**PROCESS:** Threads: Introduction to Threads, Threads in Distributed Systems, Clients: user Interfaces, Client-Side Software for Distribution Transparency, Servers: General Design Issues, Object Servers, Software Agents: Software Agents in Distributed Systems, Agent Technology, Naming: Naming Entities: Names, Identifiers, and Address, Name Resolution, The Implementation of a Name System, Locating Mobile Entities: Naming verses Locating Entities, Simple Solutions, Home-Based Approaches, Hierarchical Approaches.

**PROCESS**

A program under execution is called as a process.

**THREAD**

1. It is a light weight program.

2. **Traditional operating systems**: concerned with the “local” management and scheduling of processes.

3. **Modern distributed systems**: a number of other issues are of equal importance.

4. **There are three main areas of study**

a. Threads and virtualization within clients/servers.

b. Process and code migration.

c. Software agents.

5. Modern OSs provide “virtual processors” within which programs execute.

6. A programs execution environment is documented in the process table and assigned   
a PID.

7. To achieve acceptable performance in distributed systems, relying on the OS’s idea   
of a process is often not enough – finer granularity is required.

* The solution: Threading.

**PROBLEMS WITH PROCESSES**

1. Creating and managing processes is generally   
regarded as an expensive task (fork system call).

2. Making sure all the processes peacefully co-exist on the system is not easy (as concurrency transparency comes at a price).

3. **Threads** can be thought of as an “execution of a part of a program (in user-space)”.

4. Rather than make the OS responsible for concurrency transparency, it is left to the individual application to manage the creation and scheduling of each thread.

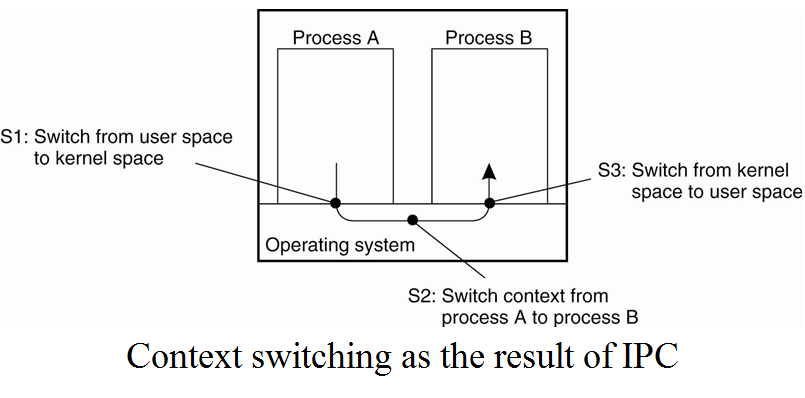
**Two Important Implications**

1.Threaded applications often run faster than non-threaded applications (as context-switches between kernel and user-space are avoided).

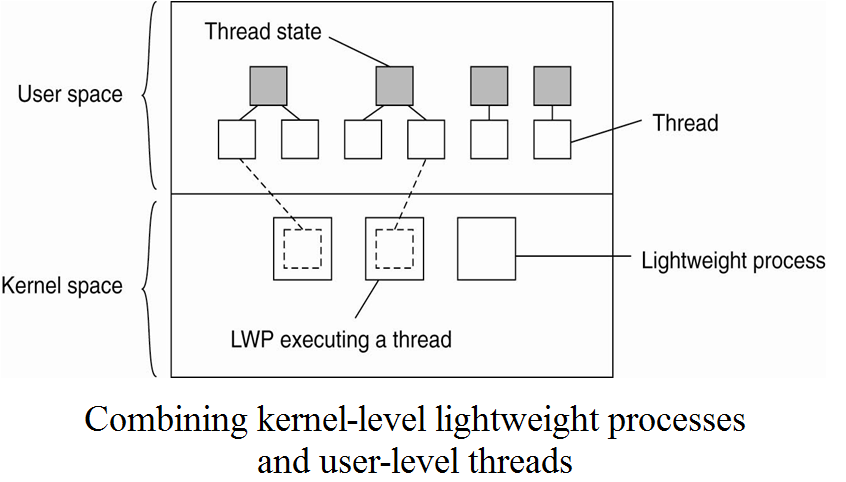
**2.** Threaded applications are harder to develop (although simple, clean designs can help here).

Additionally, the assumption is that the development environment provides a   
Threads Library for developers to use (most modern environments do).

**THREAD USAGE IN NON-DISTRIBUTED SYSTEMS**



**THREAD IMPLEMENTATION**

****

**THREADS IN NON-DISTRIBUTED SYSTEMS**

**Advantages:**

1.Blocking can be avoided

2. Excellent support for multi-processor systems (each running their own thread).

3. Expensive context-switches can be avoided.

4. For certain classes of application, the design and implementation is made considerably easier.

**THREADS IN DISTRIBUTED SYSTEMS**

**1.** Important characteristic: a blocking call in a thread does not result in the entire process   
being blocked.

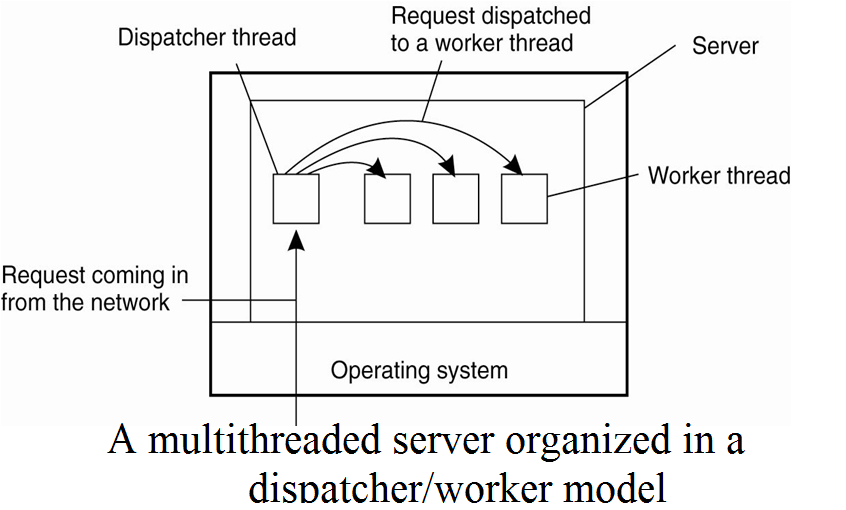
**2.** This leads to the key characteristic of threads within distributed systems:

“We can now express communications in the form of maintaining multiple logical connections at the same time (as opposed to a single, sequential, blocking process).”

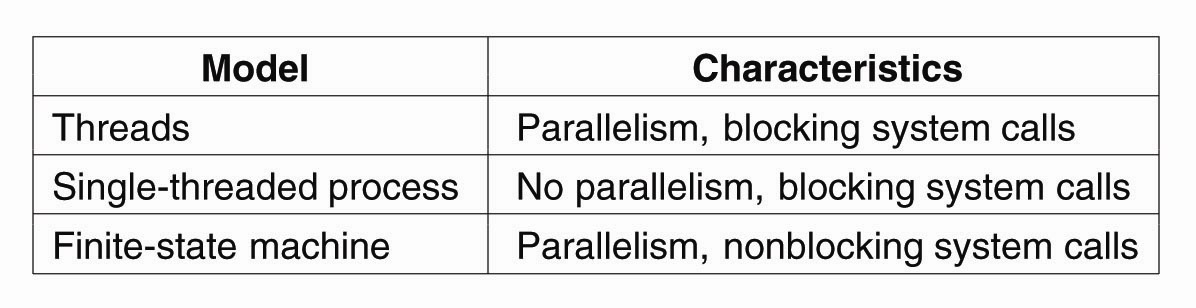
**Example: MT Clients and Servers**

* Mutli-Threaded Client: to achieve acceptable levels of perceived performance, it is often necessary to hide communications latencies.
* Consequently, a requirement exists to start communications while doing something else.
* Example: modern Web browsers.
* This leads to the notion of “truly parallel streams of data” arriving at a multi-threaded client application.
* Although threading is useful on clients, it is much more useful in distributed systems servers.
* The main idea is to exploit parallelism to attain high performance.
* A typical design is to organize the server as a single “dispatcher” with multiple threaded “workers”, as diagrammed overleaf.

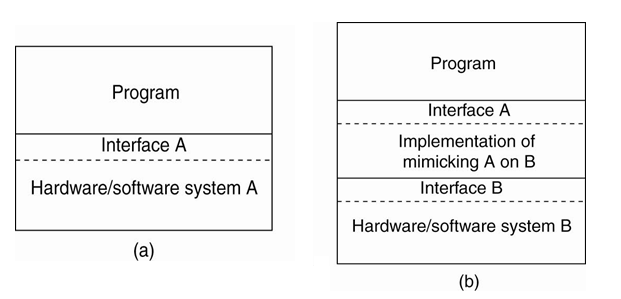
**MULTITHREADED SERVERS**



**MULTITHREADED SERVERS**

****

**THE ROLE OF VIRTUALIZATION IN DISTRIBUTED SYSTEMS**

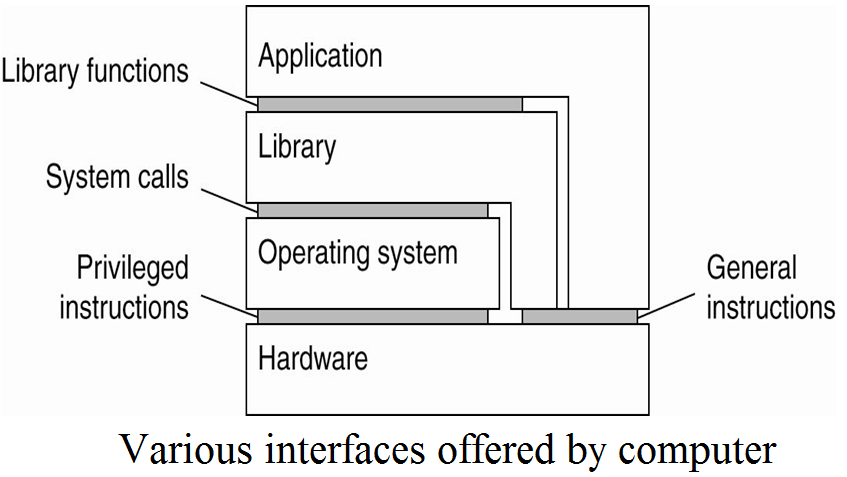
****

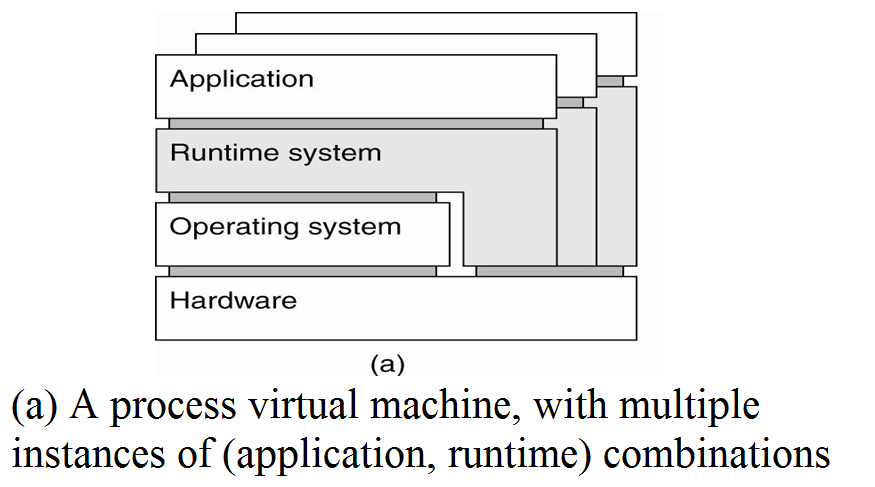
**a. General organization between a program, interface, and system**

**b. General organization of virtualizing system A on top of system B**

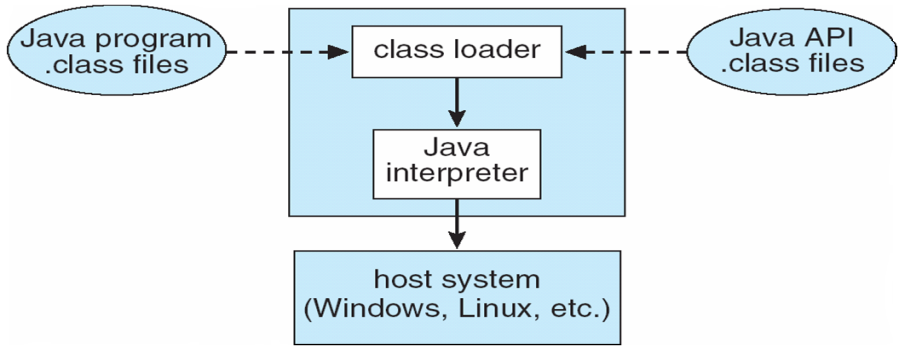
**ARCHITECTURES OF VIRTUAL MACHINES**

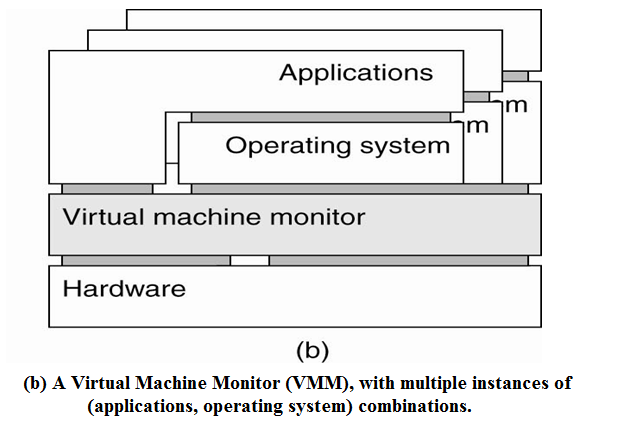
* There are interfaces at different levels.
* An interface between the hardware and software, consisting of machine instructions
  + that can be invoked by any program.
* An interface between the hardware and software, consisting of machine instructions
  + that can be invoked only by privileged programs, such as an operating system.
* An interface consisting of system calls as offered by an operating system.
* An interface consisting of library calls
  + generally forming what is known as an Application Programming Interface (API).
  + In many cases, the aforementioned system calls are hidden by an API.



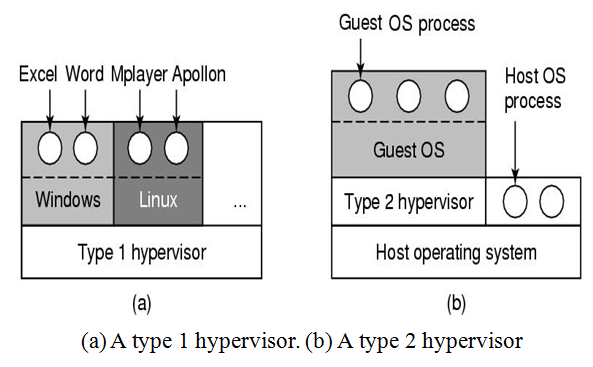


**THE JAVA VIRTUAL MACHINE**

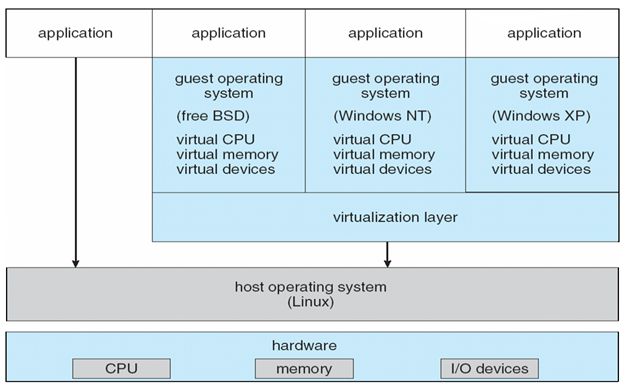
****

****

**TYPES OF VMM/HYPERVISORS**

****

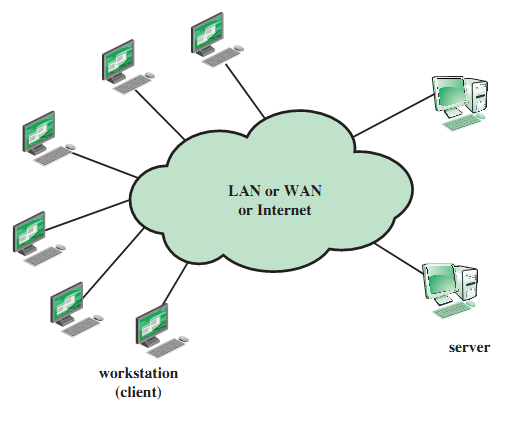
**VMWARE ARCHITECTURE**

****

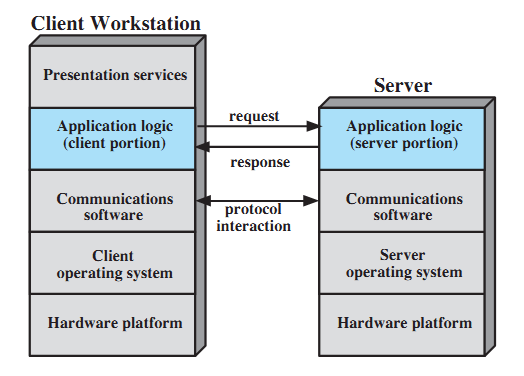
**More on Clients**

* **What’s a client?**
* **Definition: “A program which interacts with a human user and a remote server.”**
* **Typically, the user interacts with the client   
  via a GUI.**
* **Of course, there’s more to clients than simply providing a UI. Remember the multi-tiered levels of the Client/Server architecture from earlier …**

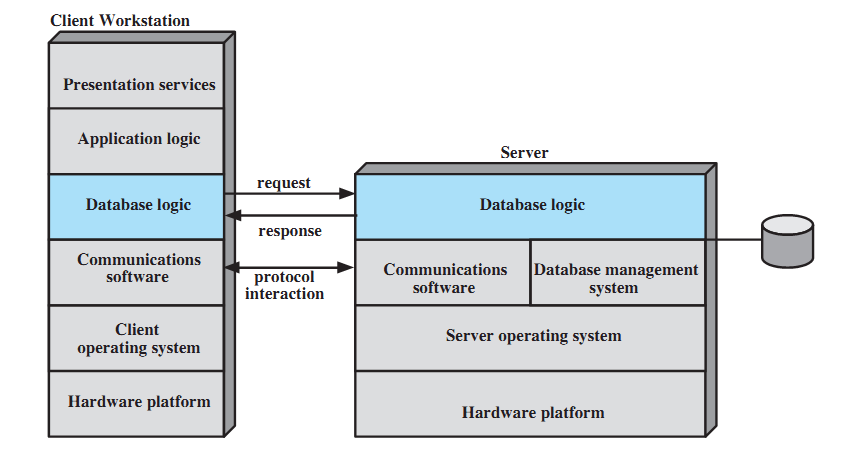
**GENERIC CLIENT/SERVER ENVIRONMENT**

****

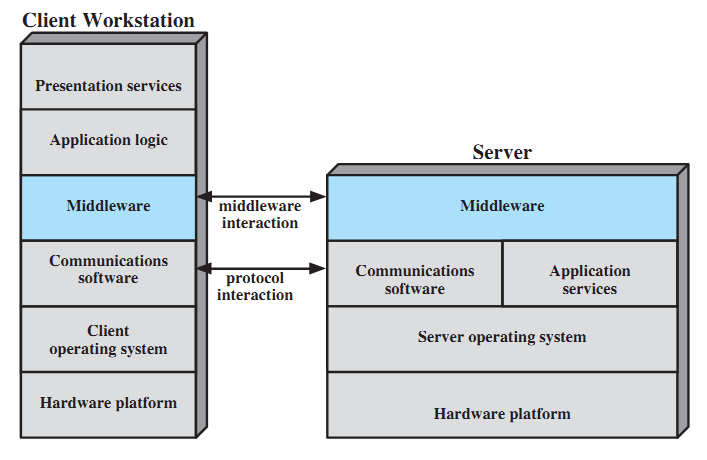
**Generic Client/Server Architecture**

****

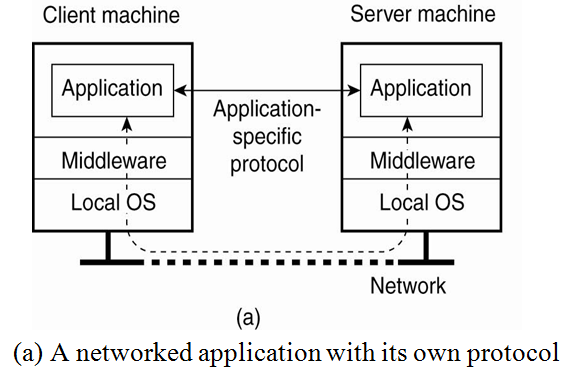
**Client/Server Architecture for Database Applications**

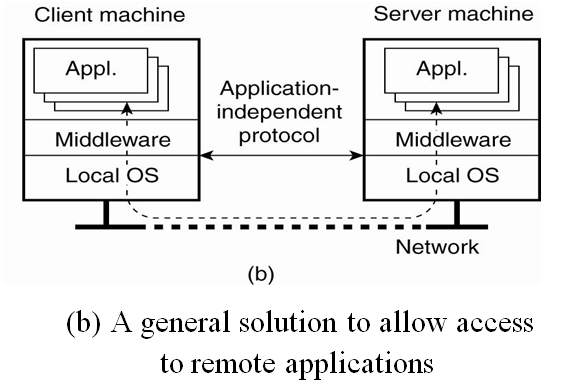
****

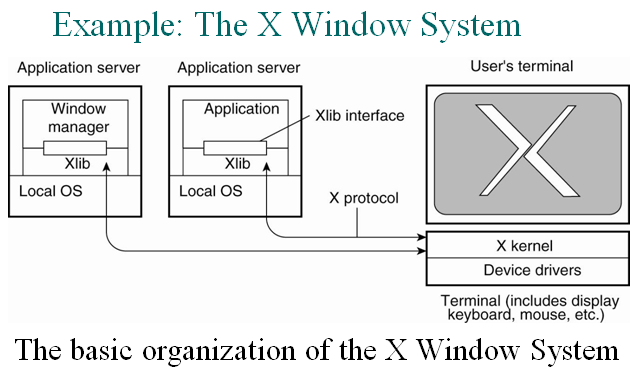
**Role of Middleware in Client/Server Architecture**

****

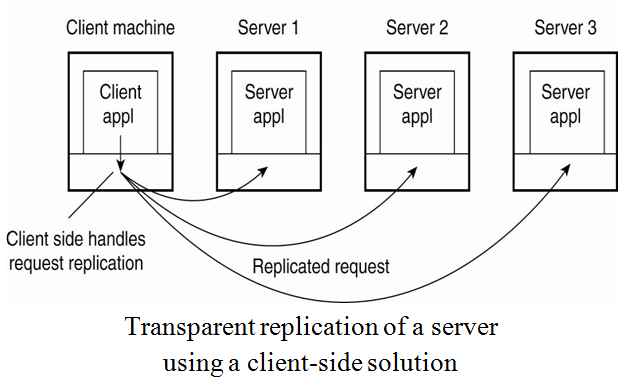
**NETWORKED USER INTERFACES**

****

****

****

**CLIENT-SIDE SOFTWARE FOR DISTRIBUTION TRANSPARENCY**

****

**More on Servers**

**SERVER**

“A process that implements a specific service on behalf of a collection of clients”.Typically, servers are organized to do one of two things:

1. Wait. 2. Service.

… Wait … service … wait … service … wait …

**SERVERS: ITERATIVE AND CONCURRENT**

1. Iterative: server handles request, then returns results to the client; any new client requests must wait for previous request to complete   
(also useful to think of this type of server as sequential).

**2.** Concurrent: server does not handle the request itself; a separate thread or sub-process handles the request and returns any results to the client; the server is then free to immediately service the next client (i.e., there’s no waiting, as service requests are processed in parallel).

**SERVER “STATES”**

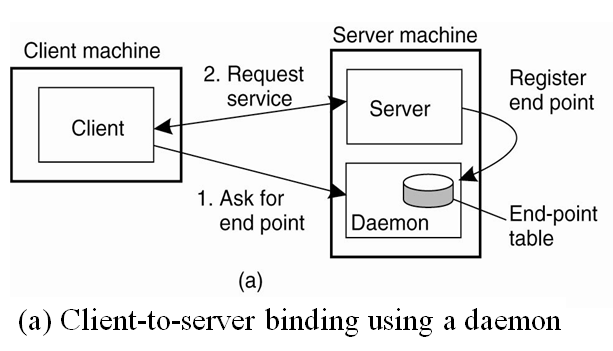
1. **Stateless servers** – no information is maintained on the current “connections” to the server. The Web is the classic example of a stateless service. As can be imagined, this type of server is **easy** to implement.

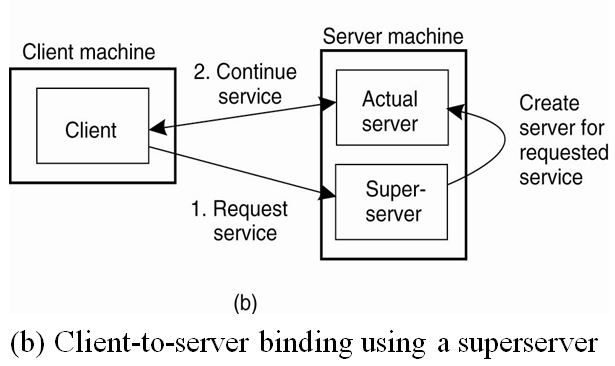
2. **Stateful servers** – information is maintained on the current “connections” to the server. Advanced file servers, where copies of a file can be updated “locally”, then applied to the main server (as it knows the state of things) - more **difficult** to implement.

**Problem: Identifying “end-points”?**

* How do clients know which end-point (or port) to contact a server at?   
  How do they “bind” to a server?
  + Statically assigned end-points (IANA).
  + Dynamically assigned end-points (DCE).
  + A popular variation:
    - the “super-server” (inetd on UNIX).

**GENERAL DESIGN ISSUES**

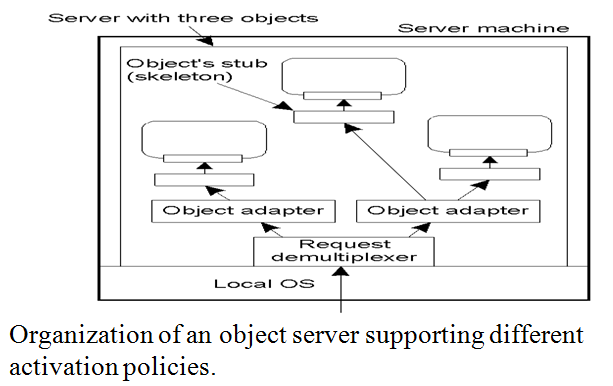




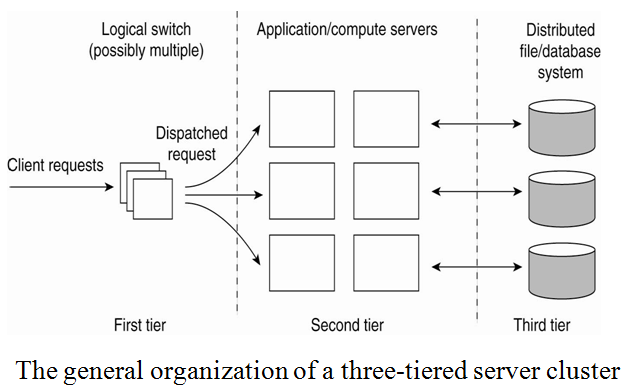
**A Special Type: Object Servers**

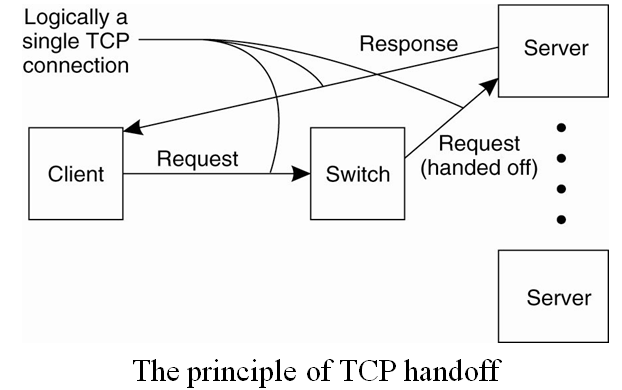
* A server tailored to support distributed objects.
* Does not provide a specific service.
* Provides a facility whereby objects can be remotely invoked by non-local clients.
* Consequently, object servers are highly adaptable.
* “A place where objects live”.

**Object Adapter**

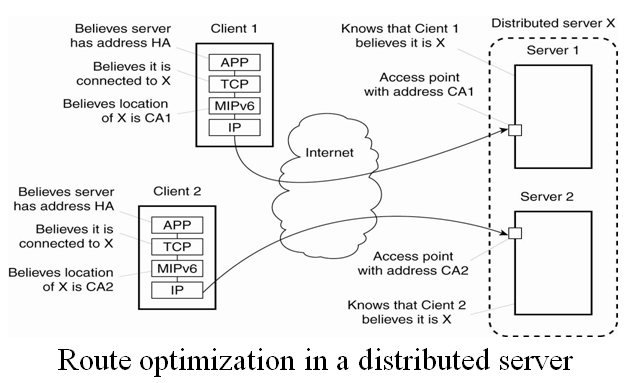


**SERVER CLUSTERS**

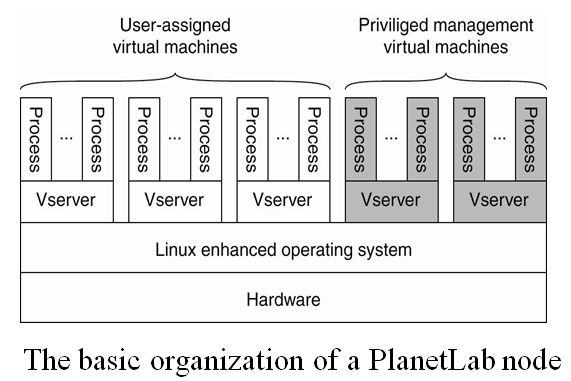




**DISTRIBUTED SERVERS**

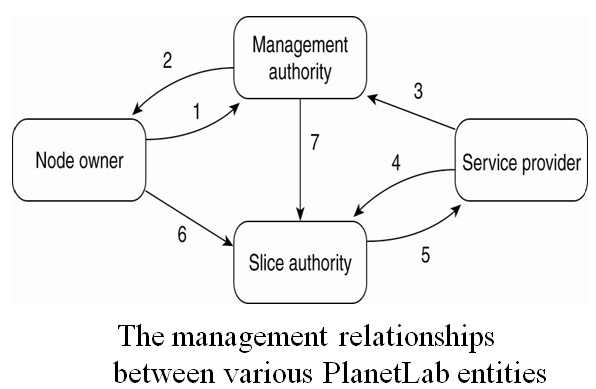
****

**MANAGING SERVER CLUSTERS**

****

**PlanetLab**

* PlanetLab management issues:
* Nodes belong to different organizations.
  + Each organization should be allowed to specify who is allowed to run applications on their nodes,
  + And restrict resource usage appropriately.
* Monitoring tools available assume a very specific combination of hardware and software.
  + All tailored to be used within a single organization.
* Programs from different slices but running on the same node should not interfere with each other.

****

* Relationships between PlanetLab entities:
* A node owner puts its node under the regime of a management authority, possibly restricting usage where appropriate.
* A management authority provides the necessary software to add a node to PlanetLab.
* A service provider registers itself with a management authority, trusting it to provide well-behaving nodes.
* A service provider contacts a slice authority to create a slice on a collection of nodes.
* The slice authority needs to authenticate the service provider.
* A node owner provides a slice creation service for a slice authority to create slices. It essentially delegates resource management to the slice authority.
* A management authority delegates the creation of slices to a slice authority.

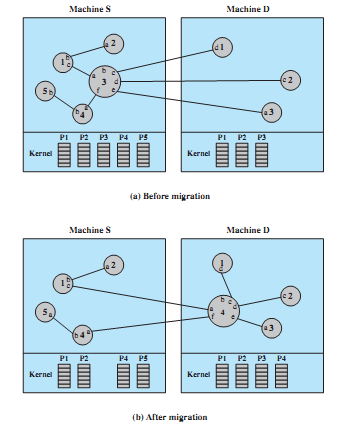
**PROCESS AND CODE MIGRATION**

* Under certain circumstances, in addition to the usual passing of data, passing code (even while it is executing) can greatly simplify the design of a DS.
* However, code migration can be inefficient and very costly.
* So, why migrate code?

**REASONS FOR MIGRATING CODE**

* Why? Biggest single reason: **better performance**.
* The big idea is to move a compute-intensive task from a heavily loaded machine to a lightly loaded machine   
  “on demand” and “as required”.

**Example of Process Migration**



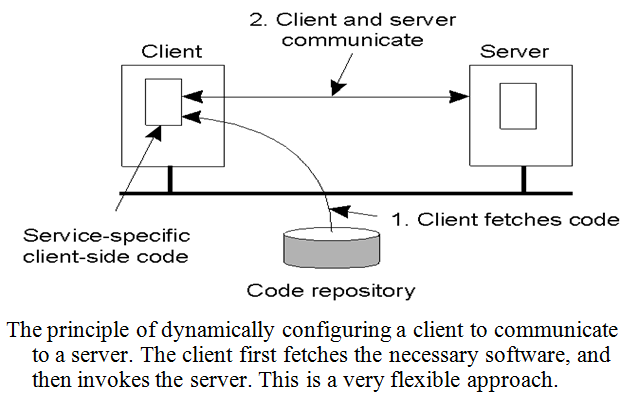
**CODE MIGRATION EXAMPLES**

* Moving (part of) a client to a server – processing data close to where the data resides. It is often too expensive to transport an entire database to a client for processing, so move the client to the data.
* Moving (part of) a server to a client –   
  checking data prior to submitting it to a server. The use of local error-checking (using JavaScript) on Web forms is a good example of this type of processing. Error-check the data close to the user, not at the server.

**“Classic” Code Migration Example**

* Searching the Web by “roaming”.
* Rather than search and index the Web by requesting the transfer of each and every document to the client for processing, the   
  client relocates to each site and indexes the documents it finds “in situ”. The index is then transported from site to site, in addition to the executing process.

**ANOTHER BIG ADVANTAGE: FLEXIBILITY**

****

**Major Disadvantage**

* **Security Concerns**.
* “Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard-disk and does not send the juiciest parts to heaven-knows-where may not always be such a good idea”.

**CODE MIGRATION MODELS**

**A RUNNING PROCESS CONSISTS OF THREE “SEGMENTS”:**

1. Code – instructions.

2. Resource – external references.

3. Execution – current state.

**MIGRATION IN HETEROGENEOUS SYSTEMS**

Three ways to handle migration (which can be combined):

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

**CODE MIGRATION CHARACTERISTICS**

**Weak Mobility**: just the code is moved – and it always restarts from its initial state.

* + e.g., Java Applets.
  + Comment: simple implementation, but limited applicability.

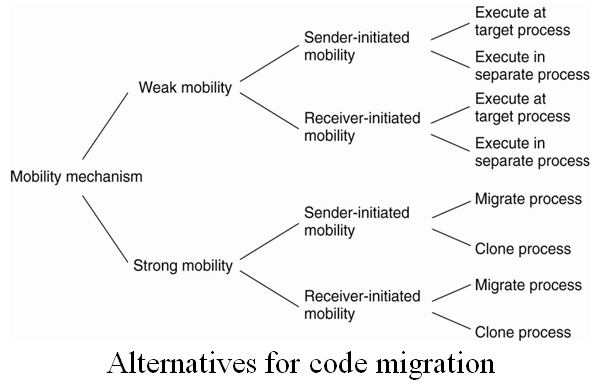
**Strong Mobility**: code & state is moved – and execution restarts from the next statement.

* + e.g., D’Agents.
  + Comment: very powerful, but hard to implement.
* Sender-Initiated vs. Receiver-Initiated.
* Which side of the communication starts the migration?
  + The machine currently executing the code (known as sender-initiated)
  + The machine that will ultimately execute   
    the code (known as receiver-initiated).

**HOW DOES THE MIGRATED CODE RUN?**

* Another issue surrounds where the migrated code executes:
  1. Within an existing process (possibly as a thread)
  2. Within it’s own (new) process space.
* Finally, strong mobility also supports the notion of “remote cloning”: an exact copy of the original process, but now running on a different machine.

**MODELS FOR CODE MIGRATION**

****

**What About Resources?**

* What makes code migration difficult is the requirement to migrate resources.
* Resources are the external references that a process is currently using, and includes (but is not limited to):
  + Variables, open files, network connections,   
    printers, databases, etc...

**TYPES OF PROCESS-TO-RESOURCE BINDING**

**1. Strongest**: Binding-by-Identifier (BI) – precisely the referenced resource, and nothing else, has to be migrated.

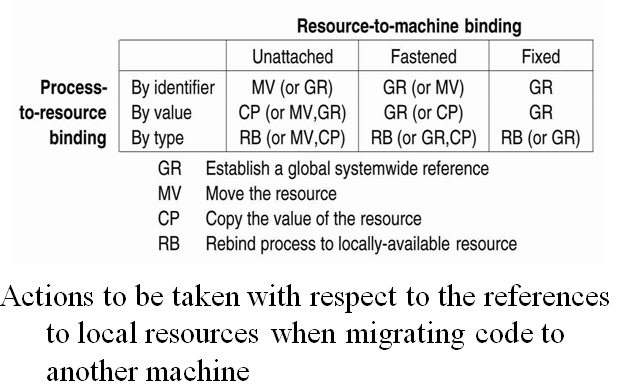
2. Binding-by-Value (BV) –   
weaker than BI, but only the value of the resource need be migrated.

**3. Weakest**: Binding-by-Type (BT) –   
nothing is migrated, but a resource of a specific type needs to be available after migration   
(e.g., a printer).

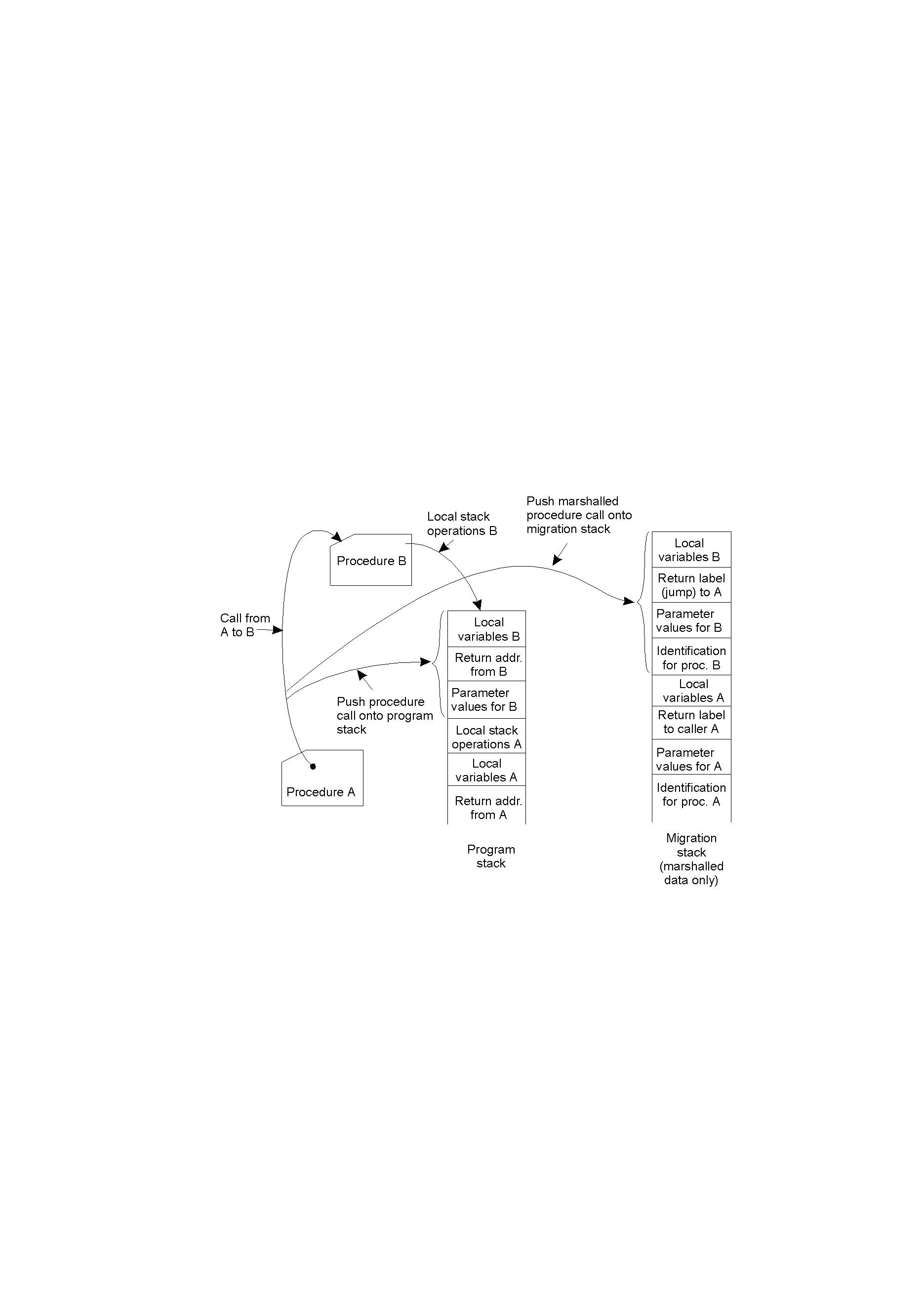
**MORE RESOURCE CLASSIFICATION**

* Resources are further distinguished as one of:
  1. **Unattached**: a resource that can be moved easily from machine to machine.
  2. **Fastened**: migration is possible, but at a high cost.
  3. **Fixed**: a resource is bound to a specific machine or environment, and cannot be migrated.
* Refer to following diagram for a good summary of resource-to-binding characteristics (to find out what to do with which resource when).

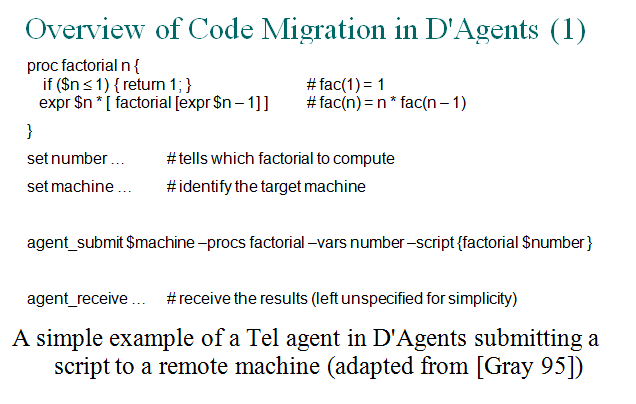
**MIGRATION AND LOCAL RESOURCES**

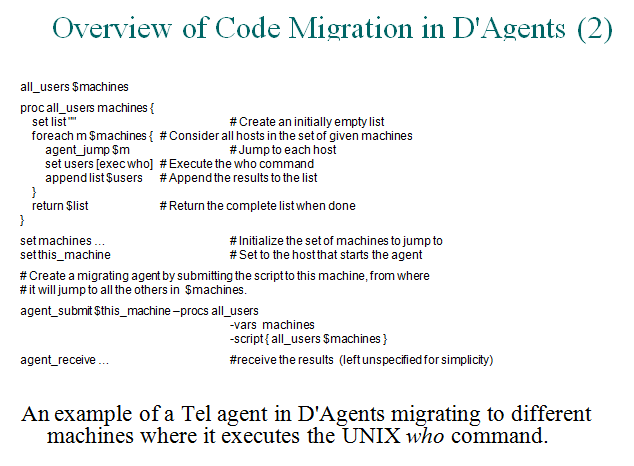


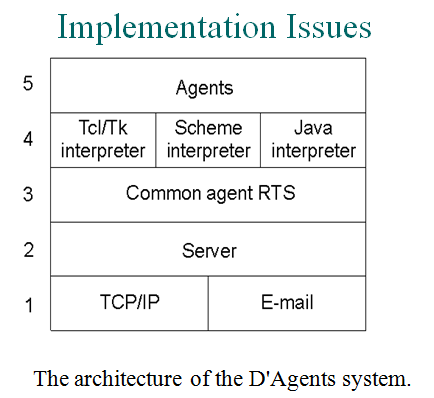
**Migration in Heterogeneous DS’s**

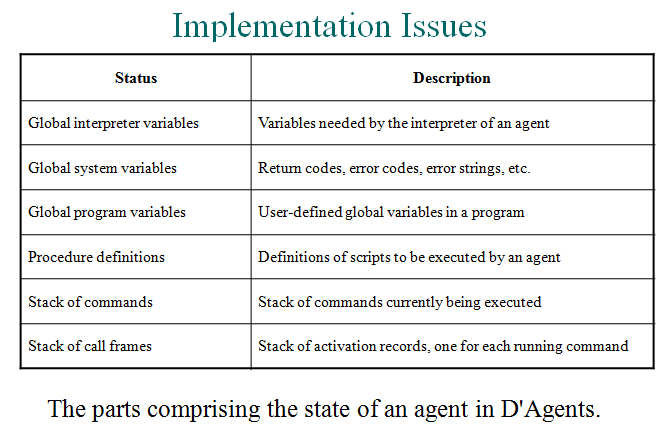
****

Using a migration stack: the principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment. Usually requires changes to the programming language and its environment.









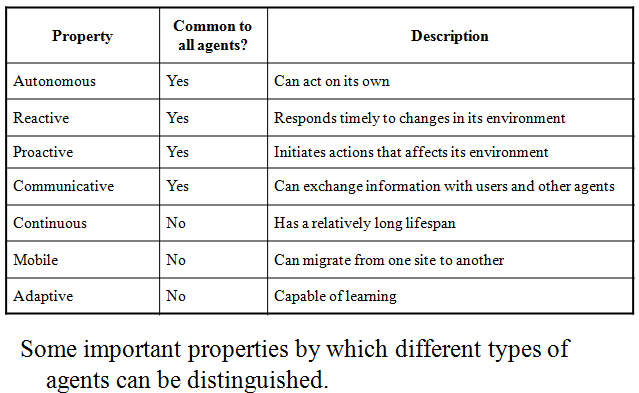
**SOFTWARE AGENTS**

* What is a software agent?
  + “An autonomous unit capable of performing a task in collaboration with other, possibly remote, agents”.
* The field of Software Agents is still immature, and much disagreement exists as to how to define what we mean by them.
* However, a number of types can be identified.

**TYPES OF SOFTWARE AGENTS**

1. Collaborative Agent – also known as “multi-agent systems”, which can work together to achieve a   
   common goal (e.g., planning a meeting).
2. Mobile Agent – code that can relocate and continue executing on a remote machine.
3. Interface Agent – software with “learning abilities” (that damned MS paperclip, and the ill-fated “bob”).
4. Information Agent – agents that are designed to collect and process geographically dispersed data and information.

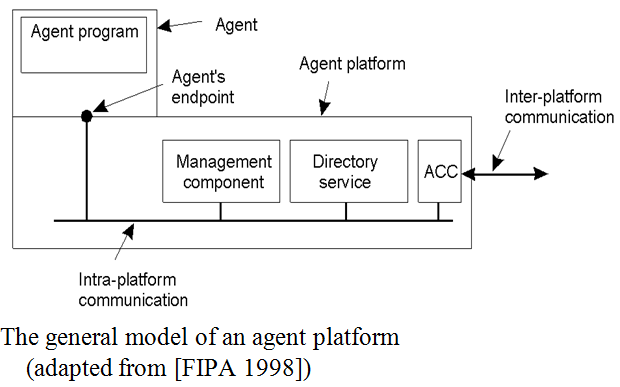
**Software Agents in Distributed Systems**

****

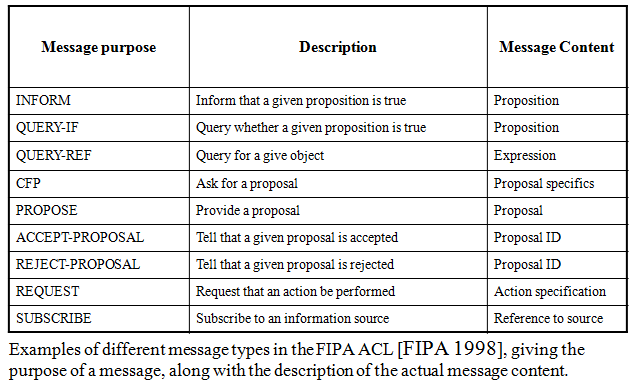
**Agent Technology – Standards**

* The general model of an agent platform has   
  been standardized by FIPA (“Foundation   
  for Intelligent Physical Agents”) located at [http://www.fipa.org](http://www.fipa.org/)
* Specifications include:
  + Agent Management Component.
  + Agent Directory Service.
  + Agent Communication Channel.
  + Agent Communication Language.

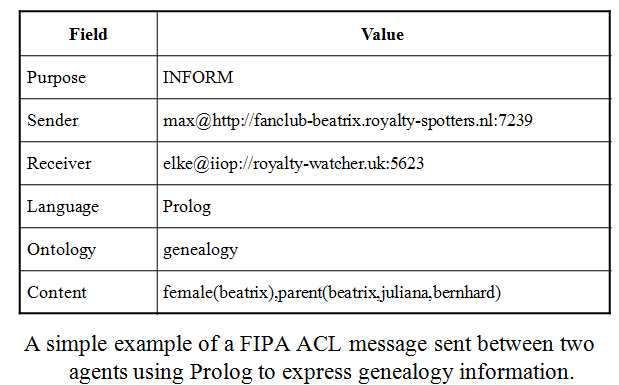
**AGENT TECHNOLOGY**



**AGENT COMMUNICATION LANGUAGES**



**AGENT COMMUNICATION LANGUAGES**



**Naming Entities**

A name in a distributed system is a string of bits or characters that is used to refer to an entity

Types of names

Address: an access point of an entity

Identifiers: a name that uniquely identifies an entity

An identifier refers to at most one entity

Each entity is referred to by at most one identifier

An identifier always refers to the same entity

Human-friendly names

Location-independent name: a name that is independent from its addresses

Name Spaces and Name Resolution

Names are organized into name spaces

A name space can be represented as a labeled, directed graph with two types of nodes

Leaf nodes and directory nodes

Absolute vs relative path names

Local names vs global names

Name Resolution: the process of looking up a name

Closure mechanism: knowing where and how to start name resolution

