ECE151 – Lecture 4

Chapter 3

Processes

Introduction to Threads

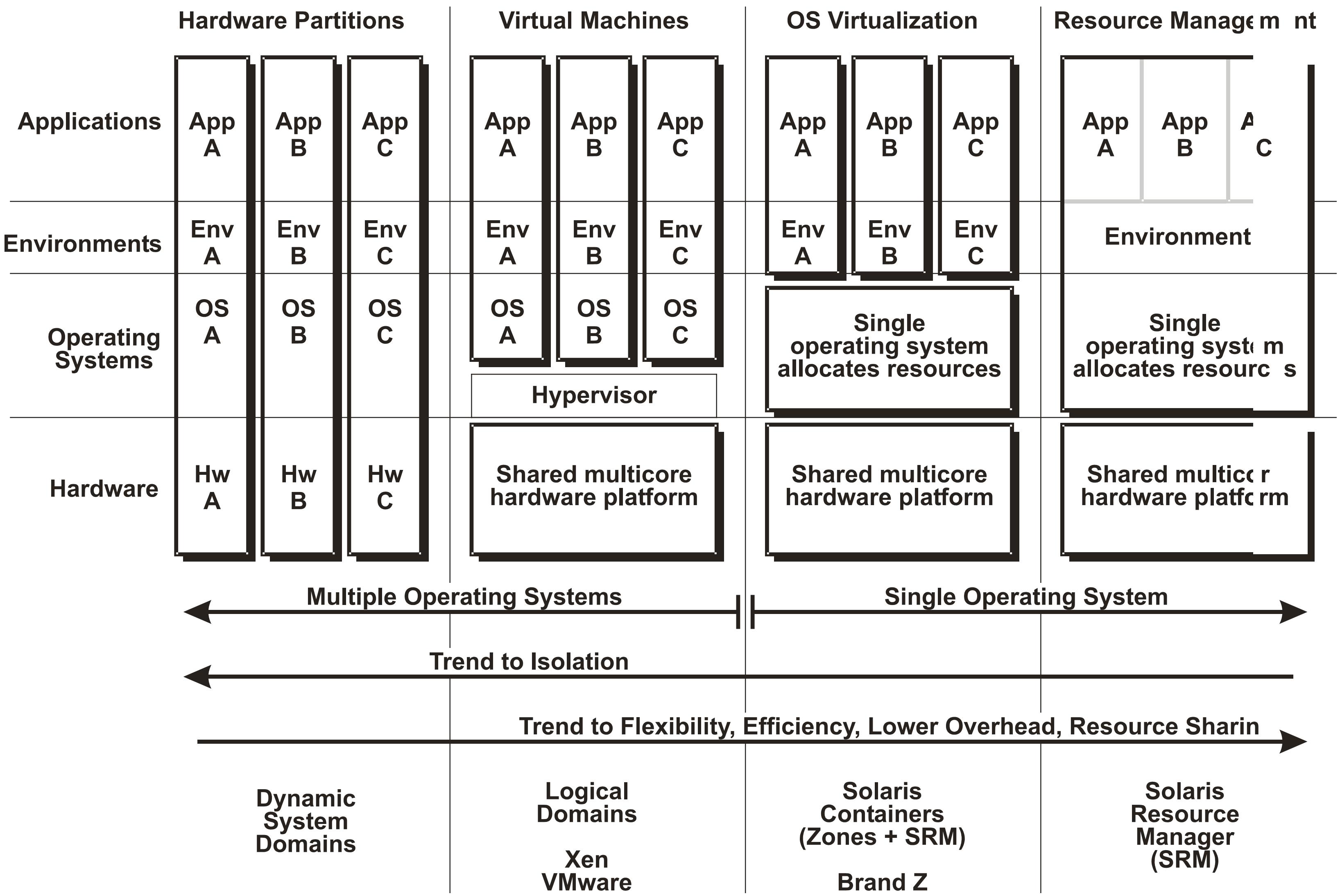
Basic idea: we build **virtual processors** in software, on top of physical processors:

Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.

Thread: A minimal software processor in whose **context** a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.

Process: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

Virtualization



Context Switching

Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).

Thread context: The minimal collection of values stored in registers and memory, used for the execution of a series of instructions (i.e., processor context, state).

Process context: The minimal collection of values stored in registers and memory, used for the execution of a thread (i.e., thread context, but now also at least MMU register values).

Context Switching

Observation 1: Threads share the same address space.

Thread context switching can be done entirely independent of the operating system.

Observation 2: Process switching is generally more expensive than thread switching; it involves getting the OS in the loop, i.e., trapping to the kernel.

Observation 3: Creating and destroying threads is much cheaper than doing so for processes.

Threads and Operating Systems

Main issue: Should an OS kernel provide threads, or should they be implemented as user-level packages?

User-space solution: We'll have nothing to do with the kernel, so all operations can be completely handled within a single process implementations can be extremely efficient.

All services provided by the kernel are done on behalf of the process in which a thread resides if the kernel decides to block a thread, the entire process will be blocked. Requires messy solutions.

In practice we want to use threads when there are lots of external events: threads block on a per-event basis if the kernel can't distinguish threads, how can it support signaling events to them.

Threads and Operating Systems

Kernel solution: The whole idea is to have the kernel contain the implementation of a thread package. This does mean that *all* operations return as system calls

Operations that block a thread are no longer a problem: the kernel schedules another available thread within the same process.

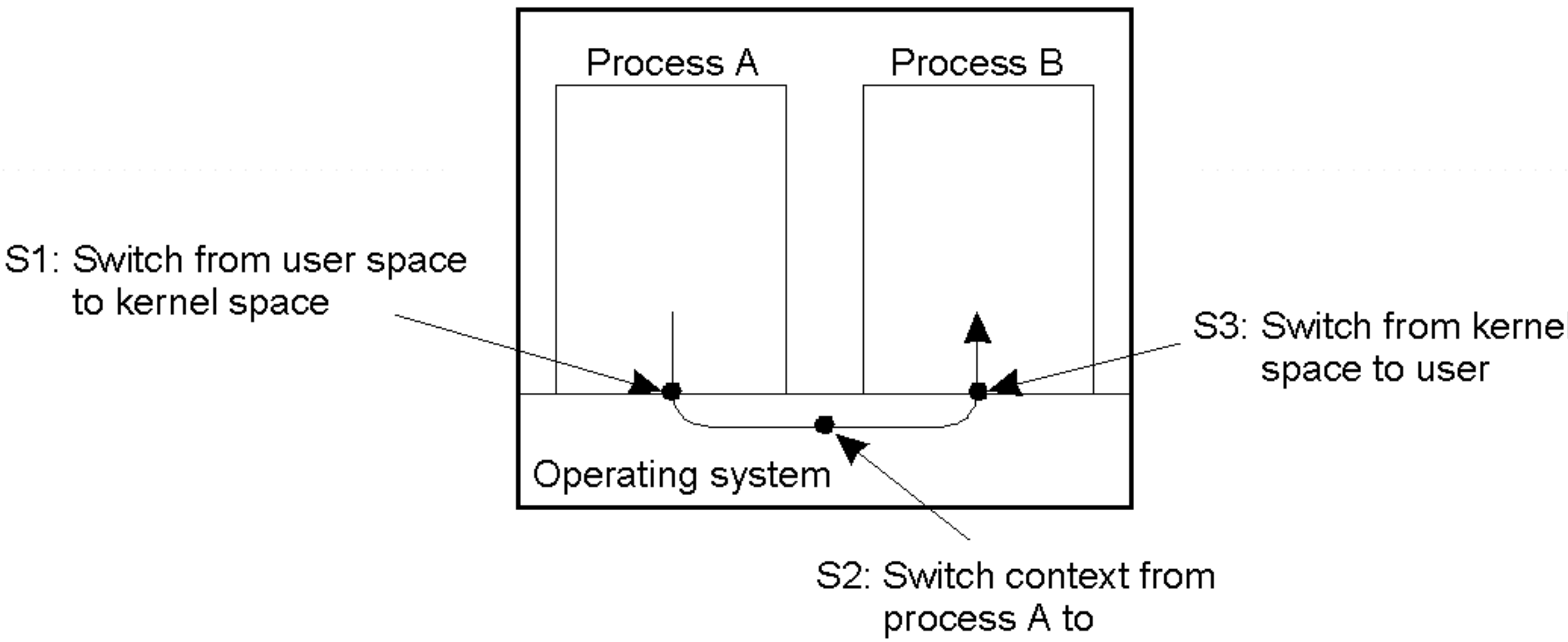
Handling external events is simple: the kernel (which catches all events) schedules the thread associated with the event.

The big problem is the loss of efficiency due to the fact that each thread operation requires a trap to the kernel.

Conclusion: Try to mix user-level and kernel-level threads into a single concept.

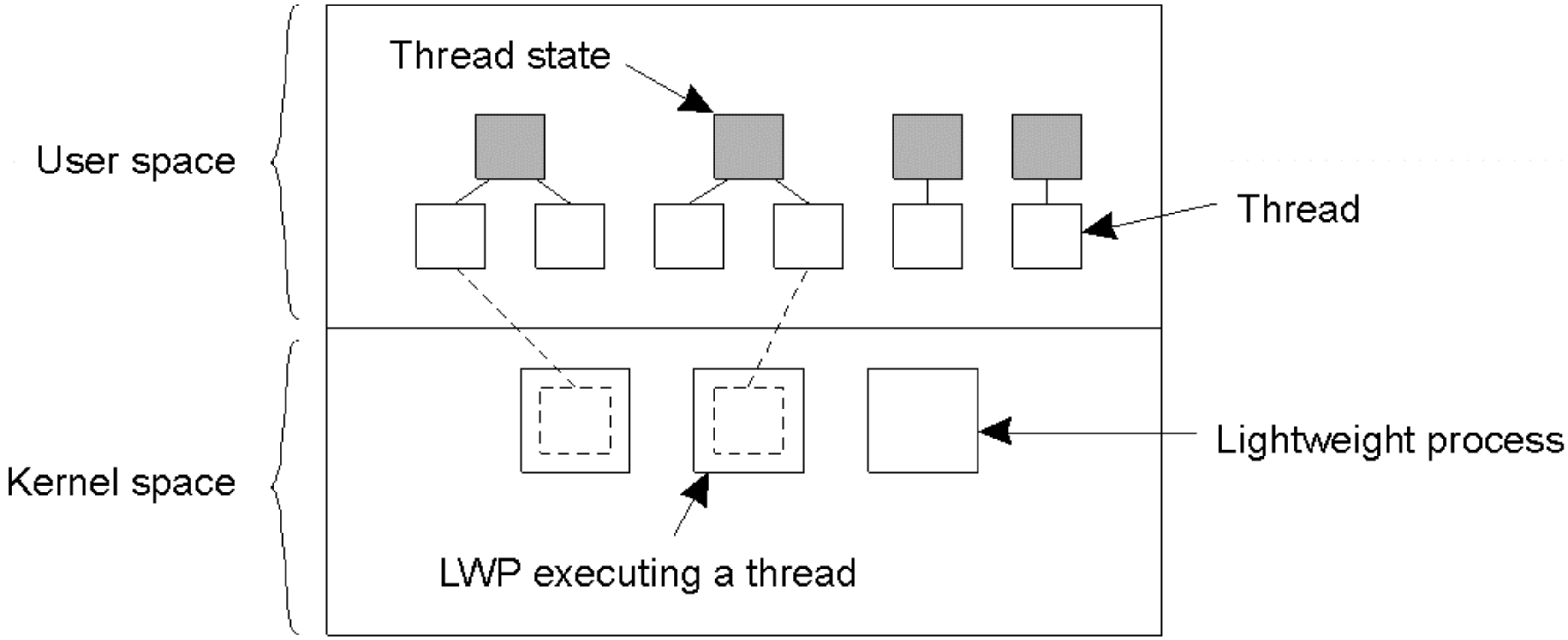
Thread Usage in Nondistributed Systems

Context switching as the result of IPC



Thread Implementation

Combining kernel-level lightweight processes and user-level threads.



Solaris Threads

When a user-level thread does a system call, the LWP that is executing that thread, blocks. The thread remains **bound** to the LWP.

The kernel can simply schedule another LWP having a runnable thread bound to it. Note that this thread can switch to *any* other runnable thread currently in user space.

When a thread calls a blocking user-level operation, we can simply do a context switch to a runnable thread, which is then bound to the same LWP.

When there are no threads to schedule, an LWP may remain idle, and may even be removed (destroyed) by the kernel.

Threads and Distributed Systems

Multithreaded clients: Main issue is hiding network latency

Multithreaded Web client:

Web browser scans an incoming HTML page, and finds that more files need to be fetched

Each file is fetched by a separate thread, each doing a (blocking) HTTP request

As files come in, the browser displays them

Multiple RPCs:

A client does several RPCs at the same time, each one by a different thread

It then waits until all results have been returned

Note: if RPCs are to different servers, we may have a linear speed-up compared to doing RPCs one after the other

Threads and Distributed Systems

Multithreaded servers: Main issue is improved performance and better structure

Improve performance:

Starting a thread to handle an incoming request is *much* cheaper than starting a new process

Having a single-threaded server prohibits simply scaling the server to a multiprocessor system

As with clients: hide network latency by reacting to next request while previous one is being replied

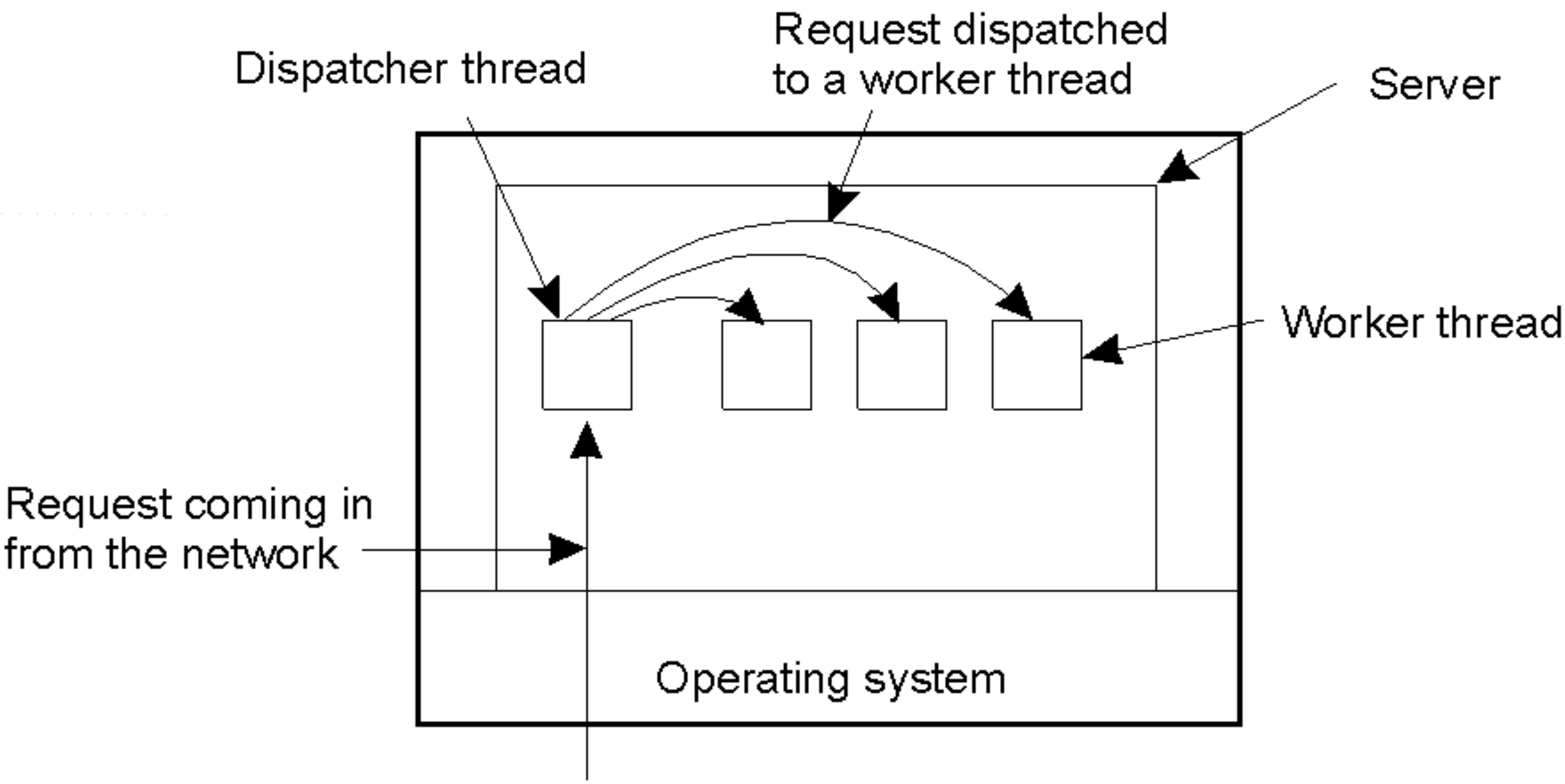
Better structure:

Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the overall structure

Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control

Multithreaded Servers (1)

A multithreaded server organized in a dispatcher/worker model.



Multithreaded Servers (2)

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Three ways to construct a server.

User Interfaces

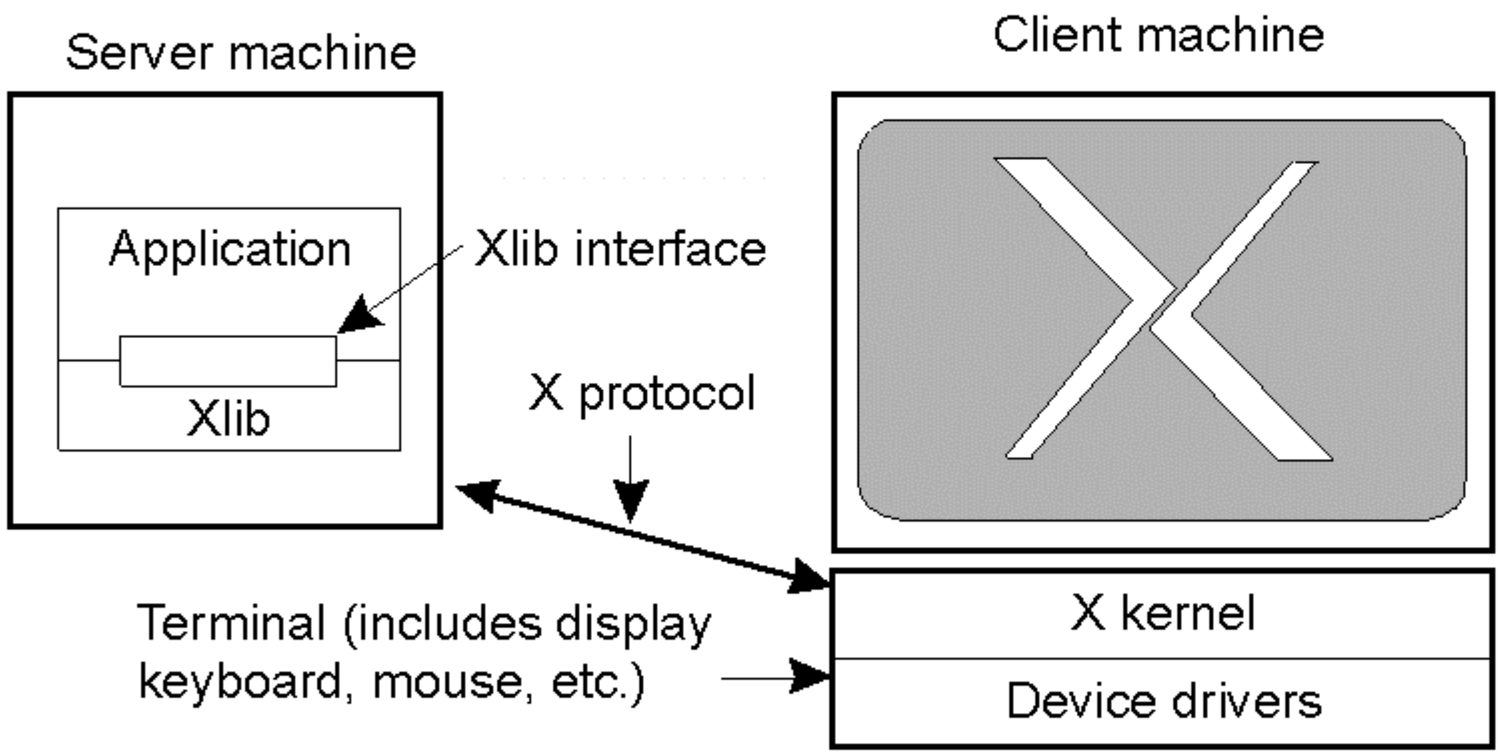
Essence: A major part of client-side software is focused on (graphical) user interfaces.

Compound documents: Make the user interface application-aware to allow interapplication communication:

drag-and-drop: move objects to other positions on the screen, possibly invoking interaction with other applications

in-place editing: integrate several applications at user-interface level (word processing + drawing facilities

The X-Window System



Client-Side Software

Essence: Often focused on providing distribution transparency

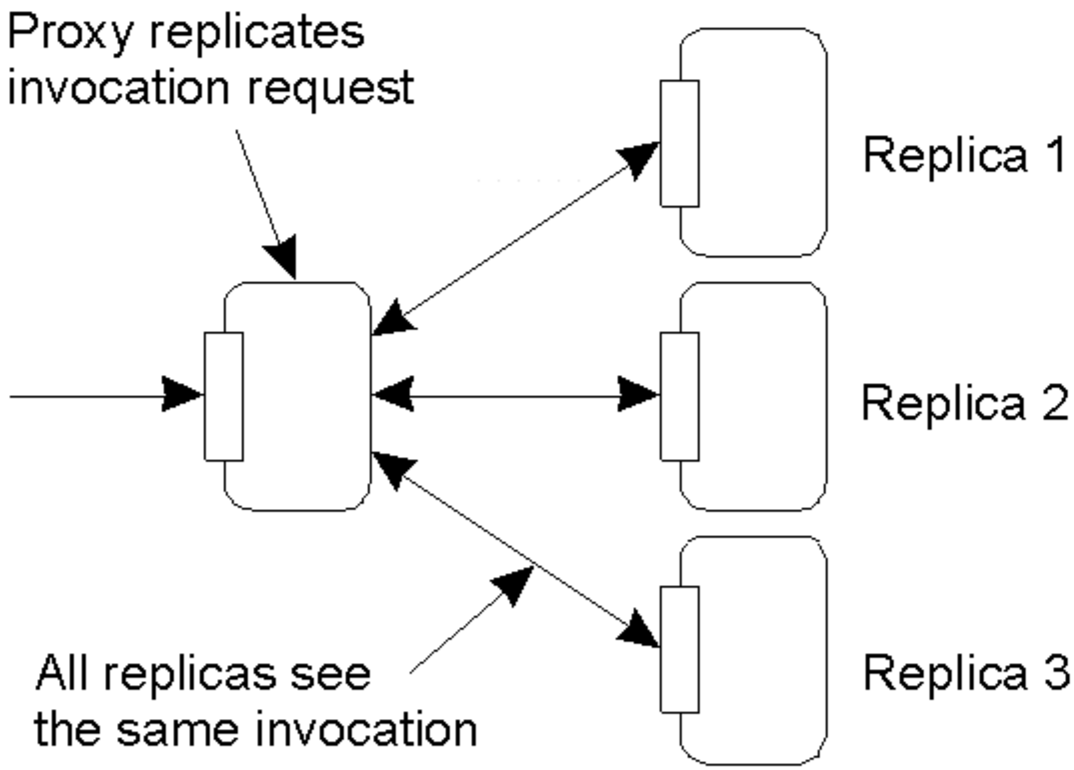
access transparency: client-side stubs for RPCs and RMIs

location/migration transparency: let client-side software keep track of actual location

replication transparency: multiple invocations handled by client stub:

failure transparency: can often be placed only at client (we're trying to mask server and communication failures).

Client-Side Software for Distribution Transparency



A possible approach to transparent replication of a remote object using a client-side solution.

Servers

Basic model: A server is a process that waits for incoming service requests at a specific transport address.

In practice, there is a one-to-one mapping between a port and a service:

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
	24	any private mail system
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)
courier	530	Xerox RPC

Servers

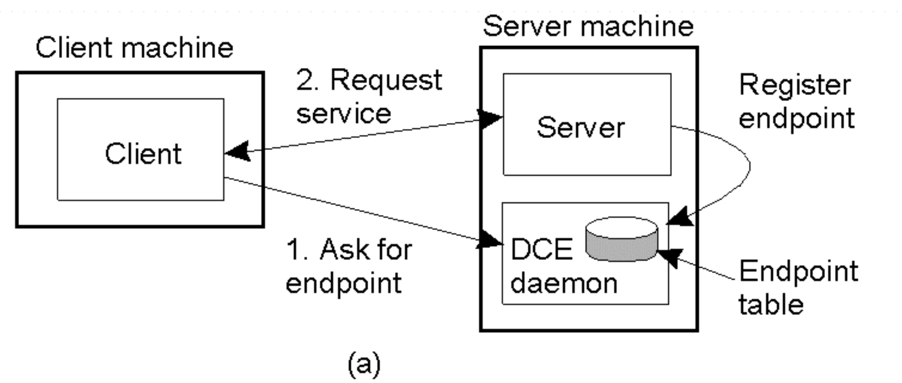
Superservers: Servers that listen to several ports, i.e., provide several independent services. In practice, when a service request comes in, they start a subprocess to handle the request (UNIX)

Iterative vs. concurrent servers: Iterative servers can handle only one client at a time, in contrast to concurrent servers

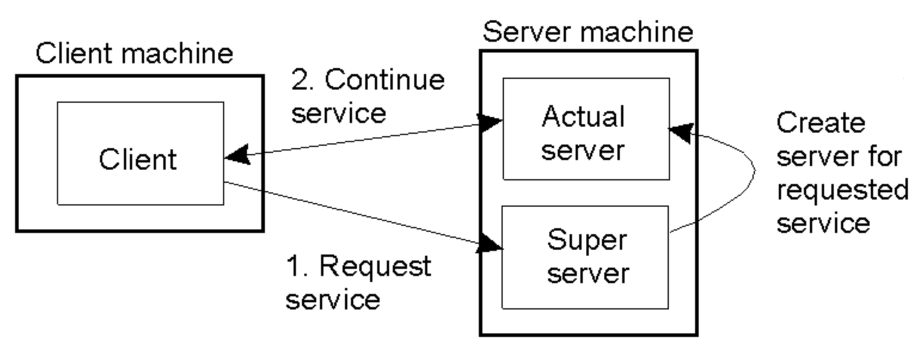
Servers: General Design Issues

a) Client-to-server binding using a daemon as in DCE

b) Client-to-server binding using a superserver as in UNIX



(a)



(b)

Out-of-Band Communication

Issue: Is it possible to *interrupt* a server once it has accepted (or is in the process of accepting) a service request?

Solution 1: Use a separate port for urgent data (possibly per service request):

- Server has a separate thread (or process) waiting for incoming urgent messages
- When urgent message comes in, associated request is put on hold
- Note: we require OS supports high-priority scheduling of specific threads or processes

Solution 2: Use out-of-band communication facilities of the transport layer:

- Example: TCP allows to send urgent messages in the same connection
- Urgent messages can be caught using OS signaling techniques

Servers and State

Stateless servers: Never keep *accurate* information about the status of a client after having handled a request:

- Don't record whether a file has been opened (simply close it again after access)
- Don't promise to invalidate a client's cache
- Don't keep track of your clients

Consequences:

- Clients and servers are completely independent
- State inconsistencies due to client or server crashes are reduced
- Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Question: Does connection-oriented communication fit into a stateless design?

Servers and State

Stateful servers: Keeps track of the status of its clients:

Record that a file has been opened, so that prefetching can be done

Knows which data a client has cached, and allows clients to keep local copies of shared data

Observation: The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is not a major problem.

Object Servers

Servant: The actual implementation of an object, sometimes containing only method implementations:

Collection of C or COBOL functions, that act on structs, records, database tables, etc.

Java or C++ classes

Skeleton: Server-side stub for handling network I/O:

Unmarshalls incoming requests, and calls the appropriate servant code

Marshalls results and sends reply message

Generated from interface specifications

Object adapter: The “manager” of a set of objects:

Inspects (as first) incoming requests

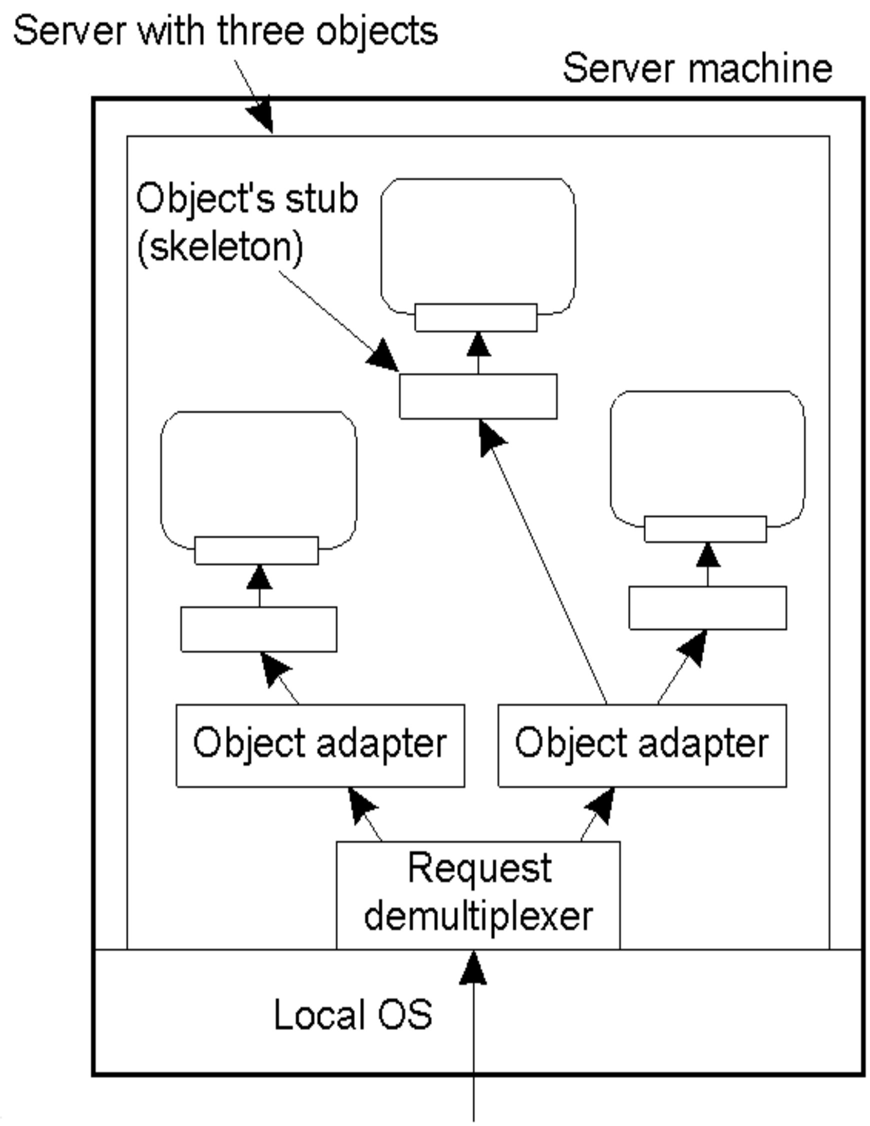
Ensures referenced object is activated (requires identification of servant)

Passes request to appropriate skeleton, following specific **activation policy**

Responsible for generating **object references**

Object Adapter

Organization of an object server supporting different activation policies.



Object Adapter

```
/* Definitions needed by caller of adapter and adapter */
#define TRUE
#define MAX_DATA 65536

/* Definition of general message format */
struct message {
    long  source                    /* senders identity */
    long  object_id;                /* identifier for the requested object */
    long  method_id;                /* identifier for the requested method  */
    unsigned  size;                 /* total bytes in list of parameters */
    char  **data;                   /* parameters as sequence of bytes */
};

/* General definition of operation to be called at skeleton of object */
typedef void (*METHOD_CALL)(unsigned, char* unsigned*, char**);

long register_object (METHOD_CALL call);          /* register an object  */
void unrigester_object (long object)id);          /* unrigester an object */
void invoke_adapter (message *request);           /* call the adapter */
```

The *header.h* file used by the adapter and any program that calls an adapter.

Object Adapter

```
typedef struct thread THREAD;                    /* hidden definition of a thread */

thread *CREATE_THREAD (void (*body)(long tid), long thread_id);
/* Create a thread by giving a pointer to a function that defines the actual  */
/* behavior of the thread, along with a thread identifier  */

void get_msg (unsigned *size, char **data);
void put_msg(THREAD *receiver, unsigned size, char **data);
/* Calling get_msg blocks the thread until of a message has been put into its  */
/* associated buffer.  Putting a message in a thread's buffer is a nonblocking */
/* operation. */
```

The *thread.h* file used by the adapter for using threads.

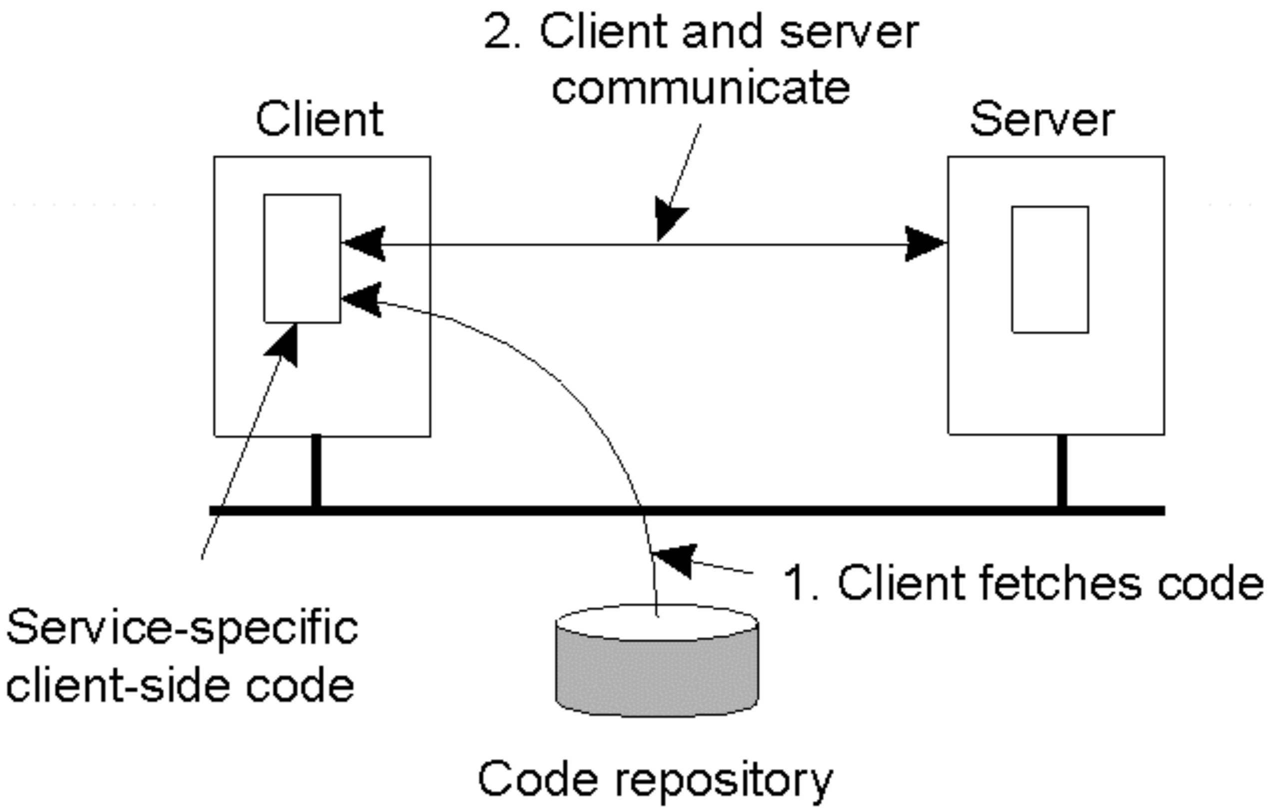
Object Adapter

The main part of an adapter that implements a thread-per-object policy.

```
#include <header.h>
#include <thread.h>
#define MAX_OBJECTS          100
#define NULL                 0
#define ANY                  -1

METHOD_CALL invoke[MAX_OBJECTS];       /* array of pointers to stubs     */
THREAD *root;                          /* demultiplexer thread           */
THREAD *thread[MAX_OBJECTS];           /* one thread per object          */

void thread_per_object(long object_id) {
    message        *req, *res;         /* request/response message       */
    unsigned       size;               /* size of messages               */
    char           **results;          /* array with all results         */

    while(TRUE) {
        get_msg(&size, (char*) &req);  /* block for invocation request   */

        /* Pass request to the appropriate stub. The stub is assumed to  */
        /* allocate memory for storing the results.                       */
        (invoke[object_id]*)(req→size, req→data, &size, results);

        res = malloc(sizeof(message)+size); /* create response message   */
        res→object_id = object_id;     /* identify object                */
        res→method_id = req.method_id; /* identify method                */
        res→size = size;               /* set size of invocation results */
        memcpy(res→data, results, size); /* copy results into response  */
        put_msg(root, sizeof(res), res); /* append response to buffer    */
        free(req);                     /* free memory of request         */
        free(*results);                /* free memory of results         */
    }
}

void invoke_adapter(long oid, message *request) {
    put_msg(thread[oid], sizeof(request), request);
}
```

Reasons for Migrating Code

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



Strong and Weak Mobility

Object components:

Code segment: contains the actual code

Data segment: contains the state

Execution state: contains context of thread executing the object's code

Weak mobility: Move only code and data segment (and start execution from the beginning) after migration:

Relatively simple, especially if code is portable

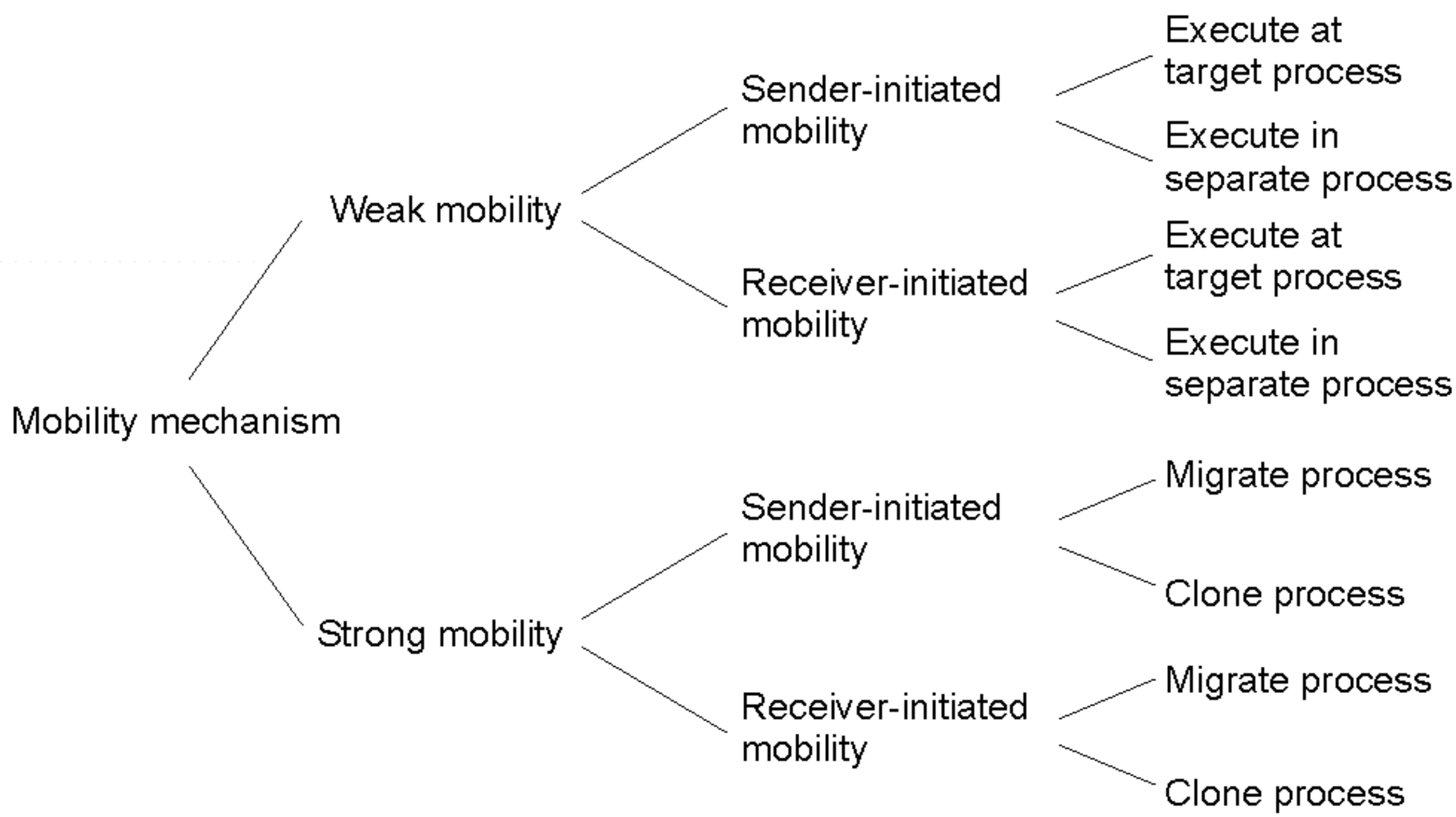
Distinguish code shipping (push) from code fetching (pull)

Strong mobility: Move component, including execution state

Migration: move the entire object from one machine to the other

Cloning: simply start a clone, and set it in the same execution state.

Models for Code Migration



Managing Local Resources

Problem: An object uses local resources that may or may not be available at the target site.

Resource types:

Fixed: the resource cannot be migrated, such as local hardware

Fastened: the resource can, in principle, be migrated but only at high cost

Unattached: the resource can easily be moved along with the object (e.g. a cache)

Object-to-resource binding:

By identifier: the object requires a specific instance of a resource (e.g. a specific database)

By value: the object requires the value of a resource (e.g. the set of cache entries)

By type: the object requires that only a type of resource is available (e.g. a color monitor)

Migration and Local Resources

Resource-to machine binding

		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV, GR)	GR (or CP)	GR
	By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

Actions to be taken with respect to the references to local resources when migrating code to another machine.

Migration in Heterogenous Systems

Main problem:

The target machine may not be suitable to execute the migrated code

The definition of process/thread/processor context is highly dependent on local hardware, operating system and runtime system

Only solution: Make use of an abstract machine that is implemented on different platforms

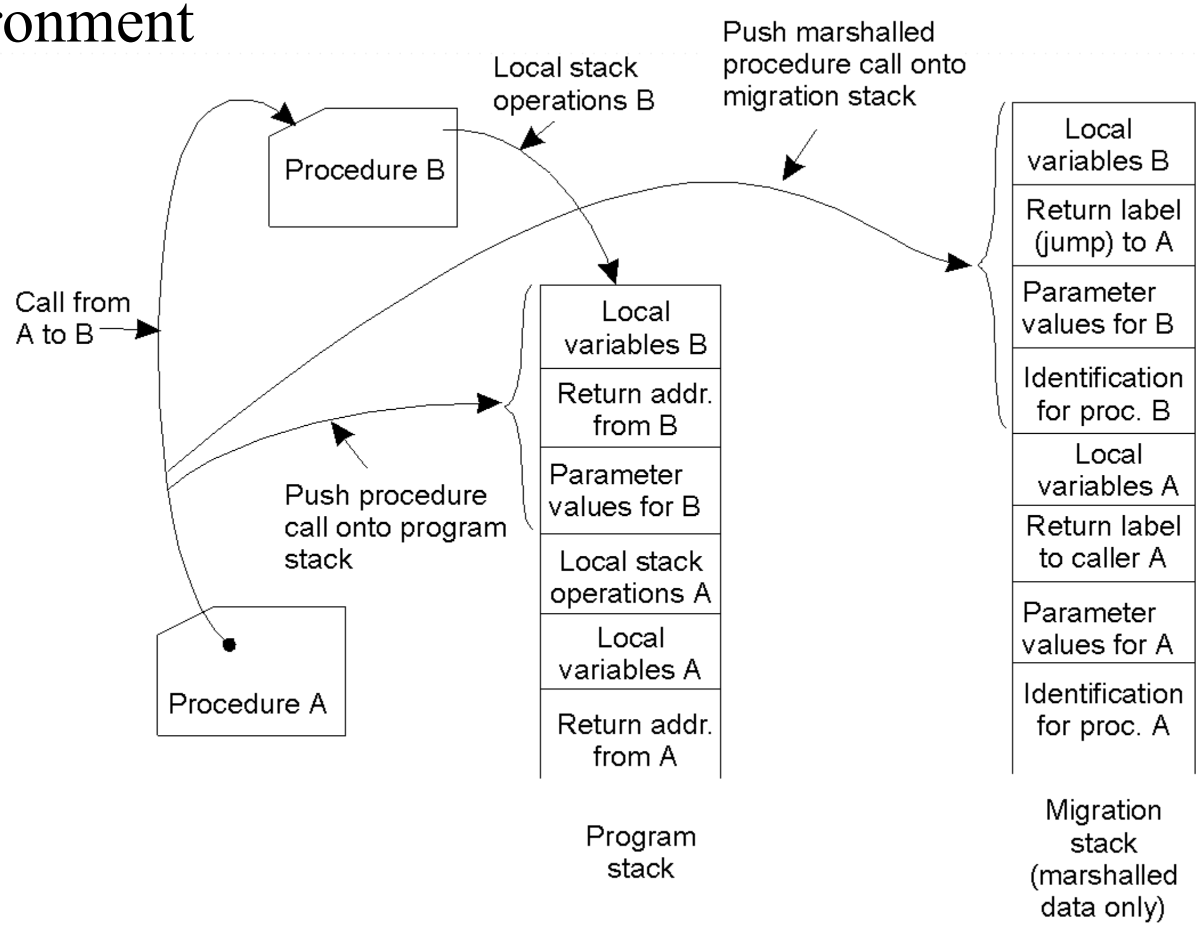
Current solutions:

Interpreted languages running on a virtual machine (Java/JVM; scripting languages)

Existing languages: allow migration at specific "transferable" points, such as just before a function call.

Migration in Heterogeneous Systems

The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment



What's an Agent?

Definition: An autonomous process capable of reacting to, and initiating changes in its environment, possibly in collaboration with users and other agents

collaborative agent: collaborate with others in a multiagent system

mobile agent: can move between machines

interface agent: assist users at user-interface level

information agent: manage information from physically different sources

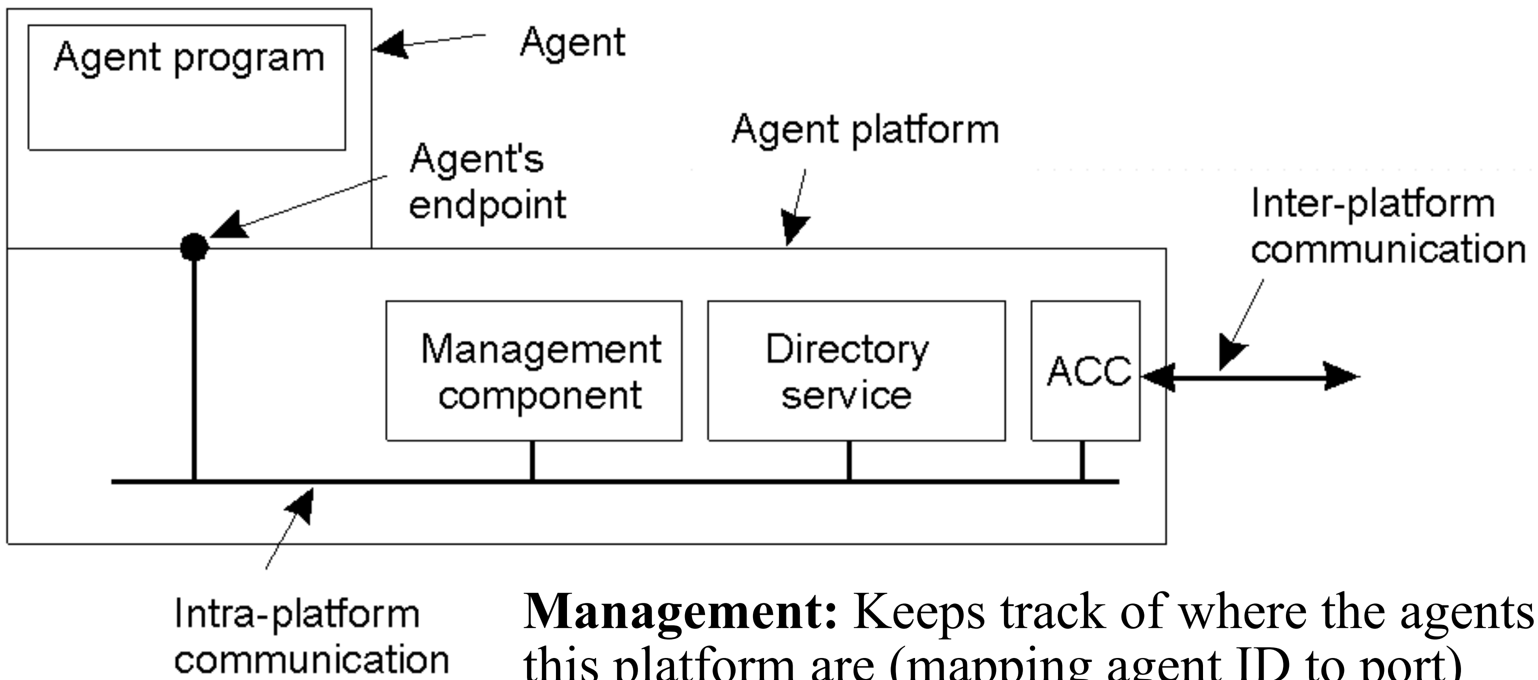
Software Agents in Distributed Systems

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.

Agent Technology

The general model of an agent platform (adapted from [fipa98-mgt]).



Management: Keeps track of where the agents on this platform are (mapping agent ID to port)
Directory: Mapping of agent names & attributes to agent IDs
ACC: Agent Communication Channel, used to communicate with other platforms

Agent Communication Languages (1)

Examples of different message types in the FIPA ACL [fipa98-acl], giving the purpose of a message, along with the description of the actual message content.

Message purpose	Description	Message Content
INFORM	Inform that a given proposition is true	Proposition
QUERY-IF	Query whether a given proposition is true	Proposition
QUERY-REF	Query for a give object	Expression
CFP	Ask for a proposal	Proposal specifics
PROPOSE	Provide a proposal	Proposal
ACCEPT-PROPOSAL	Tell that a given proposal is accepted	Proposal ID
REJECT-PROPOSAL	Tell that a given proposal is rejected	Proposal ID
REQUEST	Request that an action be performed	Action specification
SUBSCRIBE	Subscribe to an information source	Reference to source

Agent Communication Languages (2)

Field	Value
Purpose	INFORM
Sender	max@http://fanclub-beatrix.royalty-spotters.nl:7239
Receiver	elke@iiop://royalty-watcher.uk:5623
Language	Prolog
Ontology	genealogy
Content	female(beatrix),parent(beatrix,juliana,bernhard)

A simple example of a FIPA ACL message sent between two agents using Prolog to express genealogy information.

Overview of Code Migration in D'Agents (1)

A simple example of a Tel agent in D'Agents submitting a script to a remote machine (adapted from [gray.r95])

```
proc factorial n {
   if ($n ≤ 1) { return 1; }                # fac(1) = 1
  expr $n * [ factorial [expr $n – 1] ]     # fac(n) = n * fac(n – 1)
}

set number …          # tells which factorial to compute

set machine …         # identify the target machine

agent_submit $machine –procs factorial –vars number –script {factorial $number }

agent_receive …       # receive the results (left unspecified for simplicity)
```

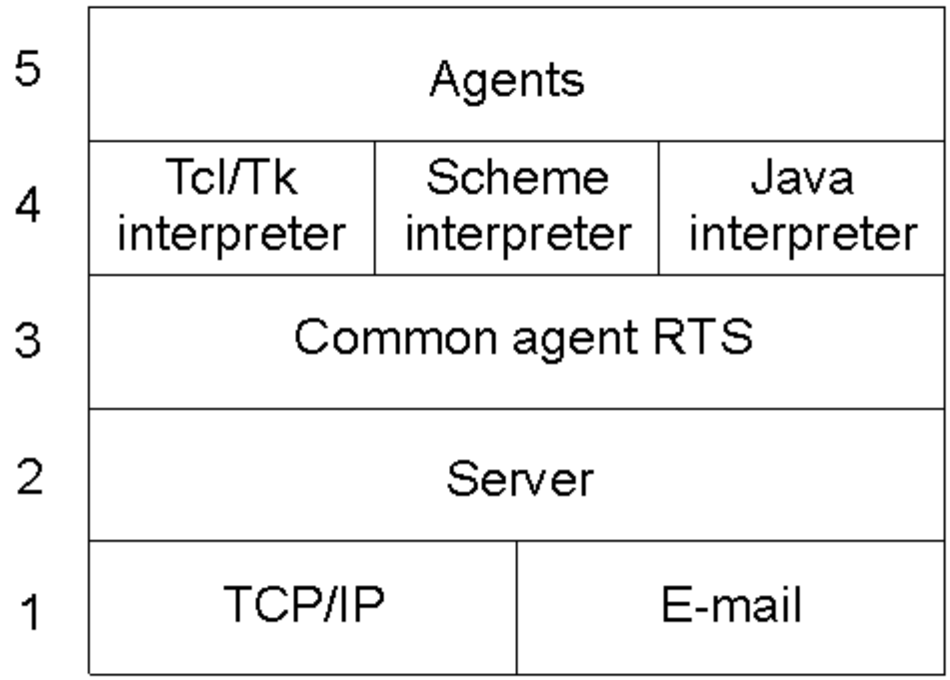
Overview of Code Migration in D'Agents (2)

An example of a Tel agent in D'Agents migrating to different machines where it executes the UNIX *who* command (adapted from [gray.r95])

```
all_users $machines

proc all_users machines {
   set list ""                    # Create an initially empty list
   foreach m $machines {          # Consider all hosts in the set of given machines
      agent_jump $m               # Jump to each host
      set users [exec who]        # Execute the who command
      append list $users          # Append the results to the list
   }
   return $list                   # Return the complete list when done
}

set machines …                    # Initialize the set of machines to jump to
set this_machine                  # Set to the host that starts the agent

# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in  $machines.

agent_submit $this_machine –procs all_users
                                  -vars  machines
                                  -script { all_users $machines }

agent_receive …                   #receive the results  (left unspecified for simplicity)
```

Implementation Issues (1)

The architecture of the D'Agents system.



Implementation Issues (2)

The parts comprising the state of an agent in D'Agents.

Status	Description
Global interpreter variables	Variables needed by the interpreter of an agent
Global system variables	Return codes, error codes, error strings, etc.
Global program variables	User-defined global variables in a program
Procedure definitions	Definitions of scripts to be executed by an agent
Stack of commands	Stack of commands currently being executed
Stack of call frames	Stack of activation records, one for each running command