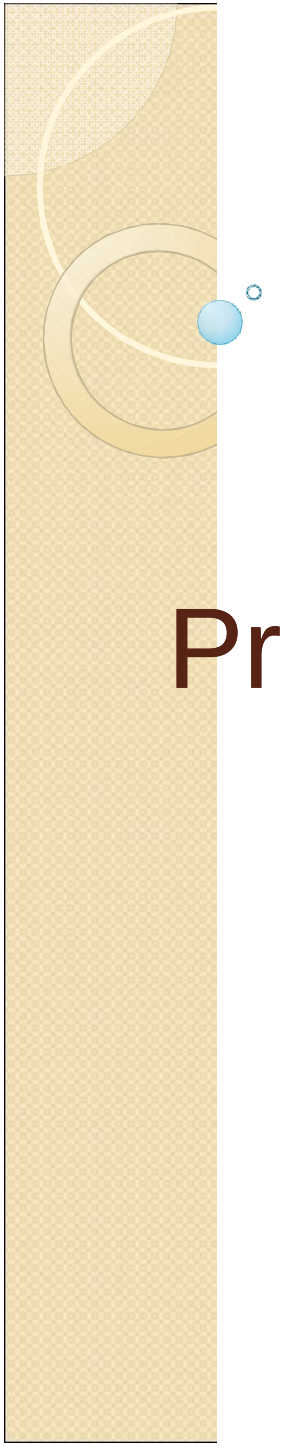
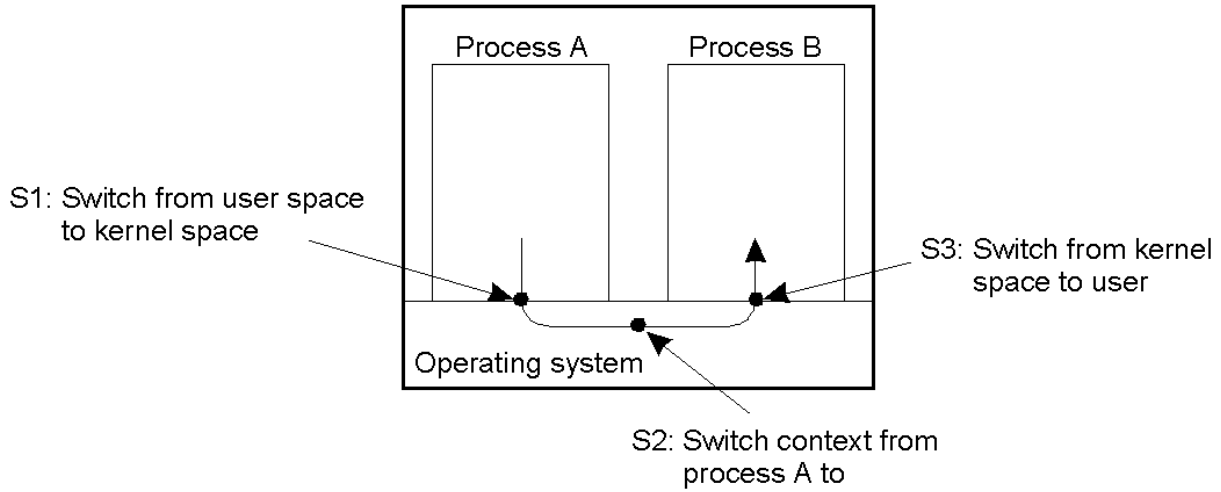


Processes

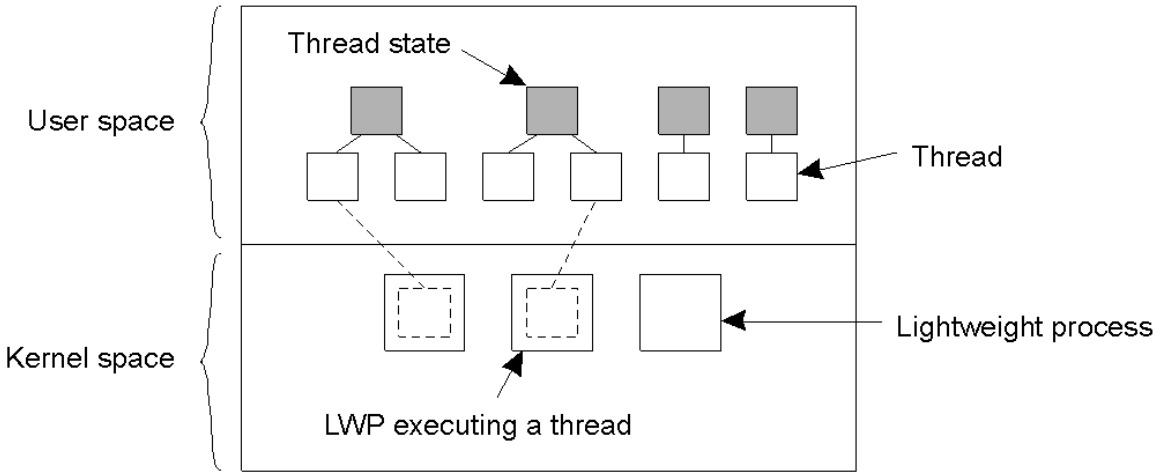
Thread Usage in Nondistributed Systems

Context switching as the result of IPC



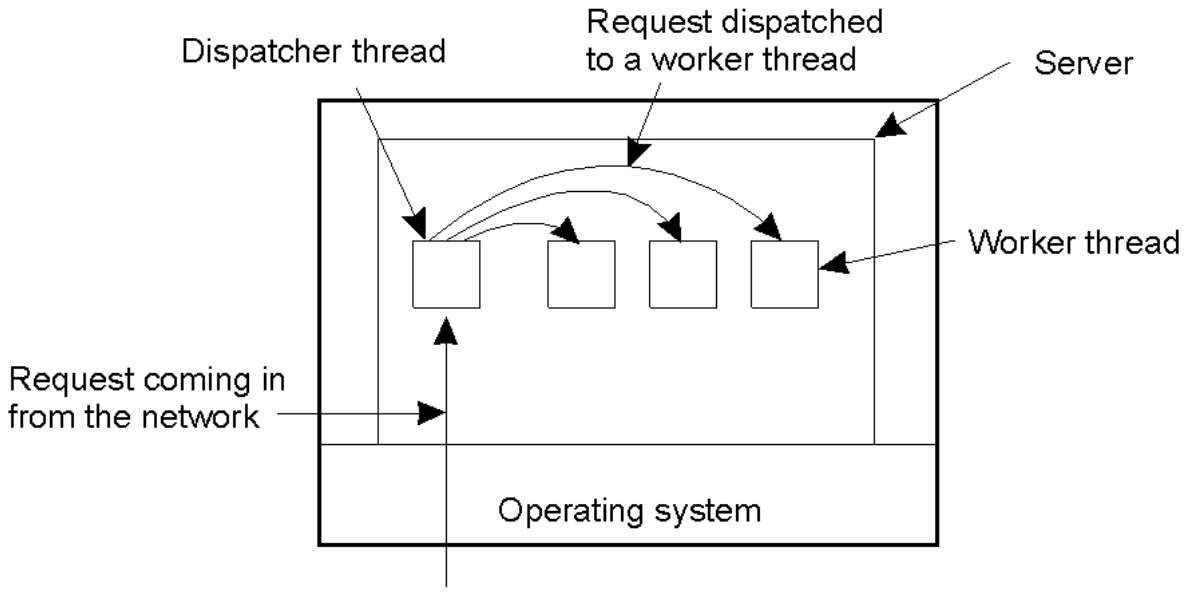
Thread Implementation

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



Multithreaded Servers (1)

A multithreaded server organized in a dispatcher/worker model.



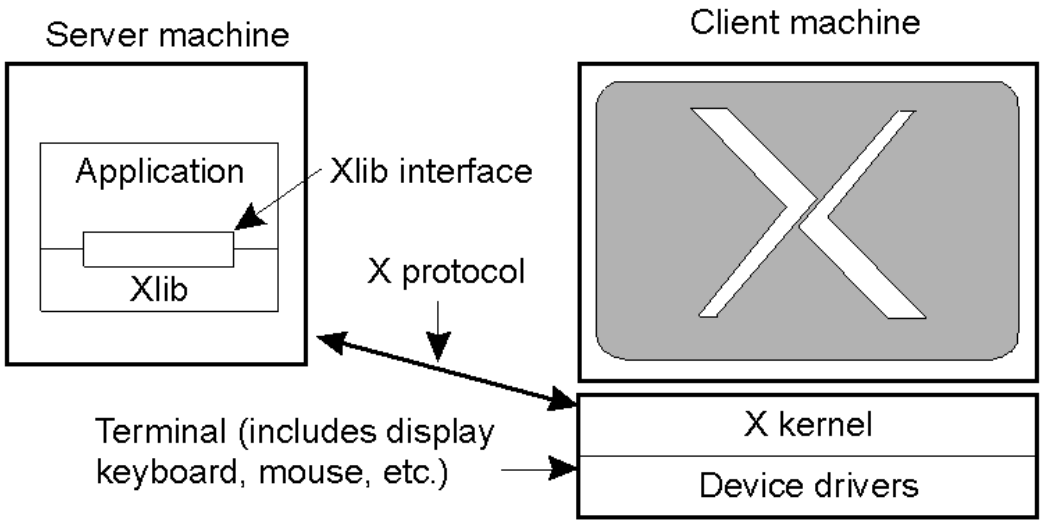
Multithreaded Servers (2)

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.

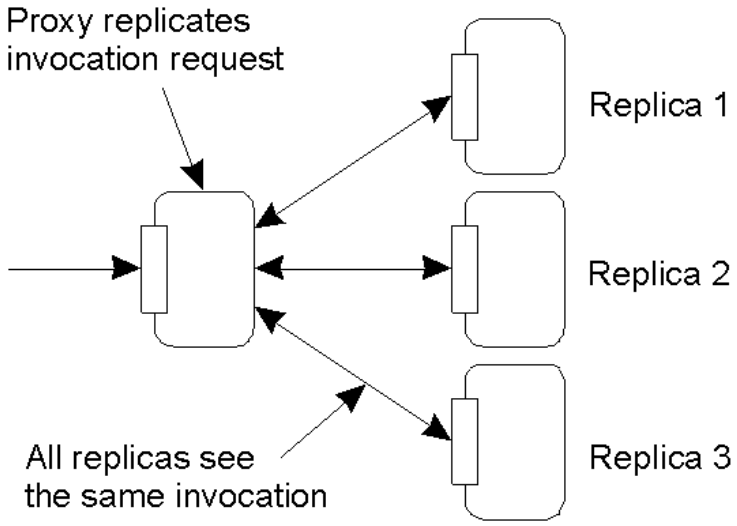
The X-Window System

The basic organization of the X Window System

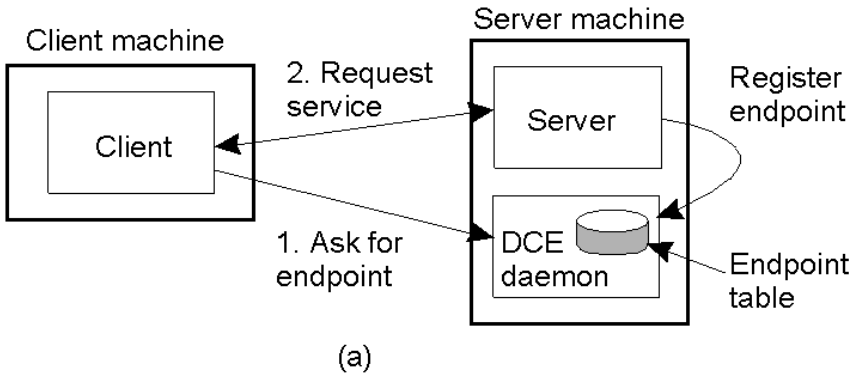


Client-Side Software for Distribution Transparency

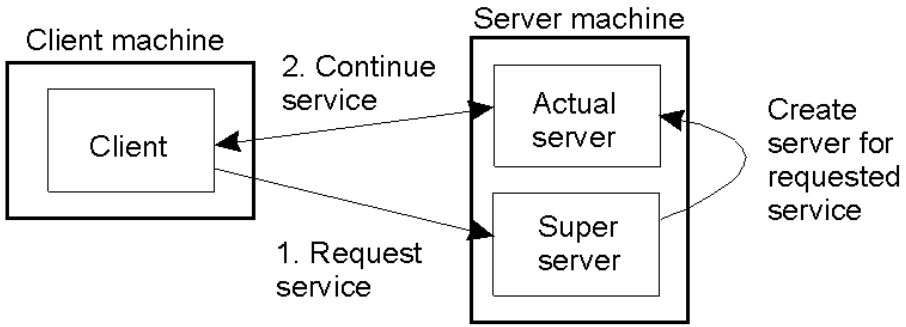
A possible approach to transparent replication of a remote object using a client-side solution.



Servers: General Design Issues



(a)

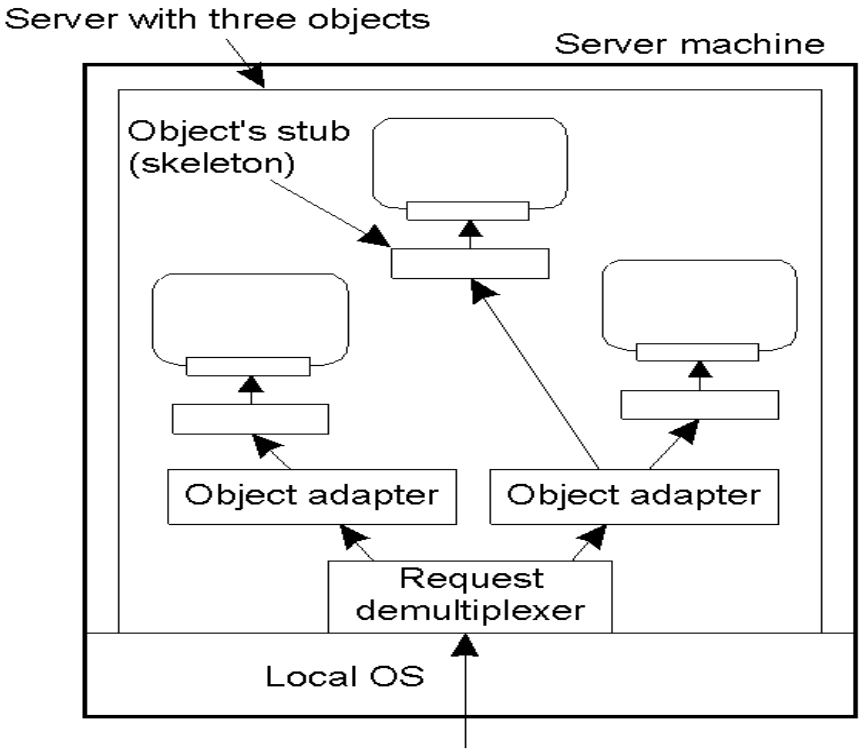


(b)

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

Object Adapter (1)

Organization of an object server supporting different activation policies.



Object Adapter (2)

```
/* 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 (3)

```
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.

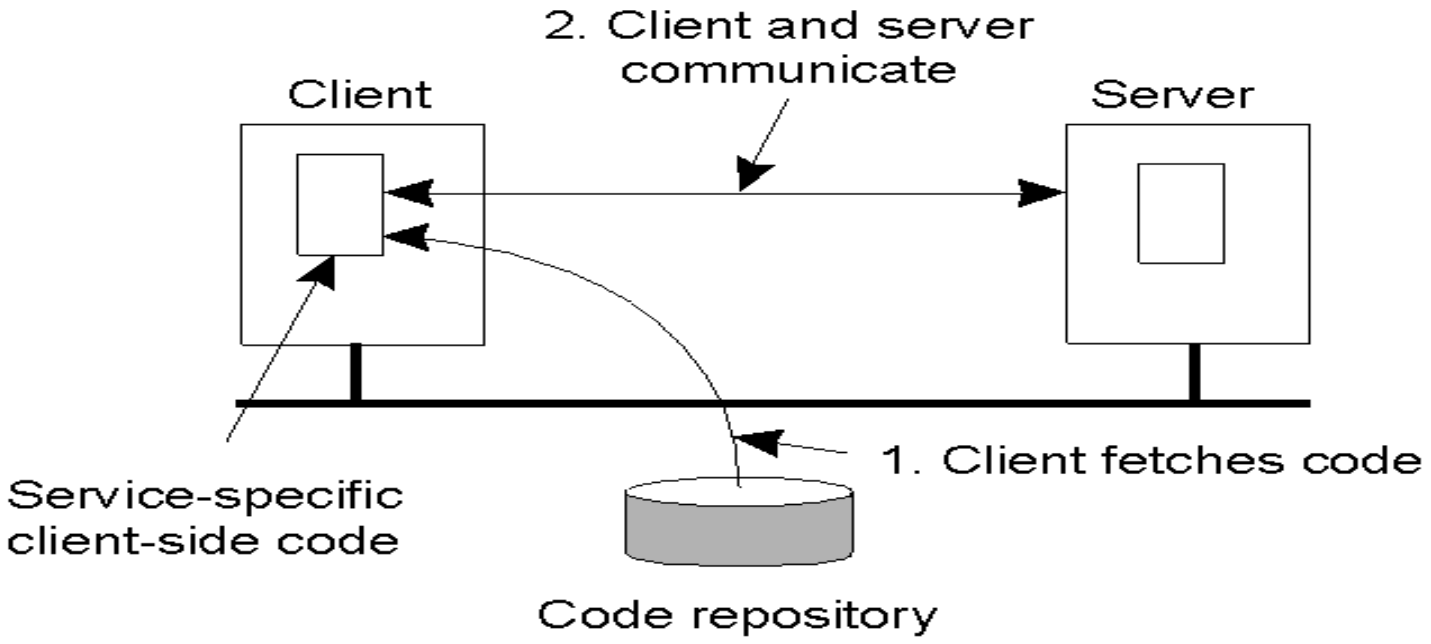
Object Adapter (4)

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

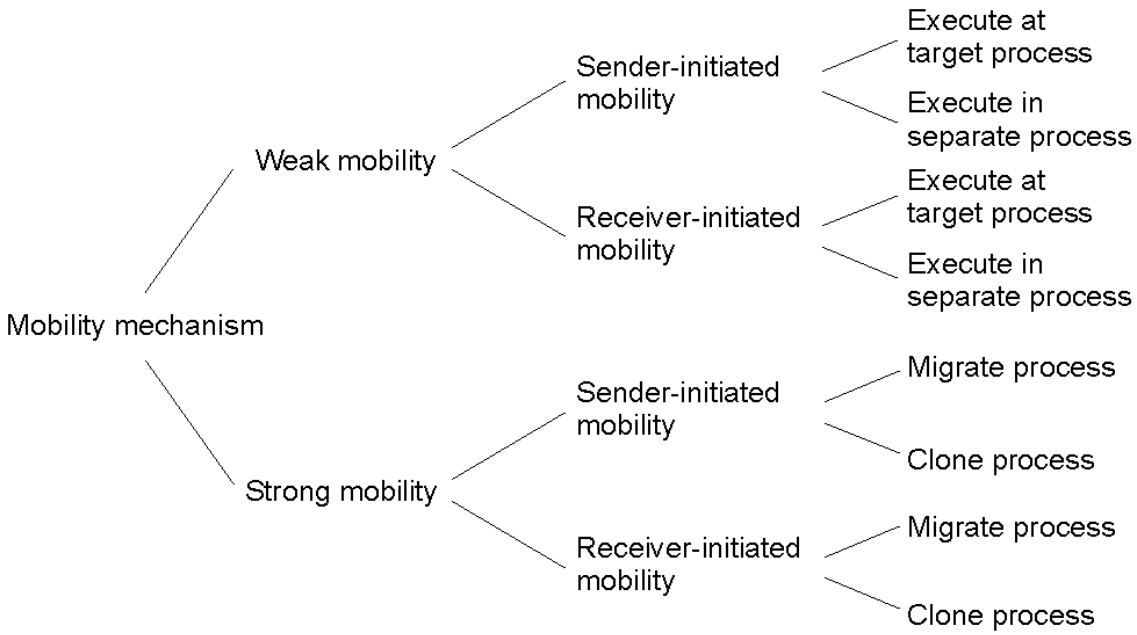
```
#include <header.h>
#include <thread.h>
#define MAX_OBJECTS        100
#define NULL               0
#define ANY                -1

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.



Models for Code Migration



Alternatives for code migration.

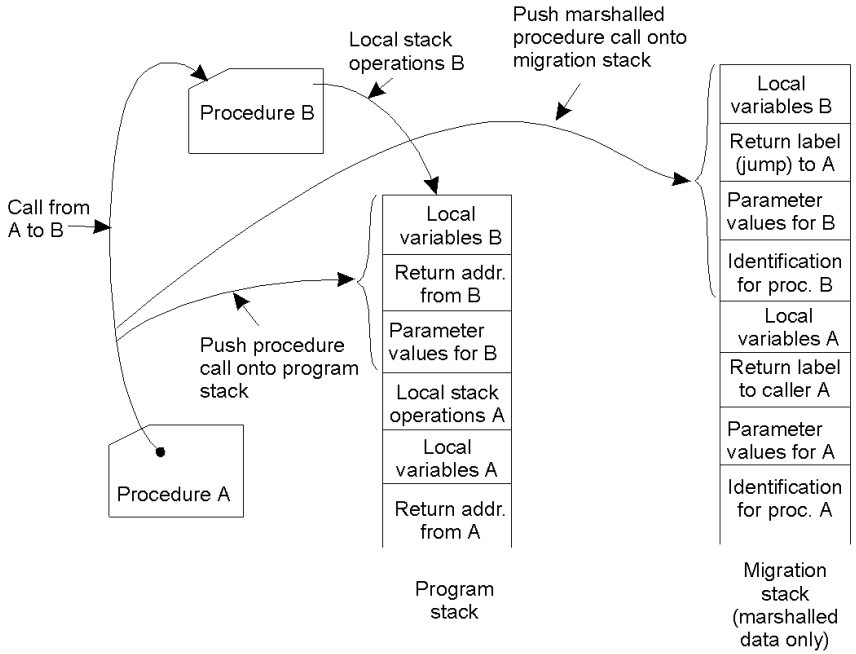
Migration and Local Resources

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 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 Heterogeneous Systems



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

Overview of Code Migration in D'Agents (1)

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

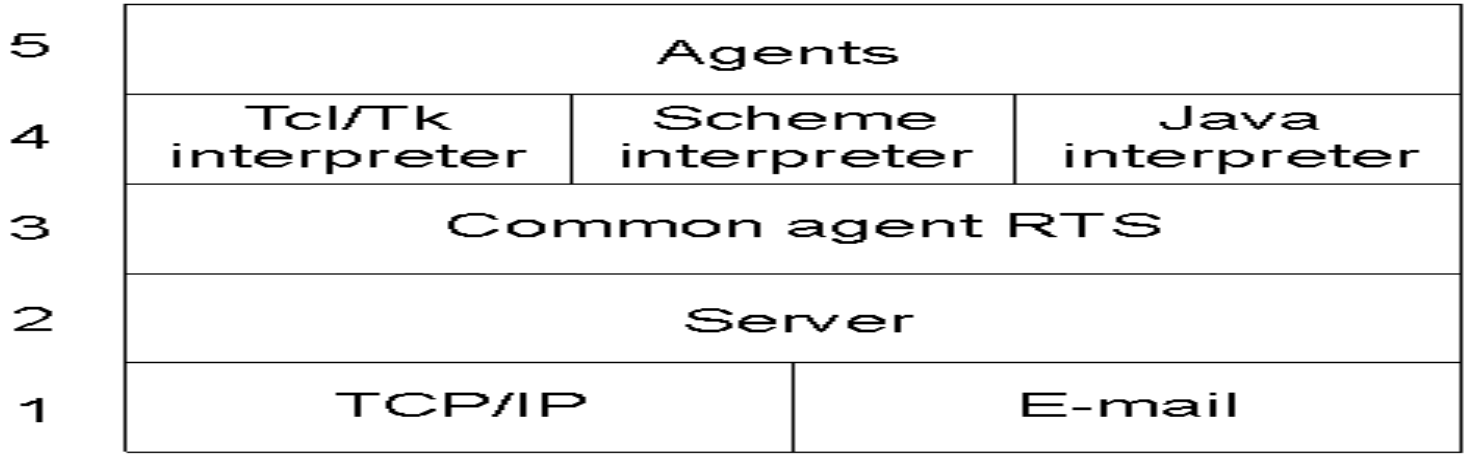
Overview of Code Migration in D'Agents (2)

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

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

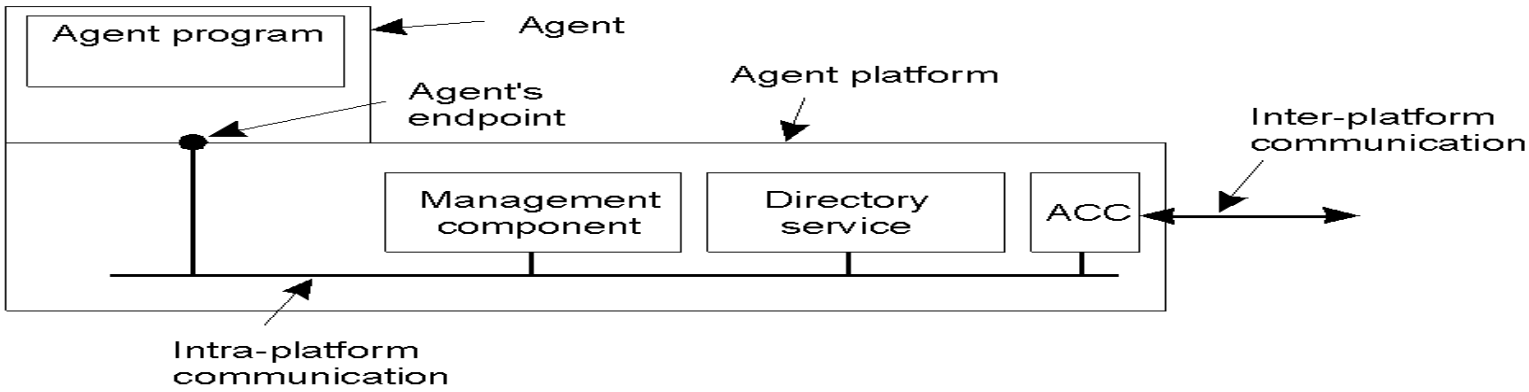
Software Agents in Distributed Systems

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]).



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.

Scope of research

- **Visualizing massively multithreaded applications with ThreadScope**

Application

- The minimal collection of values stored

in registers and memory, used for the execution of a series of instructions (i.e., processor context, state).