Distributed Systems

Principles and Paradigms

Chapter 03

(version 21st September 2001)

Maarten van Steen
Vrije Universiteit Amsterdam, Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 444 7784
E-mail: steen@cs.vu.nl, URL: www.cs.vu.nl/∼steen

Threads

- Introduction to threads
- Threads in distributed systems

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.

Context Switching (1/2)

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 (2/2)

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 as 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 (1/2)

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 (2/2)

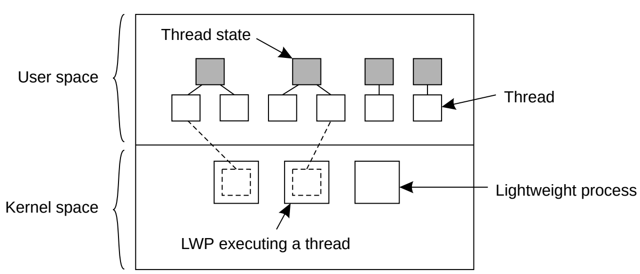
Kernel solution: The whole idea is to have the kernel contain the implementation of a thread package. This does mean that *all* operations return as system calls

- 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.

Solaris Threads (1/2)

Basic idea: Introduce a two-level threading approach: **lightweight processes** that can execute user-level threads.



Solaris Threads (2/2)

- 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 (1/2)

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 (2/2)

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:

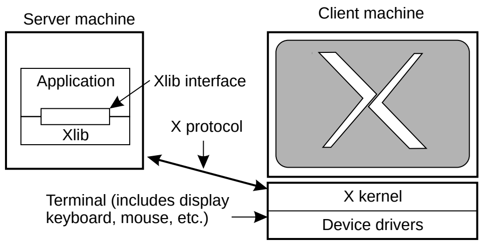
- Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the overall structure
- Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control

Clients

- User interfaces
- Other client-side software

User Interfaces

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



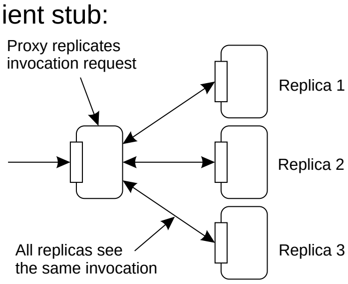
Compound documents: Make the user interface application-aware to allow interapplication communication:

- **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

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).

Servers

- General server organization
- Object servers

General Organization

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

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 `inetd`)

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

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 (1/2)

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 (2/2)

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 (1/2)

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

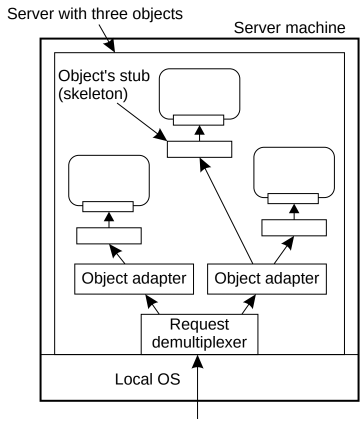
Skeleton: Server-side stub for handling network I/O:

- 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 Servers (2/2)



Observation: Object servers determine how their objects are constructed

Code Migration

- Approaches to code migration
- Migration and local resources
- Migration in heterogeneous systems

Code Migration: Some Context

	BEFORE EXECUTION		AFTER EXECUTION	
	CLIENT	SERVER	CLIENT	SERVER
CS		code state resource		code state* resource
REV	code	state resource		code state* resource
CoD	state resource	code	code state* resource	
MA	code state resource	resource	resource	code state* resource

CS: Client-Server
CoD: Code-on-demand
REV: Remote evaluation
MA: Mobile agents

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.

Managing Local Resources (1/2)

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)

Managing Local Resources (2/2)

	Unattached	Fastened	Fixed
ID	MV (or GR)	GR (or MV)	GR
Value	CP (or MV, GR)	GR (or CP)	GR
Type	RB (or MV, GR)	RB (or GR, CP)	RB (or GR)

GR = Establish global systemwide reference
MV = Move the resource
CP = Copy the value of the resource
RB = Re-bind to a locally available resource

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:

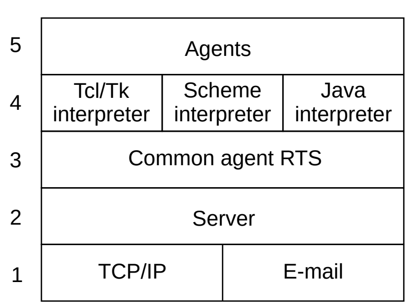
- Interpreted languages running on a virtual machine (Java/JVM; scripting languages)
- Existing languages: allow migration at specific “transferable” points, such as just before a function call.

Example: D’Agents

Overview: D’Agents is based on language interpretation providing support for

- weak and strong mobility
- agent migration
- agent cloning

Organization: Each machine is built as a five-layered system:



D'Agents: Weak migration

```
proc factorial n {
   if { $n <= 1 } { return 1; }
   expr $n * [ factorial [ expr $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
```


D'Agents: Strong migration

```
proc all_users machines {
  set list ""
  foreach m $machines {
     agent_jump $m
     set users [exec who]
     append list $users
   }
  return $list
}

set machines ...
set this_machine ...

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

agent_receive ...
```

Software Agents

- What's an agent?
- Agent technology

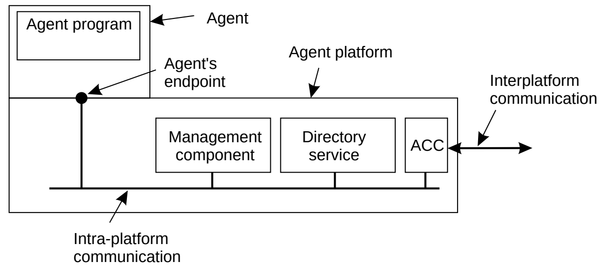
What's an Agent?

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

Property	Common?	Description
Autonomous	Yes	Can act on its own
Reactive	Yes	Responds timely to changes in its environment
Proactive	Yes	Initiates actions that affect 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

Agent Technology



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 (cf. server in D'Agents)

Agent Language

Agent Communication Language: ACL is application-level protocol, making distinction between **purpose** and **content** of a message:

Message purpose	**Description**
INFORM	Inform that a given proposition is true
QUERY-IF	Query whether a given proposition is true
QUERY-REF	Query for a given object
CFP	Ask for a proposal
PROPOSE	Provide a proposal
ACCEPT-PROPOSAL	Tell that a given proposal is accepted
REJECT-PROPOSAL	Tell that a given proposal is rejected
REQUEST	Request that an action be performed
SUBSCRIBE	Subscribe to an information source

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)