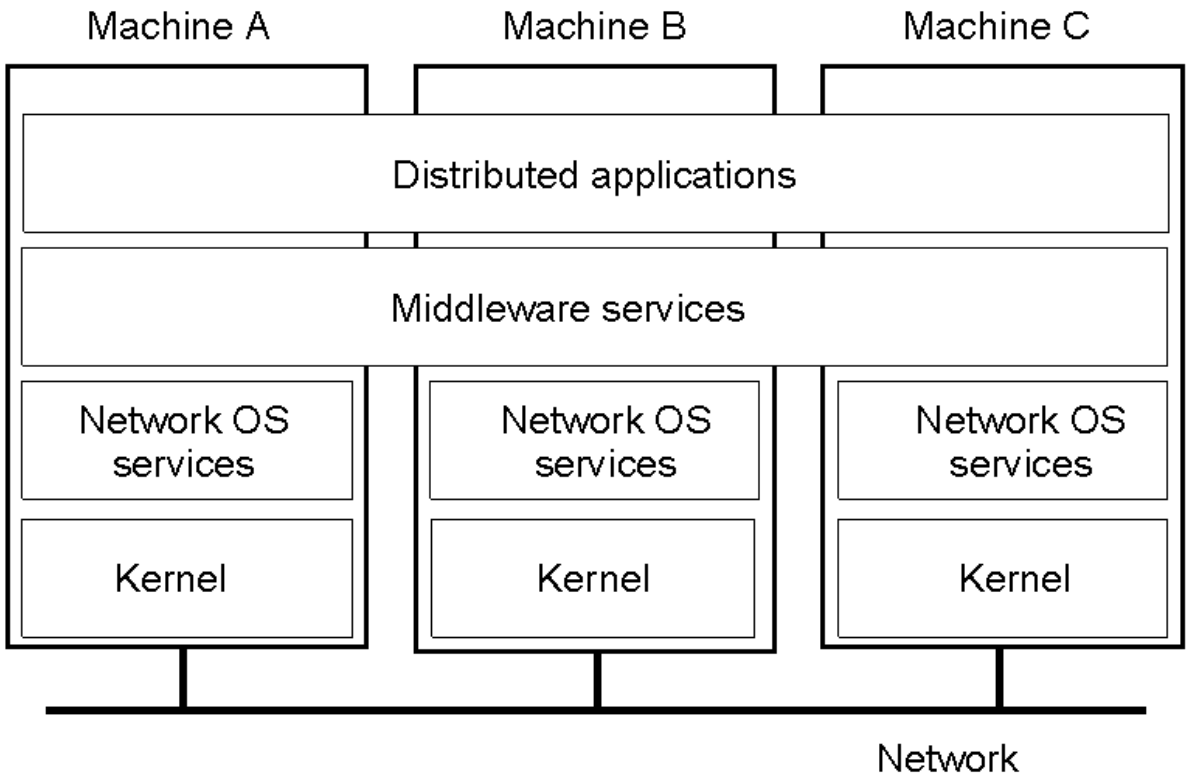
OS on each computer need not know about the other computers

OS on different computers need not generally be the same

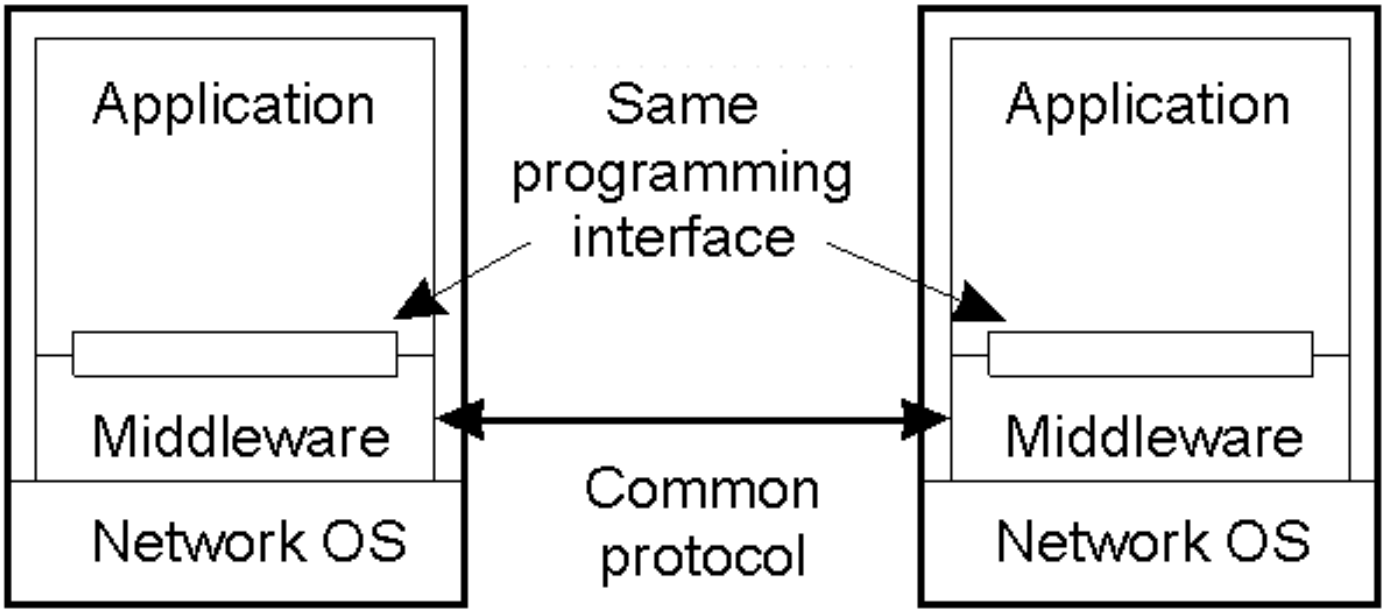
Services are generally (transparently) distributed across computers

Positioning Middleware

General structure of a distributed system as middleware.



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.

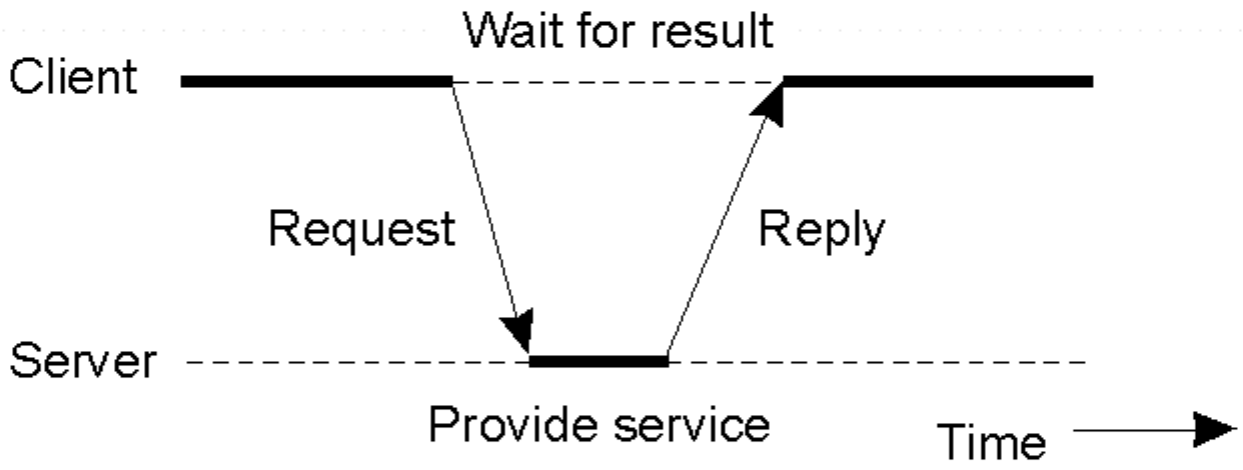
Comparison between Systems

Item	Distributed OS		Network OS	Middleware-based OS
	Multiproc.	Multicomp.		
Degree of transparency	Very High	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

A comparison between multiprocessor operating systems, multicomputer operating systems, network operating systems, and middleware based distributed systems.

Clients and Servers

General interaction between a client and a server.



Basic Client–Server Model

Servers: Generally provide services related to a *shared resource*:

- Servers for file systems, databases, implementation repositories, etc.
- Servers for shared, linked documents (Web, Lotus Notes)
- Servers for shared applications
- Servers for shared distributed objects

Clients: Allow remote service access:

- Programming interface transforming client's local service calls to request/reply messages
- Devices with (relatively simple) digital components (barcode readers, teller machines, hand-held phones)
- Computers providing independent user interfaces for specific services
- Computers providing an integrated user interface for related services (compound documents)

An Example Client and Server (1)

```
/* Definitions needed by clients and servers.          */
#define TRUE              1
#define MAX_PATH          255   /* maximum length of file name          */
#define BUF_SIZE          1024  /* how much data to transfer at once    */
#define FILE_SERVER       243   /* file server's network address        */

/* Definitions of the allowed operations */
#define CREATE            1     /* create a new file                    */
#define READ              2     /* read data from a file and return it  */
#define WRITE             3     /* write data to a file                 */
#define DELETE            4     /* delete an existing file              */

/* Error codes. */
#define OK                0     /* operation performed correctly        */
#define E_BAD_OPCODE      -1    /* unknown operation requested          */
#define E_BAD_PARAM       -2    /* error in a parameter                 */
#define E_IO              -3    /* disk error or other I/O error        */

/* Definition of the message format. */
struct message {
     long source;               /* sender's identity                    */
     long dest;                 /* receiver's identity                  */
     long opcode;               /* requested operation                  */
     long count;                /* number of bytes to transfer          */
     long offset;               /* position in file to start I/O        */
     long result;               /* result of the operation              */
     char name[MAX_PATH];       /* name of file being operated on       */
     char data[BUF_SIZE];       /* data to be read or written           */
};
```

The *header.h* file used by the client and server.

An Example Client and Server (2)

```
#include <header.h>
void main(void) {
     struct message m1, m2;                  /* incoming and outgoing messages      */
     int r;                                  /* result code                         */

     while(TRUE) {                           /* server runs forever                 */
          receive(FILE_SERVER, &m1);         /* block waiting for a message         */
          switch(m1.opcode) {                /* dispatch on type of request         */
               case CREATE:    r = do_create(&m1, &m2); break;
               case READ:      r = do_read(&m1, &m2); break;
               case WRITE:     r = do_write(&m1, &m2); break;
               case DELETE:    r = do_delete(&m1, &m2); break;
               default:        r = E_BAD_OPCODE;
          }
          m2.result = r;                     /* return result to client             */
          send(m1.source, &m2);              /* send reply                          */
     }
}
```

A sample server.

An Example Client and Server (3)

```
#include <header.h>
int copy(char *src, char *dst){                 /* procedure to copy file using the server    */
     struct message ml;                         /* message buffer                             */
     long position;                             /* current file position                      */
     long client = 110;                         /* client's address                           */

     initialize( );                             /* prepare for execution                      */
     position = 0;
     do {
          ml.opcode = READ;                     /* operation is a read                        */
          ml.offset = position;                 /* current position in the file               */
          ml.count  = BUF_SIZE;                                                                /* how many bytes to read*/
          strcpy(&ml.name, src);                /* copy name of file to be read to message    */
          send(FILESERVER, &ml);                /* send the message to the file server        */
          receive(client, &ml);                 /* block waiting for the reply                */

          /* Write the data just received to the destination file.                             */
          ml.opcode = WRITE;                    /* operation is a write                       */
          ml.offset = position;                 /* current position in the file               */
          ml.count  = ml.result;                /* how many bytes to write                    */
          strcpy(&ml.name, dst);                /* copy name of file to be written to buf     */
          send(FILE_SERVER, &ml);               /* send the message to the file server        */
          receive(client, &ml);                 /* block waiting for the reply                */
          position += ml.result;                /* ml.result is number of bytes written       */
     } while( ml.result > 0 );                  /* iterate until done                         */
     return(ml.result >= 0 ? OK : ml result);   /* return OK or error code                    */
}
```

A client using the server to copy a file.

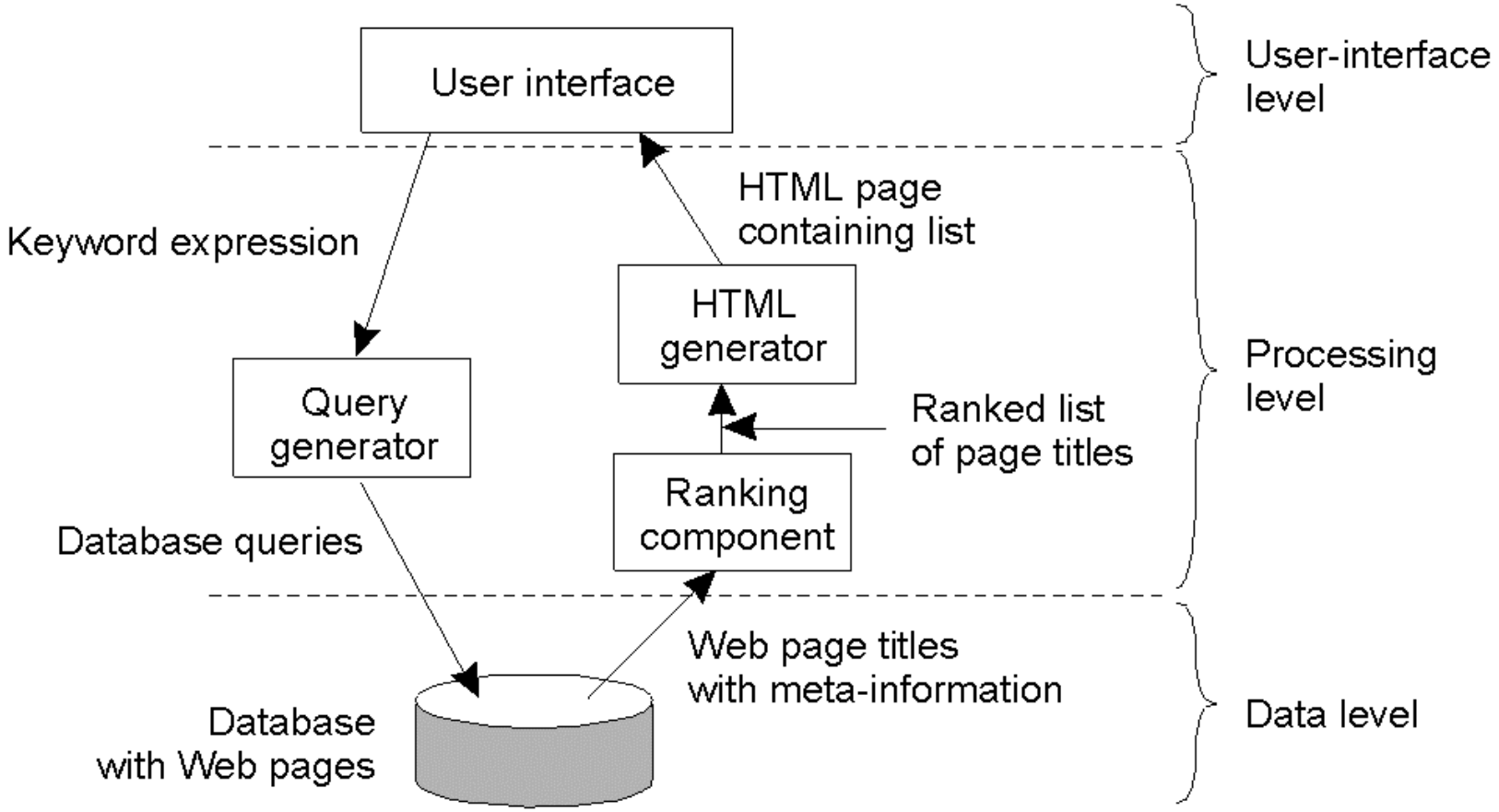
Application Layering

Traditional three-layered view:

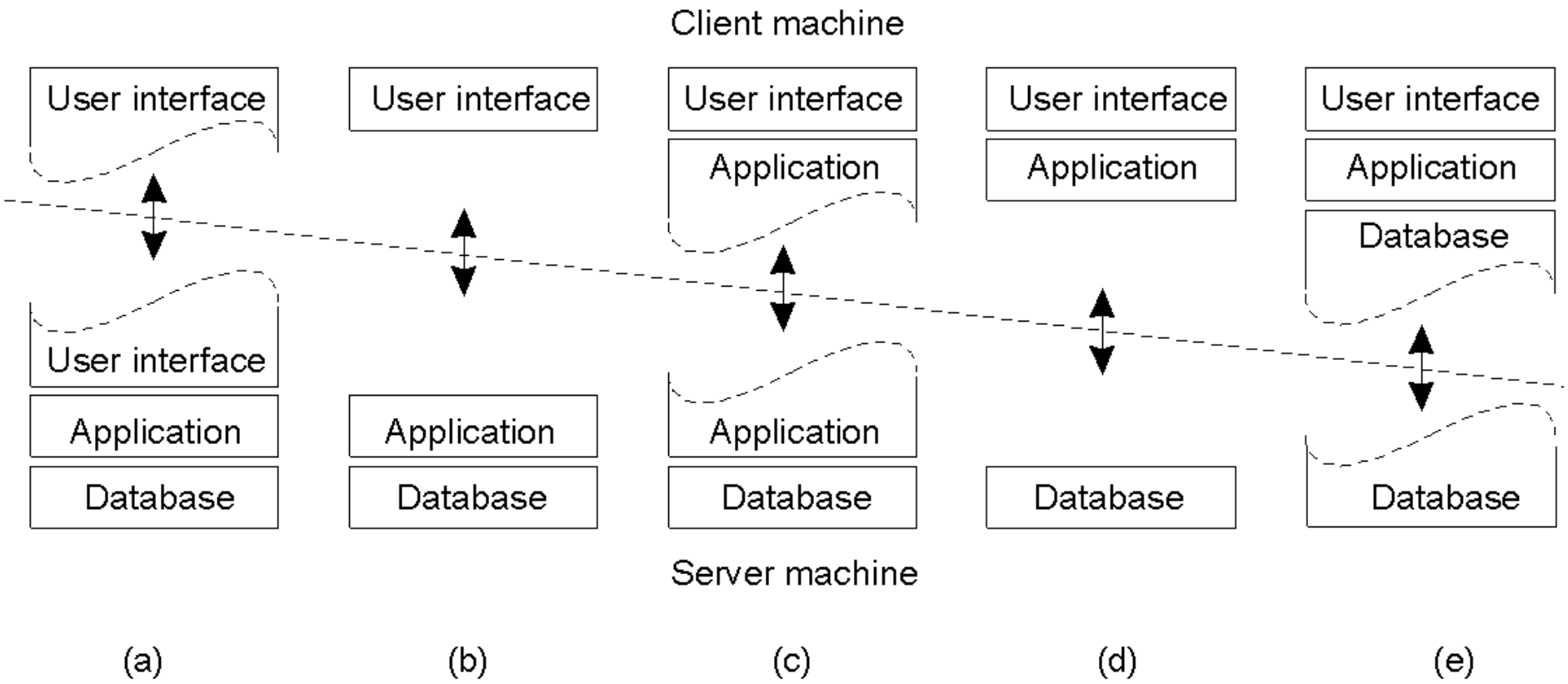
- User-interface layer contains units for an application's user interface
- Processing layer contains the functions of an application, i.e. without specific data
- Data layer contains the data that a client wants to manipulate through the application components

Observation: This layering is found in many distributed information systems, using traditional database technology and accompanying applications.

Processing Level

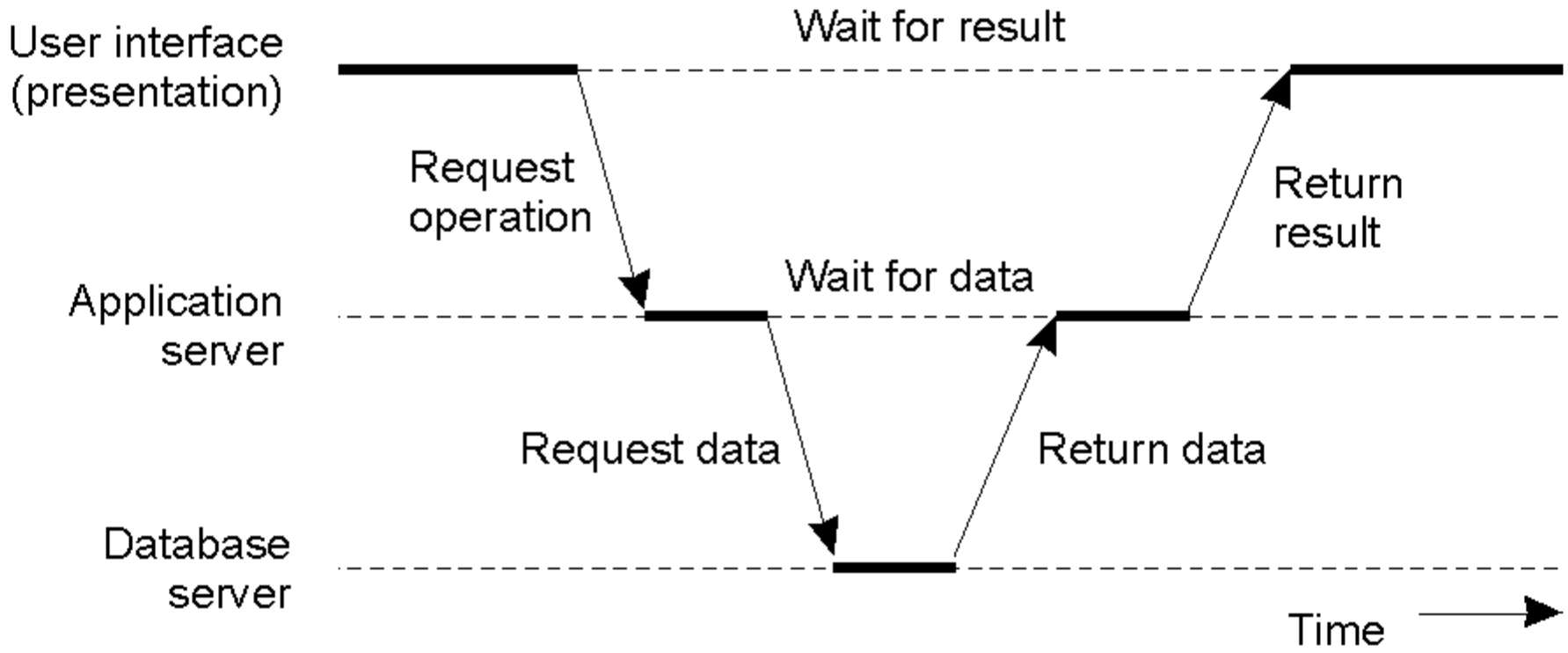


Multitiered Architectures (1)



Multitiered Architectures (2)

An example of a server acting as a client.



Modern Architectures

An example of horizontal distribution of a Web service.