UNIT II

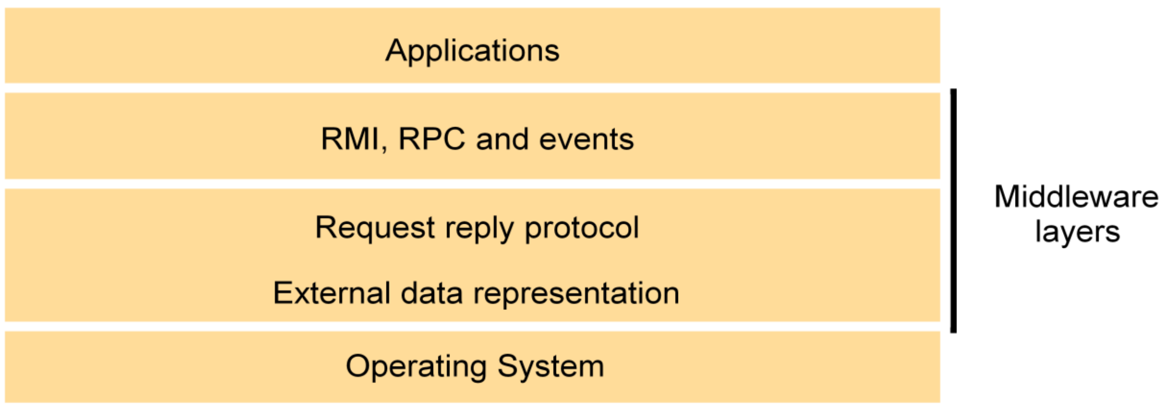
Distributed Objects and Remote Invocation-Introduction-Communication between distributed objects-Remote procedure calls-Events and notifications-Case study: Java RMI.
Operating System Support-Introduction-OS layer-Protection-Processes and threads-Communication and invocation OS architecture.

Introduction
Programming Models for Distributed Communications

– Remote Procedure Calls – Client programs call procedures in server programs
– Remote Method Invocation – Objects invoke methods of objects on distributed hosts
– Event-based Programming Model – Objects receive notice of events in other objects in which they have interest

Middleware
• Middleware: software that allows a level of programming beyond processes and message passing
– Uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocation and events
– Supports location transparency
– Usually uses an interface definition language (IDL) to define interfaces

-



Interfaces in Programming Languages
– Current PL allow programs to be developed as a set of modules that communicate with each other. Permitted interact ions between modules are defined by interfaces
– A specified interface can be implemented by different modules without the need to modify other modules using the interface

• Interfaces in Distributed Systems

– When modules are in different processes or on different hosts there are limitations
on the interactions that can occur. Only actions with parameters that are fully specified and understood can communicate effectively to request or provide services to modules in another process
– A service interface allows a client to request and a server to provide particular services
– A remote interface allows objects to be passed as arguments to and results from distributed modules
• Object Interfaces
– An interface defines the signatures of a set of methods, including arguments, argument types, return values and exceptions. Implementation details are not included in an interface.
A class may implement an interface by specifying behavior for each method in the interface. Interfaces do not have constructors.

Communication between Distributed Objects

The Object Model
Five Parts of the Object Model
– An object-oriented program consists of a collection of interacting objects
• Objects consist of a set of data and a set of methods
• In DS, object's data should be accessible only via methods

Object References
– Objects are accessed by object references
– Object references can be assigned to variables, passed as arguments, and returned as the result of a method
– Can also specify a method to be invoked on that object
Interfaces

– Provide a definition of the signatures of a set of methods without specifying their implementation
– Define types that can be used to declare the type of variables or of the parameters and return values of methods

Actions
– Objects invoke methods in other objects
– An invocation can include additional information as arguments to perform the behavior specified by the method
– Effects of invoking a method
1. The state of the receiving object may be changed
2. A new object may be instantiated
3. Further invocations on methods in other objects may occur
4. An exception may be generated if there is a problem encountered

Exceptions
– Provide a clean way to deal with unexpected events or errors
– A block of code can be defined to throw an exception when errors or unexpected conditions occur. Then control passes to code that catches the exception

Garbage Collection
– Provide a means of freeing the space that is no longer needed
– Java (automatic), C++ (user supplied)

Distributed Objects
• Physical distribution of objects into different processes or computers in a distributed system
– Object state consists of the values of its instance variables
– Object methods invoked by remote method invocation (RMI)
– Object encapsulation: object state accessed only by the object methods

Usually adopt the client-server architecture

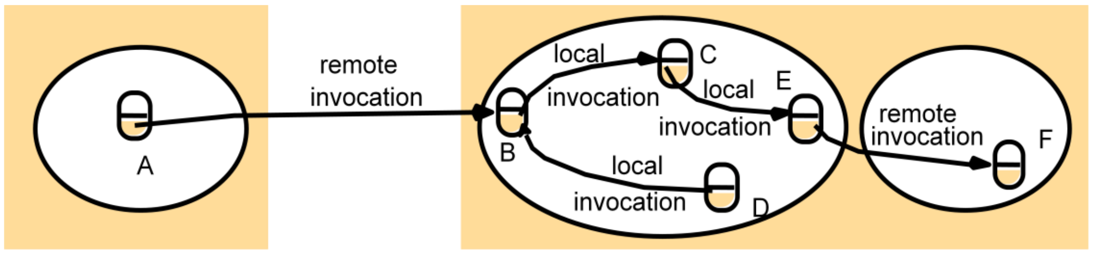
– Basic model
  • Objects are managed by servers and
  • Their clients invoke their methods using RMI
– Steps
1. The client sends the RMI request in a message to the server
2. The server executes the invoked method of the object
3. The server returns the result to the client in another message

– Other models
• Chains of related invocations: objects in servers may become clients of objects in other servers
• Object replication: objects can be replicated for fault tolerance and performance
• Object migration: objects can be migrated to enhancing performance and availability

The Distributed Object Model

Two fundamental concepts: Remote Object Reference and Remote Interface
– Each process contains objects, some of which can receive remote invocations are called remote objects (B, F), others only local invocations
– Objects need to know the remote object reference of an object in another process in order to invoke its methods, called remote method invocations
– Every remote object has a remote interface that specifies which of its methods can be invoked remotely

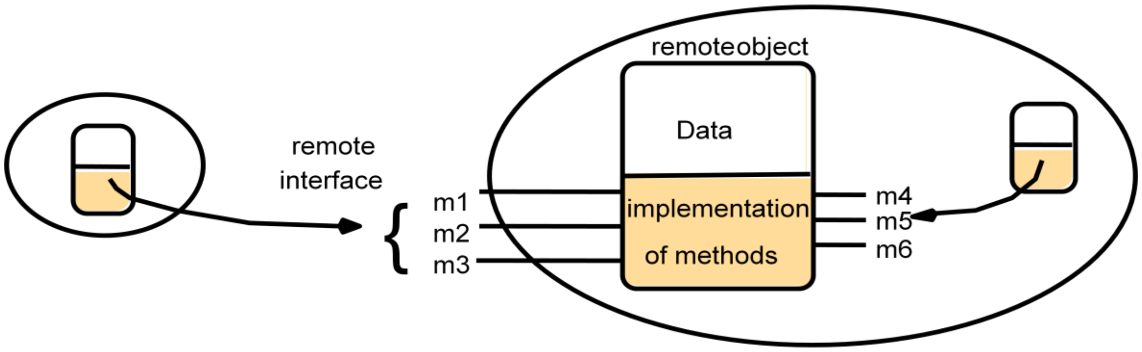
Remote and local method invocations



Five Parts of Distributed Object Model

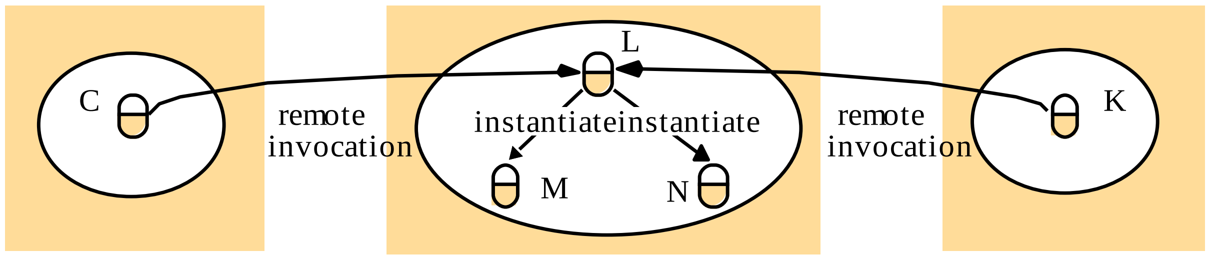
• Remote Object References
– accessing the remote object
– identifier throughout a distributed system
– can be passed as arguments
• Remote Interfaces
– specifying which methods can be invoked remotely
– name, arguments, return type
– Interface Definition Language (IDL) used for defining remote interface

Remote Object and Its remote Interface



• Actions
– An action initiated by a method invocation may result in further invocations on methods in other objects located indifference processes or computers
– Remote invocations could lead to the instantiation of new objects, ie. objects M and N of following figure.

-



• Exceptions
– More kinds of exceptions: i.e. timeout exception
- RMI should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked
• Garbage Collection
- Distributed garbage collections is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting

Design Issues for RMI

• Two design issues that arise in extension of local method invocation for RMI
  – The choice of invocation semantics

• Although local invocations are executed exactly once, this cannot always be the case for RMI due to transmission error
  – Either request or reply message may be lost
  – Either server or client may be crashed
  – The level of transparency

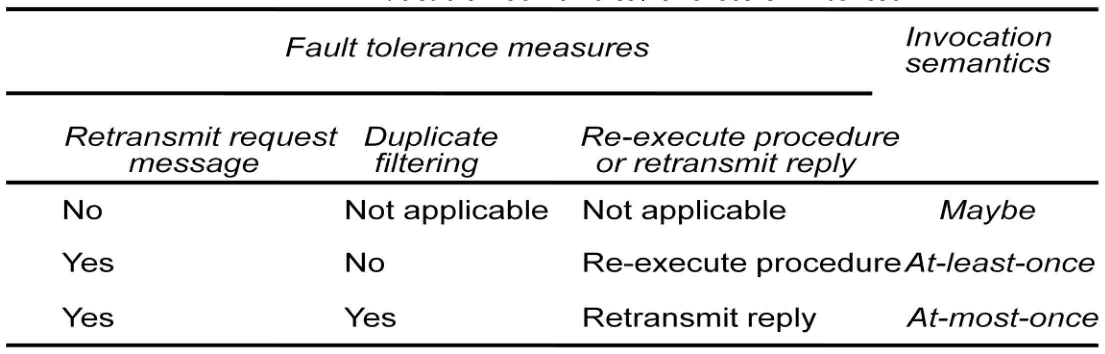
• Make remote invocation as much like local invocation as possible

RMI Design Issues: Invocation Semantics

• Error handling for delivery guarantees
  – Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed
  – Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server
  – Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations

• Choices of invocation semantics
  – Maybe: the method executed once or not at all (no retry nor retransmit)
  – At-least-once: the method executed at least once
  – At-most-once: the method executed exactly once

Invocation semantics: choices of interest



Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

RMI Design Issues: Transparency

• Transparent remote invocation: like a local call
  – marshalling/unmarshalling
  – locating remote objects
  – accessing/syntax

• Differences between local and remote invocations

-

– latency: a remote invocation is usually several order of magnitude greater than that of a local one
– availability: remote invocation is more likely to fail
– errors/exceptions: failure of the network? server? hard to tell

• syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)

– consistency on the remote machine:

• Argus: incomplete transactions, abort, restore states [as if the call was never made]

Implementation of RMI

•Communication module

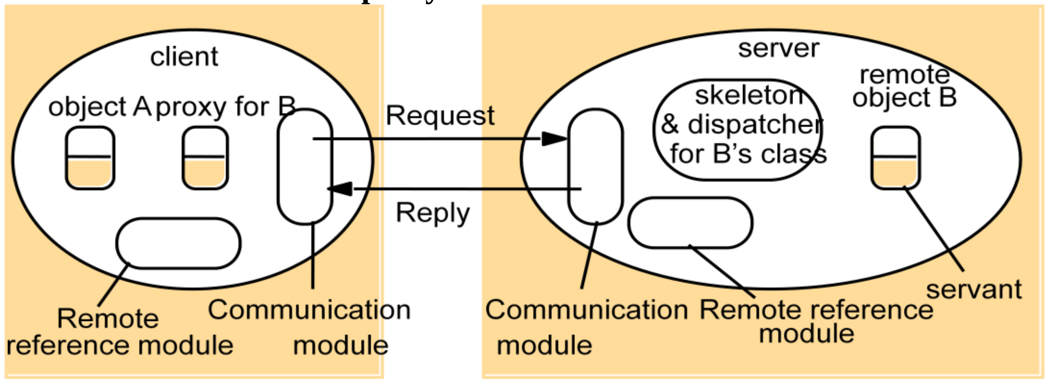
– Two cooperating communication modules carry out the request-reply protocols: message type, request ID, remote object reference

• Transmit request and reply messages between client and server
• Implement specific invocation semantics

– The communication module in the server

• selects the dispatcher for the class of the object to be invoked,
• passes on local reference from remote reference module,
• returns request

The role of proxy and skeleton in remote method invocation



• **Remote reference module**
– Responsible for translating between local and remote object references and for creating remote object references
– remote object table: records the correspondence between local and remote object references

• remote objects held by the process (B on server)
• local proxy (B on client)

– When a remote object is to be passed for the first time, the module is asked to create a remote object reference, which it adds to its table

• **Servant**
– An instance of a class which provides the body of a remote object
– handles the remote requests
•**RMI software**

– Proxy: behaves like a local object, but represents the remote object
– Dispatcher: look at the methodID and call the corresponding method in the skeleton
– Skeleton: implements the method
Generated automatically by an interface compiler

Implementation Alternatives of RMI

• Dynamic invocation
– Proxies are static—interface complied into client code
– Dynamic—interface available during run time
• Generic invocation; more info in "Interface Repository" (COBRA)
• Dynamic loading of classes (Java RMI)

•Binder
– A separate service to locate service/object by name through table mapping for names and remote object references

• Activation of remote objects
– Motivation: many server objects not necessarily in use all of the time
• Servers can be started whenever they are needed by clients, similar to inetd
– Object status: active or passive
• active: available for invocation in a running process
• passive: not running, state is stored and methods are pending
– Activation of objects:
• creating an active object from the corresponding passive object by creating a new instance of its class
• initializing its instance variables from the stored state
– Responsibilities of activator
• Register passive objects that are available for activation
• Start named server processes and activate remote objects in them
• Keep track of the locations of the servers for remote objects that it has already activated

• Persistent object stores
– An object that is guaranteed to live between activations of processes is called a persistent object
– Persistent object store: managing the persistent objects
• stored in marshaled from on disk for retrieval
• saved those that were modified
– Deciding whether an object is persistent or not:
• persistent root: any descendent objects are persistent (persistent Java, PerDiS)
• some classes are declared persistent (Arjuna system)
• Object location
– specifying a location: ip address, port #, ...
– location service for migratable objects
• Map remote object references to their probable current locations

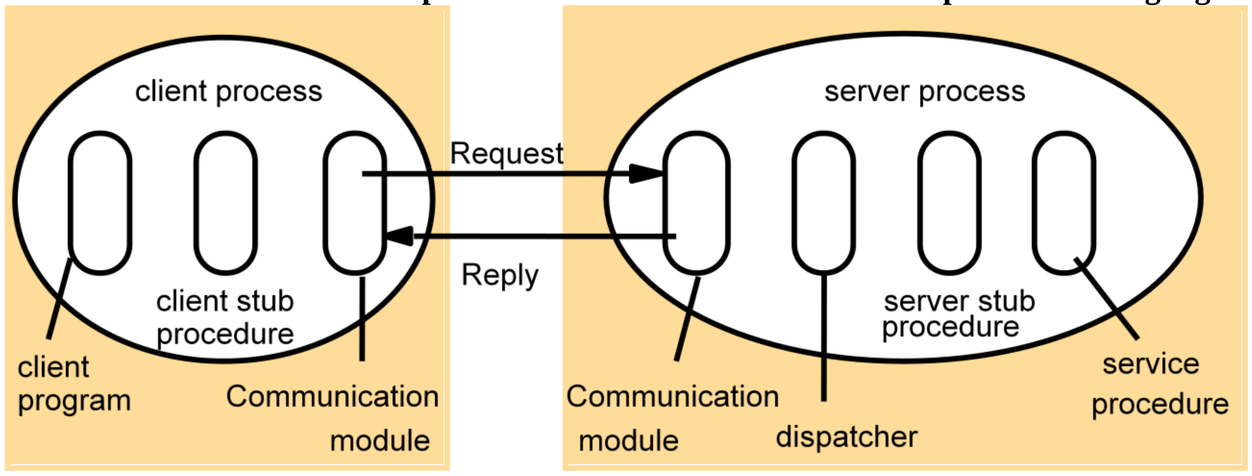
-

• Cache/broadcast scheme (similar to ARP)
– Cache locations
– If not in cache, broadcast to find it
• Improvement: forwarding (similar to mobile IP)

Distributed Garbage Collection
• Aim: ensure that an object
– continues to exist if a local or remote reference to it is still held anywhere
– be collected as soon as no object any longer holds a reference to it
• General approach: reference count
• Java's approach
– the server of an object (B) keeps track of proxies
– when a proxy is created for a remote object
• addRef(B) tells the server to add an entry
– when the local host's garbage collector removes the proxy
• removeRef(B) tells the server to remove the entry
– when no entries for object B, the object on server is deallocated

Remote Procedure Call
• client: "stub" instead of "proxy" (same function, different names)
– local call, marshal arguments, communicate the request
•server:
– dispatcher
– "stub": unmarshal arguments, communicate the results back

Role of client and server stub procedures in RPC in the context of a procedural language



Case Study: Sun RPC
•Sun RPC: client-server in the SUN NFS (network file system)
– UDP or TCP; in other unix OS as well
– Also called ONC (Open Network Computing) RPC
•Interface Definition Language (IDL)
– initially XDR is for data representation, extended to be IDL
– less modern than CORBA IDL and Java

• program numbers instead of interface names
• procedure numbers instead of procedure names
• single input parameter (structs)
– rpcgen: compiler for XDR
• client stub; server main procedure, dispatcher, and server stub
• XDR marshalling, unmarshaling

•Binding (registry) via a local binder - portmapper

– Server registers its program/version/port numbers with portmapper

– Client contacts the portmapper at a fixed port with program/version numbers to get the server port

– Different instances of the same service can be run on different computers at different ports

•Authentication

– request and reply have additional fields

– unix style (uid, gid), shared key for signing, Kerberos

Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};
```

```
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
  version VERSION {
    void WRITE(writeargs)=1;      1
    Data READ(readargs)=2;        2
  }=2;
} = 9999;
```

Events and Notifications

•Idea behind the use of events

– One object can react to a change occurring in another object

•Events

– Notifications of events: objects that represent events

• asynchronous and determined by receivers what events are interested

– event types

• each type has attributes (information in it)

• subscription filtering: focus on certain values in the attributes (e.g. "buy" events, but only "buy car" events)

•Publish-subscribe paradigm

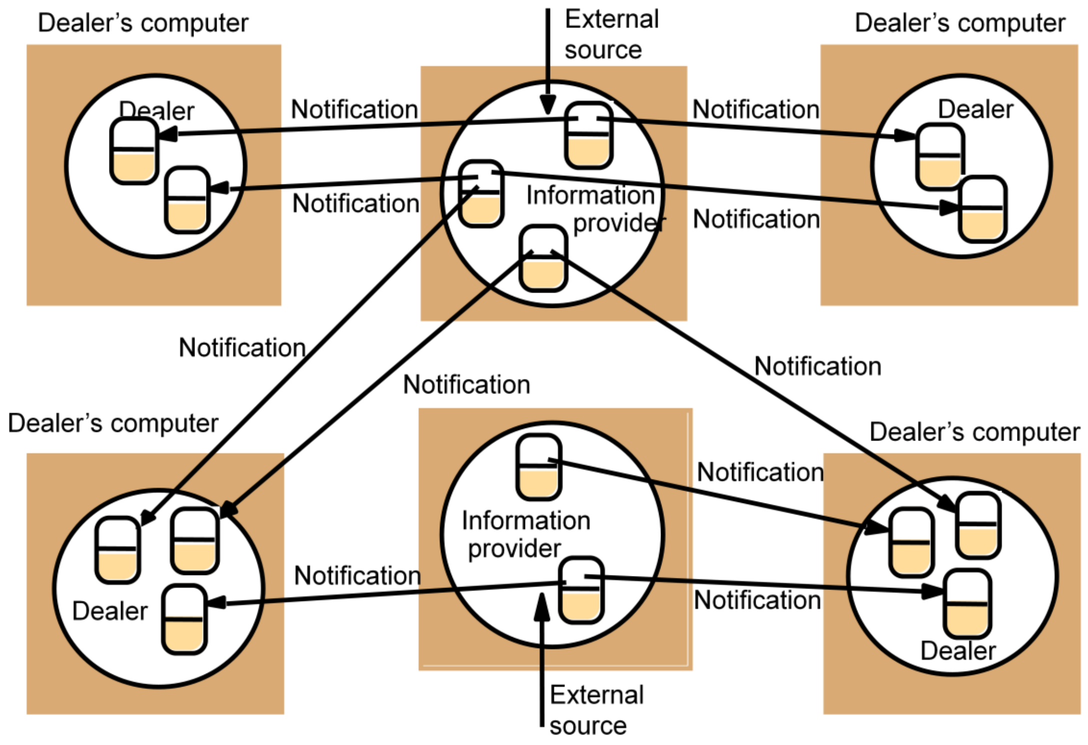
– publish events to send

– subscribe events to receive

-

•Main characteristics in distributed event-based systems
 – Heterogeneous: a way to standardize communication in heterogeneous systems
  • not designed to communicate directly
 – Asynchronous: notifications are sent asynchronously
  • no need for a publisher to wait for each subscriber--subscribers come and go

Dealing room system: allow dealers using computers to see the latest information about the market prices of the stocks they deal in

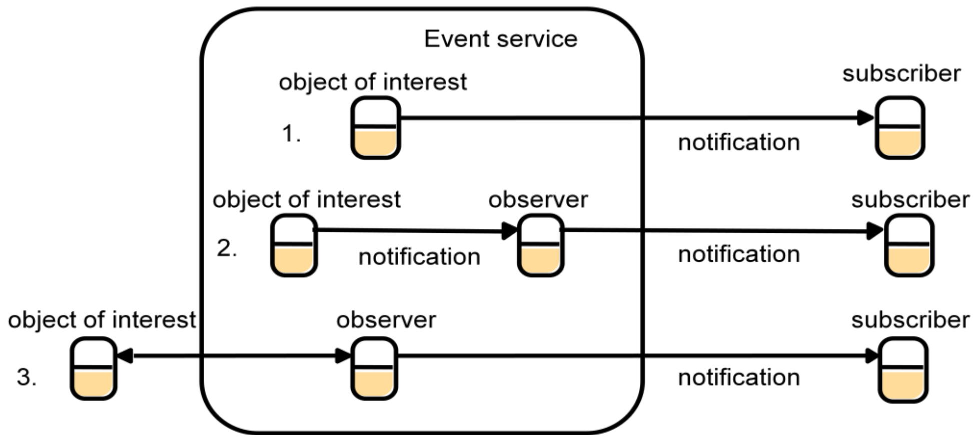


Distributed Event Notification

• Distributed event notification
 – decouple publishers from subscribers via an event service (manager)
• Architecture: roles of participating objects
 – object of interest (usually changes in states are interesting)
 – event
 – notification
 – subscriber
 – observer object (proxy) [reduce work on the object of interest]
•forwarding
• filtering of events types and content/attributes
• patterns of events (occurrence of multiple events, not just one)
• mailboxes (notifications in batch es, subscriber might not be ready)
 – publisher (object of interest or observer object)
• generates event notifications

-

Example: Distributed Event Notification



•Three cases
– Inside object without an observer: send notifications directly to the subscribers
– Inside object with an observer: send notification via the observer to the subscribers
– Outside object (with an observer)

1. An observer queries the object of interest in order to discover when events occur
2. The observer sends notifications to the subscribers

Case Study: Jini Distributed Event Specification

•Jini

–Allow a potential subscriber in one Java Virtual Machine (JVM) to subscribe to and receive notifications of events in an objectof interest in another JVM

– Main objects

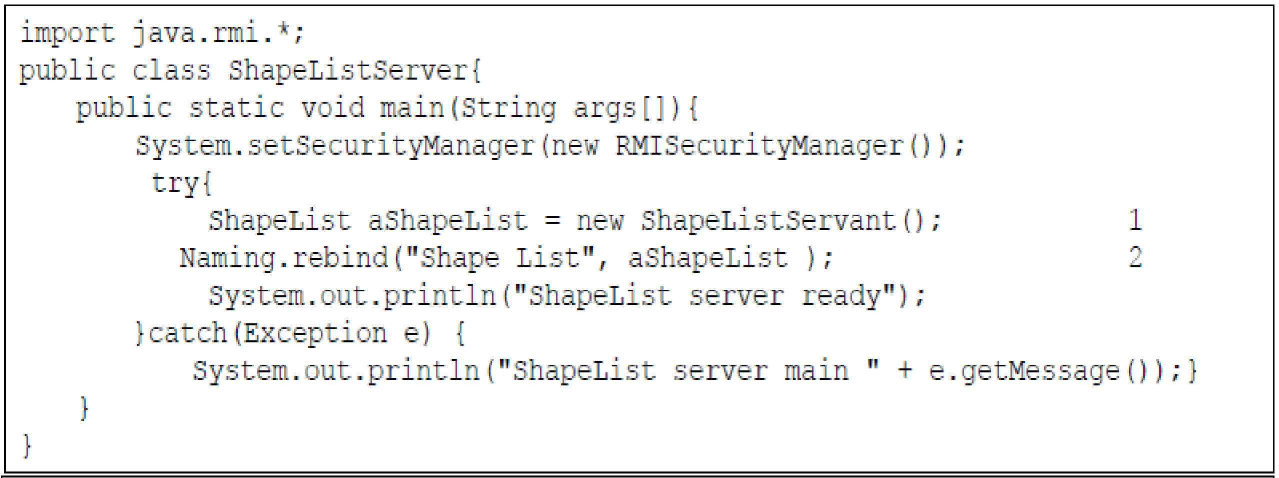
• event generators (publishers)
• remote event listeners (subscribers)
• remote events (events)
• third-party agents (observers)

– An object subscribes to events by informing the event generator about the type of event and specifying a remote event listener as the target for notification

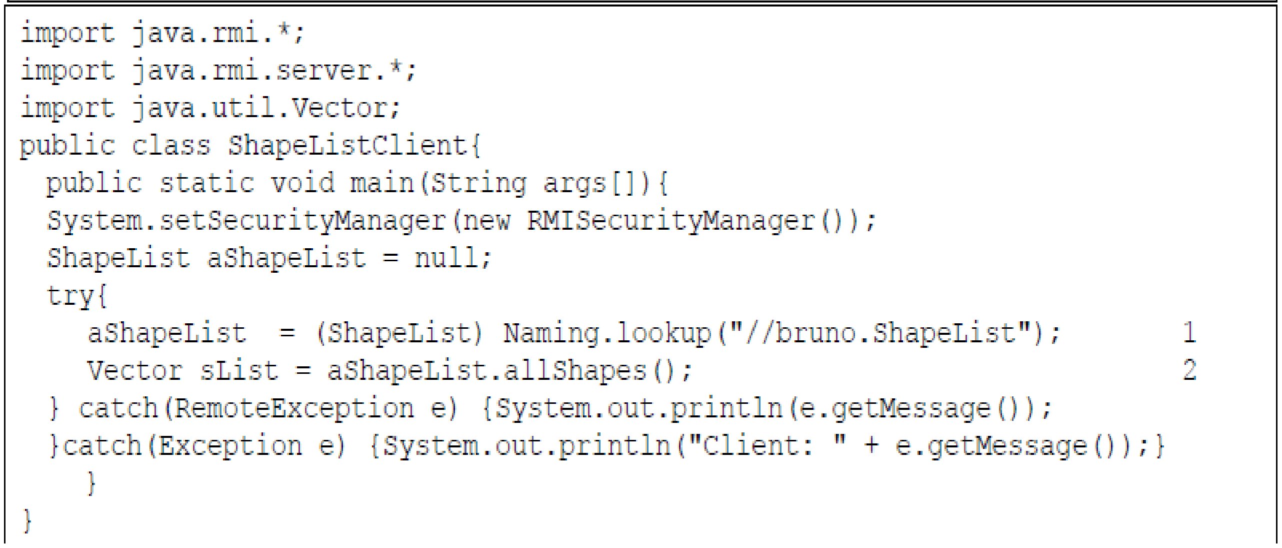
-

Case Study: Java RMI

Java Remote interfaces *Shape* and *ShapeList* *and* Java class ShapeListServant implements interface ShapeList



```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
            ShapeList aShapeList = new ShapeListServant();             1
          Naming.rebind("Shape List", aShapeList );                    2
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
           System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```



```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
  public static void main(String args[]){
  System.setSecurityManager(new RMISecurityManager());
  ShapeList aShapeList = null;
  try{
     aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");       1
     Vector sList = aShapeList.allShapes();                              2
  } catch(RemoteException e) {System.out.println(e.getMessage());
  }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
     }
}
```

Java class ShapeListServer with main and Java client of ShapreList

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
             ShapeList aShapeList = new ShapeListServant();          1
           Naming.rebind("Shape List", aShapeList );                 2
             System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
  public static void main(String args[]){
  System.setSecurityManager(new RMISecurityManager());
  ShapeList aShapeList = null;
  try{
     aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");       1
     Vector sList = aShapeList.allShapes();                              2
  } catch(RemoteException e) {System.out.println(e.getMessage());
  }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
     }
}
```

Naming class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
            ShapeList aShapeList = new ShapeListServant();                    1
                Naming.rebind("Shape List", aShapeList );                     2
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
     private Vector theList;              // contains the list of Shapes        1
     private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {          2
        version++;
            Shape s = new ShapeServant( g, version);                           3
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

-

Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
    private Vector theList;            // contains the list of Shapes         1
    private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {     2
        version++;
            Shape s = new ShapeServant( g, version);                       3
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

Java RMI Callbacks

•Callbacks

– server notifying the clients of events

– why?

• polling from clients increases overhead on server

• not up-to-date for clients to inform users

– how

• remote object (callback object) on client for server to call

• client tells the server about the callback object, server put the client on a list

• server call methods on the callback object when events occur

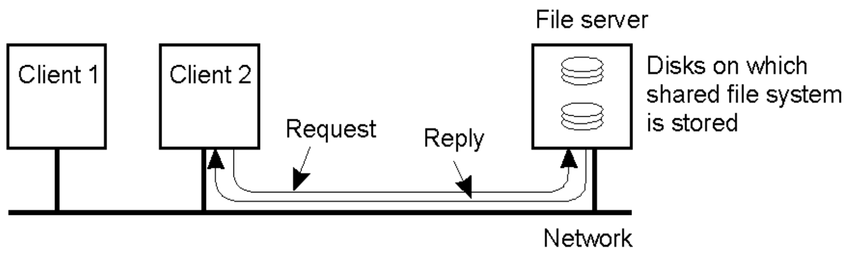
– client might forget to remove itself from the list

• lease--client expire

The task of any operating system is to provide problem-oriented abstractions of the underlying physical resources – the processors, memory, networks, and storage media. An operating system such as UNIX (and its variants, such as Linux and Mac OS X) or Windows (and its variants, such as XP, Vista and Windows 7) provides the programmer with, for example, files rather than disk blocks, and with sockets rather than raw network access. It takes over the physical resources on a single node and manages them to present these resource abstractions through the system-call interface.

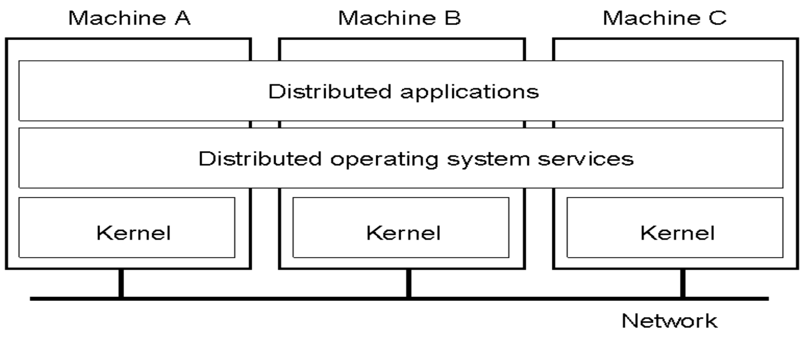
The operating system's middleware support role, it is useful to gain some historical perspective by examining two operating system concepts that have come about during the development of distributed systems: network operating systems and distributed operating systems.

Both UNIX and Windows are examples of *network operating systems*. They have a networking capability built into them and so can be used to access remote resources. Access is network-transparent for some – not all – types of resource. For example, through a distributed file system such as NFS, users have network-transparent access to files. That is, many of the files that users access are stored remotely, on a server, and this is largely transparent to their applications.



An operating system that produces a single system image like this for all the resources in a distributed system is called a *distributed operating system*

-



Middleware and the Operating System

What is a distributed OS?

- Presents users (and applications) with an integrated computing platform that hides the individual computers.
- Has control over all of the nodes (computers) in the network and allocates their resources to tasks without user involvement.
    - In a distributed OS, the user doesn't know (or care) where his programs are running.
- Examples:
    - Cluster computer systems
    - V system, Sprite

- In fact, there are no distributed operating systems in general use, only network operating systems such as UNIX, Mac OS and Windows.
- to remain the case, for two main reasons.

The first is that users have much invested in their application software, which often meets their current problem-solving needs; they will not adopt a new operating system that will not run their applications, whatever efficiency advantages it offers.

The second reason against the adoption of distributed operating systems is that users tend to prefer to have a degree of autonomy for their machines, even in a closely knit organization.

Combination of middleware and network OS

- No distributed OS in general use
    - Users have much invested in their application software
    - Users tend to prefer to have a degree of autonomy for their machines
- Network OS provides autonomy
- Middleware provides network-transparent access resource

The relationship between OS and Middleware

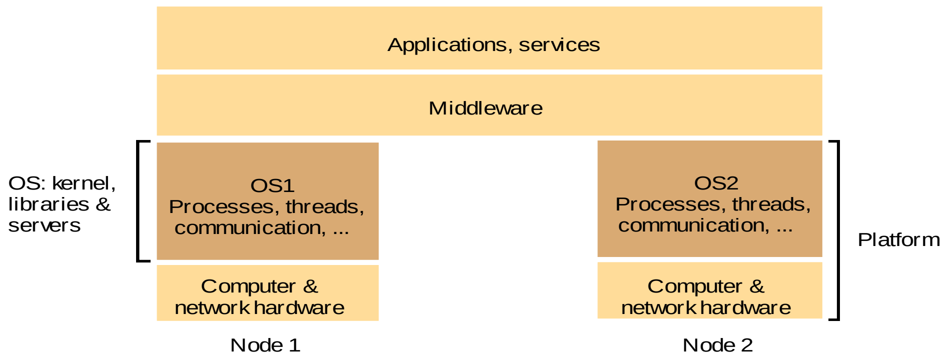
- Operating System
    - Tasks: processing, storage and communication
    - Components: kernel, library, user-level services
- Middleware
    - runs on a variety of OS-hardware combinations

-

– remote invocations

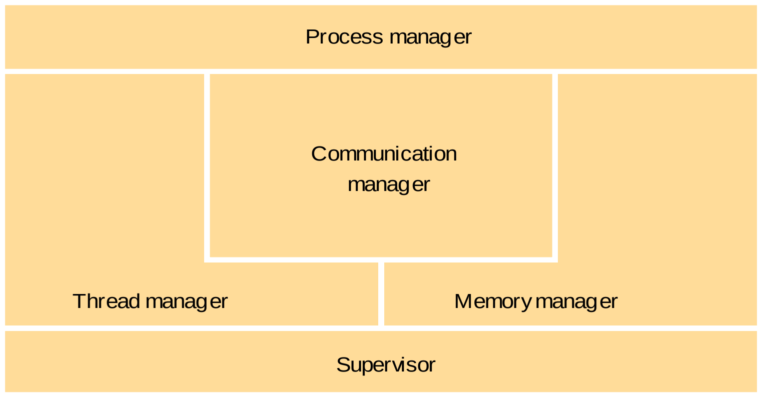
Functions that OS should provide for middleware

The following figure shows how the operating system layer at each of two nodes supports a common middleware layer in providing a distributed infrastructure for applications and services.


Applications, services
Middleware
OS: kernel, libraries & servers
OS1 Processes, threads, communication, ...
OS2 Processes, threads, communication, ...
Platform
Computer & network hardware
Computer & network hardware
Node 1
Node 2

Encapsulation: They should provide a useful service interface to their resources – that is, a set of operations that meet their clients' needs. Details such as management of memory and devices used to implement resources should be hidden from clients.
Protection: Resources require protection from illegitimate accesses – for example, files are protected from being read by users without read permissions, and device registers are protected from application processes.
Concurrent processing: Clients may share resources and access them concurrently. Resource managers are responsible for achieving concurrency transparency.
Communication: Operation parameters and results have to be passed to and from resource managers, over a network or within a computer.
Scheduling: When an operation is invoked, its processing must be scheduled within the kernel or server.

The core OS components

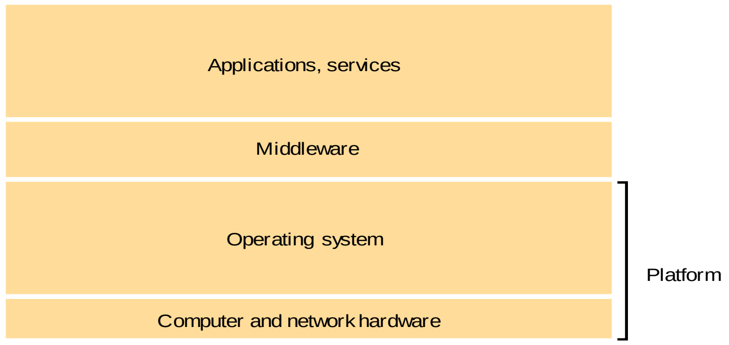


- **Process manager**
  – Handles the creation of and operations upon processes.

- **Thread manager**
  - Thread creation, synchronization and scheduling
- **Communication manager**
  - Communication between threads attached to different processes on the same computer
- **Memory manager**
  - Management of physical and virtual memory
- **Supervisor**
  - Dispatching of interrupts, system call traps and other exceptions
  - control of memory management unit and hardware caches

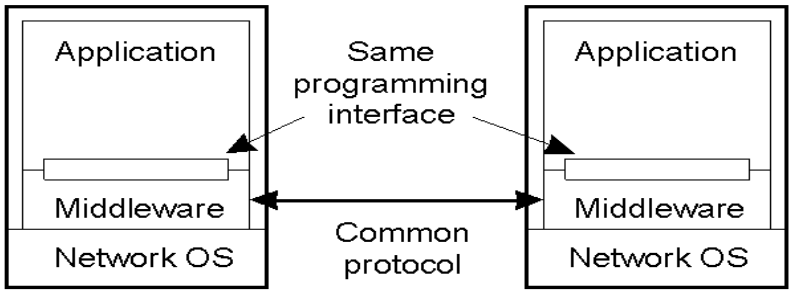
processor and floating point unit register manipulations

Software and hardware service layers in distributed systems



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.



Typical Middleware Services

- Communication
- Naming
- Persistence
- Distributed transactions
- Security

Middleware Models

- Distributed files
  - Examples?
- Remote procedure call
  - Examples?

- Distributed objects
  - Examples?
- Distributed documents
  - Examples?
- Others?
  - Message-oriented middleware (MOM)
  - Service oriented architecture (SOA)
  - Document-oriented

Middleware and the Operating System

- Middleware implements abstractions that support network-wide programming. Examples:
    - RPC and RMI (Sun RPC, Corba, Java RMI)
    - event distribution and filtering (Corba Event Notification, Elvin)
    - resource discovery for mobile and ubiquitous computing
    - support for multimedia streaming
- Traditional OS's (e.g. early Unix, Windows 3.0)
  - simplify, protect and optimize the use of local resources
- Network OS's (e.g. Mach, modern UNIX, Windows NT)
  - do the same but they also support a wide range of communication standards and enable remote processes to access (some) local resources (e.g. files).

DOS vs. NOS vs. Middleware Discussion

- What is good/bad about DOS?
  - Transparency
  - Other issues have reduced success.
  - Problems are often socio-technological.
- What is good/bad about NOS?
  - Simple.
  - Decoupled, easy to add/remove.
  - Lack of transparency.
- What is good/bad about middleware?
  - Easy to make multiplatform.
  - Easy to start something new.
    - But this can also be bad.

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

-

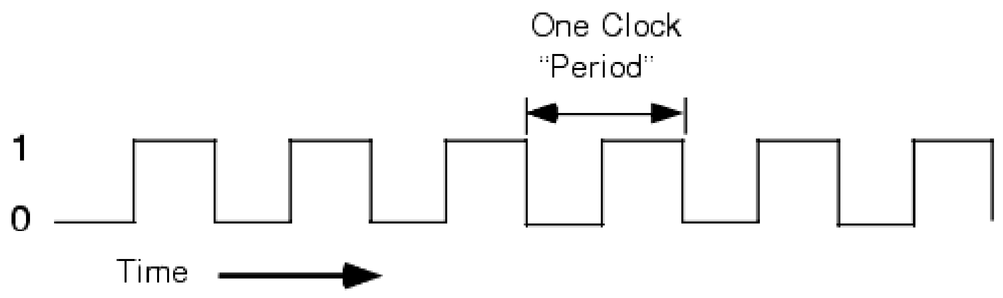
Illegitimate access

- **Maliciously contrived code**
- **Benign code**
  - contains a bug
  - have unanticipated behavior
- **Example: *read* and *write* in File System**
  - Illegal user vs. access right control
  - Access the file pointer variable directly (*setFilePointerRandomly*) vs. type-safe language
    - Type–safe language, e.g. Java or Modula-3
    - Non-type-safe language, e.g. C or C++

Kernel and Protection

- **Kernel**
  - always runs
  - complete access privileges for the physical resources
- **Different execution mode**
  - *An address space:* a collection of ranges of virtual memory locations, in each of which a specified combination of memory access rights applies, e.g.: read only or read-write
  - *supervisor* mode (*kernel process*) / *use*r mode (*user process*)
  - Interface between kernel and user processes: system call trap
- **The price for protection**
  - switching between different processes take many processor cycles
  - a system call trap is a more expensive operation than a simple method call

The System Clock



system clock frequency

clock period(One full period is also called a **clock cycle)**

"Hertz" (Hz) meaning one cycle per second

10 MHz : 100 nanoseconds

1GHz: 1 nanoseconds

Process and thread

- **Process**
  - A program in execution
  - Problem: sharing between related activities are awkward and expensive
  - Nowadays, a process consists of an *execution environment* together with one or more *threads*
  - an analogy at page 215
- **Thread**
  - Abstraction of a single activity
  - Benefits
    - Responsiveness
    - Resource sharing
    - Economy
    - Utilization of MP architectures

Execution environment

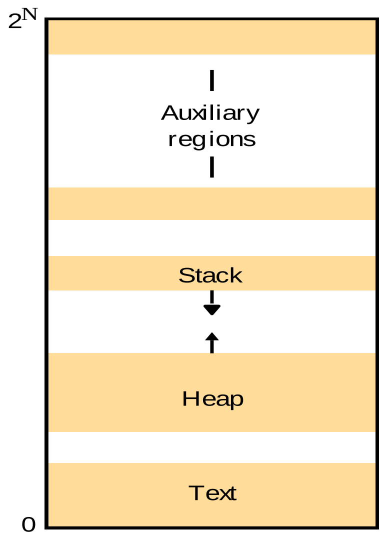
- the unit of resource management
- Consist of
  - An address space
  - Thread synchronization and communication resources such as semaphores and communication interfaces (e.g. sockets)
  - Higher-level resources such as open files and windows
- Shared by threads within a process

Address space

- **Address space**
  - a unit of management of *a process*'s virtual memory
  - Up to 2^{32} bytes and sometimes up to 2^{64} bytes
  - consists of one or more regions
- **Region**
  - an area of continuous virtual memory that is accessible by the threads of the owning process
- **The number of regions is indefinite**
  - Support a separate stack for each thread

- access *mapped file*
- Share memory between processes

- **Region can be shared**
  - Libraries
  - Kernel
  - Shared data and communication
  - Copy-on-write



Creation of new process in distributed system

- **Creating process by the operation system**
  - *Fork, exec* in UNIX
- **Process creation in distributed system**
  - The choice of a target host
  - The creation of an execution environment, an initial thread

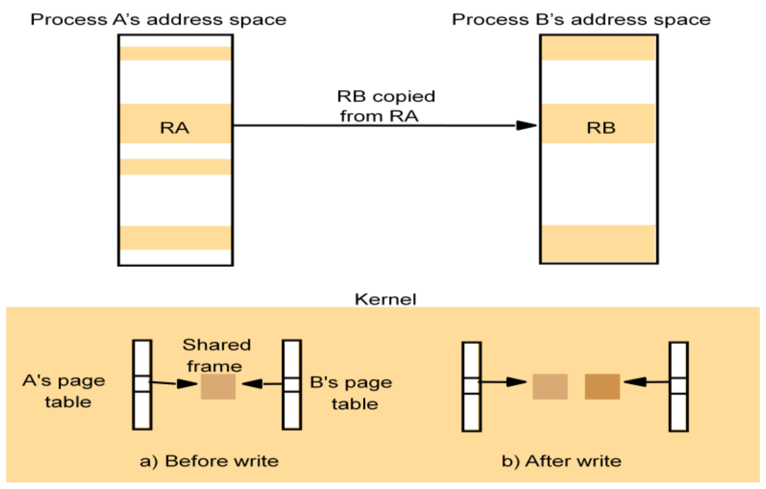
Choice of process host

- **Choice of process host**
  - running new processes at their originator's computer
  - sharing processing load between a set of computers
- **Load sharing policy**
  - Transfer policy: situate a new process locally or remotely?
  - Location policy: which node should host the new process?
    - Static policy without regard to the current state of the system
    - Adaptive policy applies heuristics to make their allocation decision
  - Migration policy: when&where should migrate the running process?

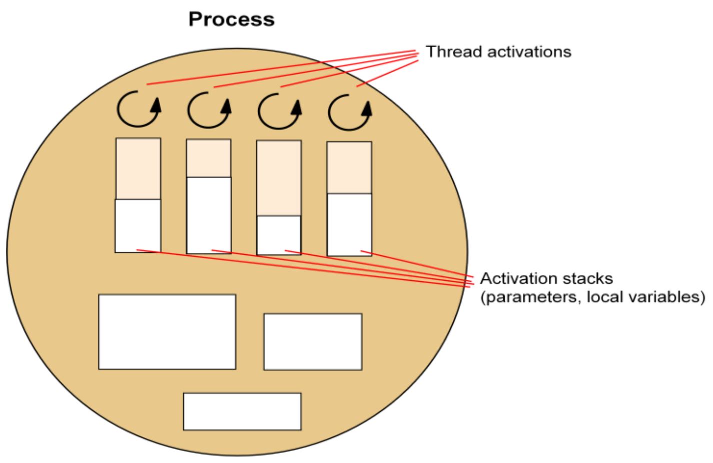
- **Load sharing system**
  - Centralized
  - Hierarchical
  - Decentralized

Creation of a new execution environment

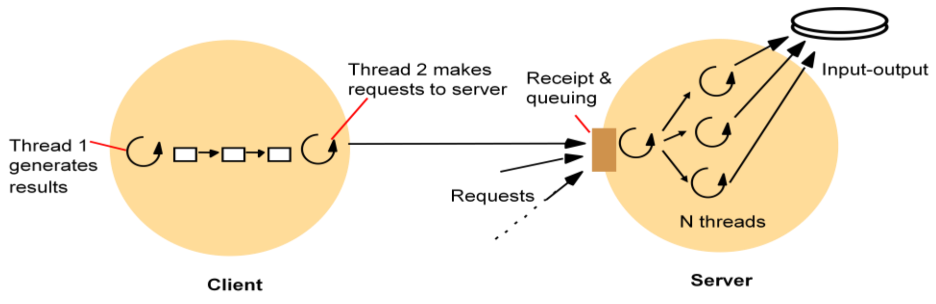
- Initializing the address space
  - Statically defined format
  - With respect to an existing execution environment, e.g. *fork*
- *Copy-on-write* scheme



Threads concept and implementation



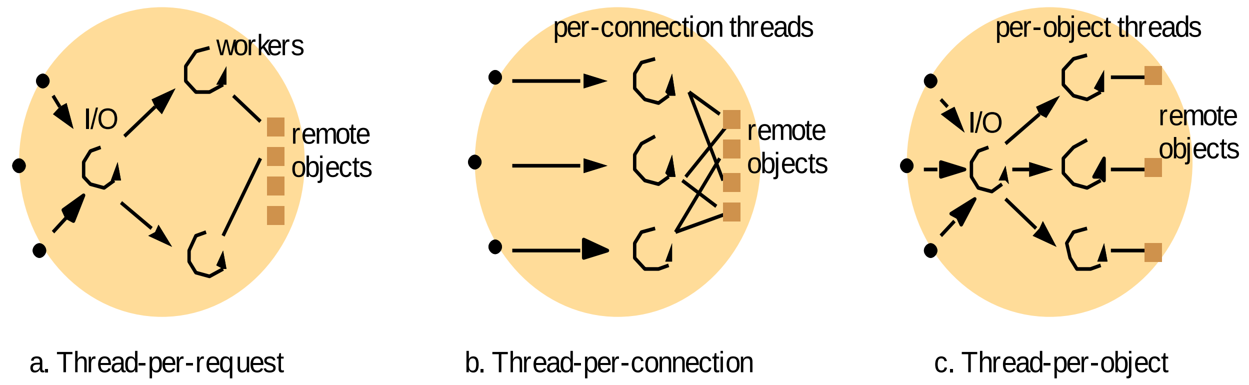
Client and server with threads



The 'worker pool' architecture

Alternative server threading architectures

(a) **would be useful for UDP-based service, e.g. NTP**
(b) **is the most commonly used - matches the TCP connection model**
(c) **is used where the service is encapsulated as an object. E.g. could have multiple shared whiteboards with one thread each. Each object has only one thread, avoiding the need for thread synchronization within objects.**



Threads versus multiple processes

- Creating a thread is (much) cheaper than a process (~10-20 times)
- Switching to a different thread in same process is (much) cheaper (5-50 times)
- Threads within same process can share data and other resources more conveniently and efficiently (without copying or messages)
- Threads within a process are not protected from each other

State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Threads implementation

Threads can be implemented:

– in the OS kernel (Win NT, Solaris, Mach)
– at user level (e.g. by a thread library: C threads, pthreads), or in the language (Ada, Java).

+ lightweight - no system calls

+ modifiable scheduler

+ low cost enables more threads to be employed

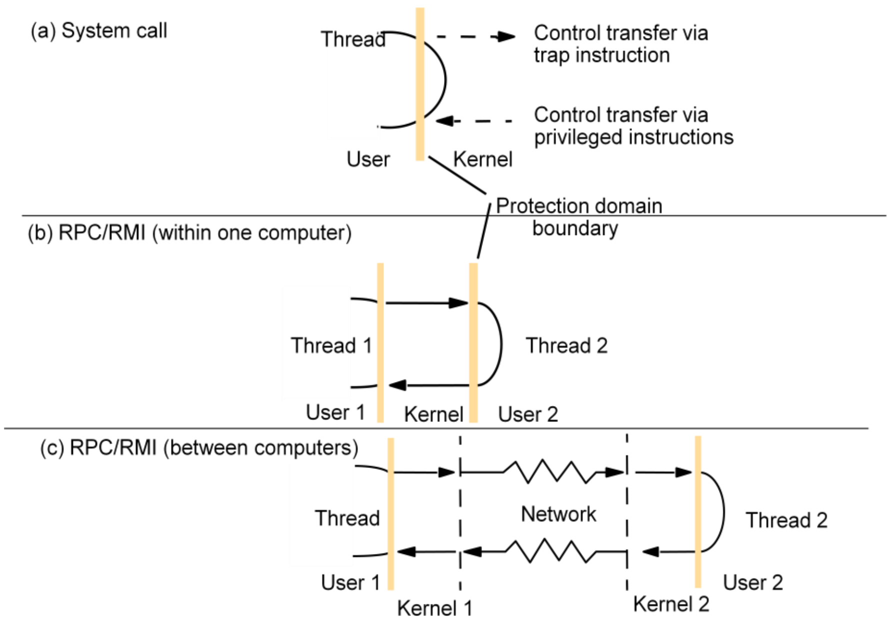
- not pre-emptive
- can exploit multiple processors
- - page fault blocks all threads

– hybrid approaches can gain some advantages of both
  - user-level hints to kernel scheduler
  - hierarchic threads (Solaris 2)
  - event-based (SPIN, FastThreads)

Implementation of invocation mechanisms

- **Communication primitives**
  - TCP(UDP) Socket in Unix and Windows
  - *DoOperation*, *getRequest*, *sendReply* in Amoeba
  - Group communication primitives in V system
- **Protocols and openness**
  - provide standard protocols that enable internetworking between middleware
  - integrate novel low-level protocols without upgrading their application
  - Static stack
    - new layer to be integrated permanently as a “driver”
  - Dynamic stack
    - protocol stack be composed on the fly
    - E.g. web browser utilize wide-area wireless link on the road and faster Ethernet connection in the office
- **Invocation costs**
  - Different invocations
  - The factors that matter
    - synchronous/asynchronous, *domain transition*, communication across a network, thread scheduling and switching
- **Invocation over the network**
  - Delay: the total RPC call time experienced by a client
  - Latency: the fixed overhead of an RPC, measured by null RPC
  - Throughput: the rate of data transfer between computers in a single RPC
  - An example
    - Threshold: one extra packet to be sent, might be an extra acknowledge packet is needed

Invocations between address spaces


(a) System call
Thread
Control transfer via trap instruction
Control transfer via privileged instructions
User
Kernel
Protection domain boundary
(b) RPC/RMI (within one computer)
Thread 1
Thread 2
User 1
Kernel
User 2
(c) RPC/RMI (between computers)
Thread
Network
Thread 2
User 1
Kernel 1
Kernel 2
User 2

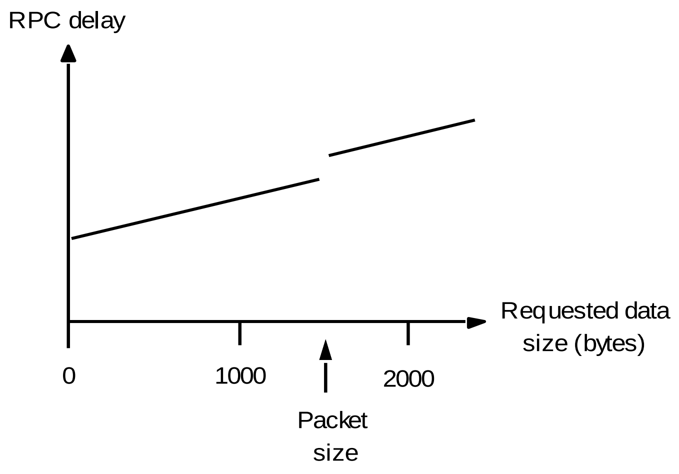
Support for communication and invocation

- The performance of RPC and RMI mechanisms is critical for effective distributed systems.
  - Typical times for 'null procedure call':
  - Local procedure call < 1 microseconds
  - Remote procedure call ~ 10 milliseconds
  - 'network time' (involving about 100 bytes transferred, at 100 megabits/sec.) accounts for only .01 millisecond; the remaining delays must be in OS and middleware - latency, not communication time.
- Factors affecting RPC/RMI performance
  - marshalling/unmarshalling + operation despatch at the server
  - data copying:- application -> kernel space -> communication buffers
  - thread scheduling and context switching:- including kernel entry
  - protocol processing:- for each protocol layer
  - network access delays:- connection setup, network latency

Improve the performance of RPC

- Memory sharing
  - rapid communication between processes in the same computer
- Choice of protocol
  - TCP/UDP
    - E.g. Persistent connections: several invocations during one
  - OS's buffer collect several small messages and send them together
- Invocation within a computer
  - Most cross-address-space invocation take place within a computer
  - LRPC (lightweight RPC)

RPC delay against parameter size



- A client stub marshals the call arguments into a message, sends the request message and receives and unmarshals the reply.
- At the server, a worker thread receives the incoming request, or an I/O threadreceives the request and passes it to a worker thread; in either case, the worker calls the appropriate server stub.
- The server stub unmarshals the request message, calls the designated procedure, and marshals and sends the reply.
- The following are the main components accounting for remote invocation delay, besides network transmission times:

Marshalling: Marshalling and unmarshalling, which involve copying and converting data, create a significant overhead as the amount of data grows.

Data copying: Potentially, even after marshalling, message data is copied several times in the course of an RPC:

1. across the user–kernel boundary, between the client or server address space and kernel buffers;
2. across each protocol layer (for example, RPC/UDP/IP/Ethernet);
3. between the network interface and kernel buffers.

Transfers between the network interface and main memory are usually handled by direct memory access (DMA). The processor handles the other copies.

Packet initialization: This involves initializing protocol headers and trailers, including checksums. The cost is therefore proportional, in part, to the amount of data sent.

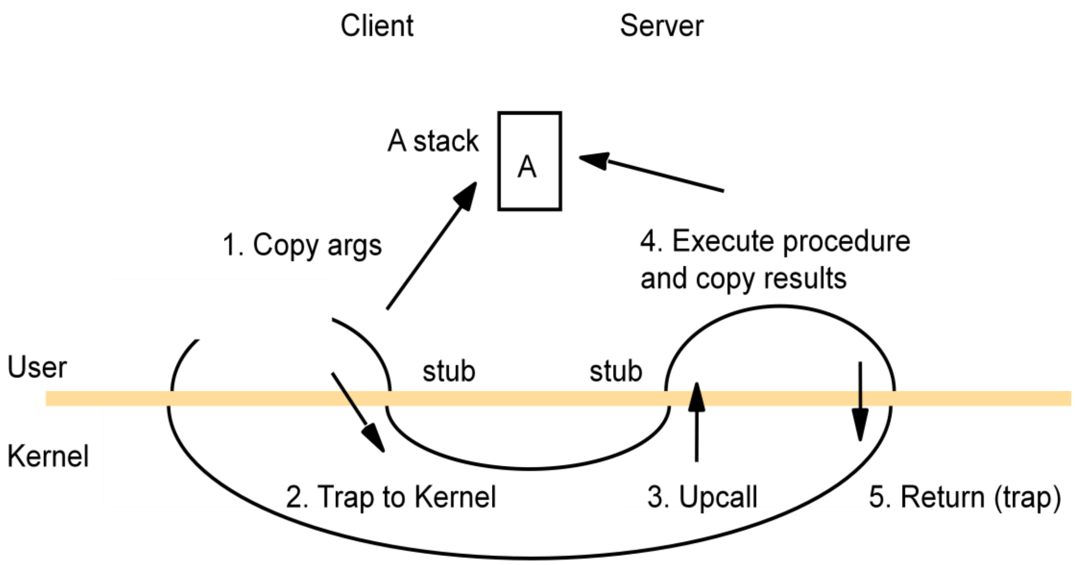
Thread scheduling and context switching: These may occur as follows:

1. Several system calls (that is, context switches) are made during an RPC, as stubs invoke the kernel's communication operations.
2. One or more server threads is scheduled.
3. If the operating system employs a separate network manager process, then each

Send involves a context switch to one of its threads.

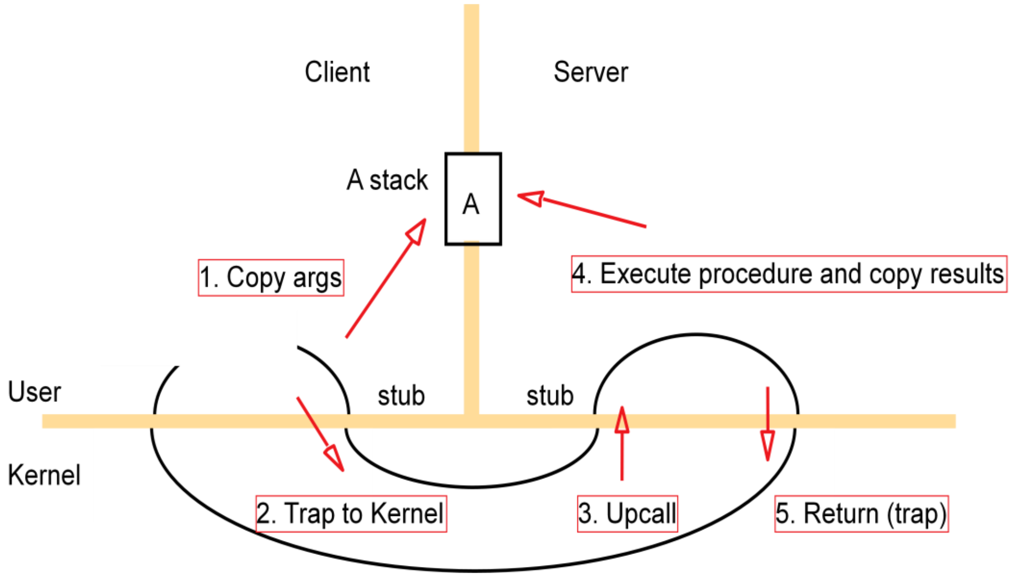
Waiting for acknowledgements: The choice of RPC protocol may influence delay, particularly when large amounts of data are sent.

A lightweight remote procedure call



Bershad's LRPC

- Uses shared memory for interprocess communication
  - while maintaining protection of the two processes
  - arguments copied only once (versus four times for convenitional RPC)
- Client threads can execute server code
  - via protected entry points only (uses capabilities)
- Up to 3 x faster for local invocations



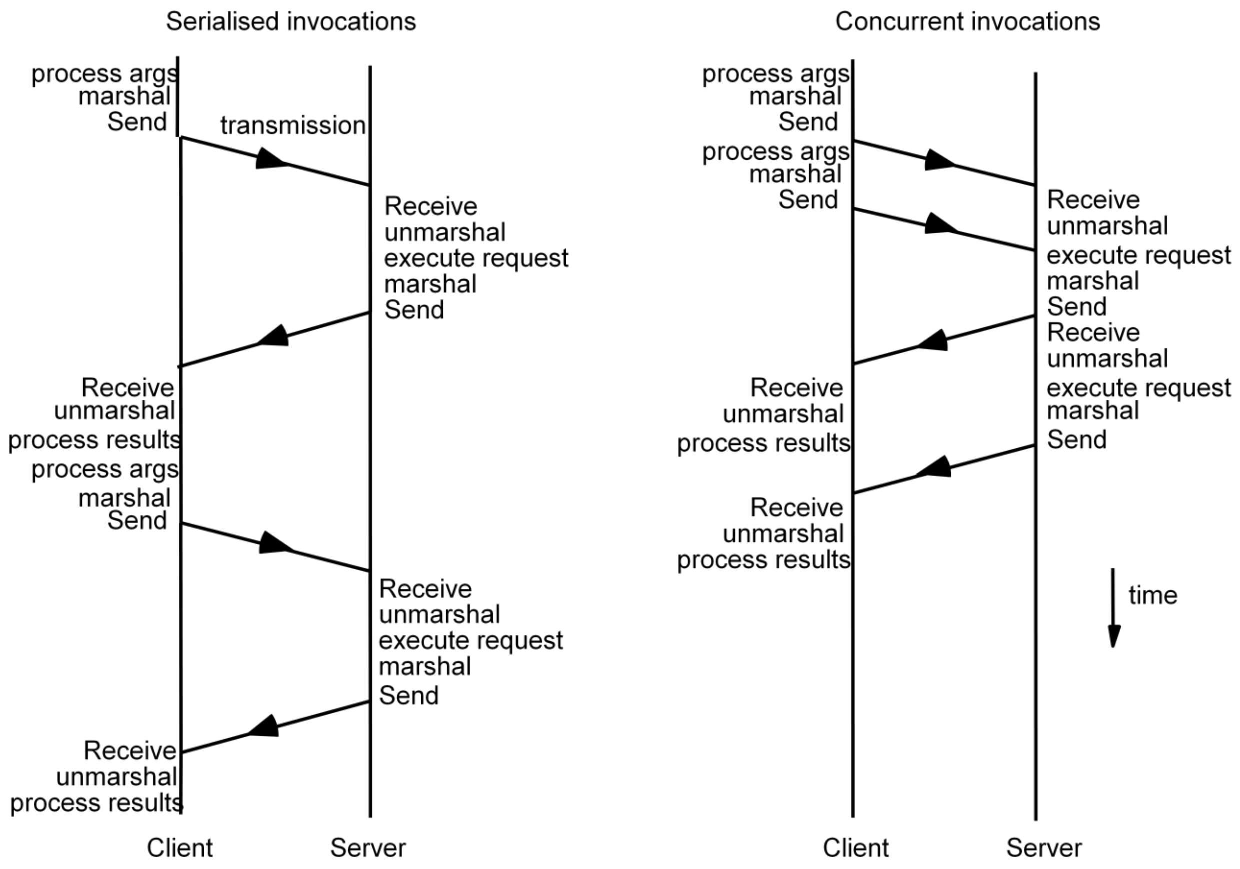
Asynchronous operation

- Performance characteristics of the Internet
  - High latencies, low bandwidths and high server loads
  - Network disconnection and reconnection.
  - outweigh any benefits that the OS can provide
- Asynchronous operation
  - Concurrent invocations
    - E.g., the browser fetches multiple images in a home page by concurrent *GET* requests
  - Asynchronous invocation: non-blocking call
    - E.g., CORBA *oneway* invocation: maybe semantics, or collect result by a separate call
- Persistent asynchronous invocations
  - Designed for *disconnected operation*
  - Try indefinitely to perform the invocation, until it is known to have succeeded or failed, or until the application cancels the invocation
  - QRPC (Queued RPC)
    - Client queues outgoing invocation requests in a stable log
    - Server queues invocation results
- The issues to programmers
  - How user can continue while the results of invocations are still not known?

The following figure shows the potential benefits of interleaving invocations (such as HTTP requests) between a client and a single server on a single-processor machine. In the serialized case, the client marshals the arguments, calls the *Send* operation and then waits until the reply from the server arrives – whereupon it *Receives*, unmarshals and then processes the results. After this it can make the second invocation.

Times for serialized and concurrent invocations

-



In the concurrent case, the first client thread marshals the arguments and calls the *Send* operation. The second thread then immediately makes the second invocation. Each thread waits to receive its results. The total time taken is liable to be lower than in the serialized case, as the figure shows. Similar benefits apply if the client threads make concurrent requests to several servers, and if the client executes on a multiprocessor even greater throughput is potentially possible, since the two threads' processing can also be overlapped.

-

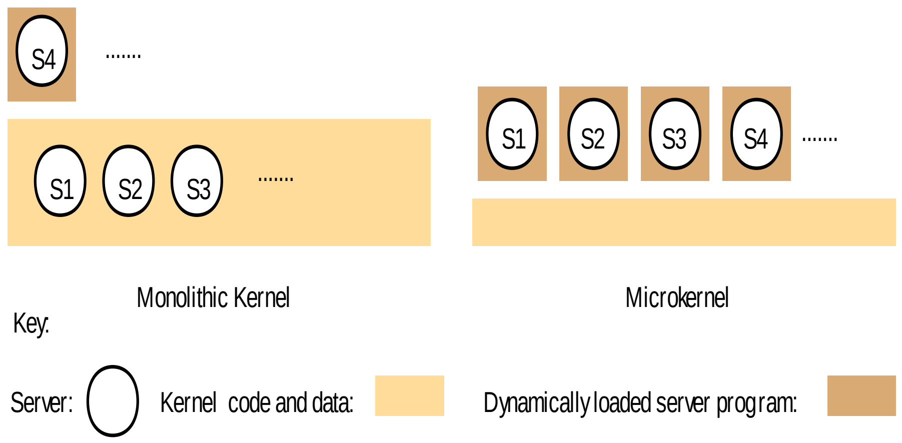
Operating System Architecture

- A key principle of distributed systems is openness.
- The major kernel architectures:
  - Monolithic kernels
  - Micro-kernels

- An open distributed system should make it possible to:
  - Run only that system software at each computer that is necessary for its particular role in the system architecture. For example, system software needs for PDA and dedicated server are different. Loading redundant modules wastes memory resources.
  - Allow the software (and the computer) implementing any particular service to be changed independent of other facilities.
  - Allow for alternatives of the same service to be provided, when this is required to suit different users or applications.
  - Introduce new services without harming the integrity of existing ones.
- A guiding principle of operating system design:
  - The separation of fixed resource management "mechanisms" from resource management "policies", which vary from application to application and service to service.
  - For example, an ideal scheduling system would provide mechanisms that enable a multimedia application such as videoconferencing to meet its real-time demands The kernel would provide only the most basic mechanisms upon which the general resource management tasks at a node are carried out.
  - Server modules would be dynamically loaded as required, to implement the required resourced management policies for the currently running applications.
  - while coexisting with a non-real-time application such as web browsing.

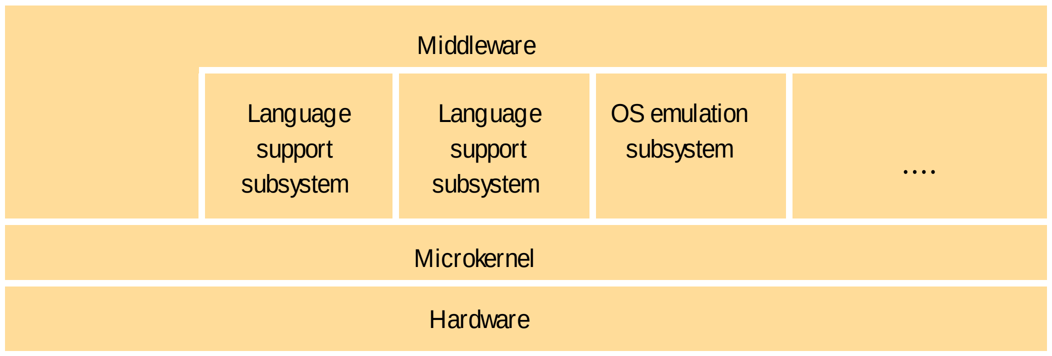
- Monolithic Kernels
  - A monolithic kernel can contain some server processes that execute within its address space, including file servers and some networking.
  - The code that these processes execute is part or the standard kernel configuration.

Monolithic kernel and microkernel



- Microkernel
  - ➢ The microkernel appears as a layer between hardware layer and a layer consisting of major systems.
    If performance is the goal, rather than portability, then middleware may use the facilities of the microkernel directly.

The role of the microkernel



The microkernel supports middleware via subsystems

- Monolithic and Microkernel comparison
  - ➢ The advantages of a microkernel
    - ❖ Its extensibility
    - ❖ Its ability to enforce modularity behind memory protection boundaries.
    - ❖ Its small kernel has less complexity.
  - ➢ The advantages of a monolithic
    - ❖ The relative efficiency with which operations can be invoked because even invocation to a separate user-level address space on the same node is more costly.

- Hybrid Approaches
  - ➢ Pure microkernel operating system such as Chorus & Mach have changed over a time to allow servers to be loaded dynamically into the kernel address space or into a user-level address space.
    In some operating system such as SPIN, the kernel and all dynamically loaded modules grafted onto the kernel execute within a single address space

Case Study of a Distributed Operating System

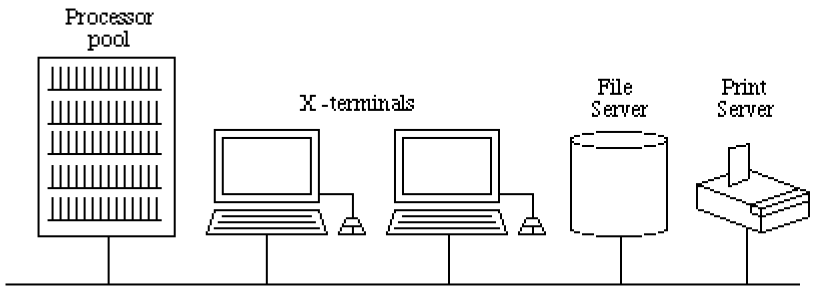
Introduction to Amoeba

- Originated at a university in Holland, 1981
- Currently used in various EU countries
- Built from the ground up. UNIX emulation added later
- Goal was to build a transparent distributed operating system
- Resources, regardless of their location, are managed by the system, and the user is unaware of where processes are actually run

---

The Amoeba System Architecture

- Assumes that a large number of CPUsare available and that each CPU ha 10s of Mb of memory
- CPUs are organised into processor pools



- CPUs do not need to be of the same architecture (can mix SPARC, Motorola PowerPC, 680x0, Intel, Pentium, etc.)
- When a user types a command, system determines which CPU(s) to execute it on. CPUs can be timeshared.
- Terminals are X-terminals or PCs running X emulators
- The processor pool doesn't have to be composed of CPU boards enclosed in a cabinet, they can be on PCs, etc., in different rooms, countries,...
- Some servers (e.g., file servers) run on dedicated processors, because they need to be available all the time

---

The Amoeba Microkernel

- The Amoeba microkernel is used on all terminals (with an on-board processor), processors, and servers
- The microkernel

  manages processes and threads

  provides low-level memory management support

  supports interprocess communication (point-to-point and group)

  handles low-level I/O for the devices attached to the machine

---

The Amoeba Servers: Introduction

- OS functionality not provided by the microkernel is performed by Amoeba servers
- To use a server, the client calls a stub procedure which marshalls parameters, sends the message, and blocks until the result comes back

Server Basics

- Amoeba uses capabilities
- Every OS data structure is an object, managed by a server
- To perform an operation on an object, a client performs an RPC with the appropriate server, specifying the object, the operation to be performed and any parameters needed.
- The operation is transparent (client does not know where server is, nor how the operation is performed)
- Capabilites

  To create an object the client performs an RPC with the server

  Server creates the object and returns a capability

  To use the object in the future, the client must present the correct capability

Bits	48	24	8	48
	Server Port	Object	Rights	Check

  The check field is used to protect the capability against forgery

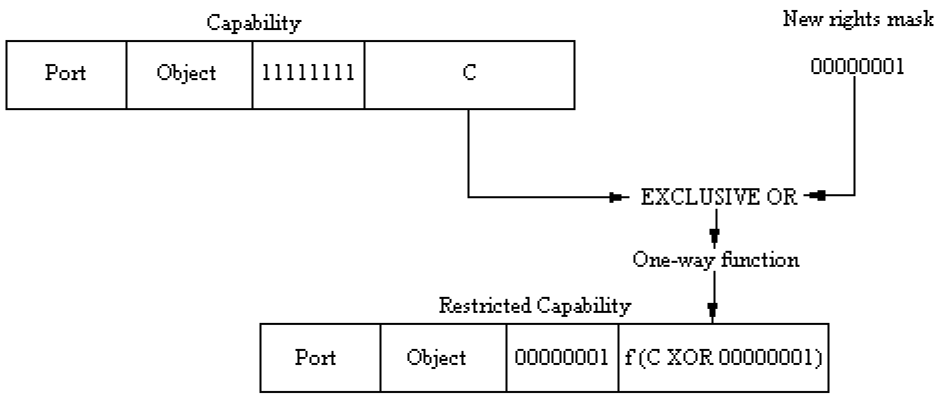
- Object protection

When an object is created, server generates random check field, which it stores both in the capability and in its own tables

The rights bits in the capability are set to on

The server sends the owner capability back to the client

Creating a capability with restricted rights



Client can send this new capability to another process

Process Management

- All processes are objects protected by capabilities
- Processes are managed at 3 levels

by process servers, part of the microkernel

by library procedures which act as interfaces

by the run server, which decides where to run the processes

- Process management uses process descriptors

Contains:

platform description

process' owner's capability

etc

Memory Management

- Designed with performance, simplicity and economics in mind
- Process occupies contiguous segments in memory
- All of a process is constantly in memory
- Process is never swapped out or paged

Communication

- Point-to-point (RPC) and Group

The Amoeba Servers

The File System

- Consists of the Bullet (File) Server, the Directory Server, and the Replication Server

The Bullet Server

- Designed to run on machines with large amounts of RAM and huge local disks
- Used for file storage
- Client process creates a file using the *create* call
- Bullet server returns a capability that can be used to *read* the file with
- Files are immutable, and file size is known at file creation time. Contiguous allocation policies used

The Directory Server

- Used for file naming
- Maps from ASCII names to capabilities
- Directories also protected by capabilities
- Directory server can be used to name ANY object, not just files and directories

The Replication Server

- Used for fault tolerence and performance
- Replication server creates copies of files, when it has time

Other Amoeba Servers

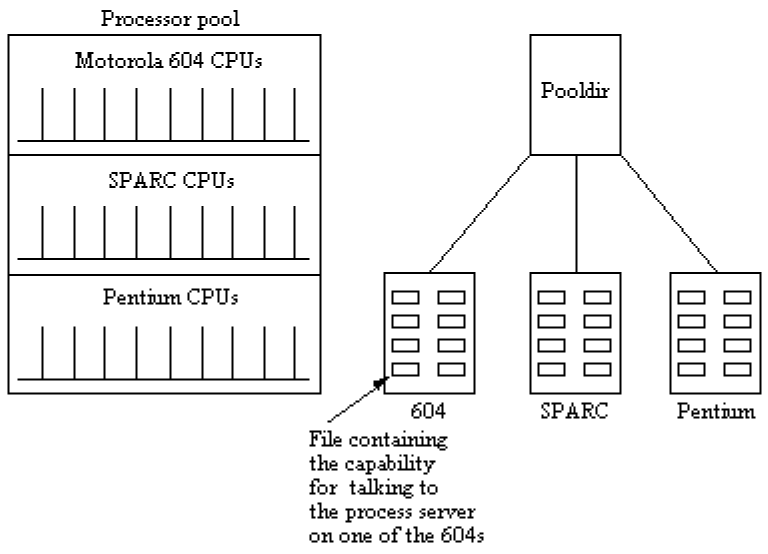
The Run Server

- When user types a command, two decisions have to be made

On which architecture should the process be run?

Which processor should be chosen?

- Run server manages the processor pools


Processor pool
Motorola 604 CPUs
SPARC CPUs
Pentium CPUs
Pooldir
604
SPARC
Pentium
File containing the capability for talking to the process server on one of the 604s

- Uses processes process descriptor to identify appropriate target architecture
- Checks which of the available processors have sufficient memory to run the process
- Estimates which of the remaining processor has the most available compute power

The Boot Server

- Provides a degree of fault tolerance
- Ensures that servers are up and running
- If it discovers that a server has crashed, it attempts to restart it, otherwise selects another processor to provide the service
- Boot server can be replicated to guard against its own failure