Processes

*Processes

Communication takes place between *processes*.

But, what's a process?

"A program in execution"

Traditional operating systems: concerned with the "local" management and scheduling of processes.

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

There are three main areas of study:

1. Threads (within clients/servers).
2. Process and code migration.
3. Software agents.

Introduction to Threads

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

A programs *execution environment* is documentated in the *process table* and assigned a *PID*.

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

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

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

Threads can be thought of as an "execution of a part of a program (in user-space)".

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.

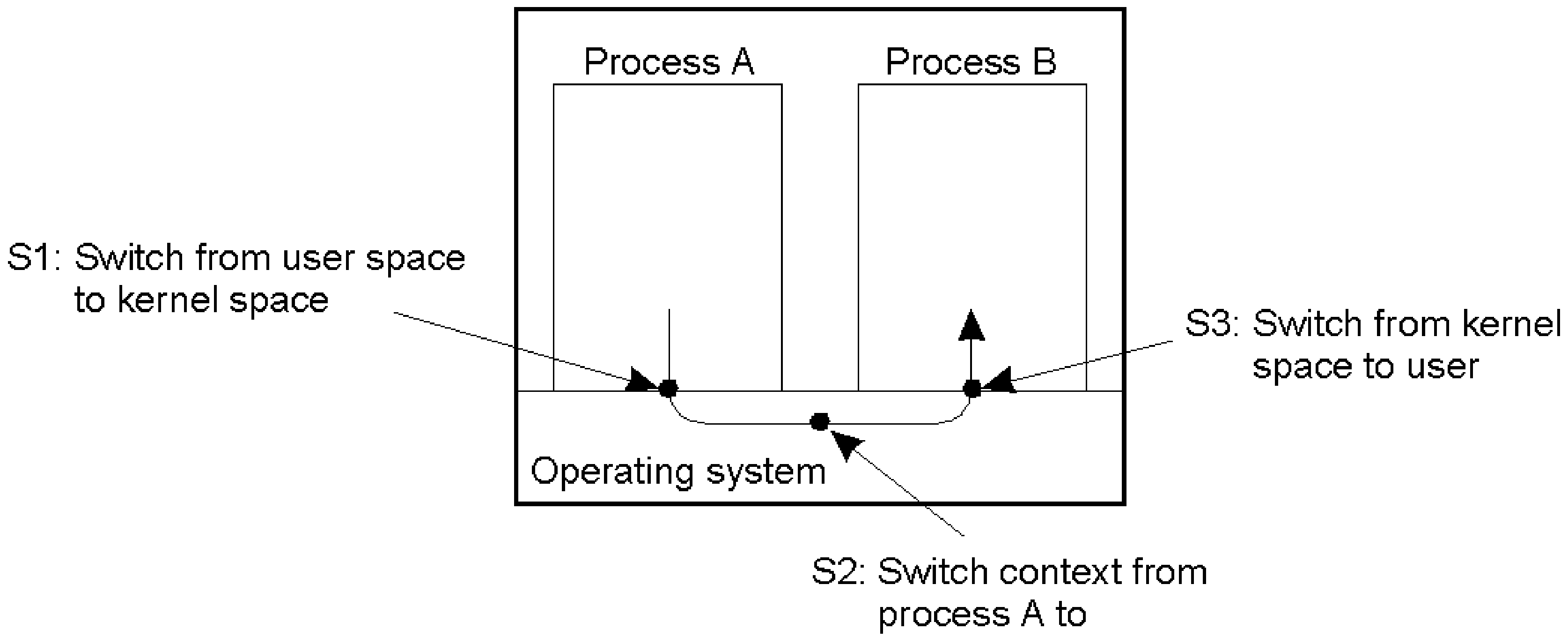
Two Important Implications

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

1. 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).

*The Cost of Context-Switches


Process A
Process B
S1: Switch from user space to kernel space
S3: Switch from kernel space to user
Operating system
S2: Switch context from process A to

Context switching as the result of IPC (Inter-process communications).

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

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

The 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.

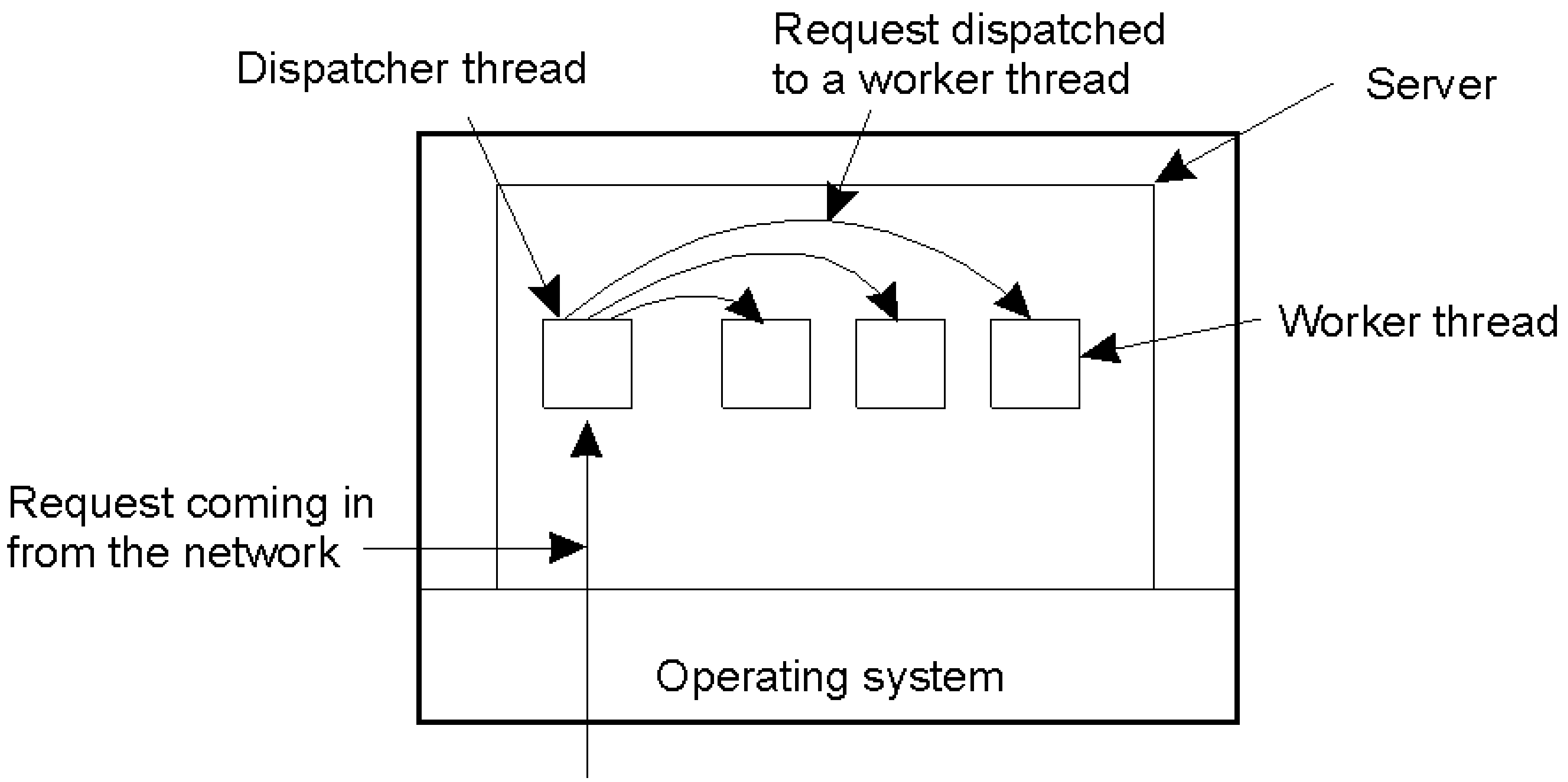
Example: MT-Servers

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 organise the server as a single “dispatcher” with multiple threaded “workers”, as diagrammed overleaf.

An Multi-Threaded Server


Dispatcher thread
Request dispatched to a worker thread
Server
Worker thread
Request coming in from the network
Operating system

A multithreaded server organized in a dispatcher/worker model.

More on Clients and Servers

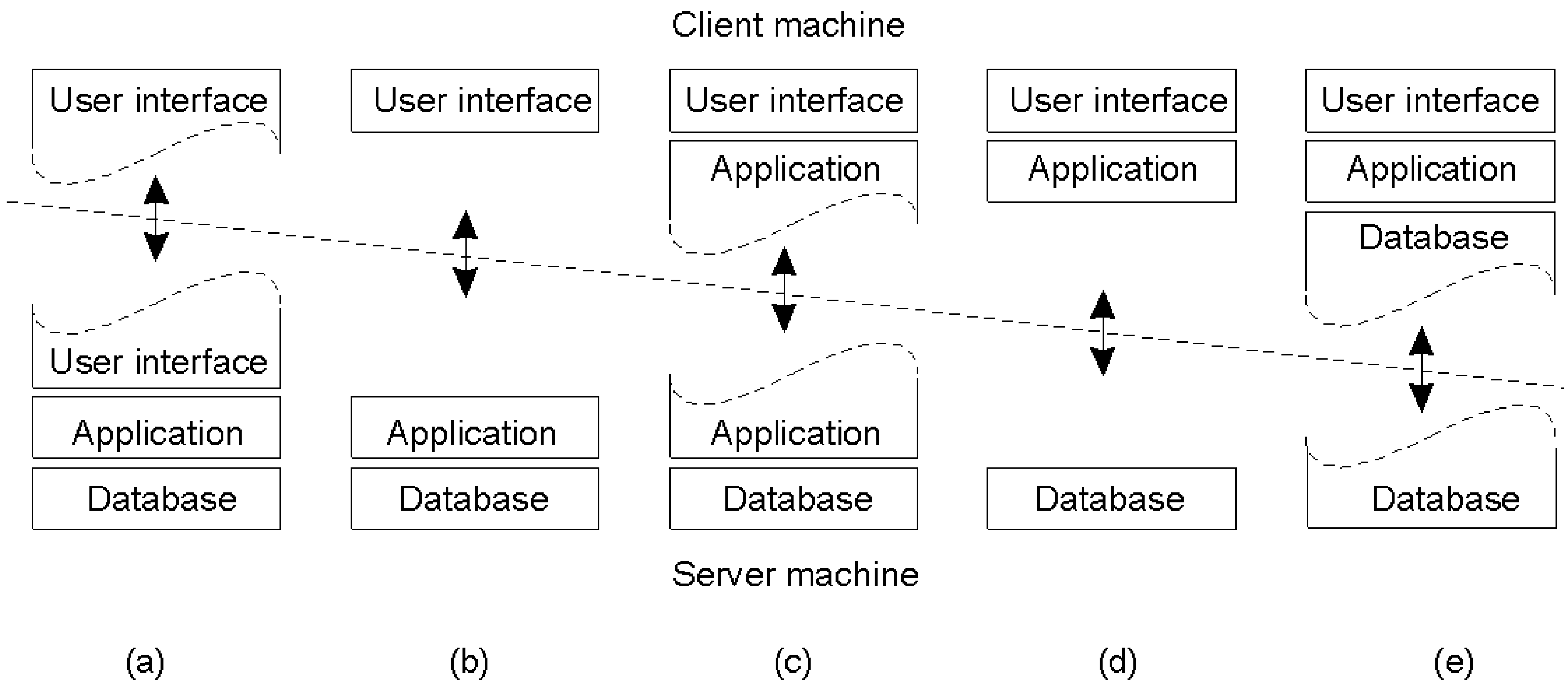
What's a client?

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 multitiered levels of the Client/Server architecture from earlier …

Multitiered Client Architectures



What’s a Server?

Definition: “A process that implements a specific service **on behalf of** a collection of clients”.

Typically, servers are organised to do one of two things:

1. Wait
2. Service

… wait … service … wait … service … wait …

Servers: Iterative and Concurrent

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*).

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*).

Problem: Identifying “end-points”?

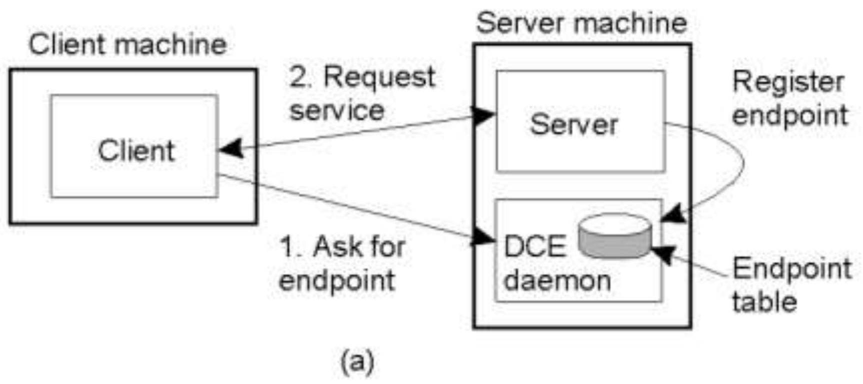
How do clients know which end-point (or port) to contact a server at? How do they “bind” to a server?

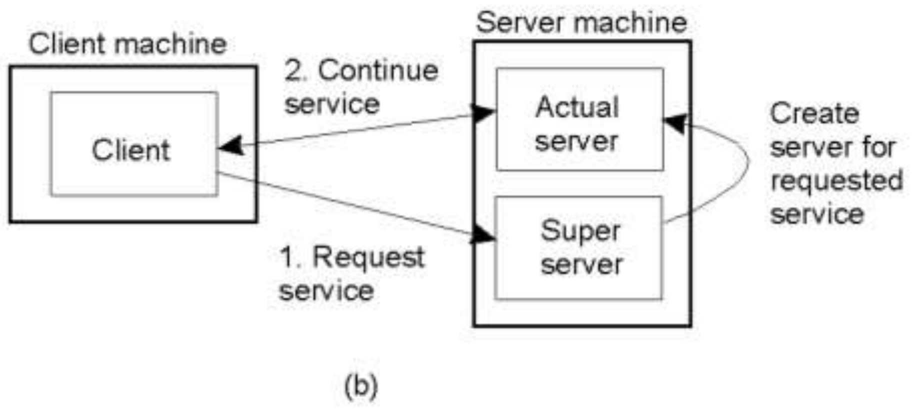
Statically assigned end-points (IANA).

Dynamically assigned end-points (DCE).

A popular variation:
the “super-server” (inetd on UNIX).

Servers: Binding to End-Points





a) Client-to-server binding using a daemon (DCE).
b) Client-to-server binding using a super-server (inetd on UNIX).

Server “States”

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.

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 the server knows the state of things). These are more **difficult** to implement.

But, what happens if something *crashes*?

(More on this later).

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".

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 distributed system.

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”.

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.

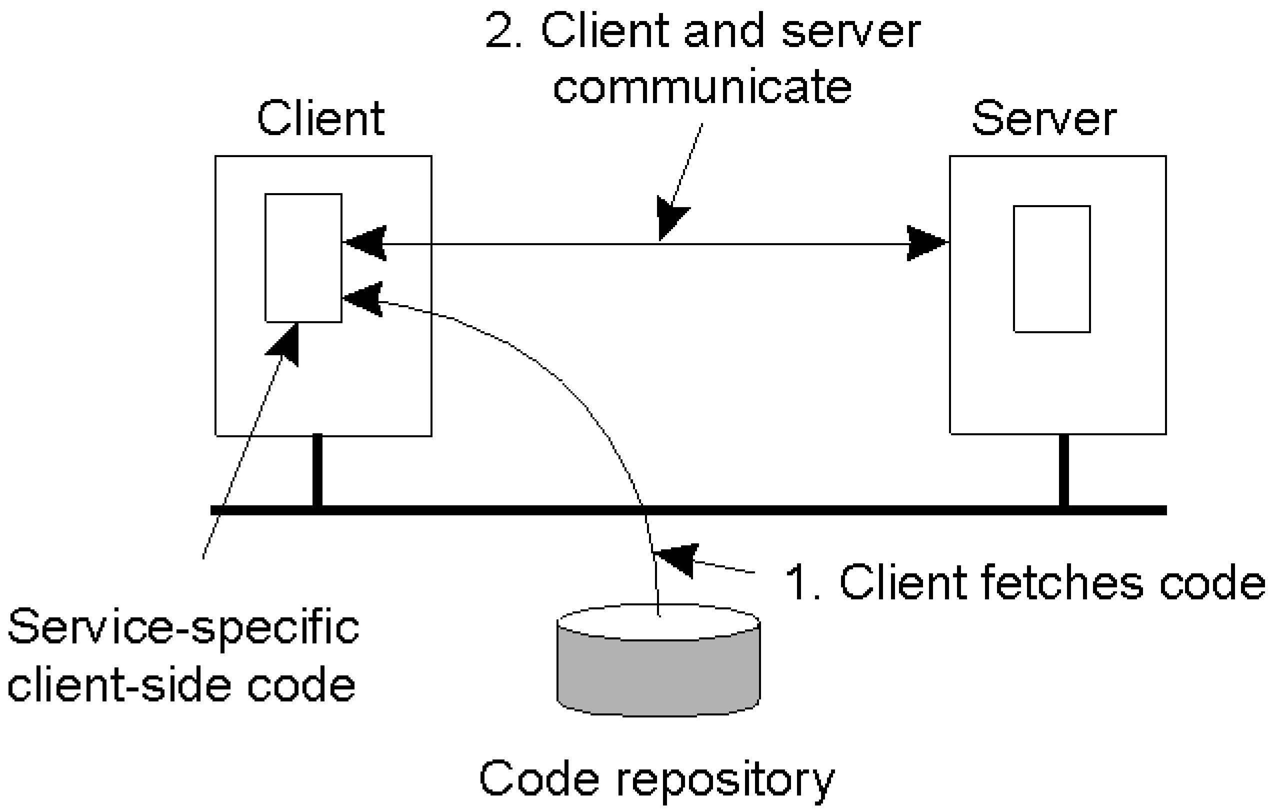
Moving (part of) a server to a client – checking data prior to submitting it to a server. The use of local error-checking (with JavaScript) on webforms is a good example of this type of processing. Error-check the data close to the user, not at the server.

“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


2. Client and server communicate
Client
Server
1. Client fetches code
Service-specific client-side code
Code repository

The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server. This is a very flexible approach.

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 “seqments”:

1. *Code* – instructions.

1. *Resource* – external references.

1. *Execution* – current state.

Code Migration Characteristics

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

Eg. Java Applets.

Comment: simple implementation, but limited applicability.

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

Eg. D'Agents.

Comment: very powerful, but hard to implement.

More Characteristics

Sender- vs. Receiver-Initiated.

Which side of the communication starts the migration?

The machine currently executing the code (known as *sender-initiated*)

or

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:

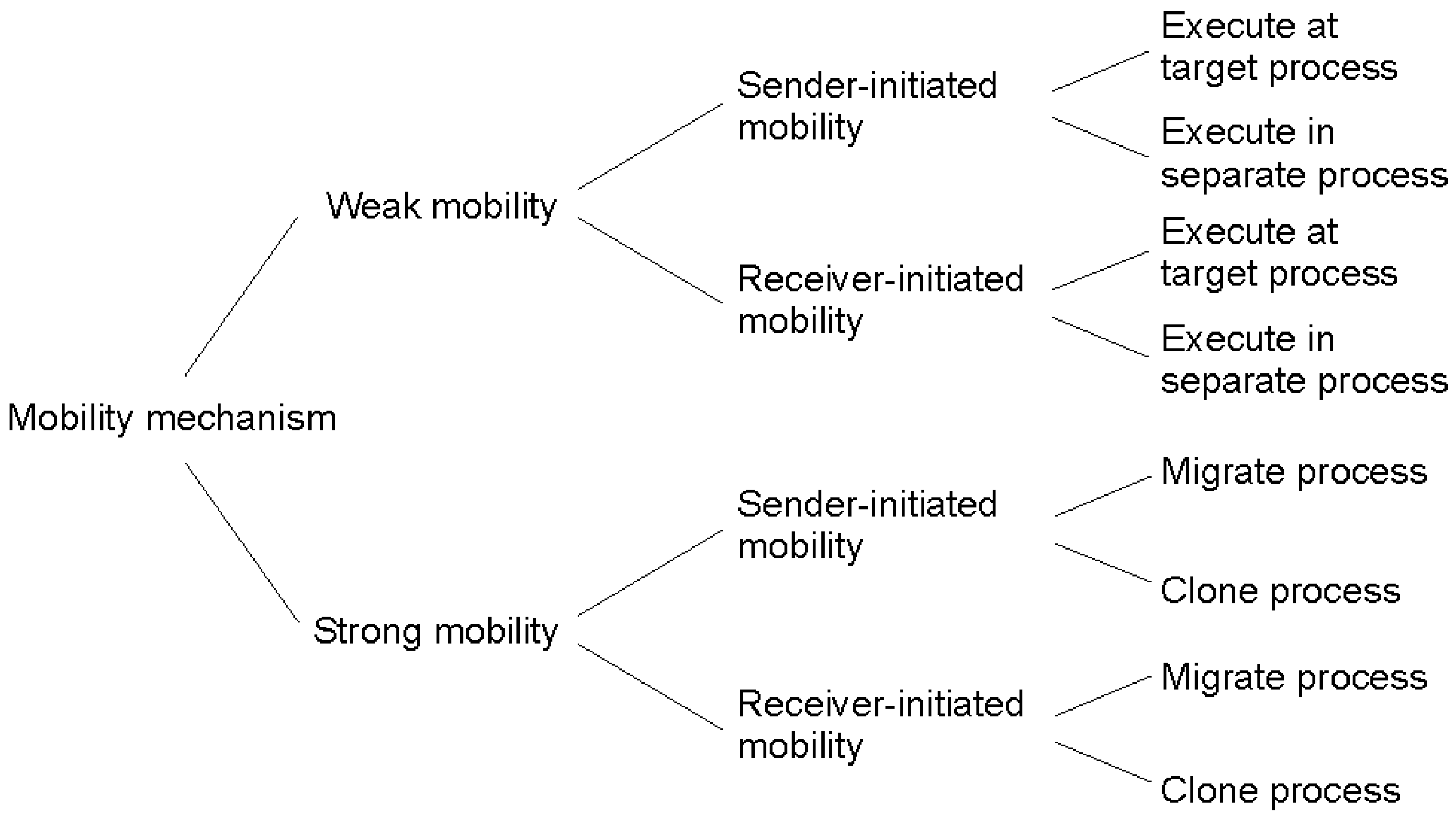
- Within an existing process (possibly as a thread)

or

- 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*.

The Models of Code Migration


Mobility mechanism
Weak mobility
Sender-initiated mobility
Execute at target process
Execute in separate process
Receiver-initiated mobility
Execute at target process
Execute in separate process
Strong mobility
Sender-initiated mobility
Migrate process
Clone process
Receiver-initiated mobility
Migrate process
Clone process

What About Resources?

This is *tricky*.

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., etc.

Types of Process-to-Resource Binding

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

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

Weakest: *binding-by-type* (BT) – nothing is migrated, but a resource of a specific type needs to be available after migration (eg, a printer).

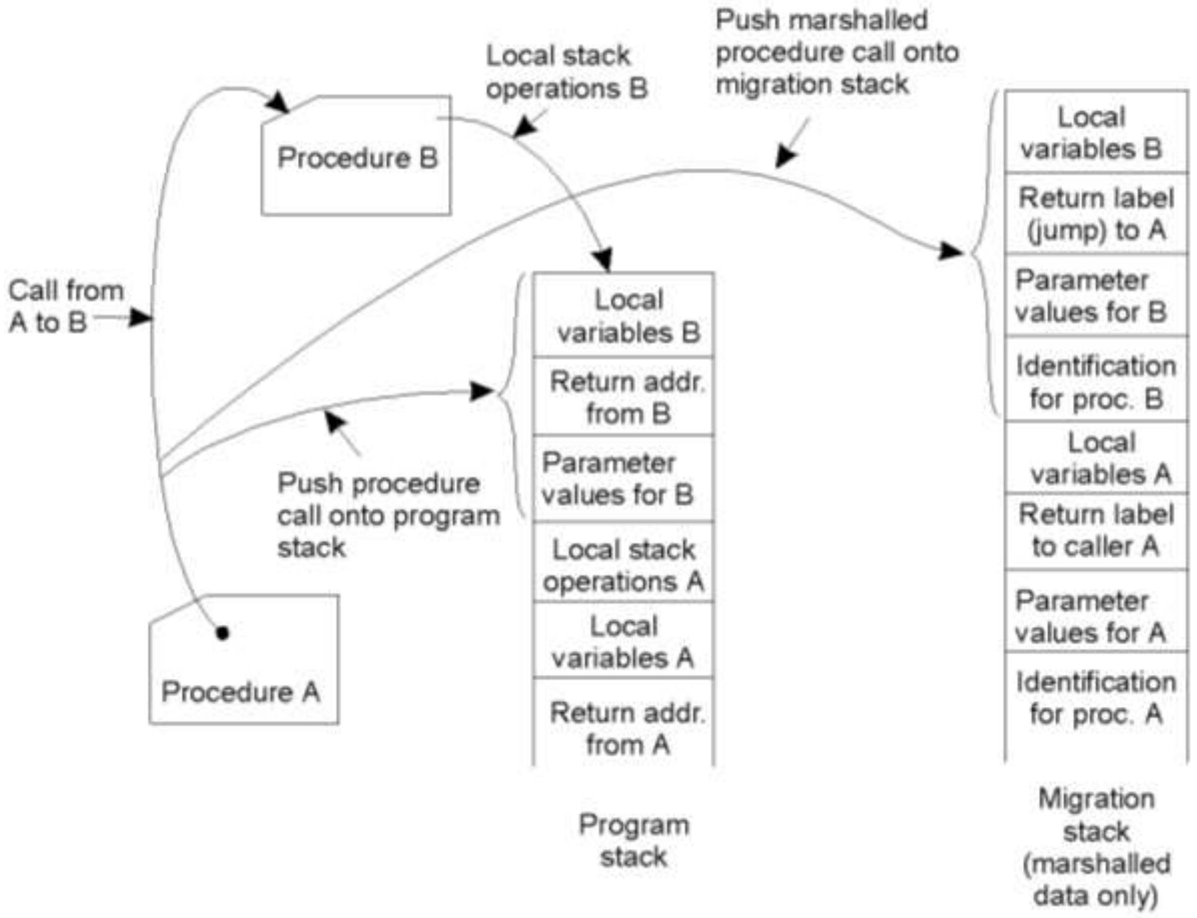
More Resource Classification

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 diagram 3-14 in the textbook for a good summary of resource-to-binding characteristics (to find out *what to do with which resource when*).

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 Agent

Collaborative Agent – also known as “multi-agent systems”, which can work together to achieve a common goal (eg, planning a meeting).

Mobile Agent – code that can relocate and continue executing on a remote machine.

Interface Agent – software with “learning abilities” (that *damned* MS paperclip, and the ill-fated “bob”).

Information Agent – agents that are designed to collect and process geographically dispersed data and information.

Software Agents in Distributed Systems

Property	Common to all agents?	Description
Autonomous	Yes	Can act on its own.
Reactive	Yes	Responds timely to changes in its environment.
Proactive	Yes	Initiates actions that affects its environment.
Communicative	Yes	Can exchange information with users and other agents.
Continuous	No	Has a relatively long lifespan.
Mobile	No	Can migrate from one site to another.
Adaptive	No	Capable of learning.

Some important properties by which different types of agents can be distinguished within the Distributed Systems world.

Agent Technology - Standards

The general model of an agent platform has been standardized by FIPA (The "Foundation for Intelligent Physical Agents") located at the http://www.fipa.org website.

Specifications include:

- Agent Management Component.
- Agent Directory Service.
- Agent Communication Channel.
- Agent Communication Language.

Summary - Processes

- Processes play a fundamental role in DS’s.
- Threads play a central role in building systems that don’t BLOCK when performing I/O – key requirement.
- The “classic” process organization model is client/server, and we looked at the various ways to organize the client and the server components.
- “Object Servers” are a special case.
- Processes can migrate from system-to-system: for performance and flexibility reasons.
- Although a simple, and easy to understand, idea, actually realizing this is not that simple (especially within heterogeneous environments).
- Standards are immature in this area, but are gaining support within the community (eg, FIPA).