Distributed Systems

Basic concepts

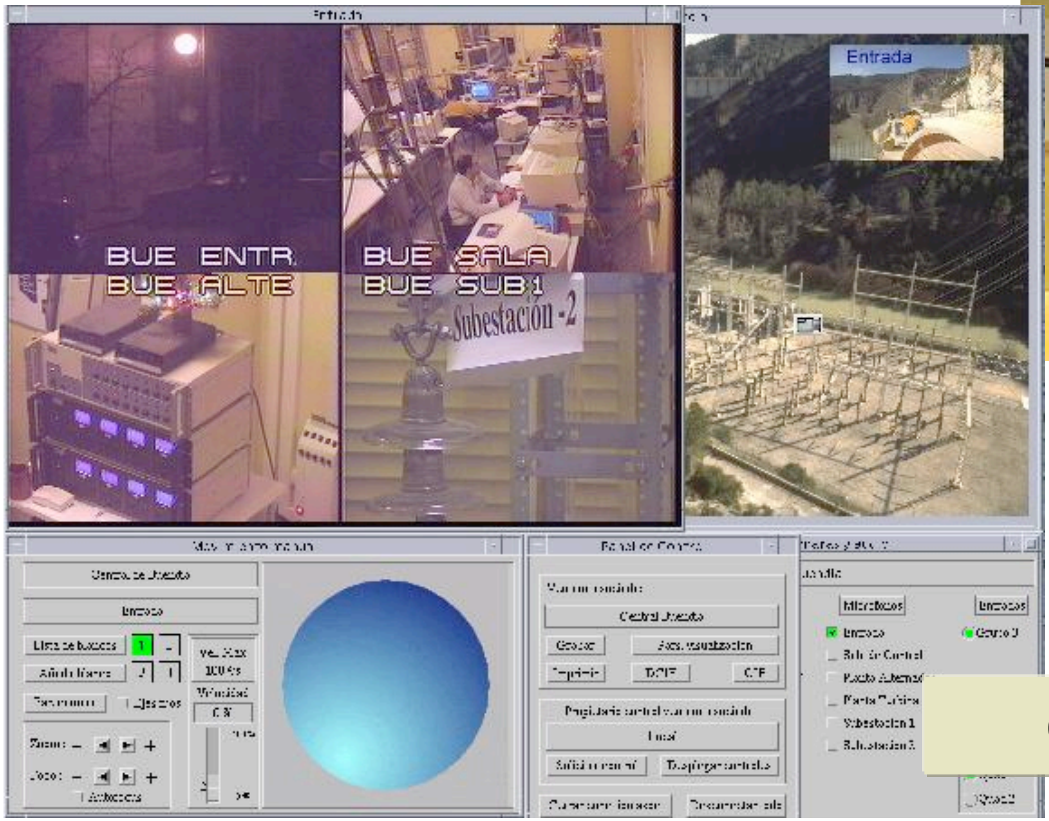
Computadores II / 2004-2005

Definition

- A **distributed system**:
  - Multiple connected CPUs working together
  - A collection of independent computers that appears to its users as a single coherent system
  - One in which components located at networked computers communicate and coordinate their actions only by passing messages.
- Examples
  - The internet
  - A local area network
  - A distributed control system
  - Mobile and ubiquitous computing
  - Battelfield management system

HYDRA

- Real time WAN video
- Remote operation of hydraulic power plants



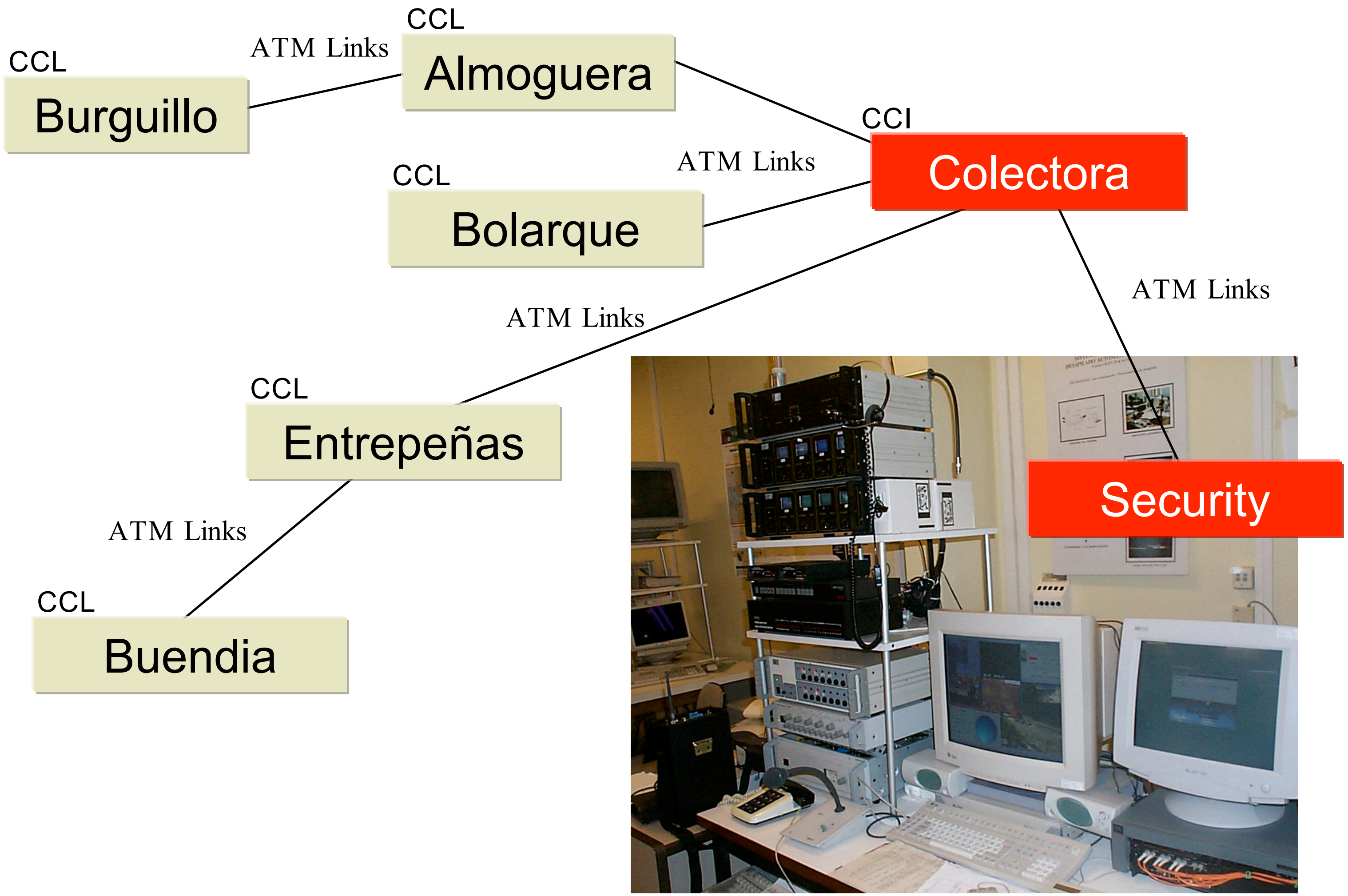
CCI

CAMERA

HYDRA : Operation Modes

- Local
  - (Almost) Classic video security system
- Remote manual
  - From CCI (Integral Control Center, operation center o another plant)
- Remote automatic
  - From CCI using events generated by SCADA
- Bidirectional Audio/Video

HYDRA : Structure



CCL

Burguillo

ATM Links

CCL

Almoguera

CCI

Colectora

ATM Links

CCL

Bolarque

ATM Links

ATM Links

CCL

Entrepeñas

ATM Links

CCL

Buendia

Security

Advantages and Disadvantages

- Advantages
  - Communication and resource sharing possible
  - Economics – price-performance ratio
  - Reliability
  - Scalability
  - Potential for incremental growth
  - Localisation
- Disadvantages
  - Complexity: design, implementation, management
  - Distribution-aware PLs, OSs and applications
  - Network connectivity essential
  - Security and privacy

Topics in Distributed Systems

- Interprocess Communication
- Processes and their scheduling
- Naming and location management
- Resource sharing, replication and consistency
- Canonical problems and solutions
- Fault-tolerance
- Security in distributed Systems
- Distributed middleware
- Further topics: web services, multimedia, real-time and mobile systems

Basic Concepts

Distributed systems and OSs

Computadores II / 2004-2005

Challenges for D-Systems

- Heterogeneity
- Openness
- Security
- Scalability
- Failure handling
- Concurrency
- Assurance
- Transparency

Heterogeneity

- Different networks, hardware, operating systems, programming languages, developers.
- We set up protocols to solve these heterogeneities.
- Middleware: a software layer that provides a programming abstraction as well as masking the heterogeneity.
- Mobile code: code that can be sent from one computer to another and run at the destination.

Openness

- The openness of DS is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs.
- Open systems are characterized by the fact that their key interfaces are published.
- Open DS are based on the provision of a uniform communication mechanism and published interfaces for access to shared resources.
- Open DS can be constrcted from heterogeneous hardware and software.

Security

- Security for information resources has three components:
  - Confidentiality: protection against disclosure to unauthorized individuals.
  - Integrity: protection against alteration or corruption.
  - Availability: protection against interference with the means to access the resources.
- Two new security challenges:
  - Denial of service attacks (DoS).
  - Security of mobile code.

Scalability

- A system is described as scalable if it remains effective when there is a significant increase in the number of resources, tasks and/or number of users.
- Challenges:
  - Controlling the cost of resources or money.
  - Controlling the performance loss.
  - Preventing software resources from running out
  - Avoiding preformance bottlenecks.

Failure handling

- When faults occur in hardware or software, programs may produce incorrect results or they may stop before they have completed the intended computation.
- Techniques for dealing with failures:
  - Detecting failures
  - Masking failures
  - Tolerating failures
  - Recovering form failures
  - Redundancy

Concurrency

- There is a possibility that several clients will attempt to access a shared resource at the same time.
- Any object that represents a shared resource in a distributed system must be responsible for ensuring that operates correctly in a concurrent environment.

Assurance

- What is possible to be assured for a localized system cannot possibly be so for a distributed system
- E.g. there are algorithms that do not have proofs of convergence when distributed

Transparency

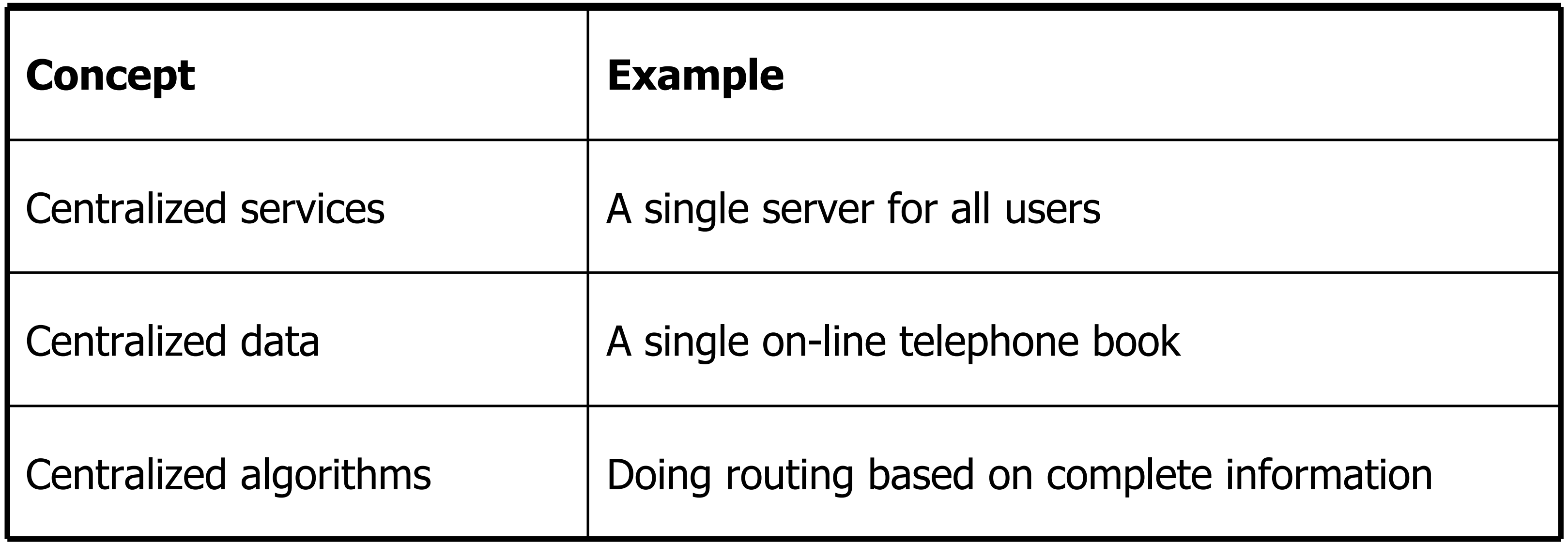
- Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components.
- Many forms of transparency:
  - Access transparency
  - Location transparency
  - Concurrency transparency
  - Replication transparency
  - Failure transparency
  - Mobility transparency
  - Technology transparency
  - Performance transparency
  - Scaling transparency

Transparency in a D-System

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

Many forms of transparency in a distributed system!

Scalability Problems



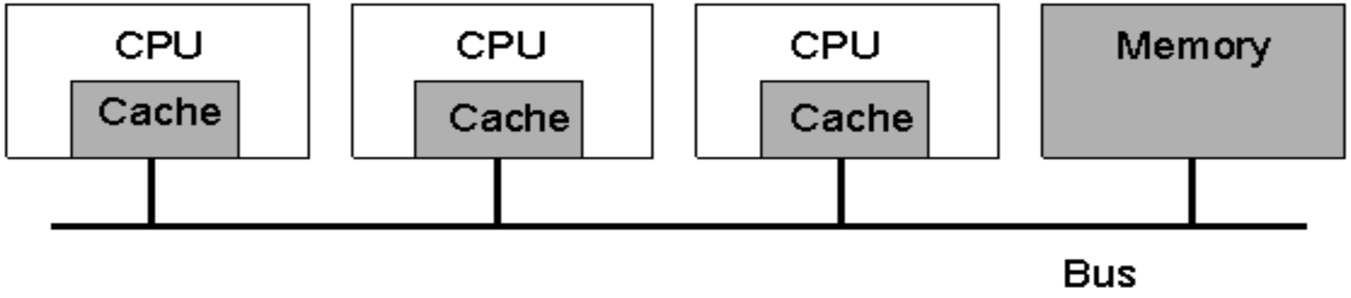
Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Examples of scalability limitations.

Multiprocessors (1)

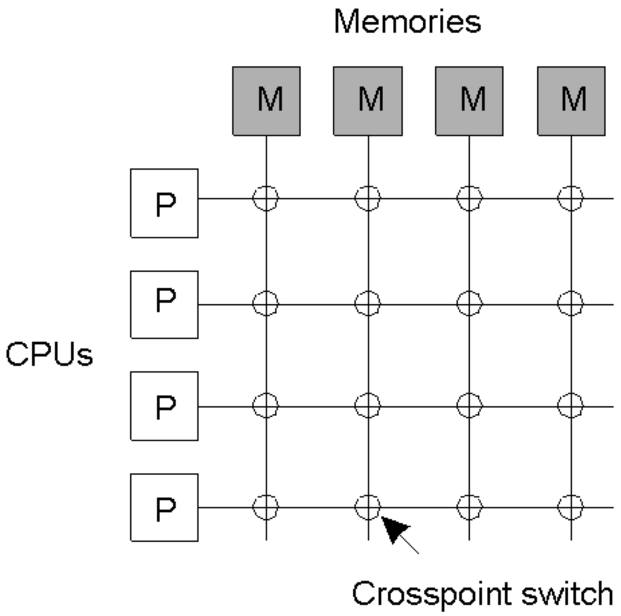
- Multiprocessor dimensions
  - Memory: could be shared or be private to each CPU
  - Interconnect: could be shared (bus-based) or switched

- A bus-based multiprocessor.

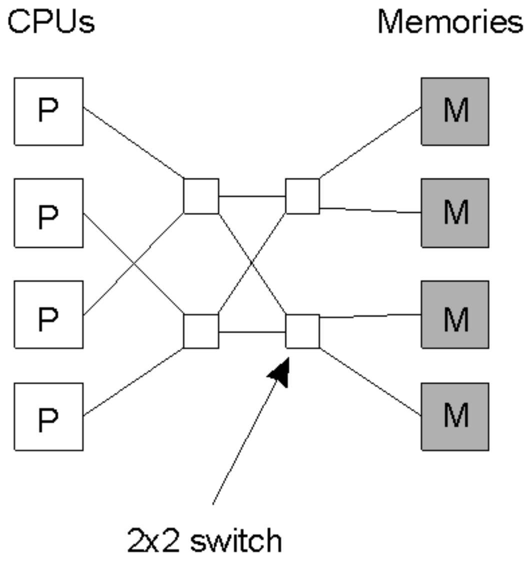


Multiprocessors (2)

- A crossbar switch
- An omega switching network



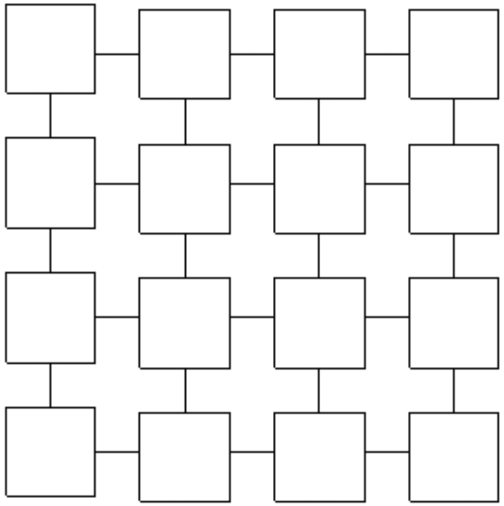
(a)



(b)

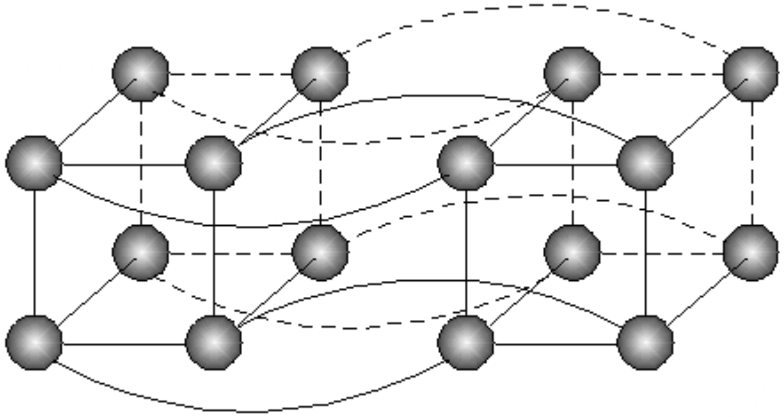
Homogeneous Multicomputers

- Grid



(a)

- Hypercube



(b)

Distributed Operating Systems

- Minicomputer model (e.g., early networks)
  - Each user has local machine
  - Local processing but can fetch remote data (files, databases)
- Workstation model (e.g., Sprite)
  - Processing can also migrate
- Client-server Model (e.g., V system, world wide web)
  - User has local workstation
  - Powerful workstations serve as servers (file, print, DB servers)
- Processor pool model (e.g., Amoeba, Plan 9)
  - Terminals are Xterms or diskless terminals
  - Pool of backend processors handle processing

Basic DOS Implementations

- Distributed OS
  - One OS / Many processors
  - Two variantes: Multiprocessor and Multicomputer
- Network-oriented OS
  - Many OSs
  - Network-level transparency
- Middleware-based OS
  - Many OSs
  - Appplication-level transparency

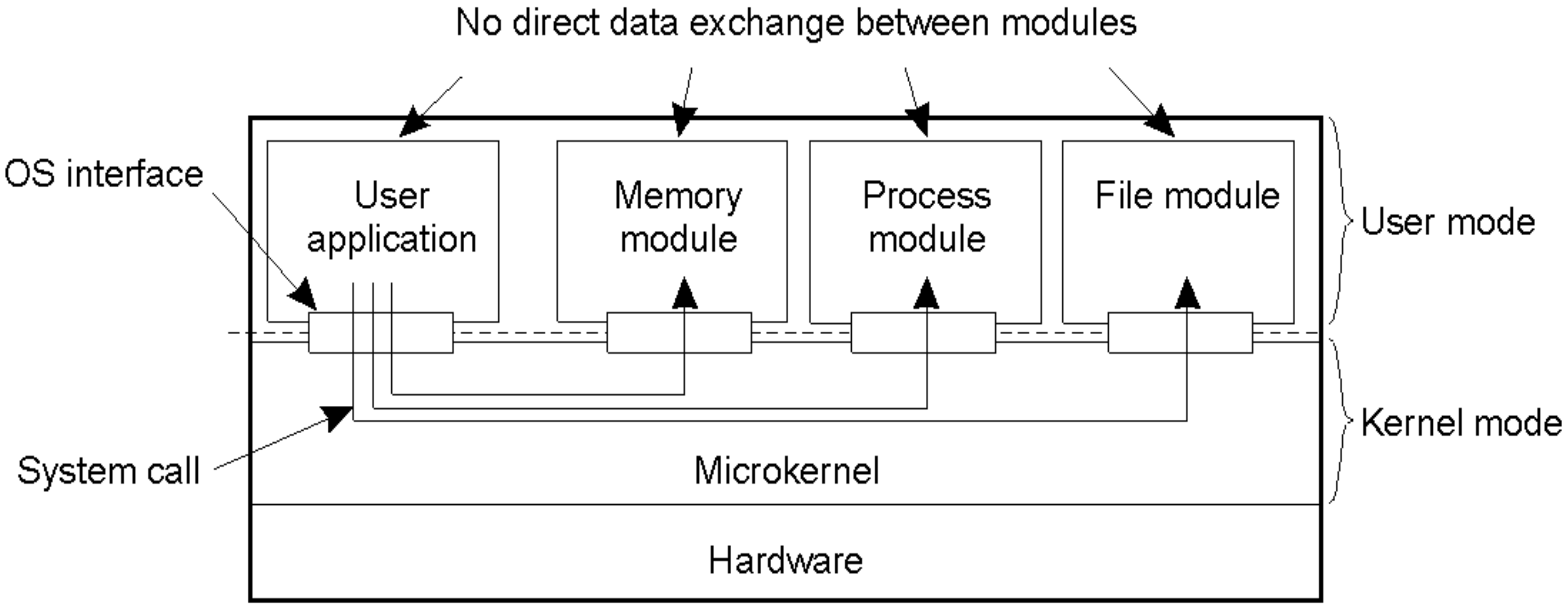
Uniprocessor Operating Systems

- An OS acts as a resource manager or an arbitrator
  - Manages CPU, I/O devices, memory
- OS provides a virtual interface that is easier to use than hardware

- Structure of uniprocessor operating systems
  - Monolithic (e.g., MS-DOS, early UNIX)
    - One large kernel that handles everything
  - Layered design
    - Functionality is decomposed into N layers
    - Each layer uses services of layer N-1 and implements new service(s) for layer N+1

Uniprocessor Operating Systems

- Microkernel architecture
  - Small kernel
  - user-level servers implement additional functionality



Distributed Operating System

- Manages resources in a distributed system
  - Seamlessly and transparently to the user
- Looks to the user like a centralized OS
  - But operates on multiple independent CPUs
- Provides transparency
  - Location, migration, concurrency, replication,…
- Presents users with a virtual uniprocessor

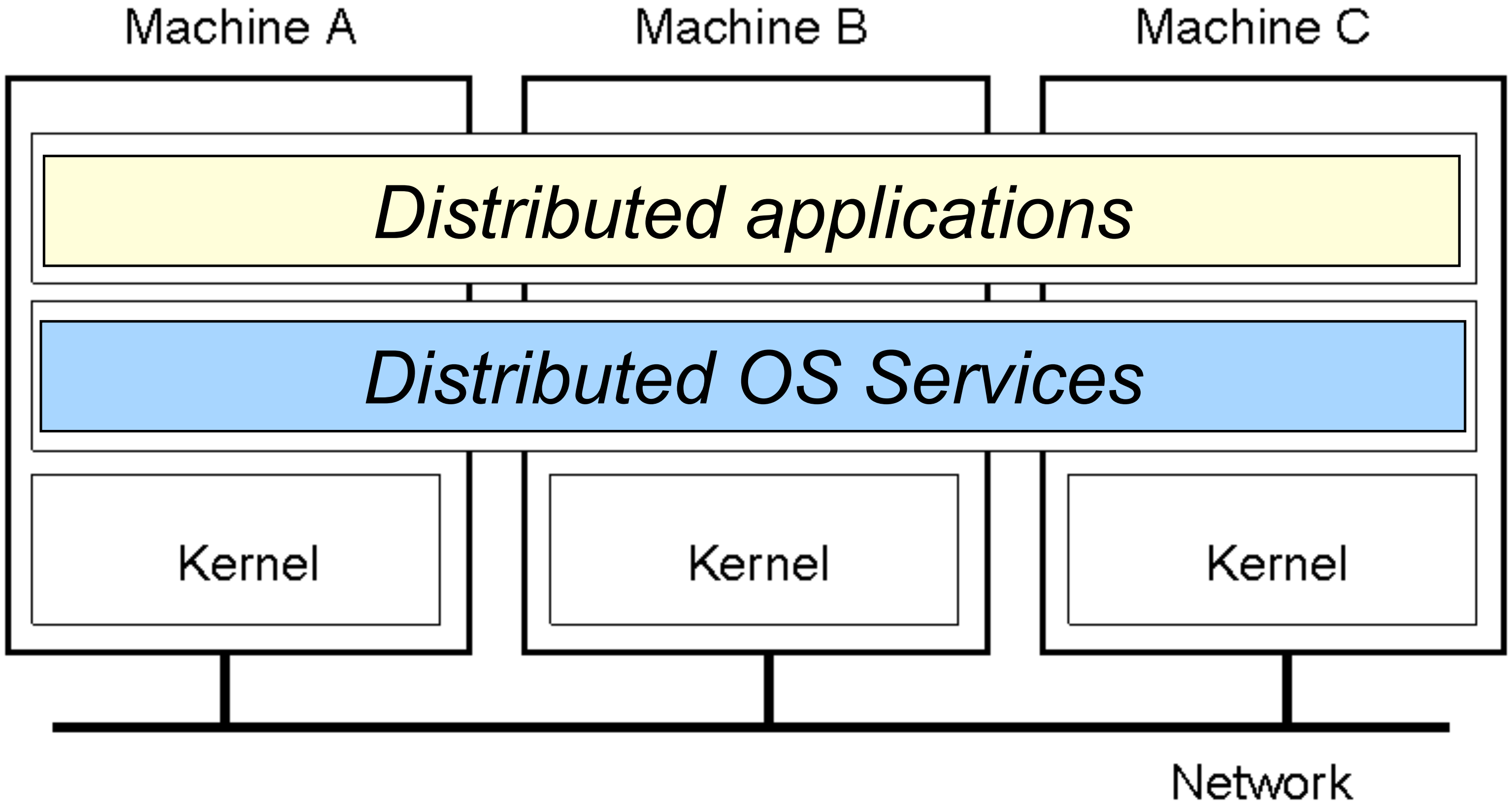
Types of Distributed OSs

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

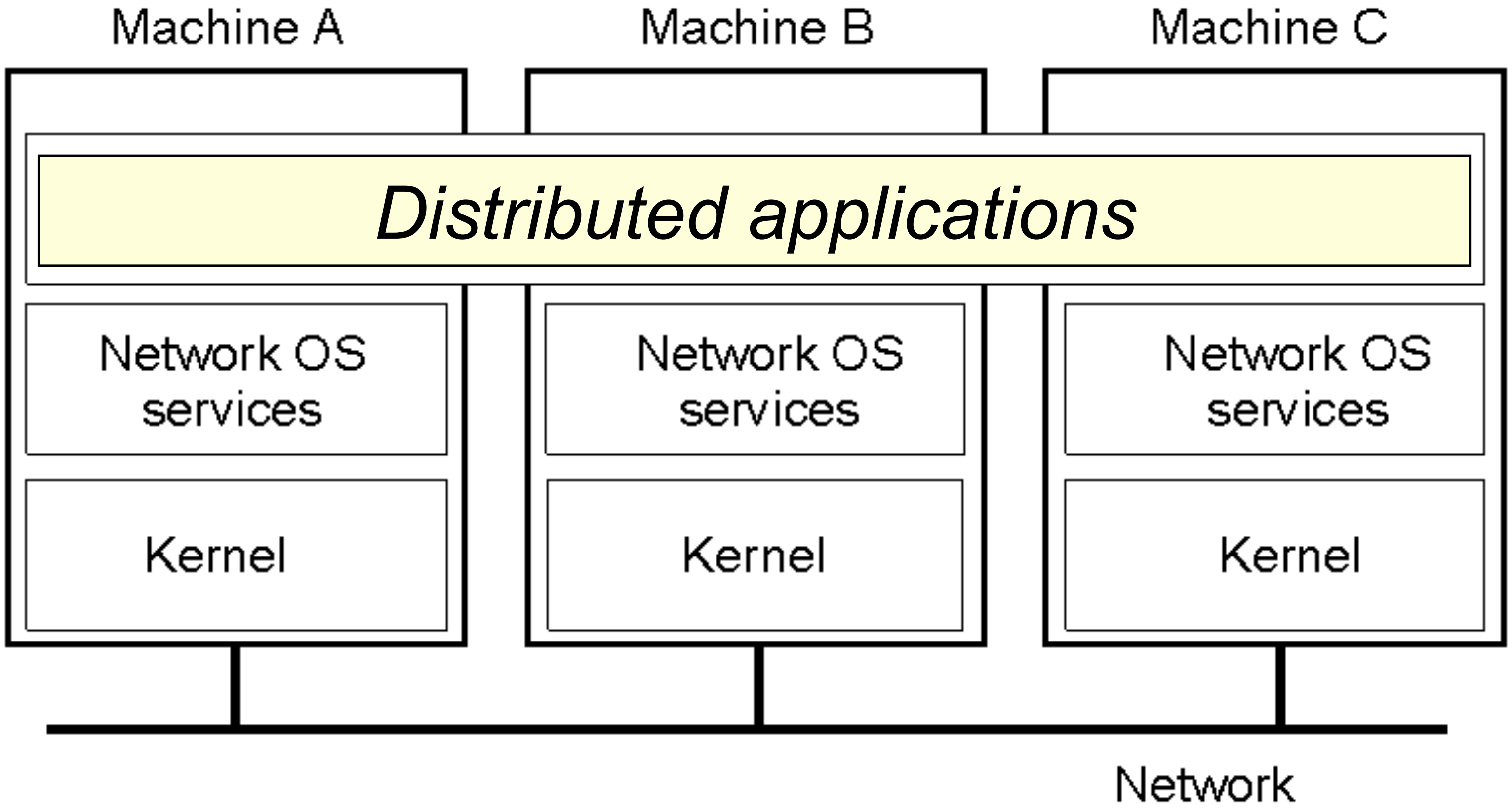
Multiprocessor OSs

- Like a uniprocessor operating system
- Manages multiple CPUs transparently to the user
- Two variants
  - SMP: Symmetric Multiprocessing
  - AMP: Asymmetric Multiprocessing
- Each processor has its own hardware cache
  - Maintain consistency of cached data

Multicomputer OSs

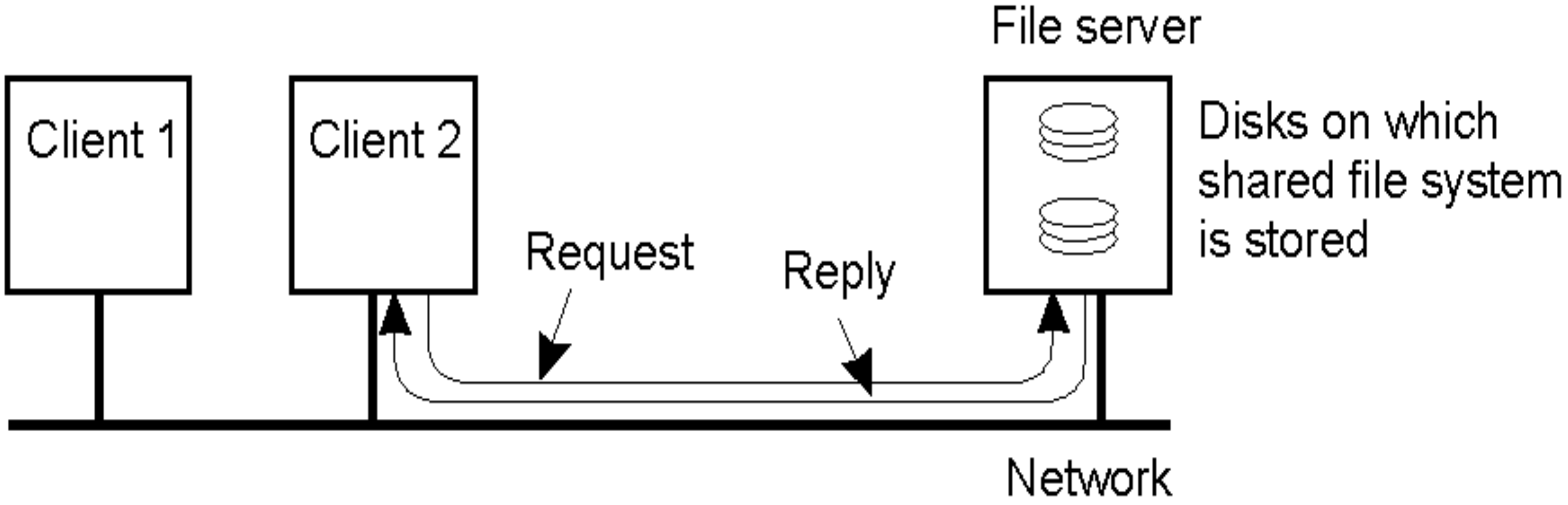


Network Operating System



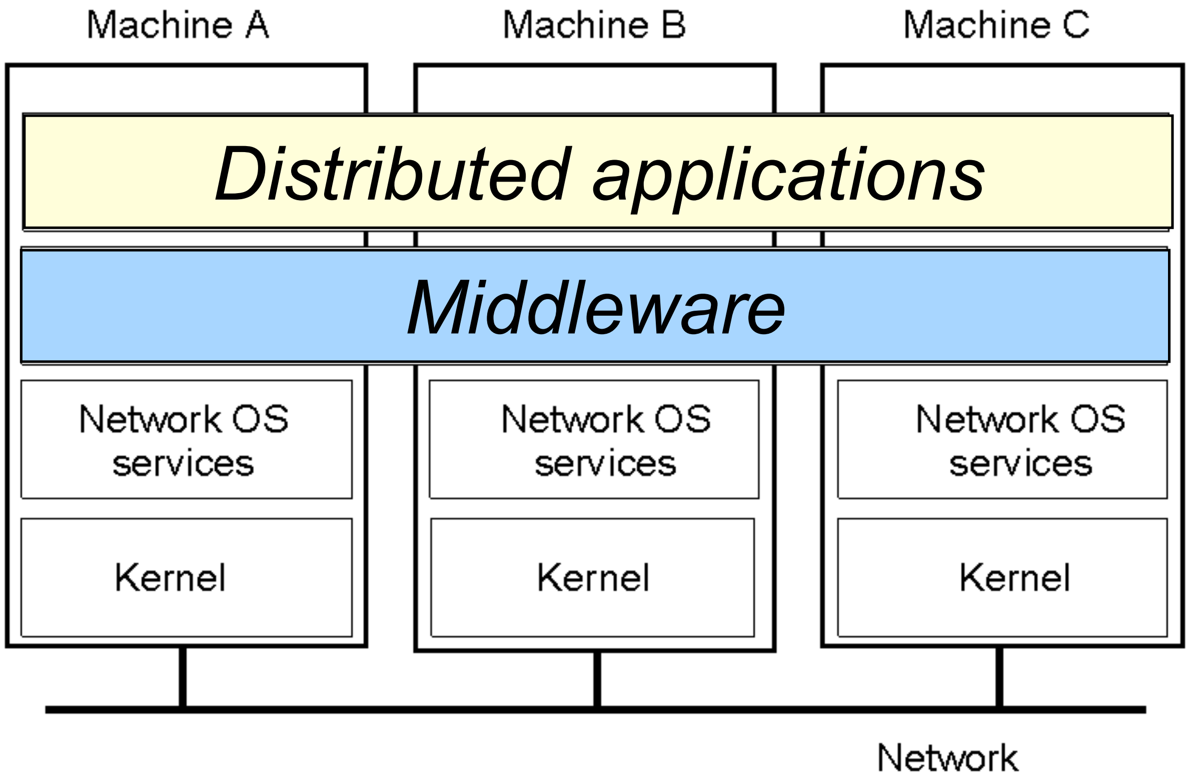
Network Operating System

- Employs a client-server model
  - Minimal OS kernel
  - Additional functionality as user processes



Middleware-based Systems

- General structure of a distributed system as middleware.



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

Communication in Distributed Systems

Basic Concepts

Computadores II / 2004-2005

Communication

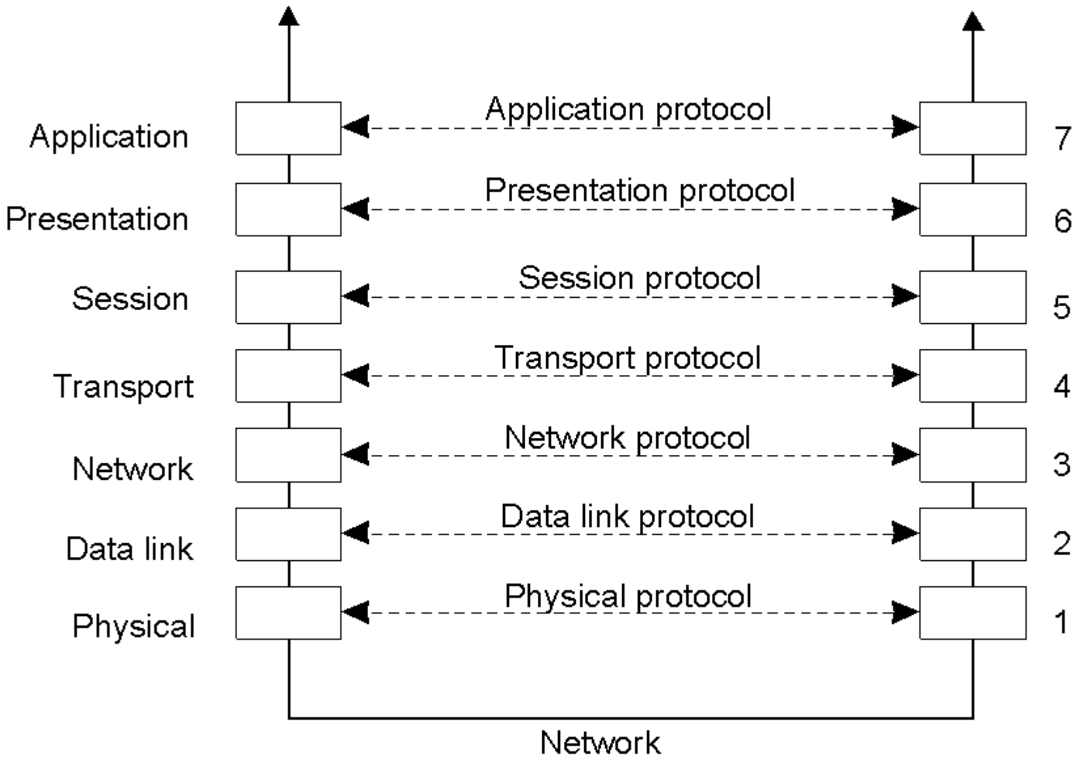
- Message-oriented Communication
- Remote Procedure Calls
  - Transparency but poor for passing references
- Remote Method Invocation
  - RMIs are essentially RPCs but specific to remote objects
  - System wide references passed as parameters
- Stream-oriented Communication
- Broker-based Middleware
  - Maximum Transparency
  - Complexity

Interprocess Communication

- **Unstructured** communication
  - Use shared memory or shared data structures
- **Structured** communication
  - Use explicit messages (IPCs)
- Distributed Systems: both need low-level communication support (why?)

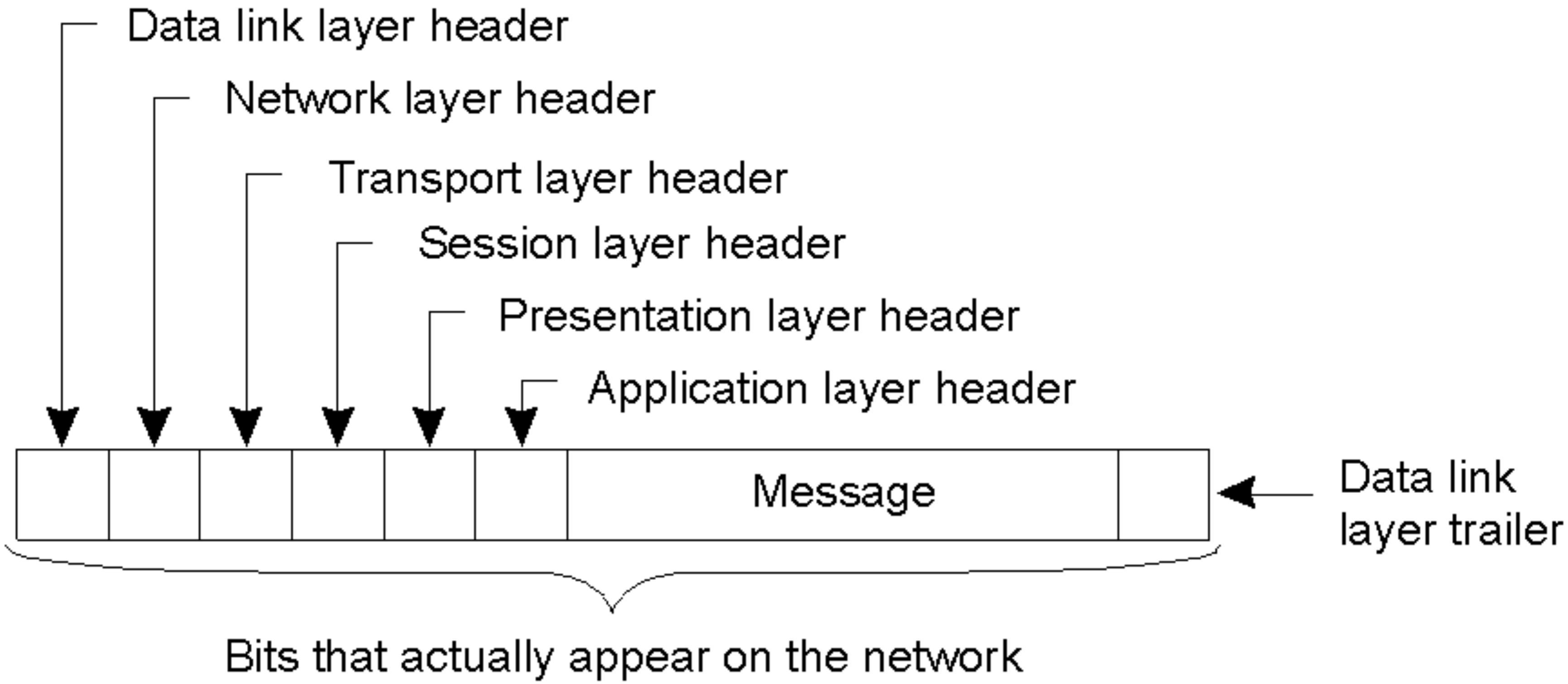
Communication Protocols

- Protocols are agreements/rules on communication
- Protocols could be connection-oriented or connectionless

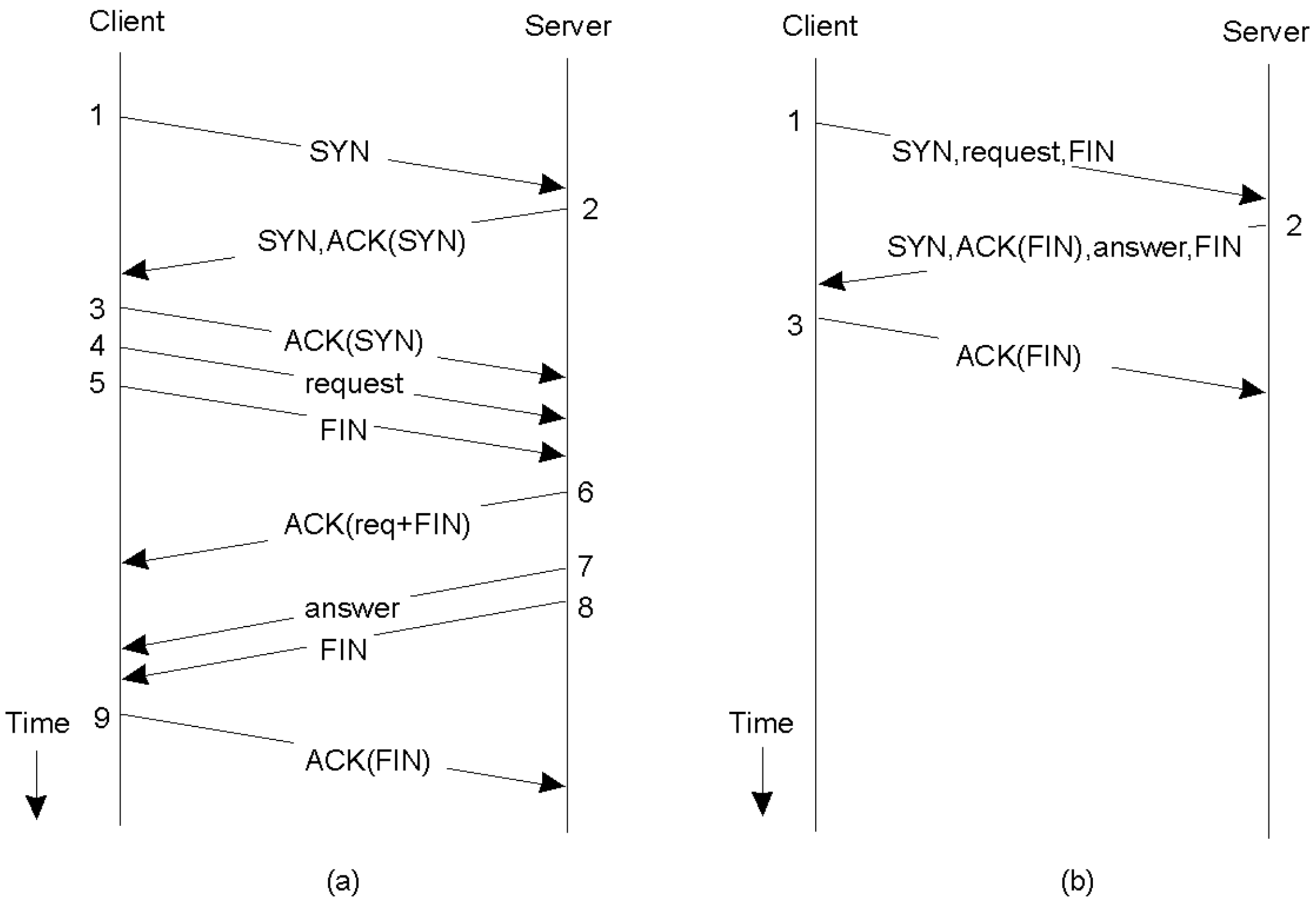


Layered Protocols

- A typical message as it appears on the network.

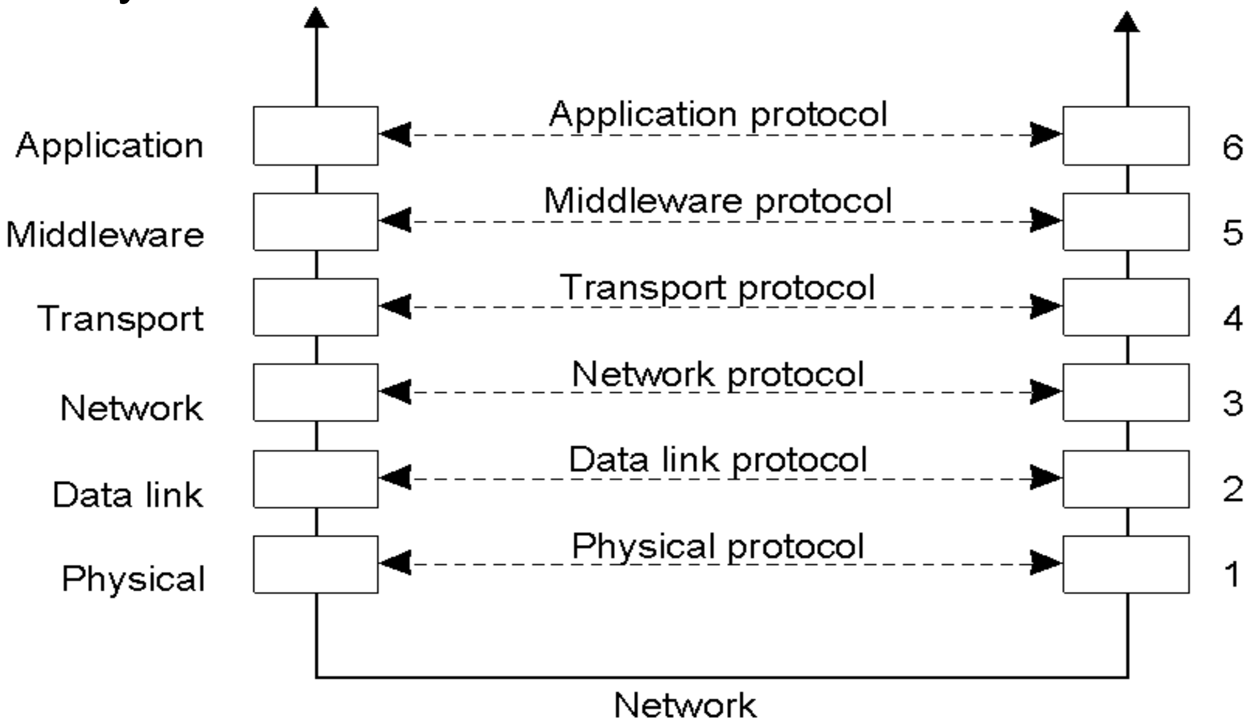


Client-Server TCP



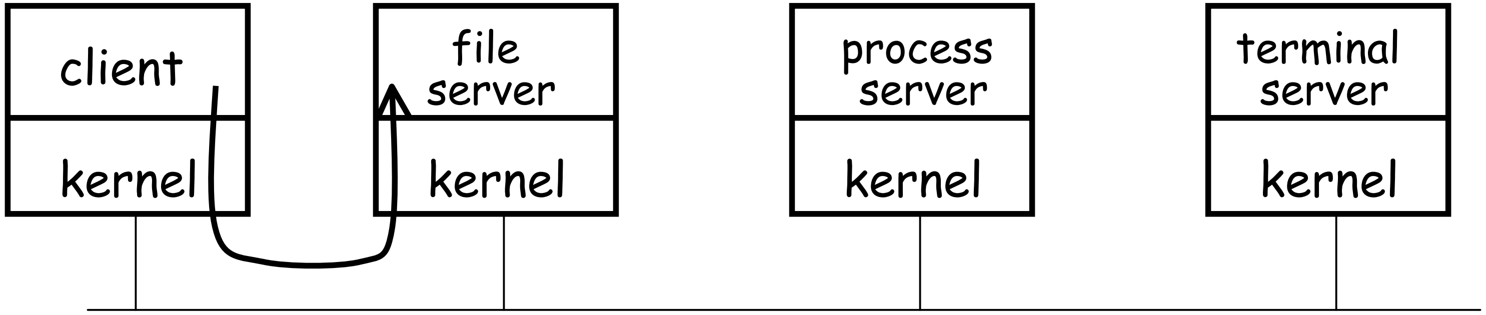
Middleware Protocols

- Middleware: layer that resides between an OS and an application
  - May implement general-purpose protocols that warrant their own layers



Client-Server Communication

- Structure: group of servers offering service to clients
- Based on a request/response paradigm
- Techniques:
  - Socket, remote procedure calls (RPC), Remote Method Invocation (RMI), Object Request Brokering (ORB)

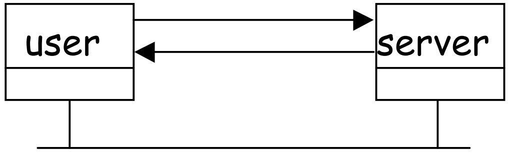


Issues in Client-Server

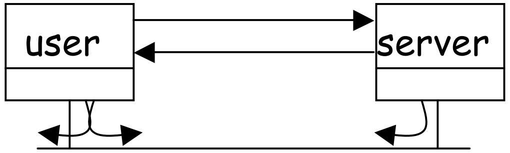
- Addressing
- Blocking versus non-blocking
- Buffered versus unbuffered
- Reliable versus unreliable
- Server architecture: concurrent versus sequential
- Scalability

Addressing Issues

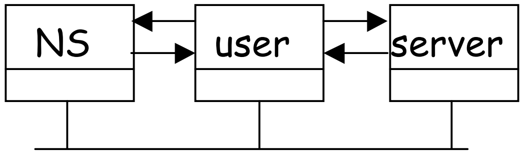
- *Question:* how is the server located?
- Hard-wired address
  - Machine address and process address are known a priori



- Broadcast-based
  - Server chooses address from a sparse address space
  - Client broadcasts request
  - Can cache response for future



- Locate address via name server



Synchronicity

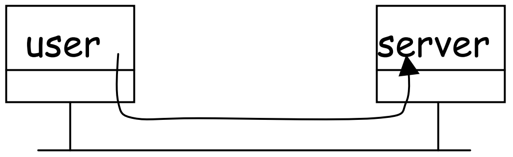
- Asynchronous communication
  - Sender continues immediately after it has submitted the message
  - Need a local buffer at the sending host
- Synchronous communication
  - Sender blocks until message is stored in a local buffer at the receiving host or actually delivered to sending
  - Variant: block until receiver processes the message

Blocking versus Non-blocking

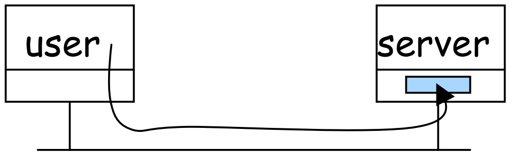
- Blocking communication (synchronous)
  - Send blocks until message is actually sent
  - Receive blocks until message is actually received
- Non-blocking communication (asynchronous)
  - Send returns immediately
  - Return does not block either

Buffering Issues

- Unbuffered communication
  - Server must call receive before client can call send

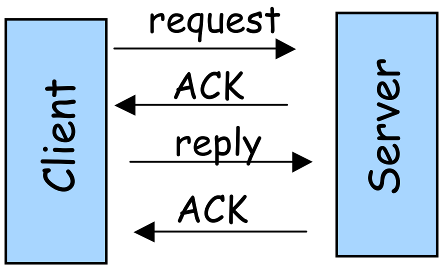


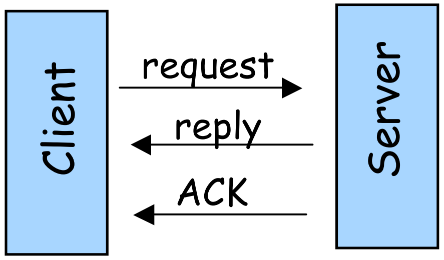
- Buffered communication
  - Client send to a mailbox
  - Server receives from a mailbox



Reliability

- Unreliable channel
  - Need acknowledgements (ACKs)
  - Applications handle ACKs
  - ACKs for both request and reply
- Reliable channel
  - Reply acts as ACK for request
  - Explicit ACK for response
- Reliable communication on unreliable channels
  - Transport protocol handles lost messages





Server Architecture

- Sequential
  - Serve one request at a time
  - Can service multiple requests by employing events and asynchronous communication
- Concurrent
  - Server spawns a process or thread to service each request
  - Can also use a pre-spawned pool of threads/processes (apache, RT-CORBA threadpools)
- Thus servers could be
  - Pure-sequential, event-based, thread-based, process-based
- Which architecture is most efficient?
  - This is application dependent

Scalability

- How can you scale the server capacity?
- Buy bigger machine!
- Replicate
- Distribute data and/or algorithms
- Ship code instead of data
- Cache

To Push or Pull ?

- Client-pull architecture
  - Clients pull data from servers (by sending requests)
  - Example: HTTP
  - Pro: stateless servers, failures are each to handle
  - Con: limited scalability
- Server-push architecture
  - Servers push data to client
  - Example: video streaming, stock tickers
  - Pro: more scalable
  - Con: stateful servers, less resilient to failure
- When/how-often to push or pull?

Group Communication

- One-to-many communication
  - Very useful for distributed applications
- Issues:
  - Group characteristics:
    - Static/dynamic, open/closed
  - Group addressing
    - Multicast, broadcast, application-level multicast (unicast)
  - Atomicity
  - Message ordering
  - Scalability

Putting it all together: Email

- User uses mail client to compose a message
- Mail client connects to mail server
- Mail server looks up address to destination mail server
- Mail server sets up a connection and passes the mail to destination mail server
- Destination stores mail in input buffer (user mailbox)
- Recipient checks mail at a later time

Email: Design Considerations

- Structured or unstructured?
- Addressing?
- Blocking/non-blocking?
- Buffered or unbuffered?
- Reliable or unreliable?
- Server architecture
- Scalability
- Push or pull?
- Group communication

Remote Procedure Call

An example of distribution technology

Computadores II / 2004-2005

Remote Procedure Calls

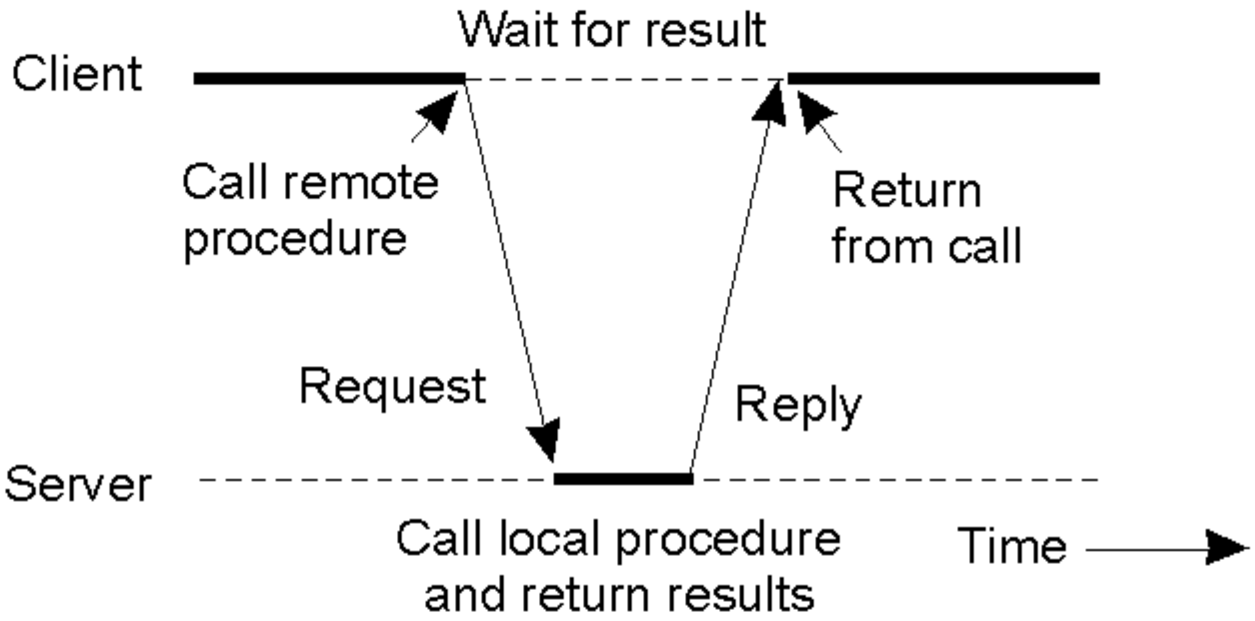
- Goal: Make distributed computing look like centralized computing
- Allow remote services to be called as procedures
  - Transparency with regard to location, implementation, language
- Issues
  - How to pass parameters
  - Bindings
  - Semantics in face of errors
- Two classes:
  - Integrated into programming language
  - Separate system service

Parameter Passing

- Local procedure parameter passing
  - Call-by-value
  - Call-by-reference: arrays, complex data structures
- Remote procedure calls simulate this through:
  - Stubs – proxies
  - Flattening – marshalling
- Related issue: global variables are not allowed in RPCs

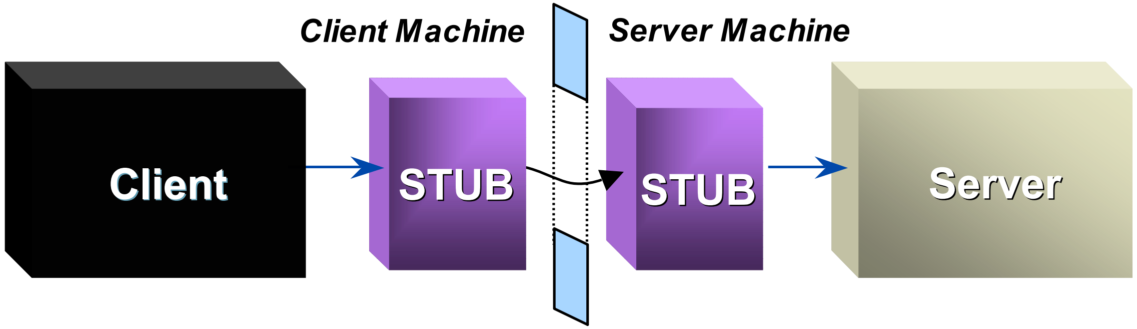
Client and Server

- Principle of RPC between a client and server program.



Stubs

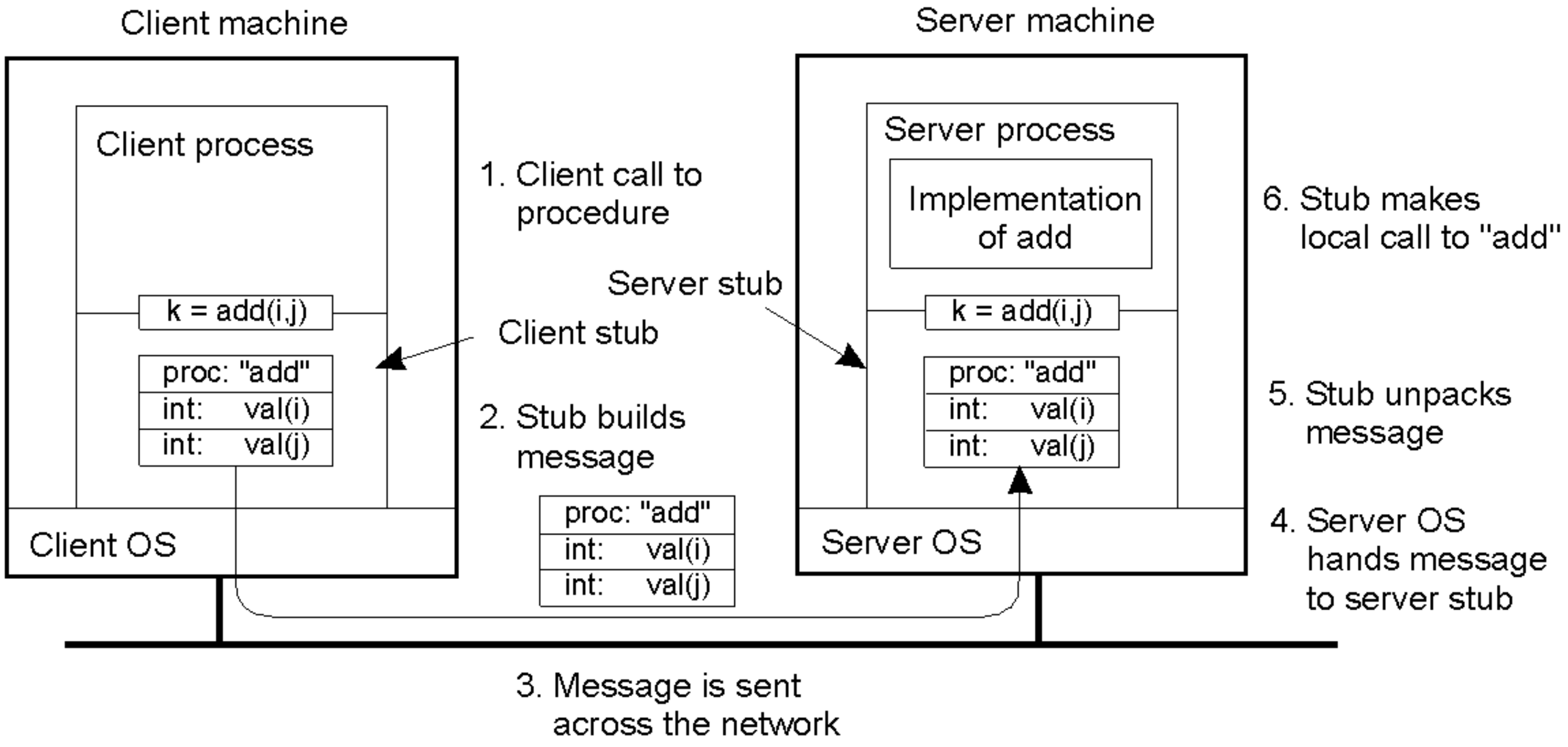
- Client makes procedure call (just like a local procedure call) to the client **stub**
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
- Packaging parameters is called ***marshalling***
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
- Simplifies programmer task



Steps in a RPC

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

Example of an RPC



Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a standard representation
  - Example: external data representation (XDR)
- Problem: how do we pass pointers?
  - If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
- What about data structures containing pointers?
  - Prohibit
  - Chase pointers over network
- Marshalling: transform parameters/results into a byte stream

Binding

- Problem: how does a client locate a server?
  - Use Bindings
- Server
  - Export server interface during initialization
  - Send name, version no, unique identifier, handle (address) to binder
- Client
  - First RPC: send message to binder to import server interface
  - Binder: check to see if server has exported interface
    - Return handle and unique identifier to client

Binding: Comments

- Exporting and importing incurs overheads
- Binder can be a bottleneck
  - Use multiple binders
- Binder can do load balancing

Failure Semantics

- **Client unable to locate server**: return error
- **Lost request messages**: simple timeout mechanisms
- **Lost replies**: timeout mechanisms
  - Make operation idempotent
  - Use sequence numbers, mark retransmissions
- **Server failures**: did failure occur before or after operation?
  - At least once semantics (SUNRPC)
  - At most once
  - No guarantee
  - Exactly once: desirable but difficult to achieve

Failure Semantics

- ***Client failure:*** what happens to the server computation?
  - Referred to as an *orphan*
  - *Extermination*: log at client stub and explicitly kill orphans
    - Overhead of maintaining disk logs
  - *Reincarnation*: Divide time into epochs between failures and delete computations from old epochs
  - *Gentle reincarnation*: upon a new epoch broadcast, try to locate owner first (delete only if no owner)
  - *Expiration*: give each RPC a fixed quantum *T*; explicitly request extensions
    - Periodic checks with client during long computations

Implementation Issues

- Choice of protocol [affects communication costs]
  - Use existing protocol (UDP) or design from scratch
  - Packet size restrictions
  - Reliability in case of multiple packet messages
  - Flow control
- Copying costs are dominant overheads
  - Need at least 2 copies per message
    - From client to NIC and from server NIC to server
  - As many as 7 copies
    - Stack in stub – message buffer in stub – kernel – NIC – medium – NIC – kernel – stub – server
  - Scatter-gather operations can reduce overheads

Case Study: SUNRPC

- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8912 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN's eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures

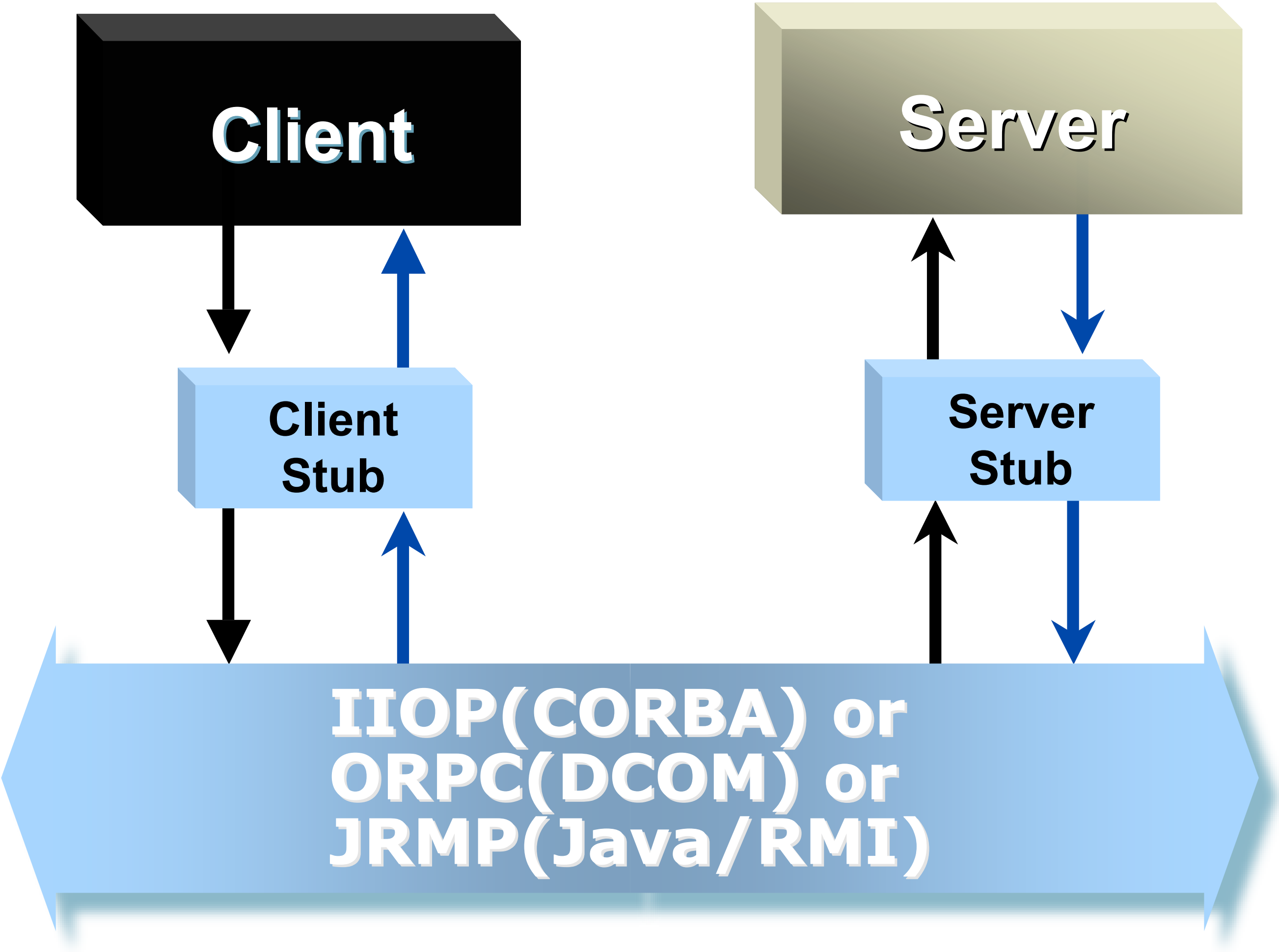
Where to go ?

What to know more on distributed systems ?

Canonical Problems

- Time ordering and clock synchronization
- Leader election
- Mutual exclusion
- Deadlock detection
- Causality
- Global state and termination detection
- Election algorithms
- Distributed synchronization and mutual exclusion
- Distributed transactions

More Independence



Literature

- **Distributed Systems**
  - Tannenbaum and Van Steen
  - Prentice Hall 2001
- **Distributed Systems - Concepts and Design**
  - Coulouris, Dollimore and Kindberg
  - Addison Wesley 2000