Chapter 2: Processes, Threads, and Agents
Beneath the aspect of communication, the organisation and management of processes has to be considered. Some aspects in this are

- *Control flows in processes*
- Organisation of processes
- Allocation of processes
- Scheduling

A distributed system is a collection of cooperating *processes.*
Processes

- A process is *a program which is currently executed*; the execution is strict sequentially.
- Very often, a large number of concurrently running user processes and system processes exist.
- Resources of processes:
  - Disk storage
  - CPU time
  - Access to I/O resources
  - . . .
- When several processes want to access the same resource, an efficient coordination of these processes is necessary.
Processes

- Processes can be in several states:

<table>
<thead>
<tr>
<th>State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>running</td>
<td>Process operations are currently executed</td>
</tr>
<tr>
<td>ready</td>
<td>A process is ready to run but waits for a processor to become free</td>
</tr>
<tr>
<td>waiting</td>
<td>A process waits for an event (e.g. a resource to finish its task)</td>
</tr>
<tr>
<td>blocked</td>
<td>A process waits for a resource used by another process</td>
</tr>
<tr>
<td>new</td>
<td>A process is created</td>
</tr>
<tr>
<td>killed</td>
<td>A process is terminated before ending its task</td>
</tr>
<tr>
<td>terminated</td>
<td>A process finishes after working on all of its instructions</td>
</tr>
</tbody>
</table>

- At each time, *only one* process can be executed on a processor!
Processes

- Transitions between process states can be described by using state diagrams
- Example:

![State Diagram]

- new
- admitted
- exit
- terminated
- ready
- interrupt
- scheduler dispatch
- running
- I/O or event completion
- waiting
- I/O or event wait
Processes

- The operating system is responsible for:
  - Creation and destruction of user processes and system processes
  - Sharing memory and CPU time
  - Provision of mechanisms for communication and synchronisation between processes
  - Intervention in case of deadlocks
- For the operating system, processes are represented as process control blocks (PCB)
- PCBs contain all information relevant for the operating system
Process Execution

- The main memory contains the operating system kernel and (build upon it) a command interpreter (CI).
- A raw division of operating systems can be made by process execution management:
  - **Single-Tasking**: a process is *exclusively* loaded into the memory when executed.
  - **Multiprogramming, Multitasking**: several processes are in memory *simultaneously*.
Multiprogramming vs. Multitasking

• Difference *Multiprogramming* ⇔ *Multitasking*

  ➢ In *multiprogramming* the user *cannot be interrupted* when using the CPU; this can cause a blocking of the whole system for processes which need high CPU capacity.

  ➢ In *multitasking* a process can get an interrupt to stop working. Additionally, a time span can be defined which gives the maximum duration of CPU usage for one process (*time-sharing*). By time-sharing, an interactive usage of the system by several processes is possible. A user executing one process seems to use the CPU exclusively.
Scheduling

• Scheduling is the process of managing the access to resources, especially the CPU, by several processes.

• Scheduling is made in different temporal dimensions:
  - The long-term scheduler (or job scheduler) chooses processes to be load into the memory
  - The short-term scheduler (or CPU scheduler) decides when already loaded processes are allowed to use the CPU

• The scheduling manages several queues:
  - Job queue – contains all processes in the system.
  - Ready queue – contains all processes load into the main memory, which are in the state ready.
  - Device queues – contains all processes currently waiting for an I/O device.
Scheduling

- Processes migrate between these queues:

- new process
- ready queue
- CPU
- terminated process

- I/O
- I/O queue
- I/O request
- time slice expires
- child executes
- fork a child
- interrupt occurs
- wait for an interrupt
Processes and Threads

*Traditional operating system*
Each process has its own address space and an own control flow. A scheduler is responsible to allocate the operating system's resources for running processes.

*Distributed systems*
A single control flow is no longer acceptable:
- Several clients can wish to access a server process simultaneously
- A speedup can be obtained by executing process operations and communication operations in parallel

For these reasons, several control flows inside a single process are needed which can share process resources and work concurrently.

⇒ Threads

Threads can be called small processes, and sometimes they are denoted as *lightweight processes*. 
**Threads**

*Execution environment for process*

- Address space
- Synchronisation and communication between threads
- Files, windows, ...
- Can only be accessed from inside the execution environment

When needed, threads can be created and destroyed dynamically.
Threads

- Each process has an own program counter, stack, registers, and address space. It works independently from other processes but is able to communicate with them.

- Each thread as well has an own program counter, stack, and registers, but there is only one address space for all threads belonging to a process. Thus, all threads can share global variables.

*States of threads*

- *running*, the thread was assigned to a processor
- *blocked* w.r.t. a semaphore
- *ready*, the threads waits on a processor to be assigned to
- *terminated*, all operations are finished, the thread waits to be destroyed
Threads

- A process can create several threads which work together and do not disturb the work of the other threads.

**Execution of threads**

A scheduler assigns CPU capacity by a given scheme to make all processes work 'simultaneously'. Inside these processes, threads are executed by using FIFO or Round Robin (time-sharing).
# Thread Management

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>self()</td>
<td>Gives back the ID of the current thread</td>
</tr>
<tr>
<td>create(name)</td>
<td>Creates a new thread</td>
</tr>
<tr>
<td>exit()</td>
<td>Finishes the current thread</td>
</tr>
<tr>
<td>join()</td>
<td>Waits for another thread to finish; possibly gives back a result</td>
</tr>
<tr>
<td>detach()</td>
<td>Detaches a thread from the other ones, i.e. no other thread will wait for results of this thread</td>
</tr>
<tr>
<td>cancel()</td>
<td>Asynchronous finishing of a thread's work</td>
</tr>
<tr>
<td>masking</td>
<td>A thread can define areas in which a <code>cancel()</code> can be delayed (masked)</td>
</tr>
<tr>
<td>prevention</td>
<td>A thread can define a cleanup handler which is executed in case of an unmasked <code>cancel()</code> e.g. for garbage collection</td>
</tr>
</tbody>
</table>
Why Threads?

+ **Speedup** by working on several process sub-tasks in parallel
+ Threads can be **created more easily** than new processes
+ A **context change** between threads is **cheaper** than the operating system's context change between processes
+ Threads inside a process **share data and resources**

– **... but: the division** of a tasks into threads is **difficult** or even impossible
– When using threads, they have to be **protected against simultaneous access** to process resources. Therefore, coordination mechanisms are needed.
  ⇒ **Additional communication overhead** and **concurrent programming** by using monitors, semaphores, ...
– **Decreasing efficiency** when inconveniently dividing a task into sub-tasks
Thread Implementation

There are two implementation possibilities:

- **User-level threads** (creating and destroying threads as well as context changing is easy. But when a thread blocks, the whole process is blocked as well.)

- **Kernel-level threads** (thread operation become as costly as process operations.)

- Efficient method: combine both thread variants
Implementation of Threaded Servers

Common model for implementation: Dispatcher/Worker model

- Dispatcher thread
- Request coming in from the network
- Worker threads
- Operating system
- Request dispatched to a worker thread
- Server
Threads in Client and Server

Client
Implement an own thread for requests. Thread 1 is not blocked while thread 2 waits for a result.

Server: Worker Pool Architecture
The server creates a fixed number of worker threads. A dispatcher thread is responsible for receiving requests and assigning them to the N worker threads. (Disadvantage: the fixed number of worker threads can sometimes be too small; additionally, many context changes between dispatcher thread and worker threads have to be made.)
Server-Threading Architectures

a. For each request, a new worker thread is created. *Disadvantage:* additional overhead for creation and destruction of threads.

b. For each connection of a client a new thread is created. This decreases the ration of creating and destroying threads. *Disadvantage:* higher response times for single connections if many requests are made on such a connection.

c. For each object a new thread is created. This thread is responsible for all connections opened by its object. The overhead (create/destroy) is further reduced, but the response times can go up.
Distribution Transparency for the Client

Replication transparency on the client's side can be realised by using threads: if a request is sent to several servers in parallel (robustness), the client could use a proxy knowing all replicates. This proxy now can use several threads, one for each replicate, to simultaneously contact all servers and collect the results for computing a final result for the client.
Example - Threads in Java

Thread(ThreadGroup group, Runnable target, String name)
creates a new thread in state SUSPENDED. The thread belongs to a group group
and is identified by name. After being created, the thread calls the method run() of
the object target.

setPriority(int newPriority), getPriority()
sets/gets the priority of a thread.

run()
the run() method of the target object is executed. If the object has no such
method, the thread executes an own (default) run() method (thread implements
Runnable).

start()
the current state of the thread is changed from SUSPENDED to RUNNABLE.

sleep(int millisecs)
the thread is suspended (state SUSPENDED) for the given time span.

yield()
transition to state READY, calling the scheduler
Example - Threads in Java

`destroy()`
- destroying the thread.

`thread.join(int millisecs)`
- the thread is blocked for the given time span. If the called `thread` terminates before the end of this time span, the blocking is cancelled.

`thread.interrupt()`
- the called `thread` is interrupted in its execution; by this, he can be invoked from a blocking call like `sleep()`.

`object.wait(long millisecs, int nanosecs)`
- the calling thread is blocked till a request `notify()` or `notifyAll()` in `object` is made; if the given time span end before this or if an `interrupt()` comes in, the calling thread is also invoked.

`object.notify(), object.notifyAll()`
- invokes the thread resp. All threads which have called `wait()` on `object`. 
Combination of Communication and Processes

*Processes* are working on (sub-)tasks

*Communication services* enable a cooperation of processes

But: as well process execution as communication can be very time-costly:

- A computer is heavily loaded and needs much time for working on a process
- Exchanging (many) data frequently

Transmission of *code*

- A process can be migrated to another host for a faster execution
- If several request (which build upon the results of the request made before) have to be made, the computation and generation of the next request can take place at the target host: no more data exchange is necessary by using the network
- A client can dynamically be enhanced by new functionalities
Dynamic Enhancements

Using a code repository, a client can dynamically be enhanced by interfaces to new servers. This enables the implementation of 'lightweight' clients without the need to determine all used services before runtime.
Models for Code Migration

Transmission of code and initial state

Weak mobility

Mobility mechanism

Additional transmission of execution state

Sending a search client

Sender-initiated mobility

e.g. Java Applets

Receiver-initiated mobility

Execute at target process

Execute in separate process

Execute at target process

Execute in separate process

Strong mobility

Sender-initiated mobility

Receiver-initiated mobility

Migrate process

Clone process

Clone process

Problem in heterogeneous systems: how to achieve a correct reconstruction of a process on the target platform?
Migration in heterogeneous Systems

Idea:

- Restrict migration to fixed points in time of the program execution: only when a sub-routine is called.
- The system manages a so-called migration stack in a machine-independent format as a copy of the normal program stack.
- The migration stack is updated when a sub-routine is called or ends.
- In case of a migration, migration stack and global program data are transmitted.
- The target host gets the migration stack, but has to get the routine code for its platform from a code repository.

Better solution: using a virtual machine

- Java
- Scripting languages
Mobile Agents

An agent is a piece of software, which works on a task

- *autonomous*,
- on behalf of a user, and
- together with other agents.

Not each migrating process is a (mobile) agent:

- *Process migration*
  
  the *operating system* decides the migration of processes. Details are hidden for the application.

- *Agent migration*
  
  the *agent* decides its migration activities itself, depending on its internal state. It is able to initiate its own migration.

Not each migrating process is directly called an agent; but in the following, only agents are talked about.
Agent Migration

**Strong agent migration**
- State of data and execution are to be transmitted
- The whole transmission process (getting state, marshalling, transmission, and unmarshalling) are transparent
- Slow and costly because of possibly large states

**Weak agent migration**
- Only data state is transmitted
- An own start method interpolates form data state, where to continue with the agent execution
- Faster, but more complicated to program

*Remote access in distributed systems:*
Communication mechanisms need the transmission of:
- Remote Procedure Call **control**
- Remote Method Invocation **control + code**
- Weak agent migration **control + code + data**
- Strong agent migration **control + code + data + state**
Mobile Agents

Steps for migration:
- Serialisation of code
- Transmission of code (and state)
- De-serialisation of code

(sometimes) additional steps:
- Class loading on host 2
- Deletion of agent from host 1 to prevent from agent copies
- Prevent agent loss
Agent Example: D′Agents

- Supports writing agents in different languages:
  - Java
  - Tcl
  - Scheme

- Communication and migration overhead not as low as possible

- Strong mobility

---

Machine A

1. Capture agent state
2. Sign/encrypt state
3. Send state to B

... searched = searched + 1; jump B; meet with search-engine; ...

Server

Agent

Machine B

4. Authenticate
5. Restore state
6. Resume agent

... searched = searched + 1; jump B; meet with search-engine; ...

...
Migration in D’Agents – Scripting Languages

```tcl
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 }

Simple example: a Tcl-Agent in D'Agents transmits a script to a remote host
all_users $machines

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

# 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 …

Simple example: the agent migrates to several hosts
Implementation

For migration to and execution on a remote host, an agent needs a special execution environment:

1: transmission by e-mail or TCP/IP
2: the server manages agents and their communication as well as authentication services
3: Common Agent RTS (Run-Time System) provides basic services for agent support: start, termination, migration, ...
4: several language interpreters allow the agents to be realised in several languages

Layer 2 to 4 enable an execution of agents in a heterogeneous environment by separating them from the underlying platform

Agent platform

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP</td>
<td>Server</td>
<td>Common agent RTS</td>
<td></td>
<td>Agents</td>
<td></td>
</tr>
<tr>
<td>E-mail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Tcl/Tk interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Scheme interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Java interpreter</td>
</tr>
</tbody>
</table>
Many agent platforms allow an execution of agents written in Java, because the concept of Java is platform-independency by using a virtual machine.

An example is the agent platform *Voyager*. This platform provides Java classes for the management and the migration of agents. As in D'Agents, the mobility can be easily embedded in the agents.

**IMobileAgentMini.java:**

```java
public interface IMobileAgentMini {
    void start();
    void goal();
}
```

Interface definition for an agent
Mobile Agents in Java (Voyager)

MobileAgentMini.java:
import java.io.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;

public class MobileAgentMini implements IMobileAgentMini, Serializable {
    public void start () {
        try {
            Agent.of(this).moveTo("//hostname:8000", "goal()");
        }
        catch (Exception exception) {
            System.err.println(exception);
            System.exit(0);
        }
    }

    public void goal () {
        System.err.println("-> done\n");
    }
}

Loading of classes for management and mobility support of agents

Serialisability for code transmission

Migration by specifying target host and port as well as code to be transmitted
Mobile Agents in Java (Voyager)

AgentStarter.java:
import com.objectspace.voyager.*;
import com.objectspace.voyager.agent.*;

public class AgentStarter {
    public static void main(String[] argv) {
        try {
            Voyager.startup();  \--------- Start of agent environment

            // add mobility part
            IMobileAgentMini mobileAgent = (IMobileAgentMini)
                Factory.create(
                    MobileAgentMini.class.getName());
            System.err.println("-\> start migration");
            mobileAgent.start();
        }
        catch(Exception exception) {
            System.err.println(exception);
            System.exit(0);
        }

        Voyager.shutdown();
    }
}
Why Mobile Agents?

Example: search a remote table for a particular value
## RMI vs. Agents in a Local Network

<table>
<thead>
<tr>
<th>Migration Type</th>
<th>Time in ms</th>
<th>One Migration Equals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration with class loading</td>
<td>1430,1</td>
<td>-</td>
</tr>
<tr>
<td>Remote CORBA communication (e.g. check status)</td>
<td>0,9</td>
<td>1557</td>
</tr>
<tr>
<td>Remote CORBA communication (100 byte)</td>
<td>1,2</td>
<td>1210</td>
</tr>
<tr>
<td>Remote CORBA communication (1 Kbyte)</td>
<td>3,8</td>
<td>372</td>
</tr>
<tr>
<td>Remote CORBA communication (10 Kbyte)</td>
<td>23,1</td>
<td>62</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Migration Type</th>
<th>Time in ms</th>
<th>One Migration Equals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration without class loading (status only)</td>
<td>19,38</td>
<td>-</td>
</tr>
<tr>
<td>Remote CORBA communication (e.g. status check)</td>
<td>0,9</td>
<td>21</td>
</tr>
<tr>
<td>Remote CORBA communication (100 byte)</td>
<td>1,2</td>
<td>16</td>
</tr>
<tr>
<td>Remote CORBA communication (1 Kbyte)</td>
<td>3,8</td>
<td>5</td>
</tr>
<tr>
<td>Remote CORBA communication (10 Kbyte)</td>
<td>23,1</td>
<td>0,8</td>
</tr>
</tbody>
</table>

CORBA: communication using RPC resp. RMI
Network Load

The slower the network, the more suitable are mobile agents:

Important in wireless networks: the risk of connection loss is a strong motivation for mobile agent usage.
Stationary vs. Migrating Agents

Not each agent needs to be mobile; there is also the concept of stationary agents. The main difference between normal objects and stationary agents is the *autonomy*.

<table>
<thead>
<tr>
<th><strong>Stationary agents</strong></th>
<th><strong>Mobile agents</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Are placed on a host for particular tasks</td>
<td>Are able to migrate to other hosts</td>
</tr>
<tr>
<td>e.g.</td>
<td>e.g.</td>
</tr>
<tr>
<td>- local search service</td>
<td>- distributed search service</td>
</tr>
<tr>
<td>- database services</td>
<td>- conferencing services</td>
</tr>
<tr>
<td>- accounting</td>
<td>- mobile communications</td>
</tr>
</tbody>
</table>

**Advantages:**
- reduction of network load
- division of tasks to several agents
- better flexibility and location transparency

**Problems:**
- location of agents
- migration
- management, ...
There is no general agreed definition of agents. On the other hand, agents can be characterised by several properties:

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

By the definition of lots of agent concepts, lots of different agent systems are originated. It would be advantageous to enable an inter-platform communication and migration. But this only is possible with lots of additional efforts.

As a solution, the Foundation for Intelligent Physical Agents (FIPA, http://www.fipa.org) has developed a general architecture model for agent platforms:

![Diagram of agent platform architecture](image)
Agent Communication Languages

Mobility was no central topic in FIPA's work. The main topic was the standardisation of a uniform Agent Communication Language (ACL) which defines basic message types. The transmission of such messages builds upon communication protocols like TCP/IP.

<table>
<thead>
<tr>
<th>Message purpose</th>
<th>Description</th>
<th>Message Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFORM</td>
<td>Inform that a given proposition is true</td>
<td>Proposition</td>
</tr>
<tr>
<td>QUERY-IF</td>
<td>Query whether a given proposition is true</td>
<td>Proposition</td>
</tr>
<tr>
<td>QUERY-REF</td>
<td>Query for a give object</td>
<td>Expression</td>
</tr>
<tr>
<td>CFP</td>
<td>Ask for a proposal</td>
<td>Proposal specifics</td>
</tr>
<tr>
<td>PROPOSE</td>
<td>Provide a proposal</td>
<td>Proposal</td>
</tr>
<tr>
<td>ACCEPT-PROPOSAL</td>
<td>Tell that a given proposal is accepted</td>
<td>Proposal ID</td>
</tr>
<tr>
<td>REJECT-PROPOSAL</td>
<td>Tell that a given proposal is rejected</td>
<td>Proposal ID</td>
</tr>
<tr>
<td>REQUEST</td>
<td>Request that an action be performed</td>
<td>Action specification</td>
</tr>
<tr>
<td>SUBSCRIBE</td>
<td>Subscribe to an information source</td>
<td>Reference to source</td>
</tr>
</tbody>
</table>
FIPA's ACL allows a message exchange with a defined semantic. Example:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>INFORM</td>
</tr>
<tr>
<td>Sender</td>
<td>max@<a href="http://fanclub-beatrix.royalty-spotters.nl:7239">http://fanclub-beatrix.royalty-spotters.nl:7239</a></td>
</tr>
<tr>
<td>Receiver</td>
<td>elke@iiop://royalty-watcher.uk:5623</td>
</tr>
<tr>
<td>Language</td>
<td>Prolog</td>
</tr>
<tr>
<td>Ontology</td>
<td>genealogy</td>
</tr>
<tr>
<td>Content</td>
<td>female(beatrix),parent(beatrix, juliana, bernhard)</td>
</tr>
</tbody>
</table>

Concluding can be said, that there are two kinds of communication between agents: *Migration* (with local method invocations) and *message exchange*.