Introduction

- *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 in this chapter:
  - Threads (within clients/servers).
  - Process and code migration.
  - Software agents.

Chapter 3.1 Threads

- Process
  - *High overhead abstraction* to execute a program
  - E.g. when context switch
    - Save the CPU context (register, PC, stack pointer), modify MMU registers, invalidate TLB cache
  - E.g. create and manage processes is generally regarded as an *expensive* task (`fork` system call).
    - Each process have its own code, data, and stack segments
  - Make sure by OS that all processes peacefully co-exist is not easy (as *concurrency transparency* comes at a price)
- Thread: sometimes called *lightweight process*

Two Important Implications

- Threaded applications often run faster than non-threaded applications
  - E.g. a web server

- Threaded applications are more harder to develop
  - E.g., require synch. mechanism

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

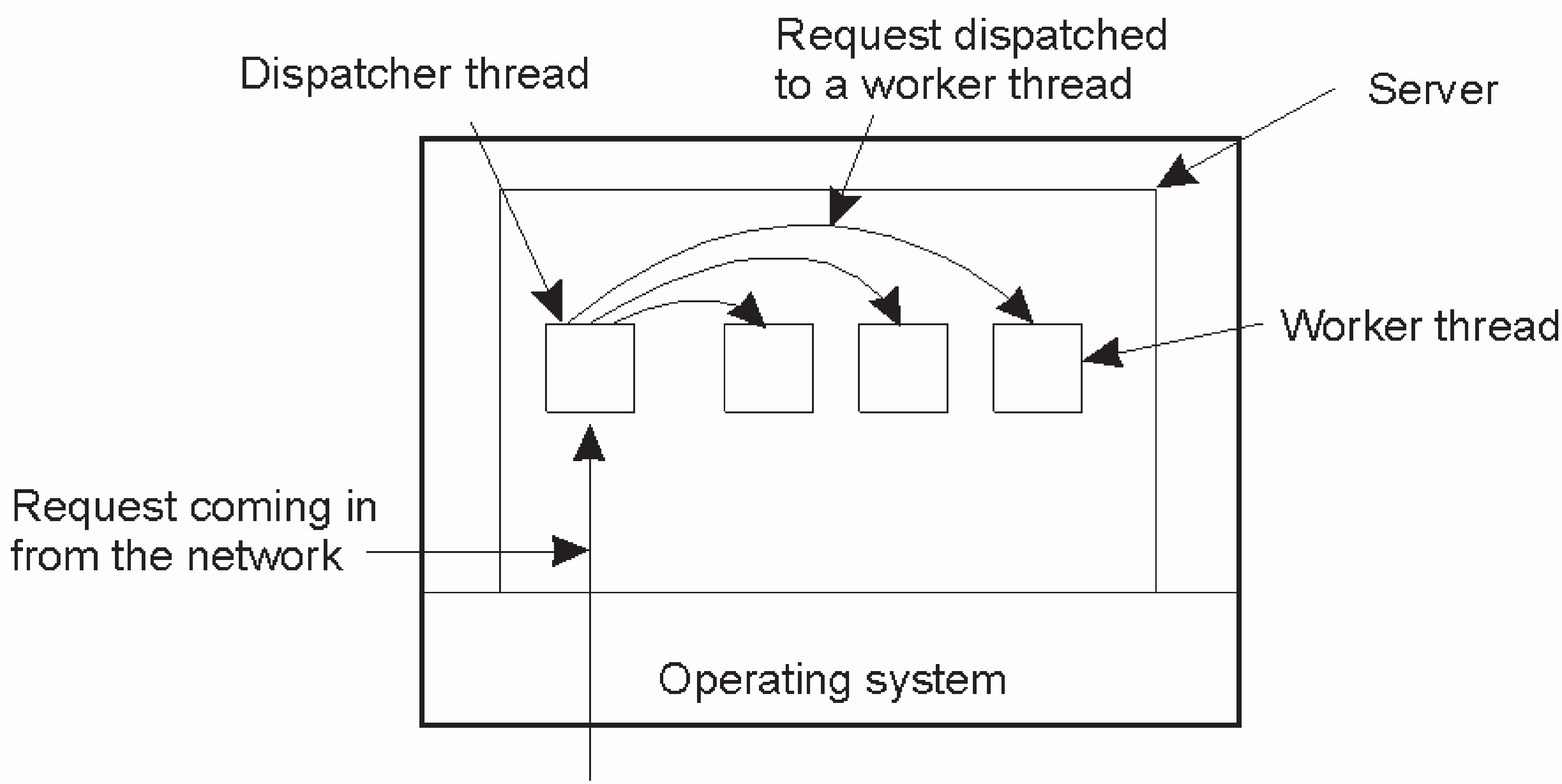
- Important characteristic: a *blocking call* in a thread does not result in the entire process being blocked.
- Lead to the key characteristic of threads within distributed systems:
- Example: Multithreaded Clients & Servers
  - Client example: web browser
  - Server example: web server

Server Implementation Schemes

- Single-threaded server
- Multi-threaded server
- Finite-state machine server
  - Use non-blocking system call
  - Hard to program

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

An Multi-Threaded Server



A multithreaded server organized in a dispatcher/worker model.

Chapter 3.2: Clients

- User-interface is required
  - The X window system
  - Compound documents
- Client-side middleware functions
  - Cooperate with the server to provide access transparency
    - Include location, migration, and relocation transparency

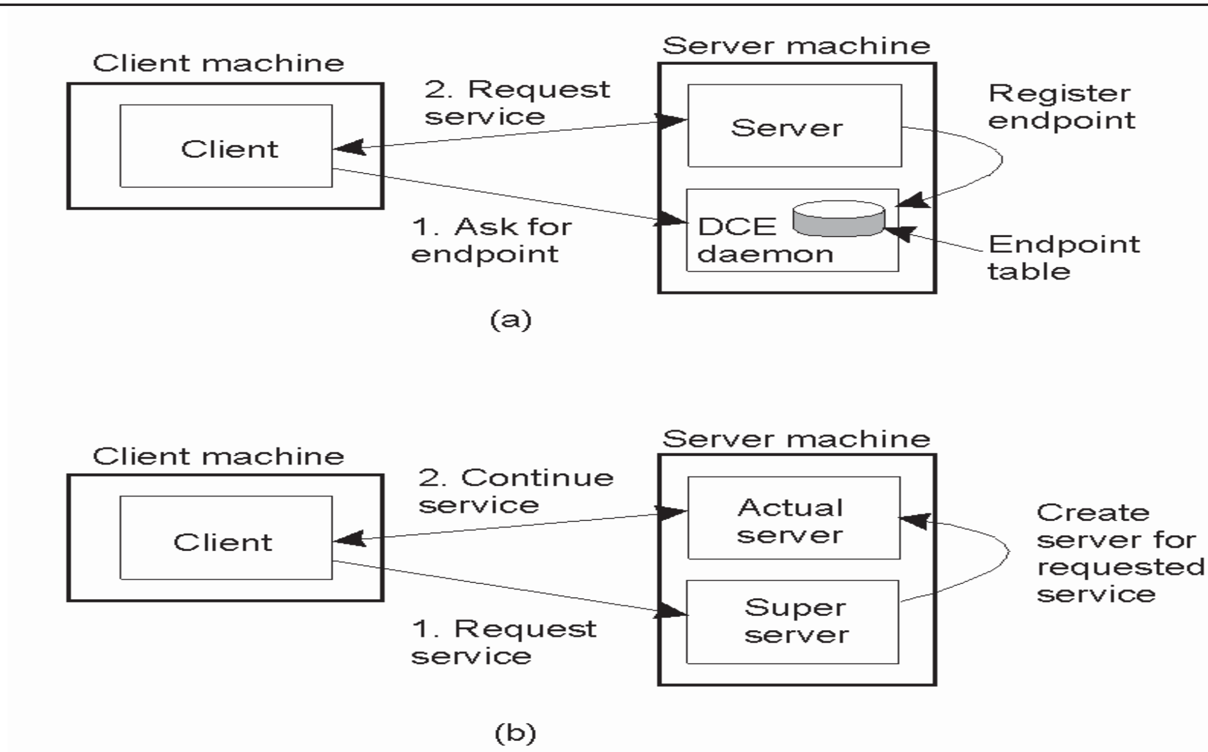
Chapter 3.3: Servers

- Two types of servers:
  - ***Iterative server***: server handles request, then returns results to the client
    - Any new client requests *must wait* for previous request to complete
    - Useful to think of this type of server as ***sequential***
  - ***Concurrent server***
    - Server does not handle the request itself
      - A separate thread handles the request and returns any results to the client
      - The server is then free to immediately service the next client
    - There's no waiting, as service requests are processed in ***parallel***

General Design Issues: Identifying "end-points"

- How do the clients ***bind*** to a server?
  - Client connects to a ***endpoint*** (also called a ***port***) at the server machine
- Three approaches to get the endpoint
  - Statically assigned end-points for well-known services
    - HTTP: 80, FTP: 21
  - Dynamically assigned end-points but a special daemon listening to a well-known port (DCE)
  - ***Superserver*** listening to a set of ports
    - *inetd* daemon in UNIX
    - When a request comes in, fork a process to handle the request

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).

General Design Issues: Interrupted Server

- Whether and how a server can be interrupted?
  - E.g. abort a huge FTP file upload
- Two approaches
  - User abruptly exits the client applications
  - Send ***out-of-band*** data communication
    - Another endpoint for such purpose (i.e., control channel v.s. data channel)
    - Or TCP's ***urgent data***

General Design Issues: Server States

- Stateless or stateful servers
  - ***Stateless Server***: no information is maintained on the current “connections” to the server.(web server)
  - ***Stateful server***: information is maintained on the current “connections” to the server
  - Example:
    - A stateful server for file open operation
      - Maintain which client has which files open
    - A stateless server sends requests that are self contained
      - Client provides full name and offset.

General Design Issues: Server States (Cont.)

- Advantages of Stateless servers
  - More fault-tolerant
  - No open, close calls needed
  - no limits on number of open files
  - no table is required, more space efficient
  - crash recovery is less problematic and quick
- Advantages of Stateful servers
  - Shorter request messages
  - Better performance
  - Read-ahead possible
  - File locking possible

Object Servers

- In a distributed object-based paradigm there is another type of server
  - An Object Server- a server tailored to support distributed objects.
  - Difference: object servers don't provide any specific service
    - Specific service is provided by the objects that reside in the server
- *Act as a place where objects live*
  - Only provides a facility to invoke local objects, based on the requests from remote client
  - Easy to change services by adding/removing local objects

Policy for Invoking Objects by an Object Server

- A object server needs how to invoke an object
  - Which code to execute, which data to be operated…
- Solutions
  - Tree all objects look alike and only one way to invoke (DCE approach)
    - But bad since inflexible
  - Different policies about object creation
    - Created only when invoked and destroyed when no requests are bound to it
    - Create all local objects at startup and remain there

Policy for Invoking Objects by an Object Server (Cont.)

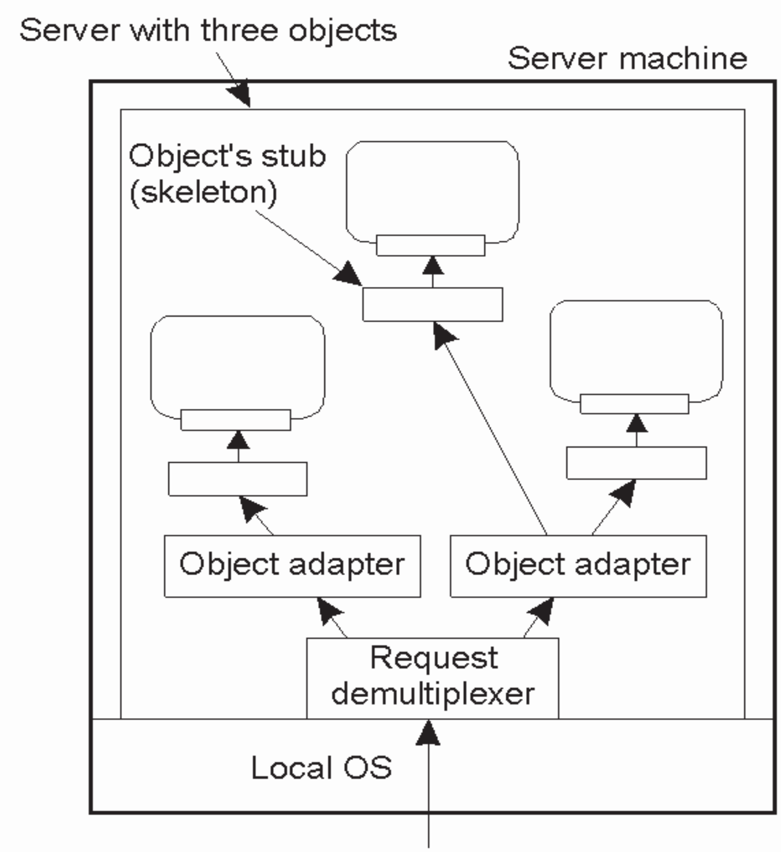
- Solutions (Cont.)
  - Different policies about memory separation
    - Each object is places in its own segment (no sharing)
    - Each object may share their code
  - Different policies about threading model
    - One single thread
      - can only service one outstanding request
    - One thread for each object
      - can only service one req per object
    - One thread for each method

Object Adapter

- Decisions on how to invoke an object is called the ***activation policy***
- The object must first be brought to the server's address space before it can be invoked
- We need a mechanism to group objects according to policies
- Such mechanism is known as "***Object Adapters***" or "***Object Wrappers***"
  - A software implemented a specific activation policy

Object Adapter (Cont.)

- An object adapter has one or more objects under its control
- There can be several object adapters in the same server
- Object adapter are unaware of the specific interfaces of the objects
  - Just extract the object reference
  - Then pass request to the object



Object Adapter header.h

```
/* Definitions needed by caller of adapter and adapter */
      #define TRUE
      #define MAX_DATA 65536
/* Definition of general message format exchange between adapter and client*/
      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* used by the adapter and any program that calls an adapter.

thread.h used in 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 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.

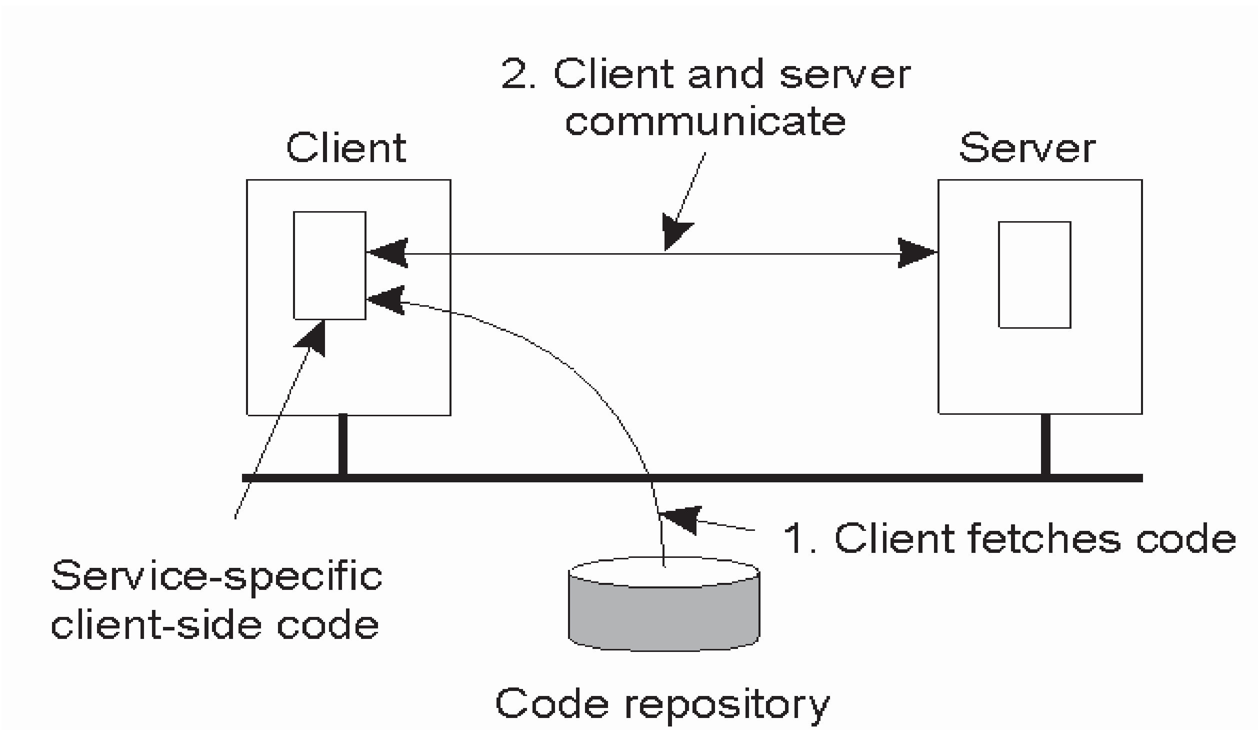
Object Adapter's Main Code

```
#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);
}
```

Chapter 3.4: Code Migration

- Data passing vs. program passing
- Code migration, i.e., process migration
  - An entire process is moved from one machine to another
- Why ?
  - Performance improvement: load balance
    - Move a compute-intensive task from a *heavily loaded* machine to a *lightly loaded* machine "on demand" and "as required"
    - *Moving (part of) a client process to the server* – processing data close to where the data resides
    - *Moving (part of) a server process to a client* – checking data prior to submitting it to a server.
  - Improve parallelism
    - Lots of mobile agent for searching in the Web
  - Increase flexibility
    - Client does not need all SW preinstalled, but just download and execute on demand
- Problem: security issue

One Big Advantage: Flexibility



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.

Models for Code Migration

- What needs to be passed?
  - Code, execution status, pending signals,… etc
- A running process consists of three "segments":
  - *Code segment*: instructions
  - *Resource segment*: reference to external resources (file…)
  - *Execution segment*: current state (private data, stack, PC)
- **Weak Mobility** : only code segment and some initialization data is moved
  - Program always starts from initial state (e.g. Java applets)
- **Strong Mobility**: execution segment passed as well
  - Execution restarts from the next statement (e.g. D'Agents)

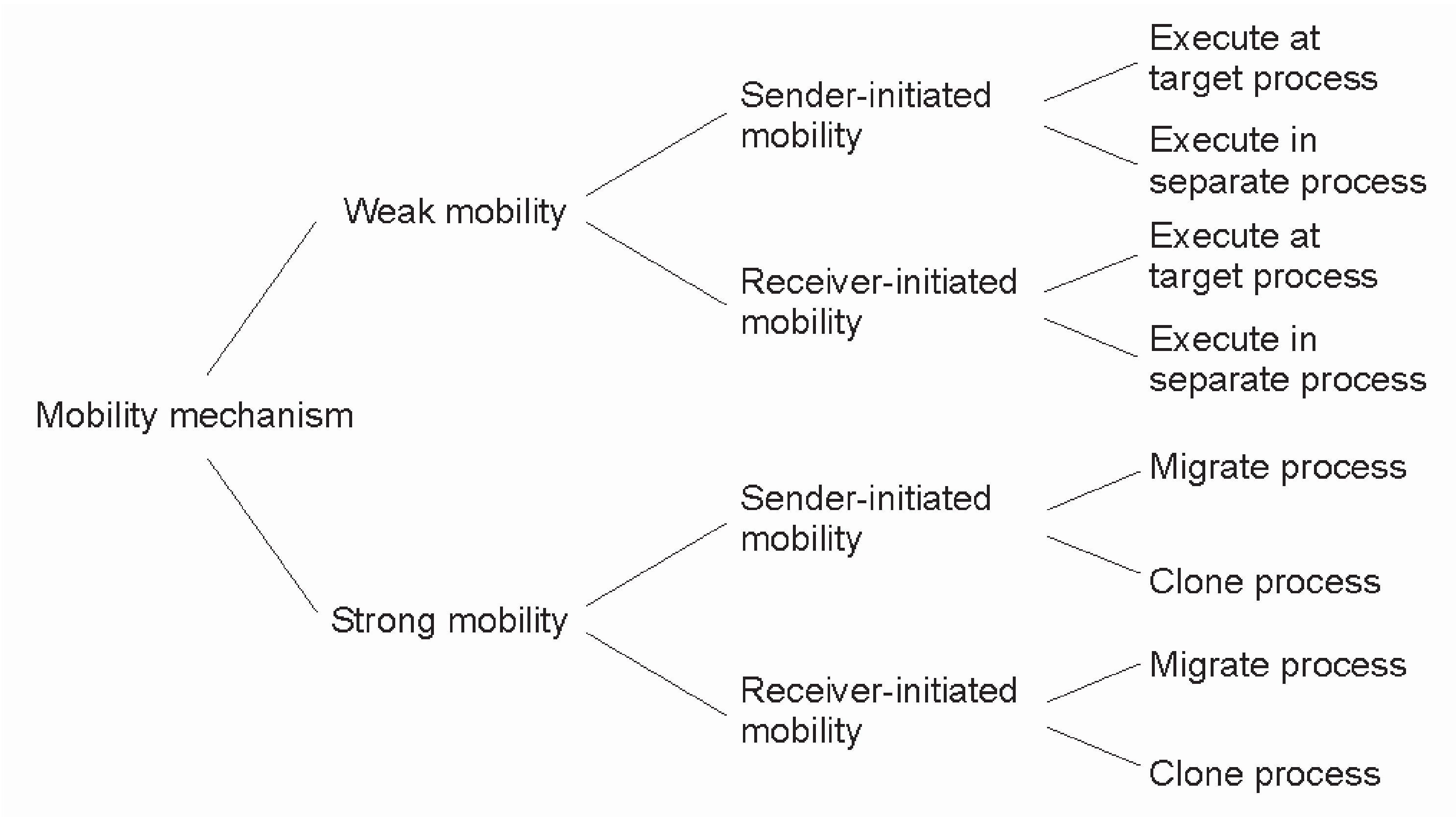
Models for Code Migration (Cont.)

- Which side of the communication starts the migration?
- *sender-initiated*
  - The machine currently executing the code
- *receiver-initiated*
  - The machine that will ultimately execute the code
  - Easy to implement since less security issue

Models for Code Migration (Cont.)

- If weak mobility, another issue surrounds where the migrated code executes
  - Executed by the target process
    - Java applets run in the browser process
  - A separate process.
- Strong mobility also supports the notion of "remote cloning"
  - *an exact copy of the original process, but now running on a different machine*.
  - Not migrated, but just clone a new process on a different machine
    - Like UNIX fork but the client runs on a remote machine

The Models of Code Migration



Migration and Local Resources

- Resource segment: what makes code migration difficult
- 3 types of process-to-resource bindings
  - ***Binding-by-identifier*** **–** the strongest that precisely the referenced resource, and nothing else, has to be migrated
    - E.g. when a process uses an URL
  - ***Binding-by-value*** **–** weaker than BI, but only the value of the resource need be migrated
    - A program relying on a libraries (use the locally available one)
  - ***Binding-by-type*** **–** nothing is migrated, but a resource of a specific type needs to be available after migration
    - E.g. local devices like monitors, printers

Migration and Local Resources (Cont.)

- 3 types of resource-to-machine bindings
  - ***Unattached resources***: a resource that can be moved easily from machine to machine (e.g. files)
  - ***Fastened resource***: migration is possible, but at a high cost (e.g. local databases, complete web sites)
  - ***Fixed resources***: a resource is bound to a specific machine or environment, and cannot be migrated. (e.g. local devices, ports)

9 Possible Combinations

Resource-to-machine binding

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

GR: establish a global systemwide reference

MV: move the resource

CP: copy the value of the resource

RB: rebind process to locally available resource

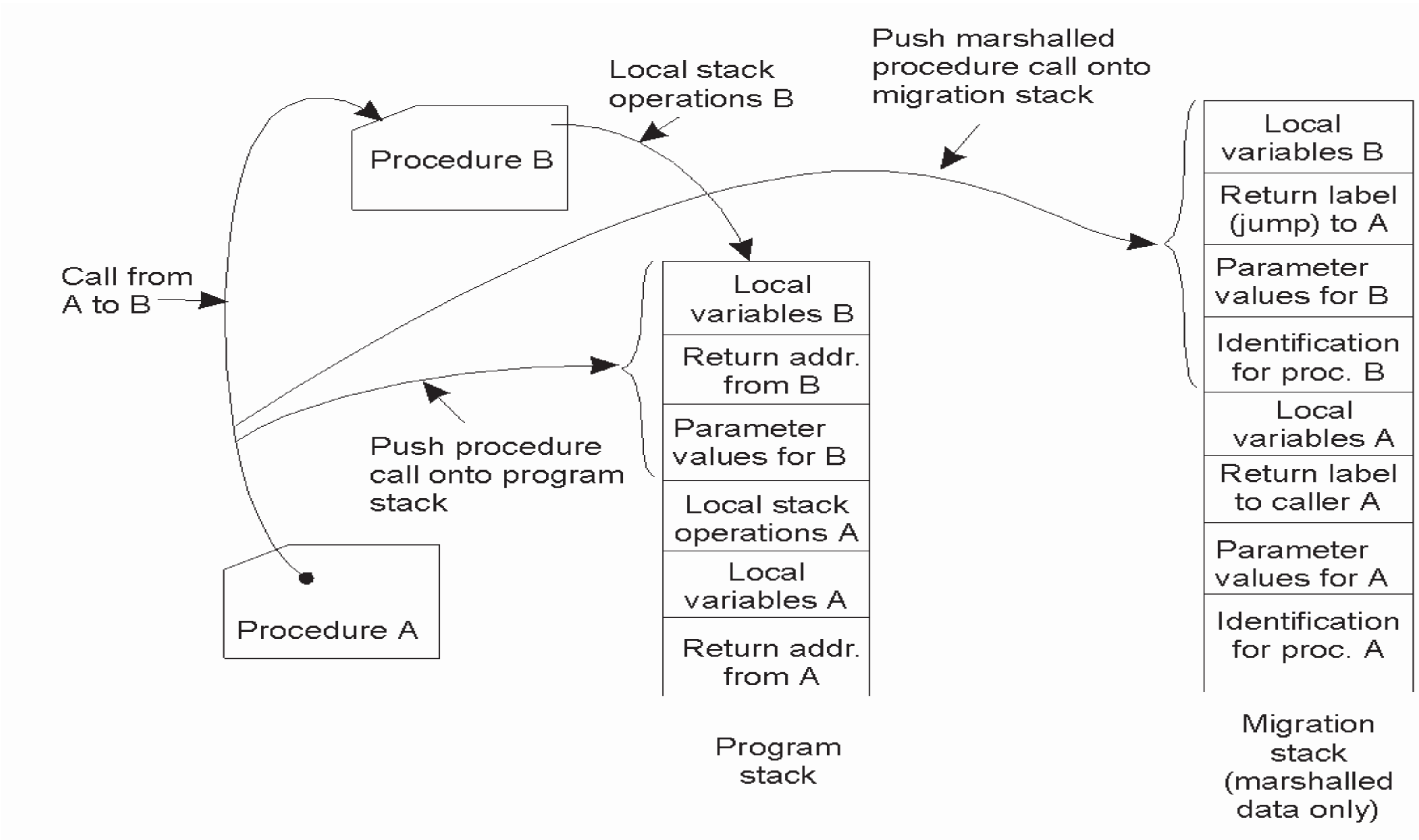
Migration in Heterogeneous System

- Heterogeneous System: different OS and machine architecture
- If week mobility
  - No runtime information (execution segment) needed to be transferred
  - Just compiler the source code to generate different code segments, one for each potenital target mamchine
- But how execution segment is migrated at strong mobility?
  - Execution segment: data private to the process, stack, PC
  - Idea: avoid having execution depend on platform-specific data (like register values in the stack)

Migration in Heterogeneous System (Cont.)

- Solutions:
  - First, code migration is restricted to specific points
    - Only when a next subroutine is called
  - Then, running system maintains its own copy of the stack, called ***migration stack***, in a machine-independent way
    - Updated when call a subroutine or return from a subroutine
  - Finally, when migrate
    - The *global program-specific data* are marshaled along with the *migration stack* are sent
    - The dest. load the code segment fit for its arch. and OS.
  - It only works if compiler supports such as stack and a suitable runtime system

Migration Stack



Migration in Heterogeneous System (Cont.)

- Recently, code migration in heterogeneous system is being attacked by ***virtual machine***
  - *Stript language* and *highly portable language* like Java
  - Source code is complied into an machine-independent intermediate language
  - Source code (script language) or intermediate code (byte-code in Java) is interpreted in a platform-dependent ***virtual machine***
  - Drawback: stuck with a specific language

D'Agents Example

- A middleware solution
  - Supports various forms of code migration in heterogeneous system by virtual machine approach
    - Sender-initiate weak mobility
    - Strong mobility by process migration
    - Strong mobility by process cloning
  - Code is written in Tcl, Java, or Scheme

Sender-Initiated Weak Mobility

```
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)
```

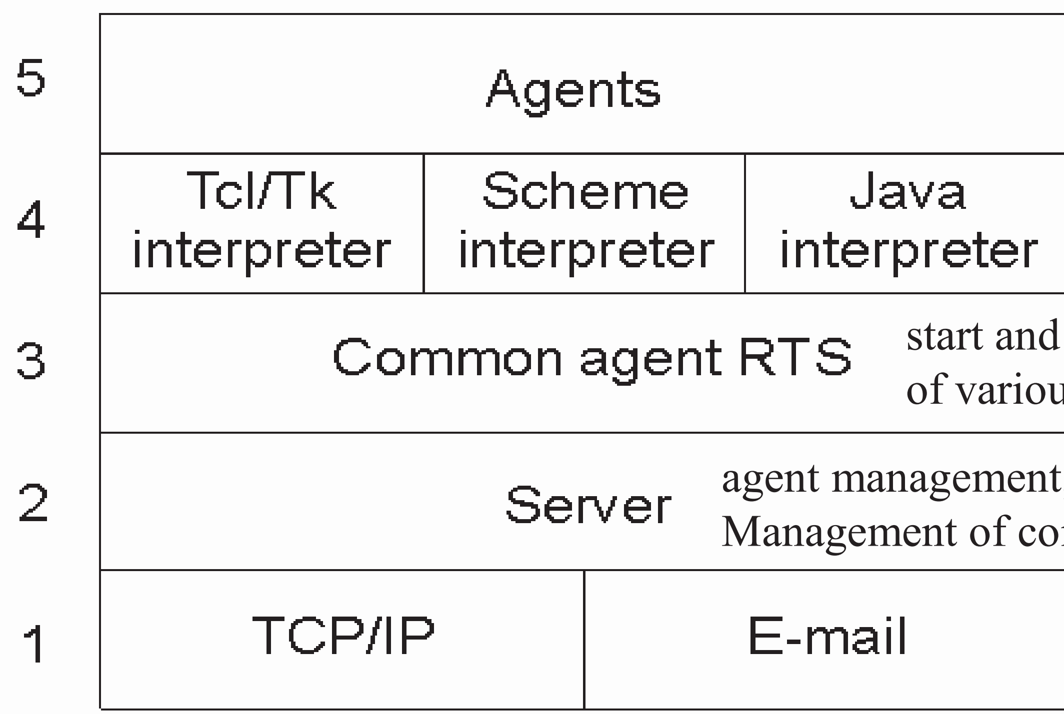
A simple example of a Tcl agent in D'Agents submitting a script to a remote machine

Strong Mobility by Process Migration

```
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)
```

An example of a Tcl agent in D'Agents migrating to different machines where it executes the UNIX *who* command

Implementation Issues of D'Agent



start and end an agent, implementation of various migration operations

agent management, authentication, Management of communication between agents

Implementation Issues of D'Agent (Cont.)

Status	**Description**
Global interpreter variables	Variables needed by the interpreter of an agent (e.g., which event handler to invoke when a packet is received)
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

The parts comprising the state of an agent in D'Agents: when an agent migrate to another machine, above *four tables* and *two stacks* are marshaled and shipped to the target machine

Stack of command: contains all the necessary fields to actually execute a Tcl command (e.g., operand value)
Stack of call frame: a stack of activation record, or call frame

Chapter 3.5: Software Agents

- Software agent
  - An autonomous unit capable of performing a task in collaboration with other, possibly remote, agents
  - An autonomous process capable of reacting to, and initiating changes in, its environment, possibly in collaboration with users and other agents
- System properties of agents
  - ***Collaborative Agent*** –also known as "multi-agent systems", agents are work together to achieve a common goal
  - ***Mobile Agent*** – has the capability to move between different machines

Agent Properties

- *Interface Agent*
  - Assist an end user in the use of one or more applications
  - Has "learning abilities"
    - The more often it interacts with user, the better its assistance
  - E.g. agents that bring buyers and sellers together
    - Know what its owner is looking for, or has to offer
- *Information Agent*
  - Manage info from many different sources
    - Ordering, filtering, and collating…
  - E.g., an e-mail agent may be capable of filtering unwanted mail

Software Agents Classifications

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.

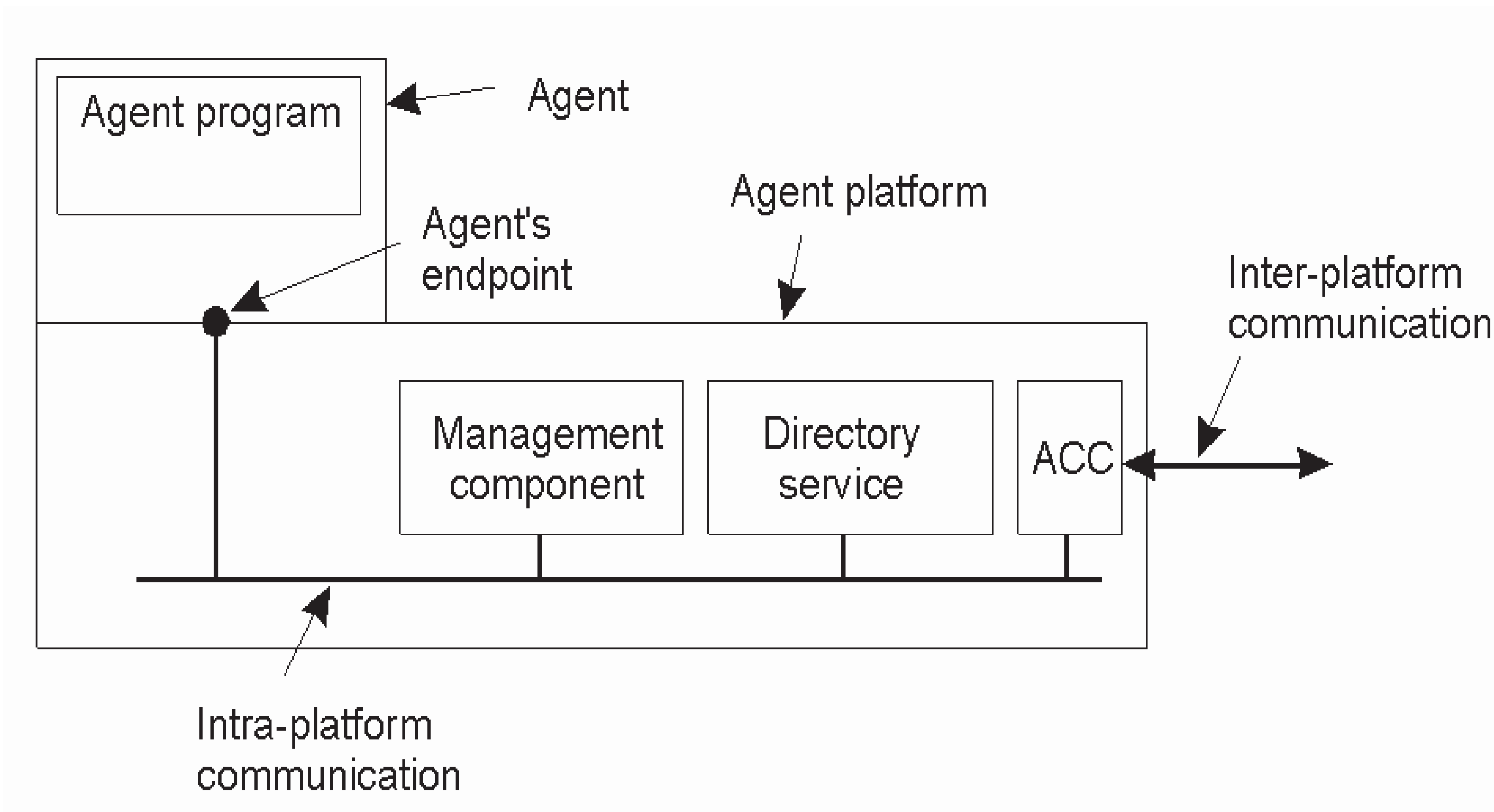
Agent Technology

- The general model of an agent platform has been standardized by FIPA ("Foundation for Intelligent Physical Agents")
  - An agent platform can support multiple SW agents
  - Located at the http://www.fipa.org
- Specifications include
  - Agent Management Component.
    - Facilities for creating and deleting agents…..
    - Mapping from a globally unique ID to a local port
  - Agent Directory Service.
    - Look up what other agents have to offer based on given attributes (like Yellow page)_
  - Agent Communication Channel (ACC)
    - Responsible for reliable and ordered point-to-point communication
  - Agent Communication Language (ACL)

Agent Communication Language (ACL)

- An application-level protocol
- Take care the kind of information that agents communicate
- Define
  - Message purpose
  - Message content
    - Should provide enough information to all receiver to interpret the content
    - Contain a field to identify the language or encoding scheme
    - If no such a field, other field is needed to identify standardized mapping from symbols to their meaning
      - Such an mapping is called "***ontology***"

FIPA Agent Technology



ACL Message Example

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)

Know how to interpret the content

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