Distributed Systems Fundamentals

Distributed Systems

Misc. Course Details

- **A.Tannenbaum and Van Steen**, Distributed Systems Prentice Hall 2001
- **G. Coulouris**, **J. Dollimore** and **T. Kindburg**. *Distributed Systems, Concepts and Design*, Addison Wesley, 2001 (ISBN: 0201-61918-0)
- **J. Bacon**. *Concurrent Systems*, Addison Wesley, 1998 (ISBN: 0-201-17767-6) -- Chapters related to distributed systems

Translated to polish

Definition of a Distributed System

- A distributed system:
  - Multiple connected CPUs working together
  - A collection of independent computers that appears to its users as a single coherent system
- Examples: parallel machines, networked machines

Definition of a Distributed System

Tanenbaum & Van Renesse (1985)

"A distributed operating system is one that looks to its users like an ordinary centrialized operation system, but runs on multiple, independent CPUs. The key concept here is transparency, in other words, the use of multiple processors should be invisible (transparent) to the user. Another way of expressing the same idea is to say that the user views the system as a virtual uniprocessor, not as a collection of distinct machines."

Advantages and Disadvantages

- Advantages
  - Communication and resource sharing possible
  - Economics – price-performance ratio
  - Relibility, scalability
  - Potential for incremental growth
- Disadvantages
  - Distribution-aware PLs, OSs and applications
  - Network connectivity essential
  - Security and privacy

Transparency in a Distributed System

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource exists in several copies that have to be consistent
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.

Scalability Problems

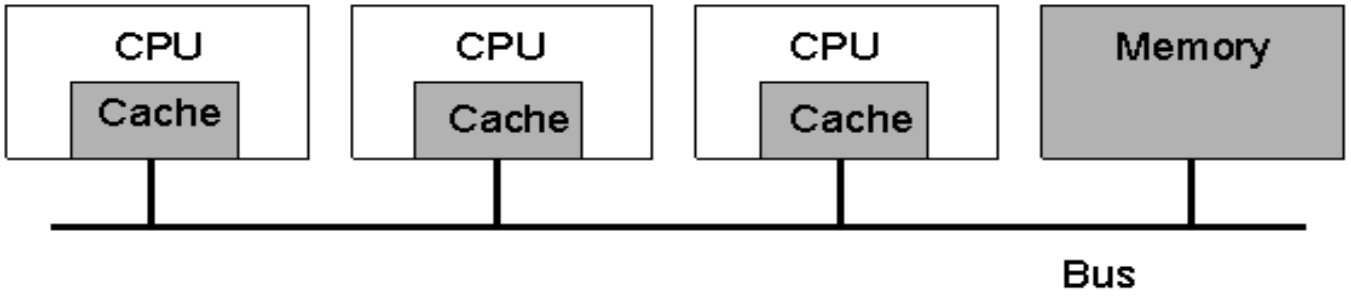
Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Examples of scalability limitations.

Hardware Concepts: Multiprocessors (1)

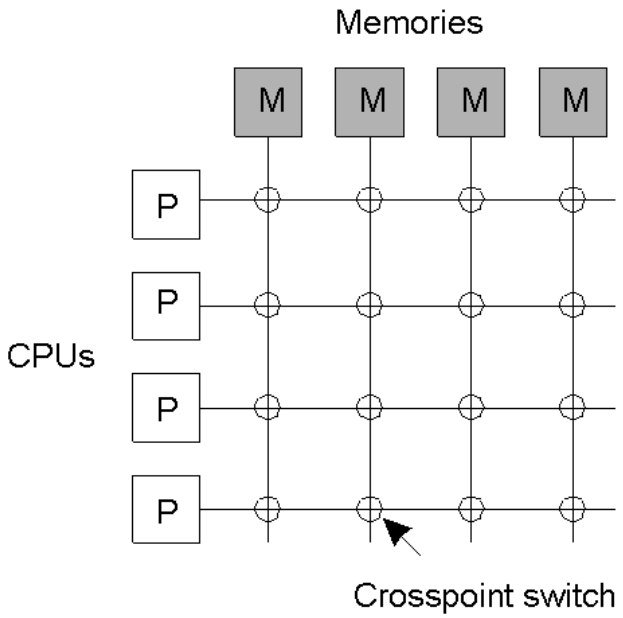
- Multiprocessor dimensions
  - Memory: could be shared or be private to each CPU
  - Interconnect: could be shared (bus-based) or switched

- A bus-based multiprocessor.

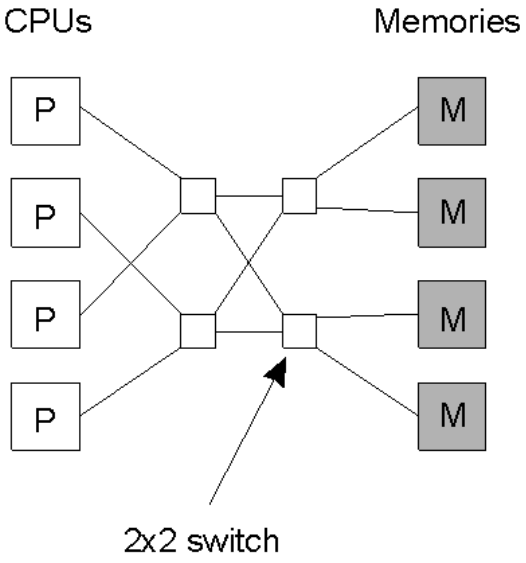


Multiprocessors (2)

a) A crossbar switch b) An omega switching network



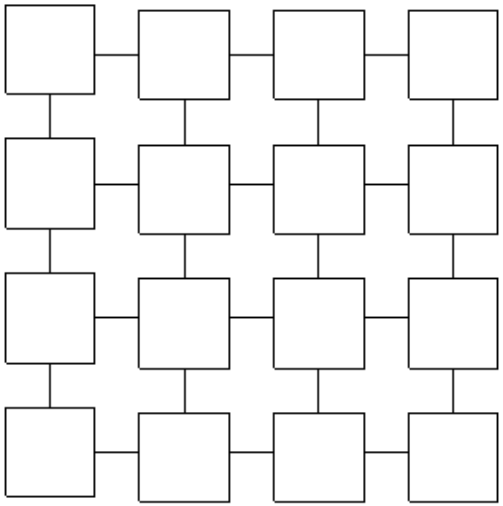
(a)



(b)

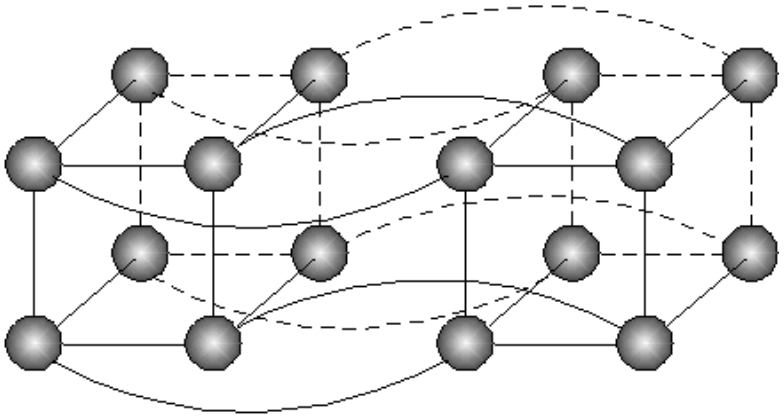
Homogeneous Multicomputer Systems

a) Grid



(a)

b) Hypercube



(b)

Uniprocessor Operating Systems

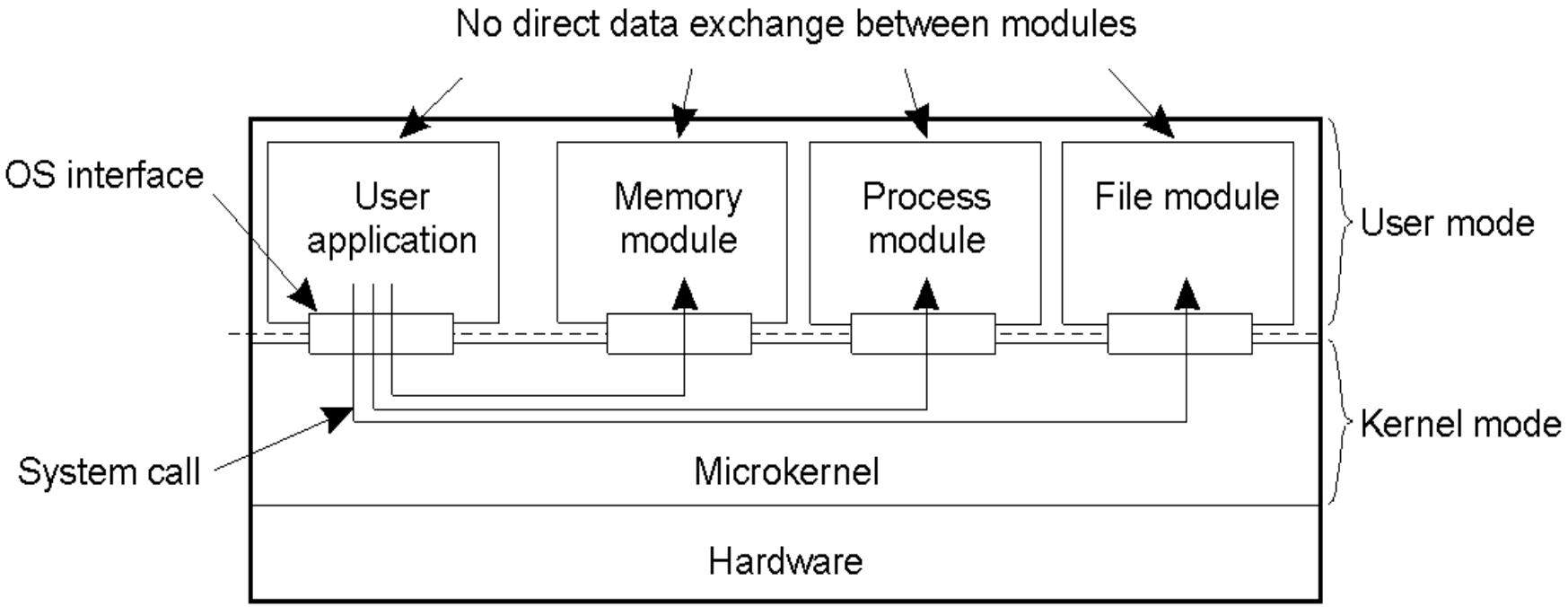
- An OS acts as a resource manager or an arbitrator
  - Manages CPU, I/O devices, memory
- OS provides a virtual interface that is easier to use than hardware

- 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

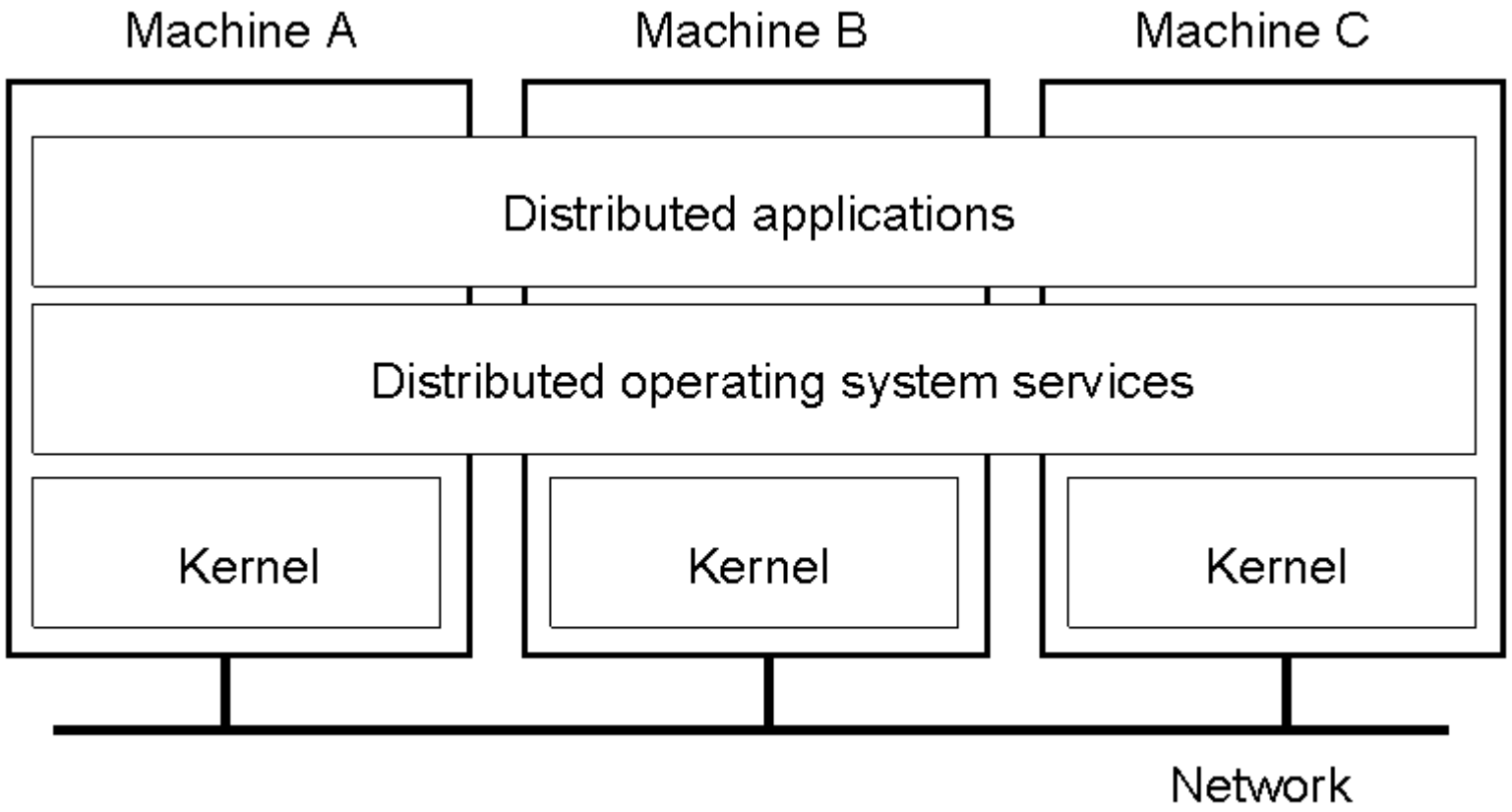
Types of Distributed OSs

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

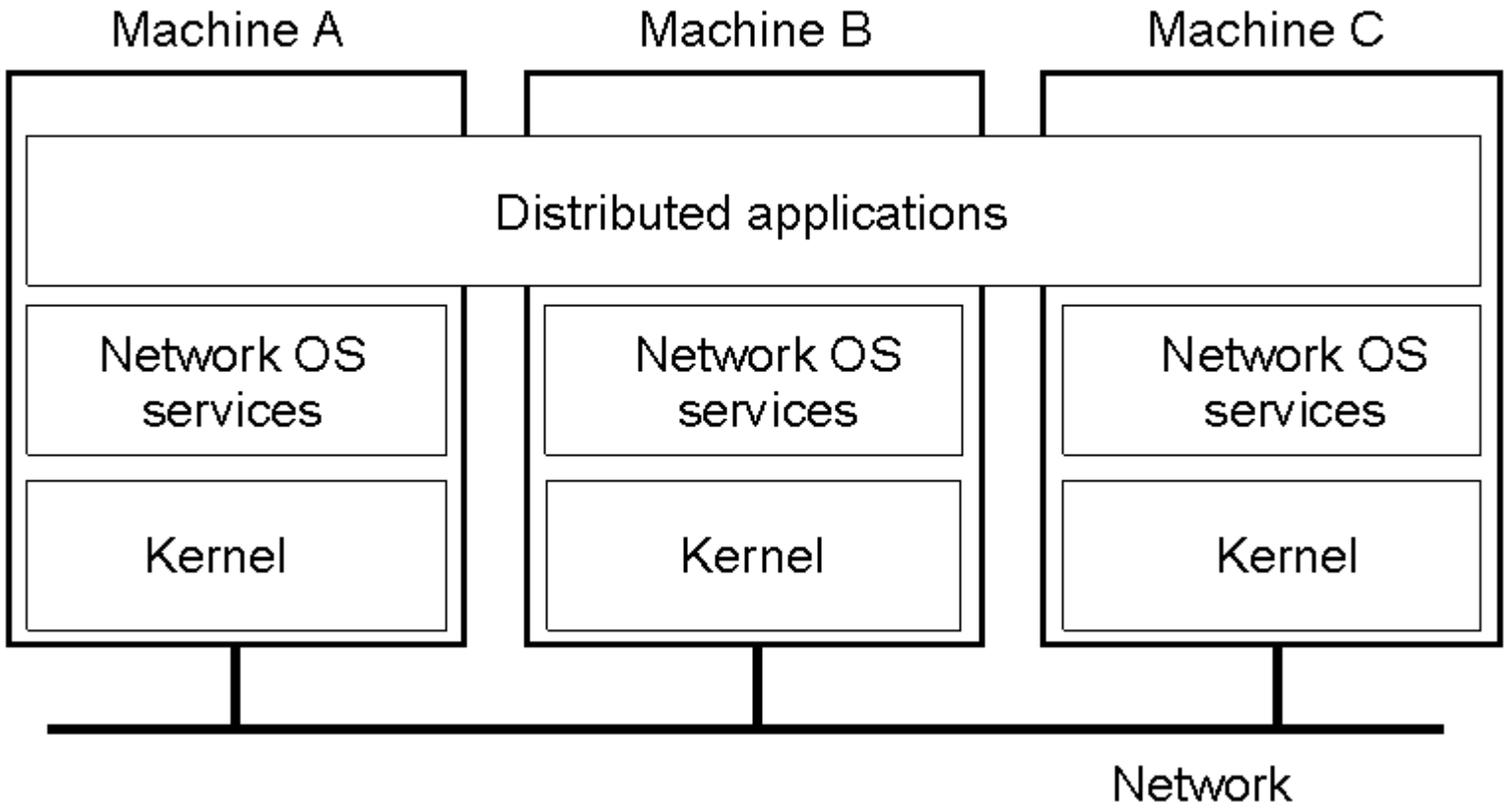
Multiprocessor Operating Systems

- Like a uniprocessor operating system
- Manages multiple CPUs transparently to the user
- Each processor has its own hardware cache
  - Maintain consistency of cached data

Multicomputer Operating Systems

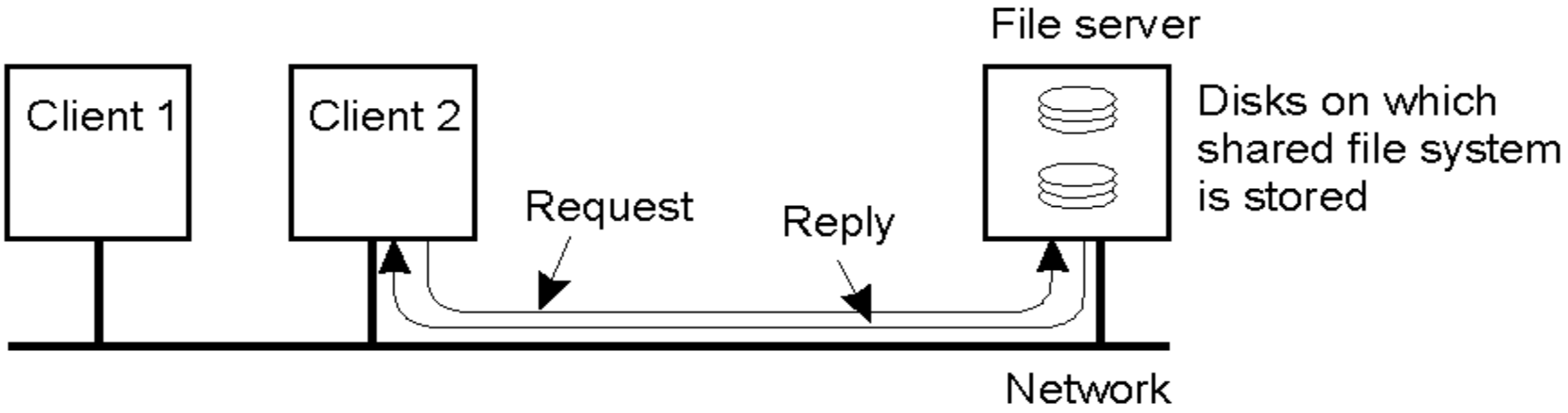


Network Operating System



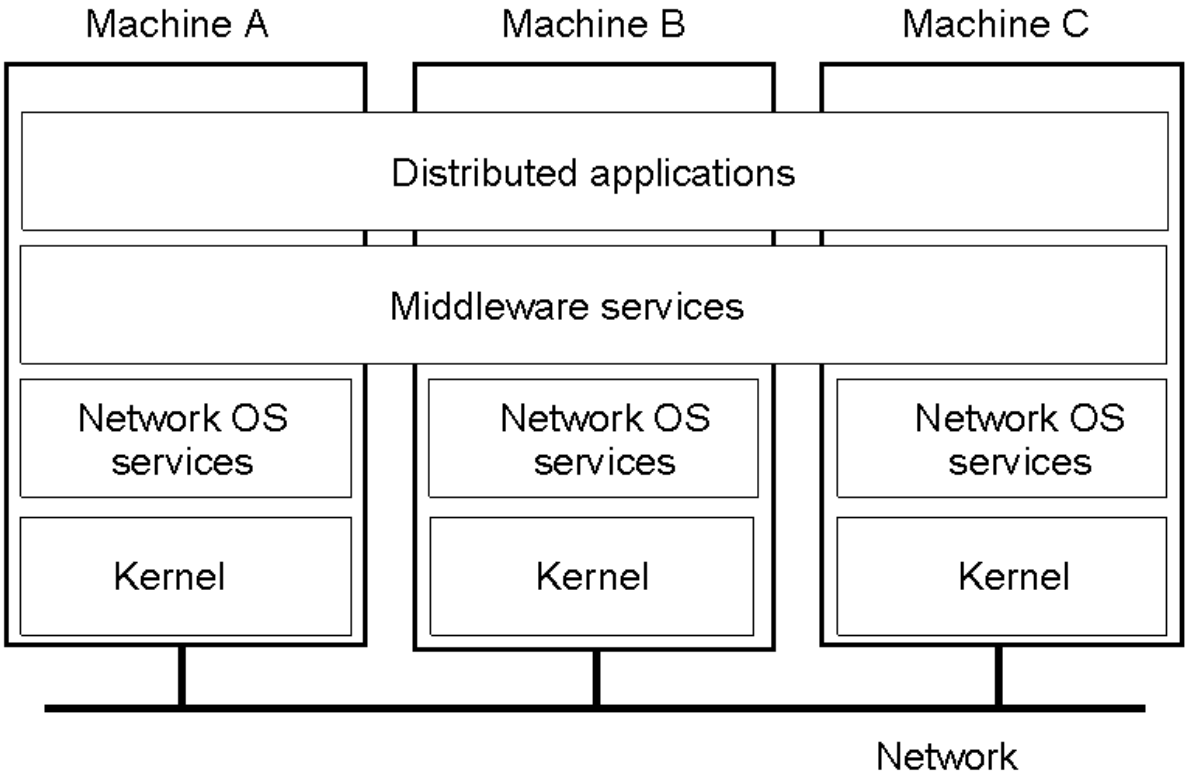
Network Operating System

- 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

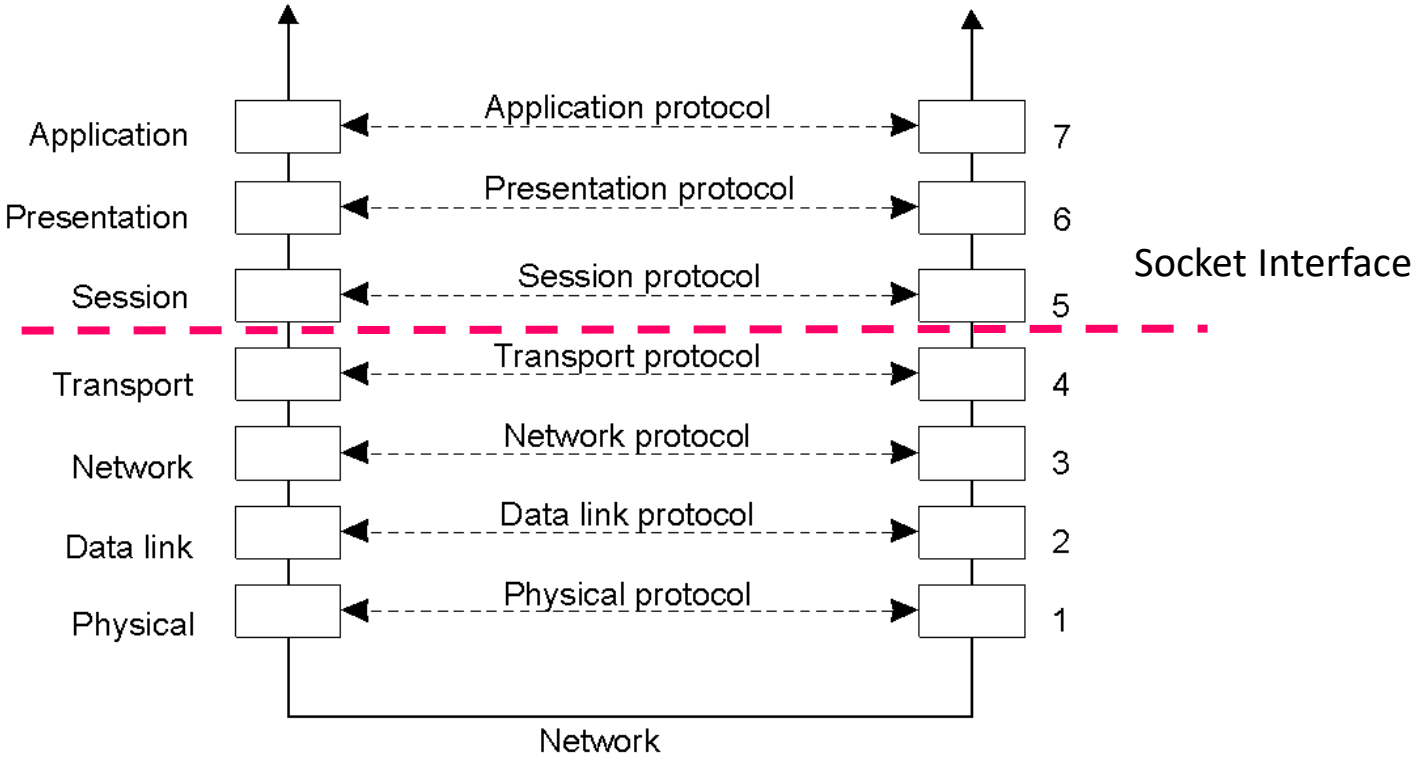
- Message-oriented Communication
- Remote Procedure Calls
  - Transparency but poor for passing references
- Remote Method Invocation
  - RMIs are essentially RPCs but specific to remote objects
  - System wide references passed as parameters
- Stream-oriented Communication

Communication Between Processes

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

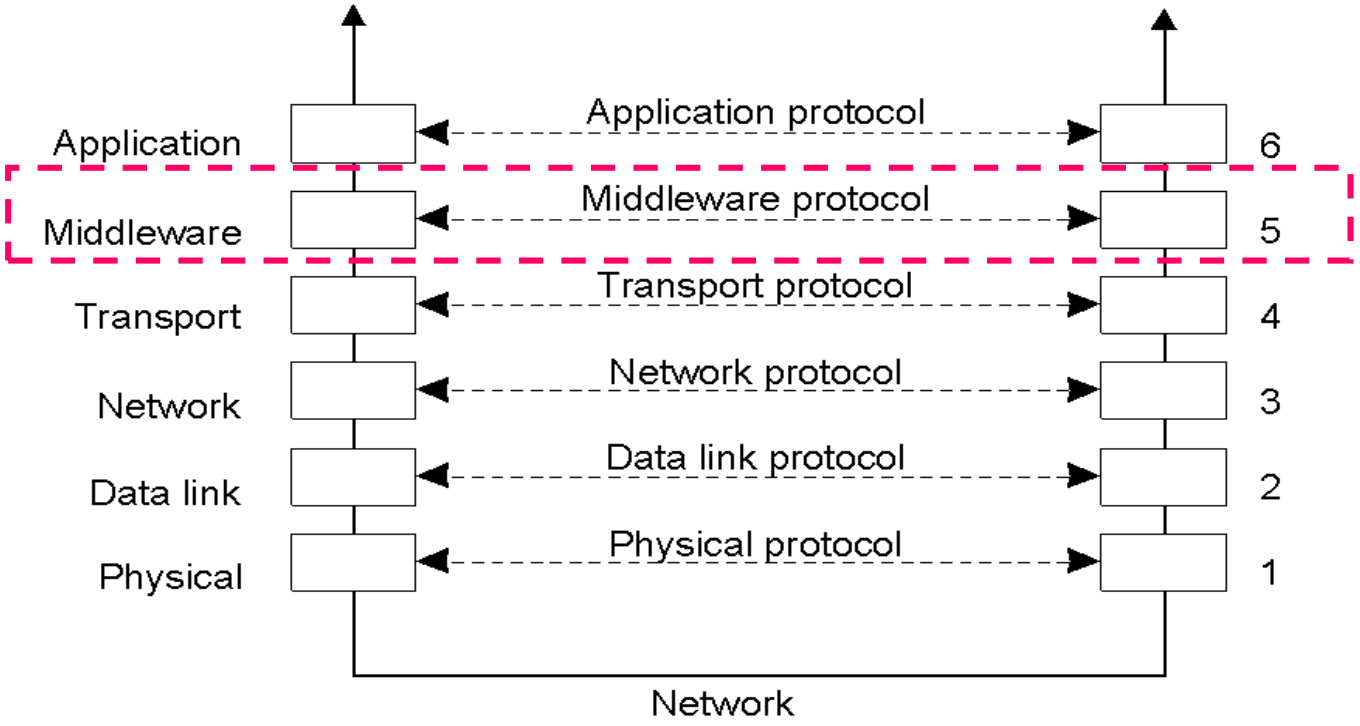
Communication Protocols

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



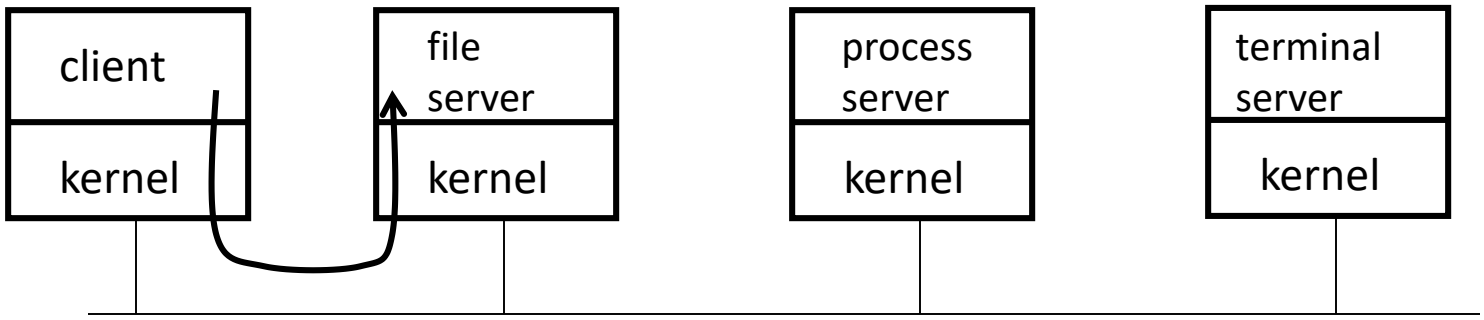
Middleware Protocols

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



Client-Server Communication Model

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

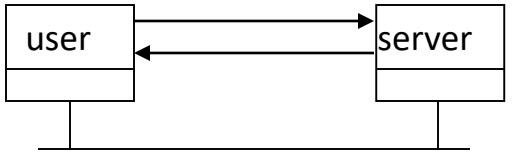


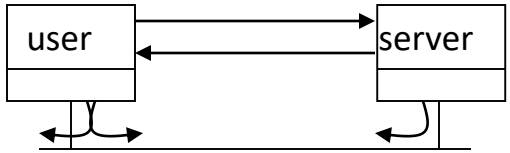
Issues in Client-Server Communication

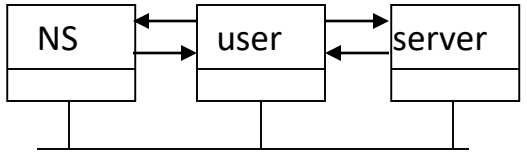
- Addressing
- Blocking versus non-blocking
- Buffered versus unbuffered
- Reliable versus unreliable
- Server architecture: concurrent versus sequential
- Scalability

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





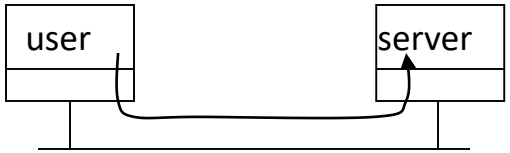


Blocking versus Non-blocking

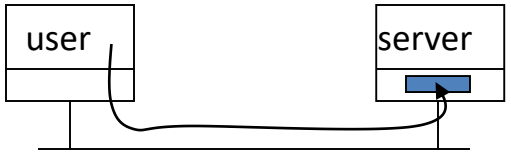
- Blocking communication (synchronous)
  - Send blocks until message is actually sent
  - Receive blocks until message is actually received
- Non-blocking communication (asynchronous)
  - Send returns immediately
  - Receive does not block either
- Examples:

Buffering Issues

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

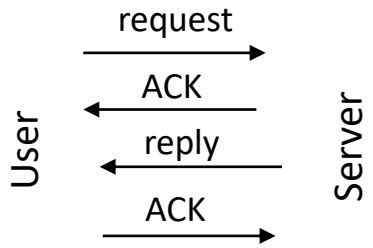


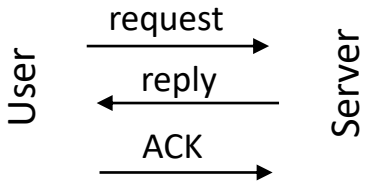
- Buffered communication
  - Client send to a mailbox
  - Server receives from a mailbox



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, e.g. TCP





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)
- Thus servers could be
  - Pure-sequential, event-based, thread-based, process-based
- Discussion: which architecture is most efficient?

Scalability

- *Question:*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: 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

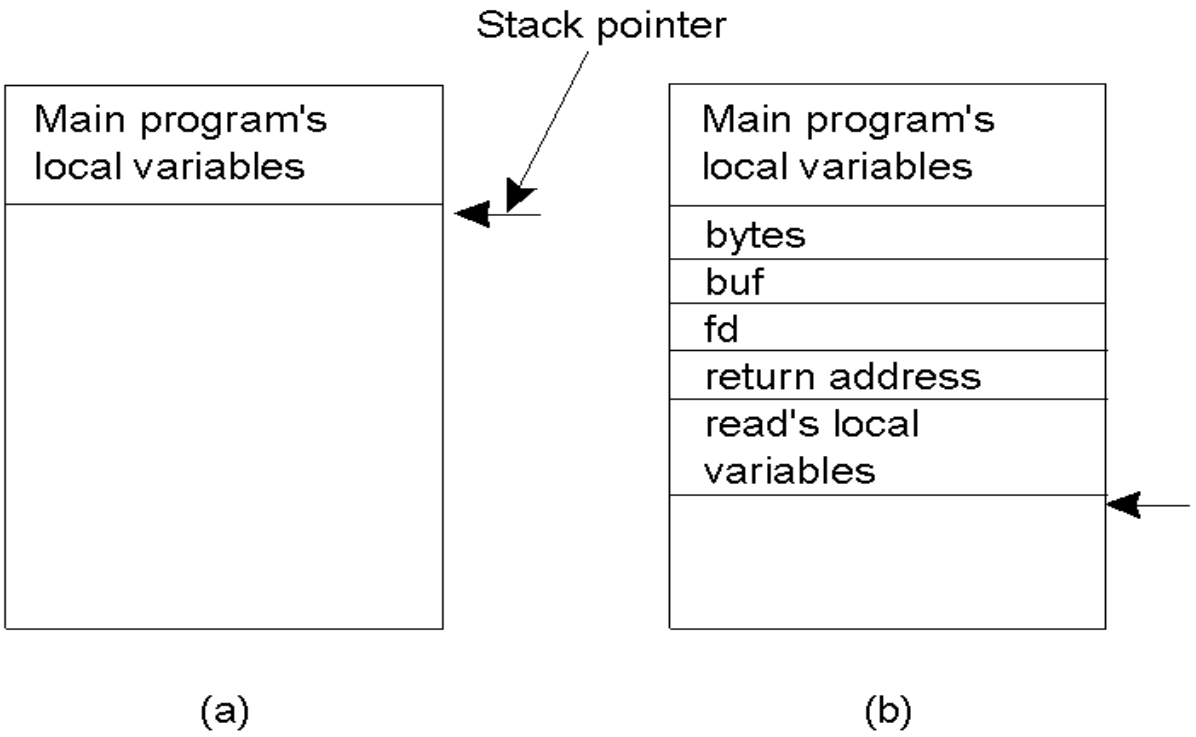
Remote Procedure Calls

- 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 prog, language and separate

Conventional Procedure Call

a) Parameter passing in a local procedure call: the stack before the call to read

b) The stack while the called procedure is active

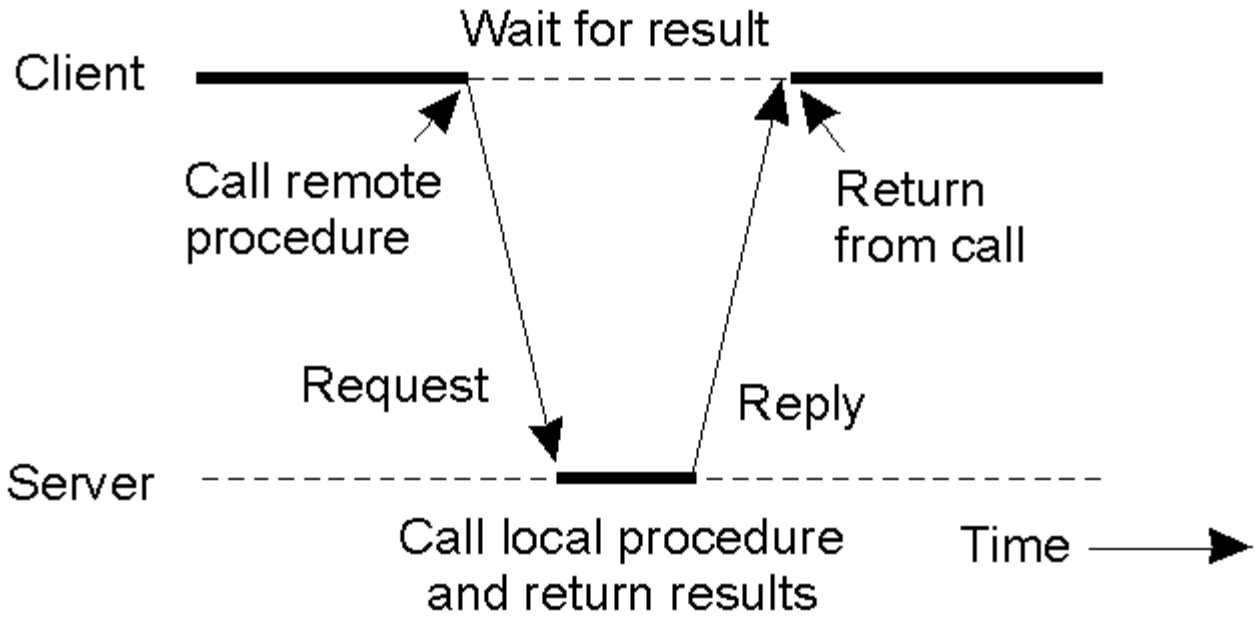


Parameter Passing

- 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 Stubs

- Principle of RPC between a client and server program.



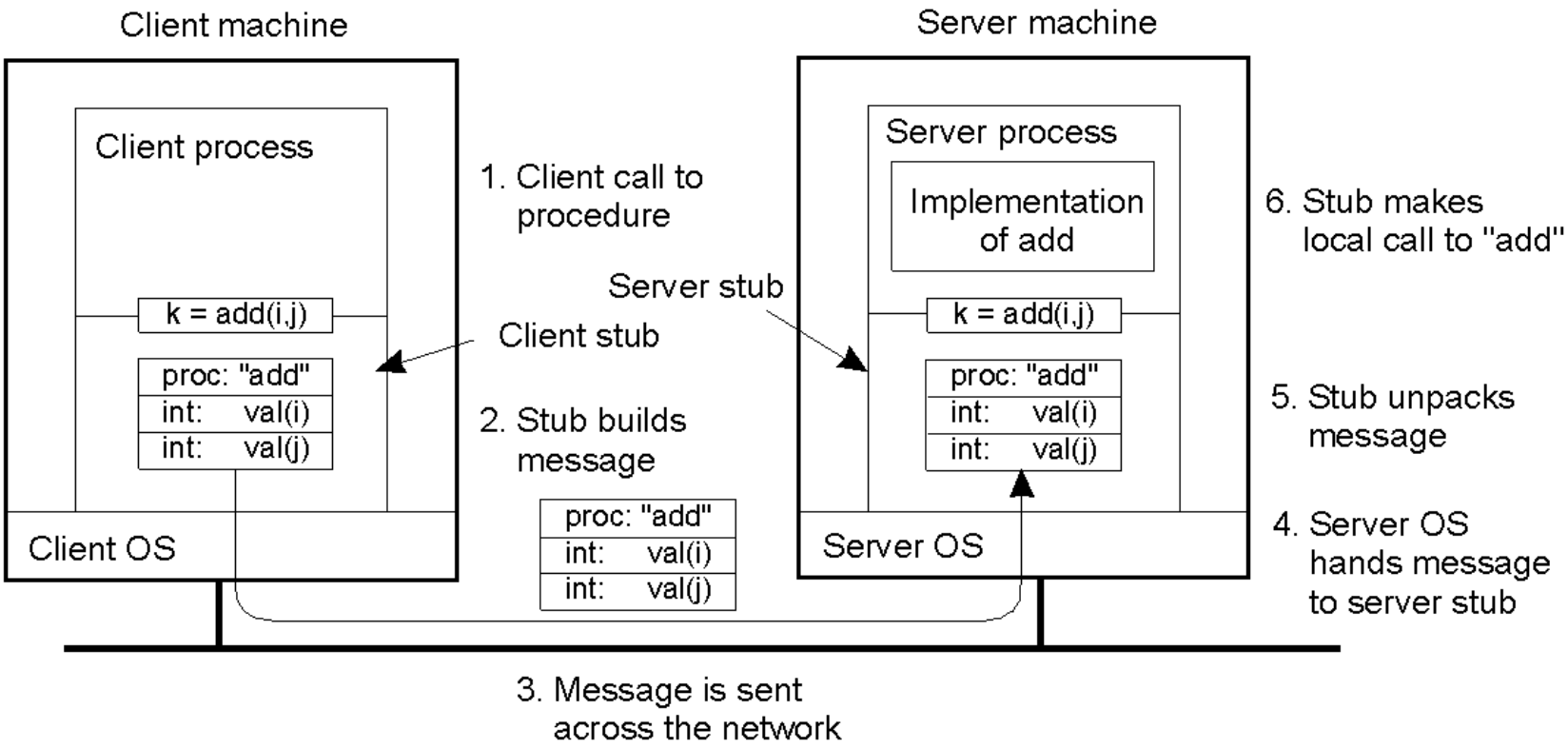
Stubs

- 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 is called *marshalling*
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
  - Simplifies programmer task

Steps of a Remote Procedure Call

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

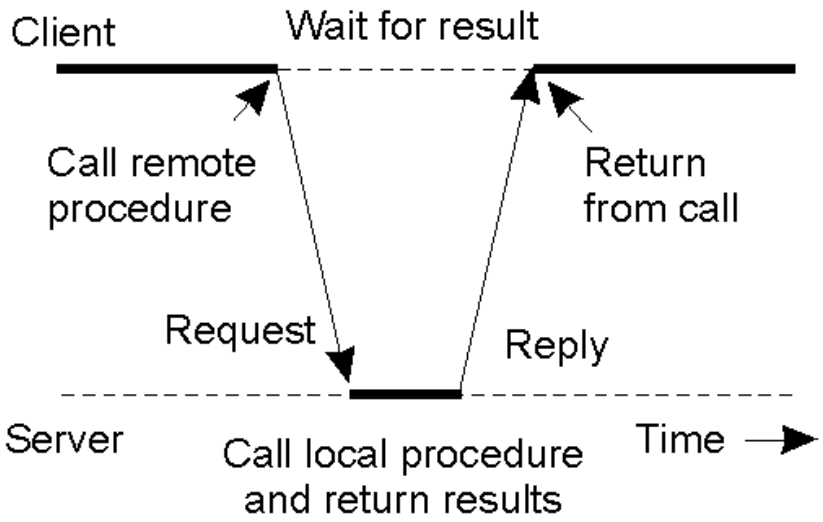
Implementation Issues

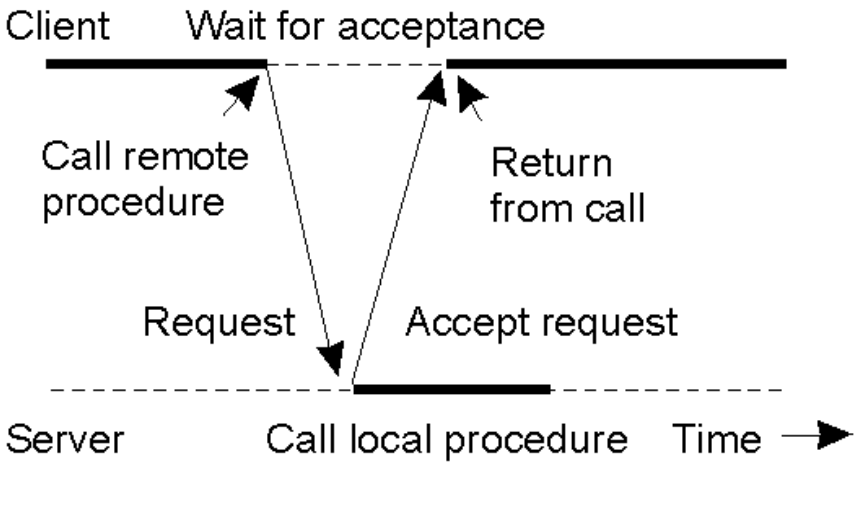
- Choice of protocol [affects communication costs]
  - Use existing protocol (UDP) or design from scratch
  - Packet size restrictions
  - Reliability in case of multiple packet messages
  - Flow control
- Copying costs are dominant overheads
  - Need at least 2 copies per message
    - From client to 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

Other RPC Models

- Asynchronous RPC
  - Request-reply behavior often not needed
  - Server can reply as soon as request is received and execute procedure later
- Deferred-synchronous RPC
  - Use two asynchronous RPCs
  - Client needs a reply but can’t wait for it; server sends reply via another asynchronous RPC
- One-way RPC
  - Client does not even wait for an ACK from the server
  - Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server).

Asynchronous RPC



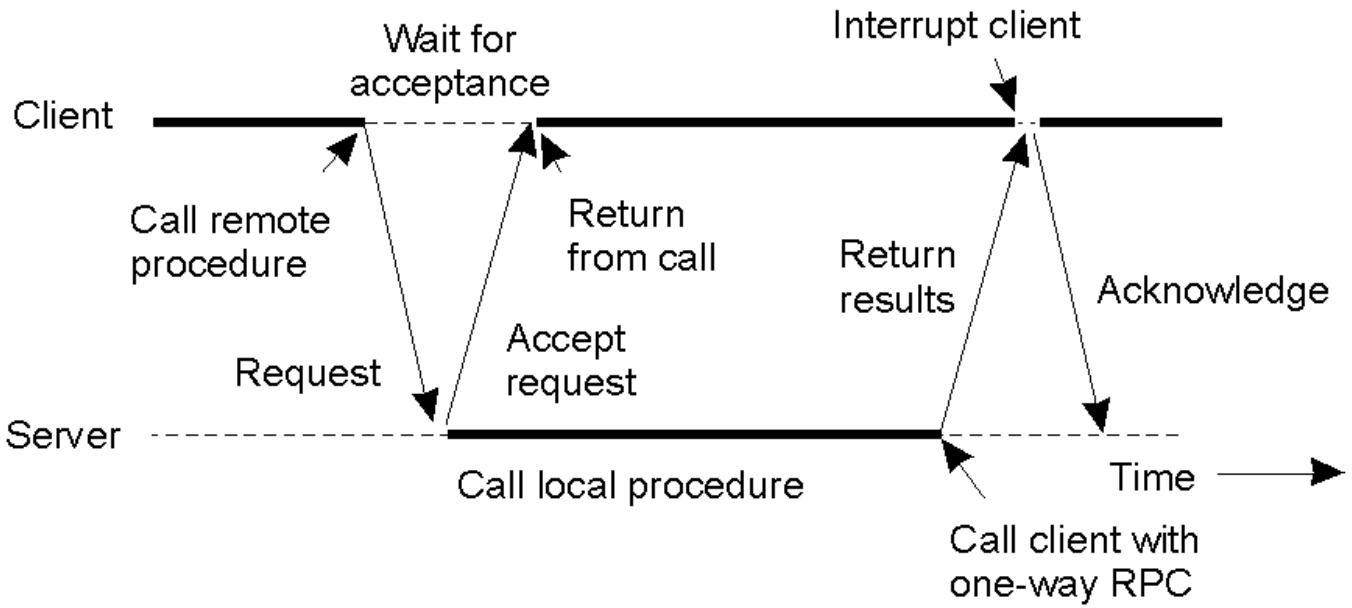


a) The interconnection between client and server in a traditional RPC

b) The interaction using asynchronous RPC

Deferred Synchronous RPC

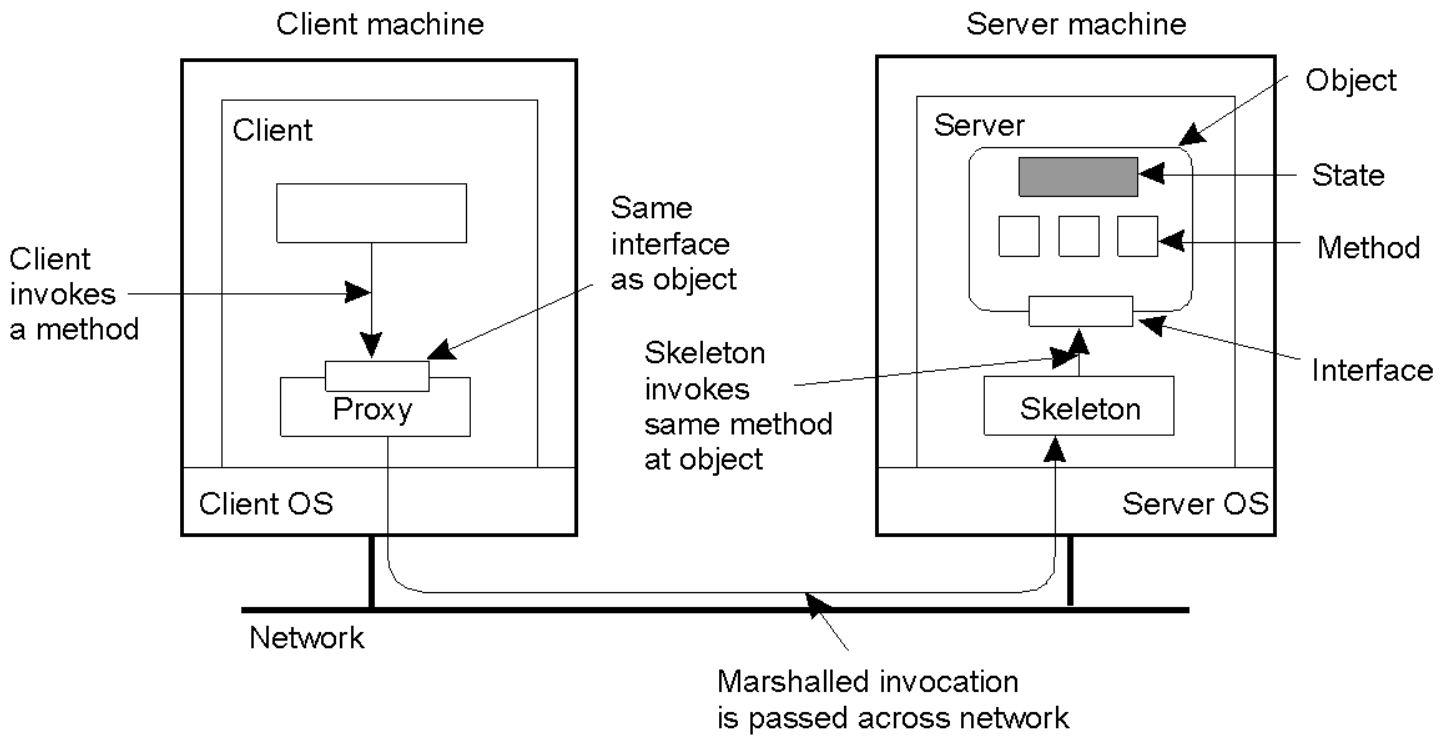
- A client and server interacting through two asynchronous RPCs



Remote Method Invocation (RMI)

- RPCs applied to *objects,* i.e., instances of a class
  - *Class:* object-oriented abstraction; module with data and operations
  - Separation between interface and implementation
  - Interface resides on one machine, implementation on another
- RMIs support system-wide object references
  - Parameters can be object references

Distributed Objects



- When a client binds to a distributed object, load the interface ("proxy") into client address space
  - Proxy analogous to stubs
- Server stub is referred to as a skeleton

Proxies and Skeletons

- Proxy: client stub
  - Maintains server ID, endpoint, object ID
  - Sets up and tears down connection with the server
  - [Java:] does serialization of local object parameters
  - In practice, can be downloaded/constructed on the fly (why can’t this be done for RPCs in general?)
- Skeleton: server stub
  - Does deserialization and passes parameters to server and sends result to proxy

Binding a Client to an Object

```
Distr_object* obj_ref;          //Declare a systemwide object reference
obj_ref = …;                    // Initialize the reference to a distributed object
obj_ref-> do_something();       // Implicitly bind and invoke a method
```

(a)

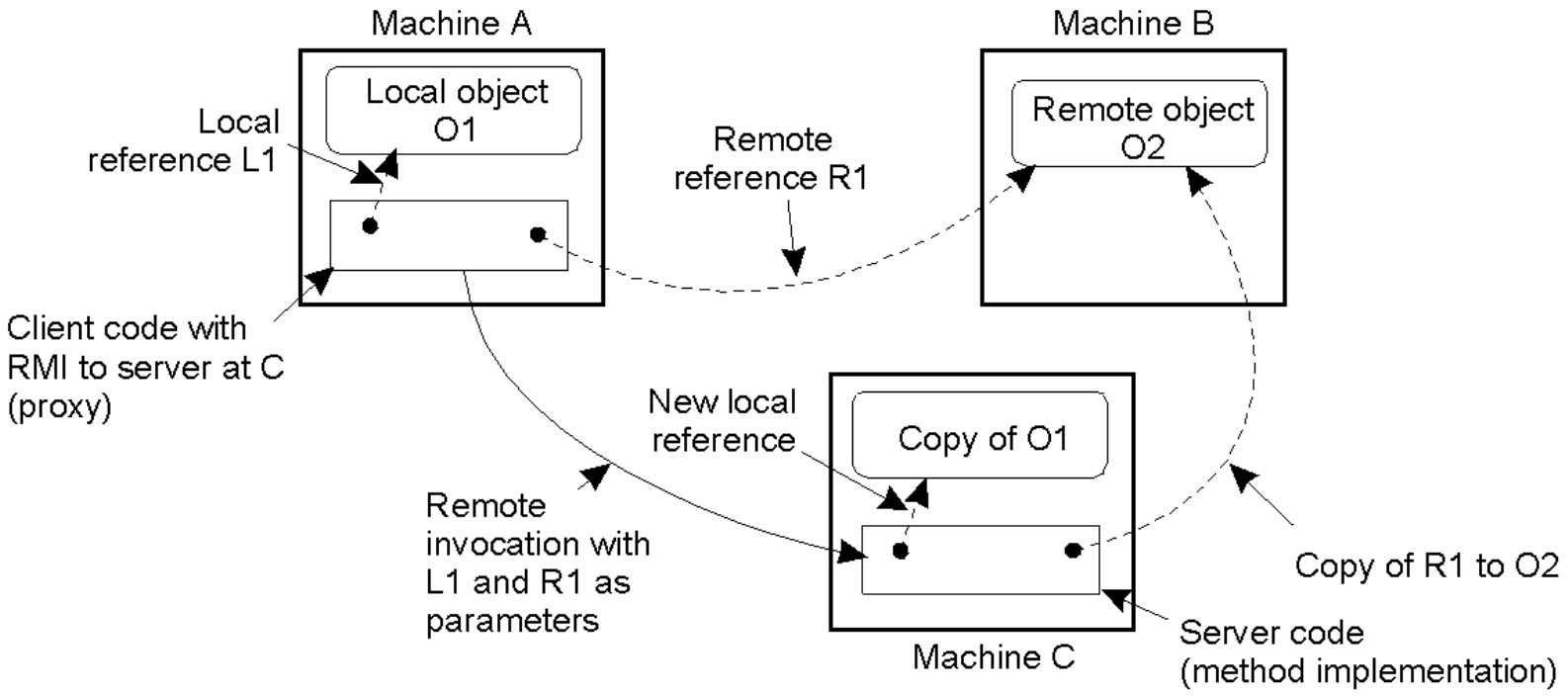
```
Distr_object obj_ref;           //Declare a systemwide object reference
Local_object* obj_ptr;          //Declare a pointer to local objects
obj_ref = …;                    //Initialize the reference to a distributed object
obj_ptr = bind(obj_ref);        //Explicitly bind and obtain a pointer to the local proxy
obj_ptr -> do_something();      //Invoke a method on the local proxy
```

(b)

a) (a) Example with implicit binding using only global references
b) (b) Example with explicit binding using global and local references

Parameter Passing

- Less restrictive than RPCs.
  - Supports system-wide object references
  - [Java] pass local objects by value, pass remote objects by reference



Java RMI

- Server
  - Defines interface and implements interface methods
  - Server program
    - Creates server object and registers object with “remote object” registry
- Client
  - Looks up server in remote object registry
  - Uses normal method call syntax for remote methods
- Java tools
  - Rmiregistry: server-side name server
  - Rmic: uses server interface to create client and server stubs