Sistemas Distribuídos

About Java and Programming Methodologies

António Rui Borges

Summary

- *Java*
  - *Historical note*
  - *Main features*
- *General description techniques of a problem solution*
- *Programming methodologies*
  - *Procedural or imperative programming*
  - *Modular programming*
  - *Object oriented programming*
  - *Concurrent programming*
  - *Distributed programming*
- *Suggested reading*

Historical note - 1

Java, originally called *Oak*, was created in the mid of the 90s by a *Green* project working group of Sun MicroSystems, headed by James Gosling

- its syntactic structure is based on the programming languages C and C++
- the initial target area for the language was building control and communication environments for embedded systems of consumer electronic appliances
  - its popularity, however, arises in a different context, that of *internet*: first, with *applets* (small programs written in Java that can be executed inside a *browser*) and, more recently, with *web services*
- in a somewhat broader perspective, Java tends to be the language of choice to write distributed applications, since it provides an integrated and uniform environment for interprocess communication over a computer network (Java in action).

Historical note - 2

Java initial objectives, as a programming language, included

- *to be totally independent of the underlying hardware platform*, this has lead to be simultaneously thought of as a *language* and as an *execution environment*: thus, programs can be transferred in run time and executed in any node of the processing mesh
- *to be inherently robust*, minimizing programming errors: it is, therefore, organized as a strongly semantic language where some C++ features, of multiple inheritance and of operator overloading, for instance, were eliminated for being judged potential sources of error; a mechanism of implicit management of dynamic memory (*garbage collection*) was also introduced and any attempt of definition of data structures outside the scope of the `class` constructor was forbidden; in this sense, some people claim that ***Java is C++ done right***
- *to promote security* in environments that are continuously exchanging information and sharing code: *pointers* were thus eliminated, trying to prevent that non-authorized programs were able to access data structues resident in memory.

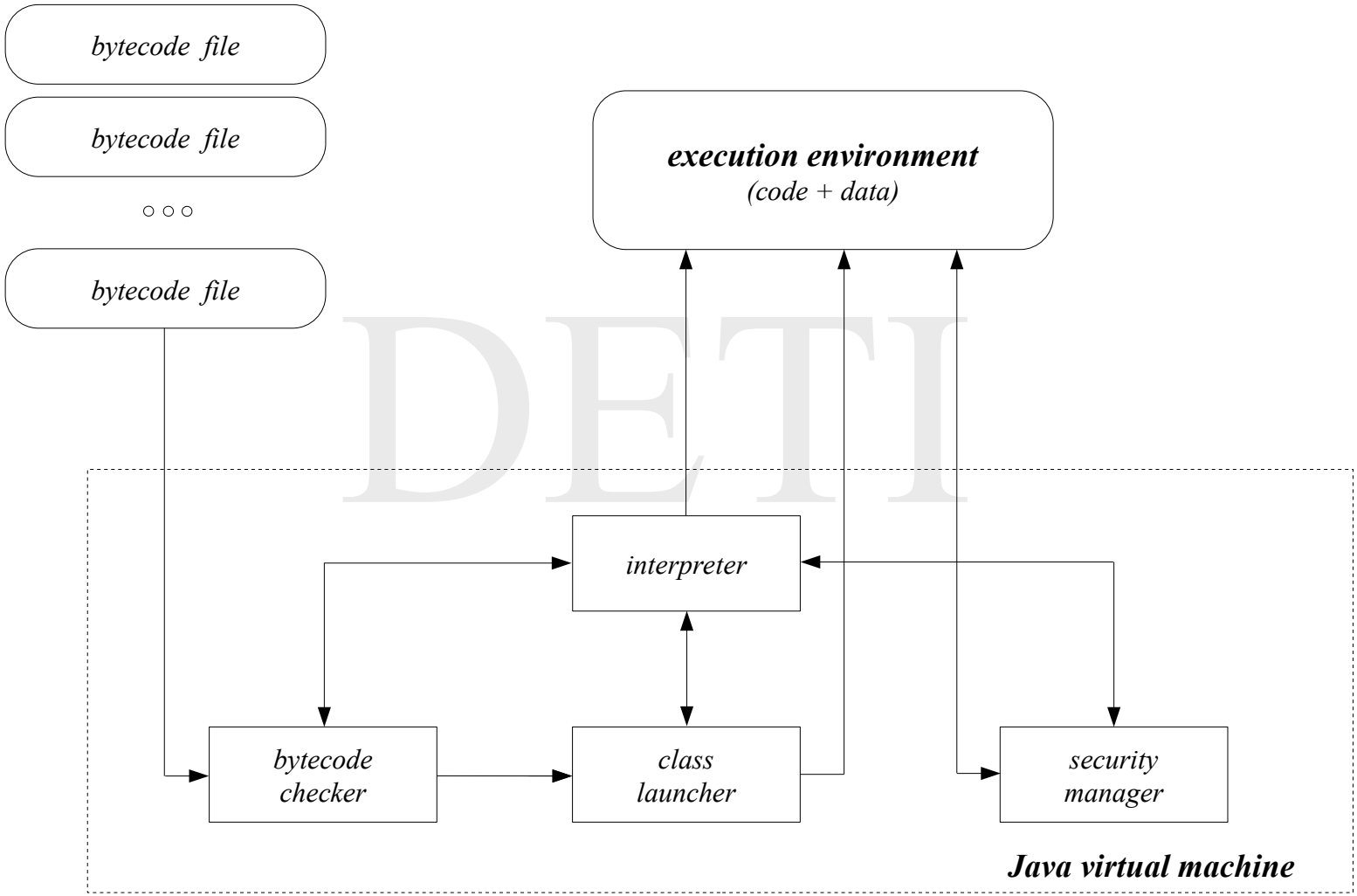
Main features - 1

- the language follows the principles of the *object-oriented paradigm*
  - there is a minimum number of semantic constructs, enabling a compact and inherently safe description
  - the inheritance mechanism is strictly linear
  - the `interface` constructor, as a special case of the `class` constructor, was introduced to enable a precise and clear separation between specification and implementation and to serve as a connecting point between variables of distinct data types
- it provides concurrency constructs
  - there is an explicit support for building *multithreading* environments
  - *objects* are implicitly transformed into *monitors* [of the Lampson / Redell type] through synchronization of its public methods
- it supports distributed programming by providing libraries which implement the two major communication paradigms
  - message passing: *sockets*
  - shared variables: *remote method of invocation* (RMI)

Main features - 2

- the Java execution environment, called *Java virtual machine* (JVM), constitutes a *middleware* layer which makes the applications to be totally independent of the hardware platform and the operating system where they run, implementing a three-layered security model that protects the system against non-trustable code
  - the *bytecode checker* parses the bytecode presented to execution and ensures that the basic rules of the Java grammar are obeyed
  - the *class launcher* delivers the required data types to the Java interpreter
  - the *security manager* deals with the problems which put potentially at stake the application related system security, controlling the access conditions of the running program to the file system, to the network, to external processes and to the window system

Main features - 3



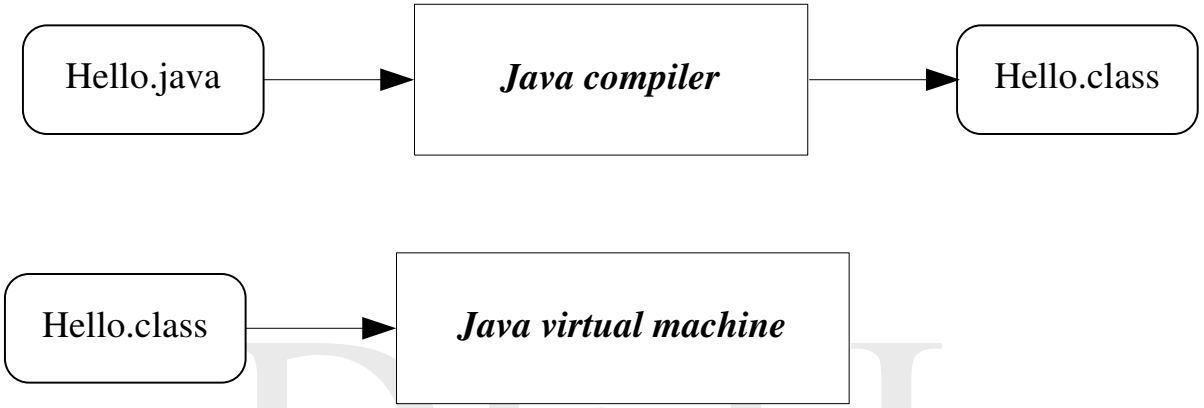
Main features - 4

- it supports internationalization
  - ASCII code is replaced by Unicode in the internal representation of characters to allow the consideration of the alphabets and ideographic symbols of different world languages
  - separation of locale information from the executable code, together with the storage of text elements outside the source code and its dynamic access, ensures that there will be a conformance of applications to the language and other specific cultural traits of the final user
- it includes tools that ease the production of program documentation
  - by taking into account the *documentation comments* inserted in the source files to describe the reference data types, its internal data structures and its access methods, it becomes trivial to generate documentation in *html* format through the application of the `javadoc` tool.

Trivial example - 1

```
/**
 *    General description:
 *     in this case, it may be the program prints the sentence
 *      <em>Hello <b>person name</b>, how are you?</em> in the monitor screen.
 *
 *    @author António Rui Borges
 *    @version 1.0 - 13/2/2017
 */

public class Hello

{
  /**
   *    Main program,
   *      it is implemented by the main method of the data type.
   *    <p>Any application must contain a static method named main in its
   *       start data type.</p>
   *
   *    @param args name of the person to salute
   */

   public static void main (String [] args)

   {
      System.out.println ("Hello " + args[0] + ", how are you?");
   }
}
```

Trivial example - 2



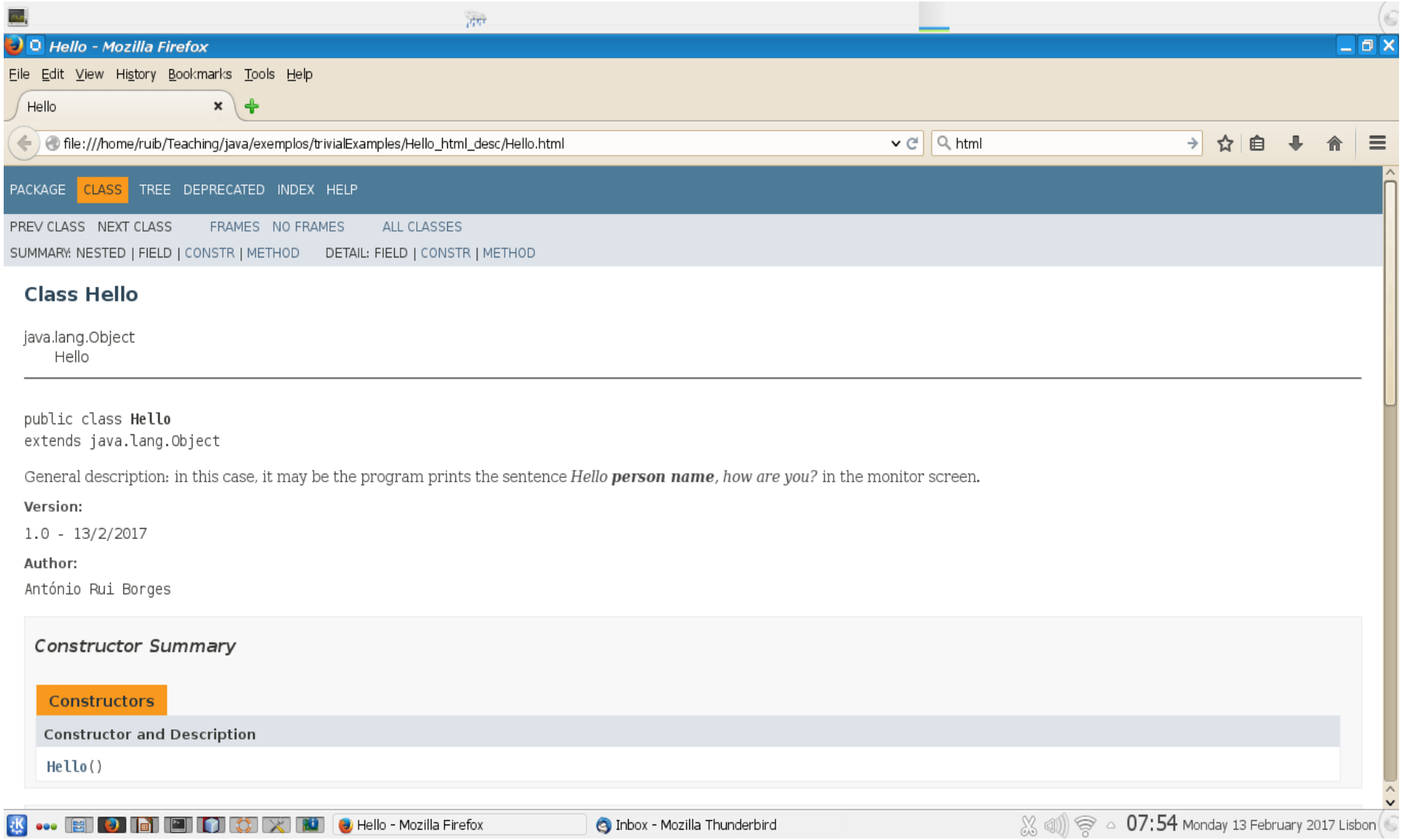

```
[ruib@ruib-laptop1 trivialExamples]$ javac Hello.java
[ruib@ruib-laptop1 trivialExamples]$ java Hello Pedro

Hello Pedro, how are you?

[ruib@ruib-laptop1 trivialExamples]$ ll
total 24
-rw-r--r-- 1 ruib ruib 599 Feb 13 07:42 Hello.class
-rw-r--r-- 1 ruib ruib 654 Feb 13 08:06 Hello.java
-rwxr--r-- 1 ruib ruib 131 Feb 13 06:42 Hello.javadoc
-rw-r--r-- 1 ruib ruib 424 Feb 13 08:07 HelloWorldApp.class
-rw-r--r-- 1 ruib ruib 149 Nov 26  2015 HelloWorldApp.html
-rw-r--r-- 1 ruib ruib 374 Feb 13 08:03 HelloWorldApp.java
[ruib@ruib-laptop1 trivialExamples]$
```

Trivial example - 3

Documentation production

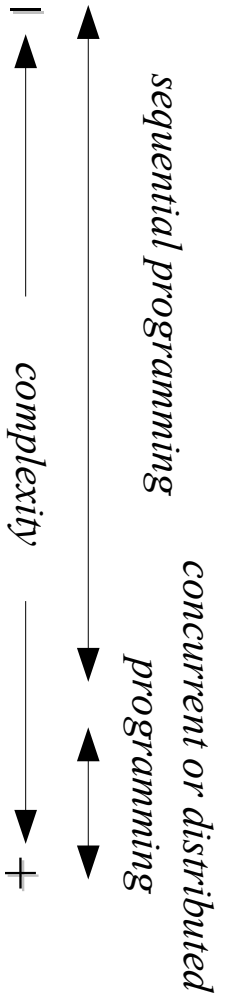
```
[ruib@ruib-laptop1 trivialExamples]$ cat Hello.javadoc
javadoc -d Hello_html_desc -author -version -breakiterator -charset "UTF-8" Hello.java
ln -s Hello_html_desc/Hello.html Hello.html

[ruib@ruib-laptop1 trivialExamples]$ ./Hello.javadoc
Loading source file Hello.java...
Constructing Javadoc information...
Creating destination directory: "Hello_html_desc/"
Standard Doclet version 1.8.0_31
Building tree for all the packages and classes...
Generating Hello_html_desc/Hello.html...
Generating Hello_html_desc/package-frame.html...
Generating Hello_html_desc/package-summary.html...
Generating Hello_html_desc/package-tree.html...
Generating Hello_html_desc/constant-values.html...
Building index for all the packages and classes...
Generating Hello_html_desc/overview-tree.html...
Generating Hello_html_desc/index-all.html...
Generating Hello_html_desc/deprecated-list.html...
Building index for all classes...
Generating Hello_html_desc/allclasses-frame.html...
Generating Hello_html_desc/allclasses-noframe.html...
Generating Hello_html_desc/index.html...
Generating Hello_html_desc/help-doc.html...
[ruib@ruib-laptop1 trivialExamples]$
```

Trivial example - 4



General description techniques of a problem solution

- *hierarchical decomposition*
  - recourse is made to a metalanguage of description, preferably similar to the programming language that will be used
  - information is encapsulated through
    - the construction of suitable data types to the characteristics of the problem
    - the definition of new operations in the context of the language
  - strict data dependencies are established
- *decomposition in interactive autonomous structures*
  - precise specification of an interaction mechanism
  - clear separation between *interface* and *implementation*
    - data abstraction and protection
    - virtualization of access operations
- *decomposition in interactive autonomous entities*
  - precise specification of a communication model
  - definition and implementation of synchronization mechanisms.



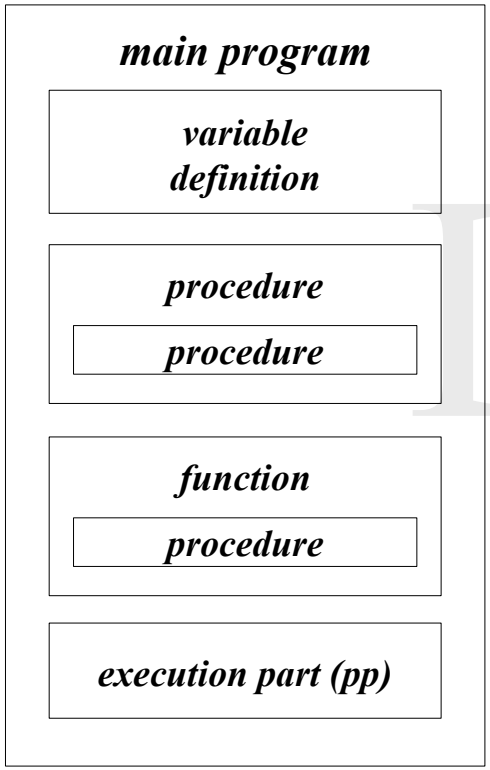
Programming methodologies - 1

Procedural or imperative programming

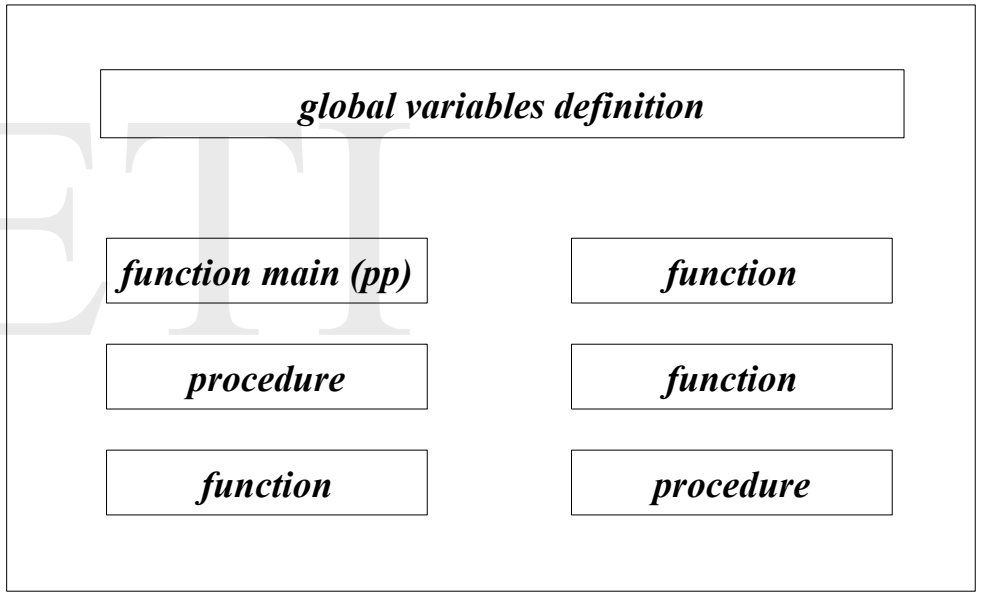
- emphasis is placed in a more or less accurate implementation of the hierarchical decomposition of the solution
- complexity is controlled by the intensive use of
  - *functions* and *procedures*, as a means to describe operations at different levels of abstraction
  - *data type constructors*, to organize the associated information in the most suitable way to the characteristics of the problem
- variable space is *concentrated*
  - all relevant data to the problem solution is defined at the *main program* level, or is global; variables local to the remaining functions and procedures only require temporary storage (they only exist when the functions or the procedures are invoked)
  - access of the different operations to data is conceived in strict obedience to the principle "*what they need to know*", using a communication model based on parameter transfer

Programming methodologies - 2

Procedural or imperative programming



hierarchical organization of a source file – Pascal like



horizontal organization of a source file – C language like (sea of functions structure)

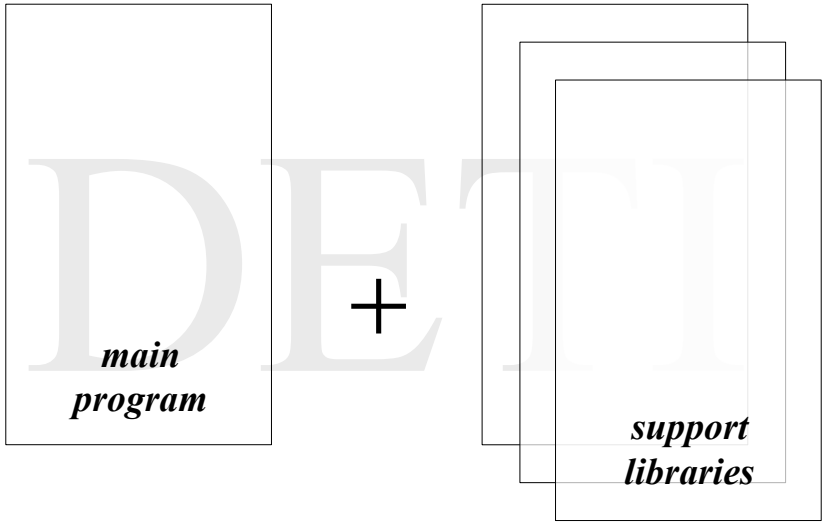
Programming methodologies - 3

Procedural or imperative programming

- an issue that immediatly comes to mind is *code reuse*
- different operations, implemented by functions and procedures in the context of a specific application, are often useful in other applications
- its automatic inclusion in the code of new applications can be done by splitting the source file into multiple files, each targeted for separate compilation and put together later on in a single executable file at the linking stage
- thus, the code of a particular application is scattered by
  - a source file that contains the main program and, eventually, other private code
  - multiple source files that contain well-defined functionalities, implemented by functions and procedures which were independently written and which will be now put to work in the context of the present application, the so called *support libraries*

Programming methodologies - 4

Procedural or imperative programming



- it becomes necessary in this type of organization to insert in the piecewise source files, whenever its code refers to external operations, the name of the source file where they are declared (*interface file*) to ensure consistency at compile time.

Programming methodologies - 5

Modular programming

- as the description complexity of the solution increases, the centralized management of the space of variables becomes progressively more difficult and emphasis moves from operation conception to the development of a finer data organization
- the space of variables becomes in fact *distributed* and the concept of *module* arises as a more convenient means to deal with its management
  - a *module* is, by definition, an autonomous structure, described in a separate source file, that encapsulates a well-defined functionality owning in general a proprietary space of variables and a set of operations to manipulate it
  - the space of variables is usually *internal*, which means that access to it can only be done by the specified operations, the so-called *access primitives*
  - it should be noted that, when the proprietary space of variables has no external meaning, a *module* becomes what was formerly called a *support library*

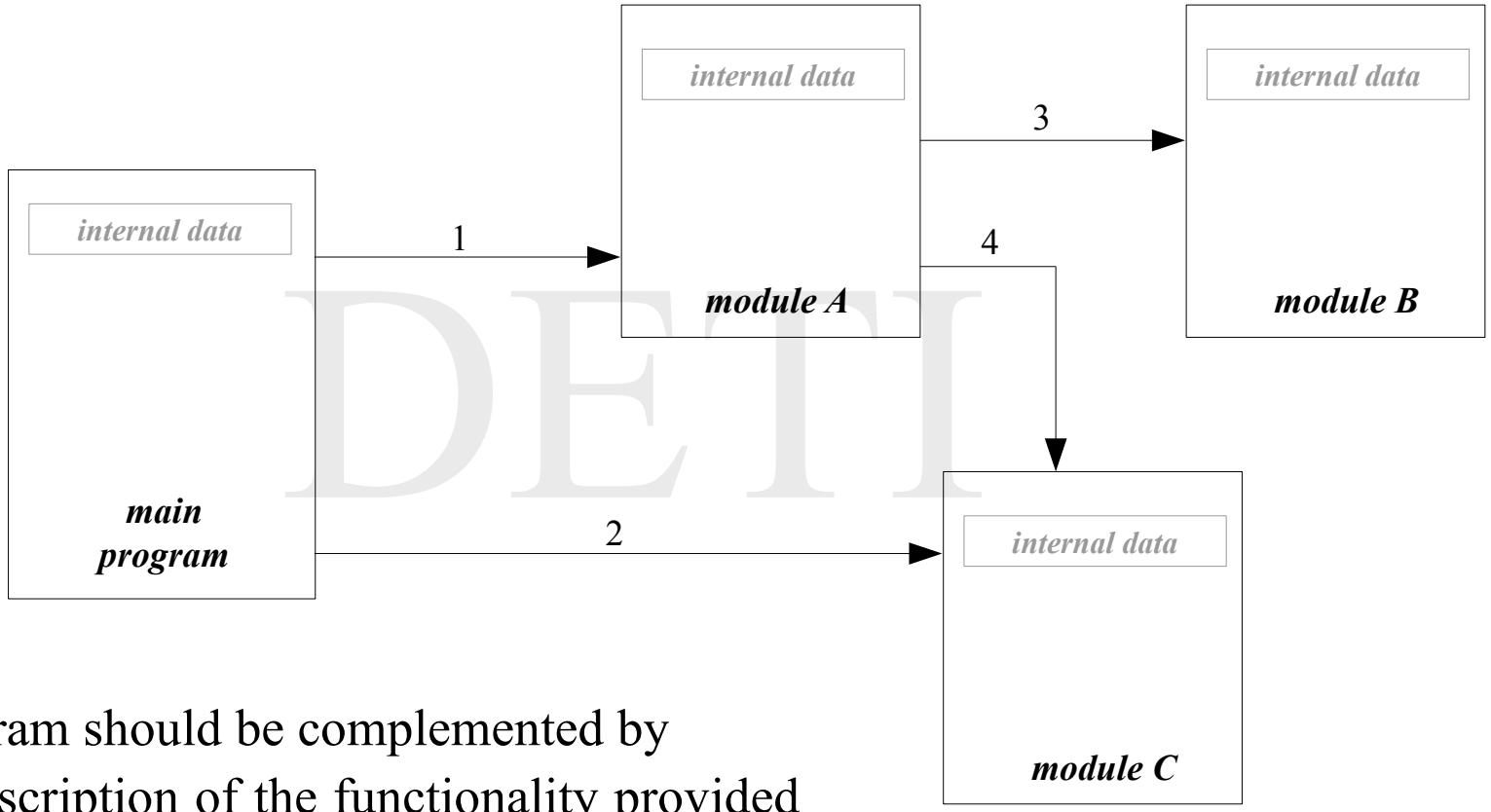
Programming methodologies - 6

Modular programming

- the application of a functional approach to solution decomposition generates the specification of a set of autonomous interacting structures and one is able to deal with much more complex problems through the creation of a clear task division and the promotion of cooperative work
  - the description is now done by specifying precisely which are the interacting structures at play and by establishing for each of them what is its role in the interaction and how access to it is to be made
  - the obvious separation between access specification, the *interface*, and implementation allows its use in the design of other modules, even before the original module has been fully concluded and validated

Programming methodologies - 7

Modular programming



- the diagram should be complemented by
  - a description of the functionality provided by each of the modules
  - a listing of the operations involved in the interaction

Programming methodologies - 8

Modular programming

- the consistent application of a *divide and conquer* strategy, which underlies any description methodology and, particularly, when a functional decomposition of the solution is carried out, enables the development of *robust* code, embodied by
  - *design for test*: as a module is an autonomous structure, it functionality can and should be validated according to the previously established specification before its inclusion in the application it is aimed to be a part of
  - *programming by contract*: the programmer, to whom it was assigned the task of its implementation, should devise means that ensure that only the described functionality is made available at *runtime*
    - all access primitives must return *status* information about the operation: *success* or *an error has occurred*, with a specific indication of the error in the latter case
    - the consistency of the internal data structure must be ensured prior to the execution of the first operation (*pre-invariant*)
    - whenever an operation is called, the value of each variable in the input parameter list must be checked to assert if it is within the allowed range, before operation execution
    - the consistency of the internal data structure must be ensured after operation execution (*post-invariant*).

Programming methodologies - 9

Object-oriented programming

- a modular implementation of a functional decomposition may become rather inflexible in terms of *code reuse* whenever the new application requires
  - the multiple instantiation of a module
  - the introduction of small changes to it
- the programmer inevitably tries to deal with the situation by adjusting the pre-existing modules to the present requirements, leading to a progressive pulverization of his/her repertoire of implemented functionalities and the consequent loss of efficiency on managing support software for the development of future applications
- a possible solution to this conundrum is an approach of increased abstraction

Programming methodologies - 10

Object-oriented programming

- the first problem may be solved by an *extension* to the concept of *data type*
- in fact, if it were possible to associate with each module instantiation a different variable, the ambiguity is removed and multiple instantiations become trivial
- in this sense, the notion of *data type* as a set of rules which allow the storage in memory of values with certain properties, is now extended to contemplate not only the storage of an aggregate of values, but also the identification of the operations that can be executed on them
- although improperly, these data types are sometimes called *abstract data types*; *reference data types*, or *user-defined data types*, are better names which are also commonly used

Programming methodologies - 11

Object-oriented programming

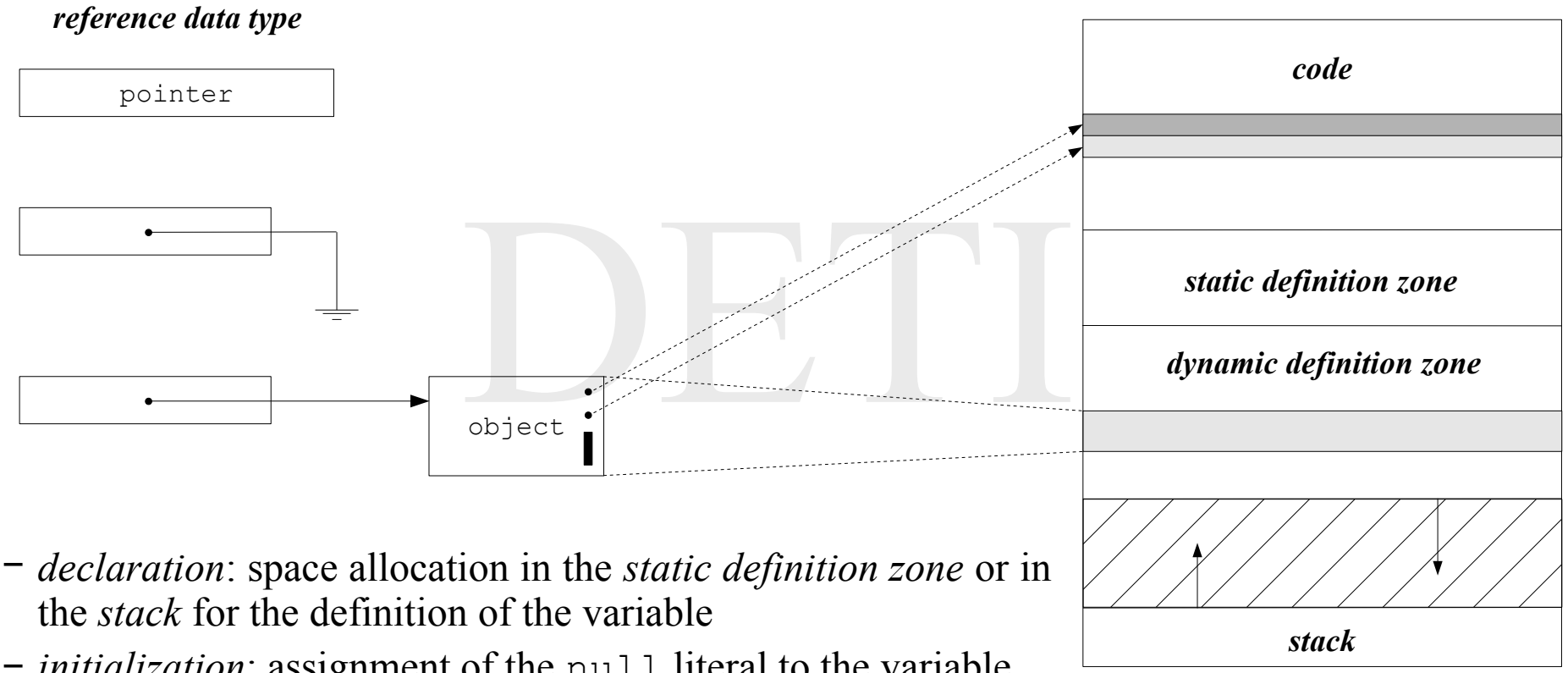
- programming languages that support them, have a constructor, whose traditional name is `class`, which basically allows the association of a module definition to an identifier
- from this point on, it becomes possible to declare variables of this type
- an important detail is that the pure declaration of variables of this type does not allocate memory space for its storage, it only allocates memory space for a *pointer*
- space allocation for the value itself is only made by the *type instantiation* in *runtime*; then, space is allocated in the dynamic definition zone of the process address space and the consequent initialization of the internal data structure takes place; furthermore, whenever the variable is no longer necessary, the storage space in dynamic memory should be freed
- the instantiated values are called *objects*, thus the paradigm name

Programming methodologies - 12

Object-oriented programming

- in *object-oriented* terminology, one calls *fields* to the variables of the internal data structure of the reference data type, and *methods* to the access primitives; an *object* is then determined by its *state* (set of values presently stored in the different fields) and by its *behavior* (set of access methods provided)
- programming languages that support the paradigm, usually allow *data abstraction* in the specification of the internal data structure, giving rise to generic data types with a wider application range
  - instead of creating a data type which implements a stack for character storage, for instance, one may create a data type which implements a stack for the storage of any kind of values
- furthermore, one may still ensure that a single type of values is stored there in *run-time* by using *parametrization*
  - taking the former example, one may create a data type that implements a stack for the storage of values of an unspecified data type `T` and postpone to the declaration and the subsequent instantiation of variables of this type the information about what type `T` really means

Object-oriented programming



- *declaration*: space allocation in the *static definition zone* or in the *stack* for the definition of the variable
- *initialization*: assignment of the `null` literal to the variable
- *instantiation*: space allocation in the *dynamic definition zone* for *object* creation (in *runtime*)

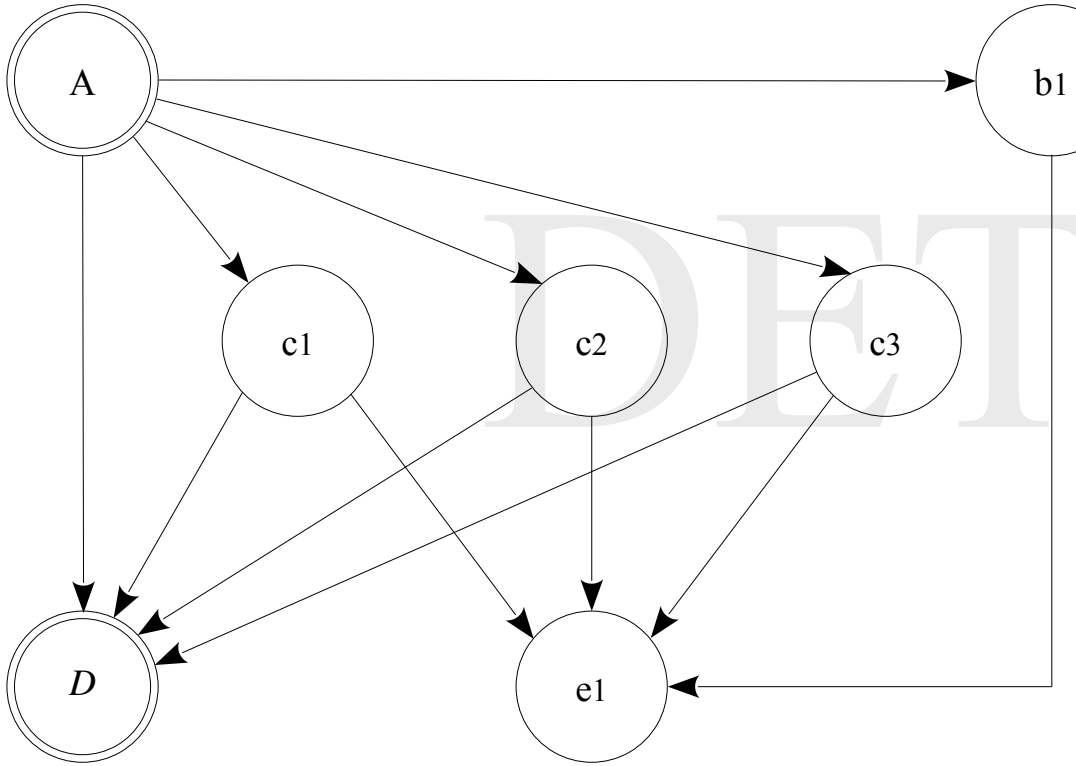
Programming methodologies - 14

Object-oriented programming

- a program organized according to the object-oriented paradigm consists of multiple source files, each defining a particular data type
- as it happens in modular programming, the interaction description is portrayed by a diagram that includes the *non-instantiated* and the *instantiated* data types, the former named by the type identifiers and the latter by the identifiers of the associated variables; the way they interact is depicted by an oriented graph which expresses the geometry of access; the diagram should be complemented with a description of the functionality each data type provides and by the listing of the operations
- as a rule, *non-instantiated* data types represent the *main program* and groups of generic operations organized in *libraries*; all the remaining data types are in principle *instantiated*

Programming methodologies - 15

Object-oriented programming



`A` – data type that contains the main program
`b1` – instantiation of data type `B`
`c1` – instantiation of data type `C`
`c2` – instantiation of data type `C`
`c3` – instantiation of data type `C`
`D` – data type that contains generic methods
`e1` – instantiation of data type `E`
→ – instantiation, access to a field or method invocation

sea of non-instantiated and instantiated data types (classes and objects)

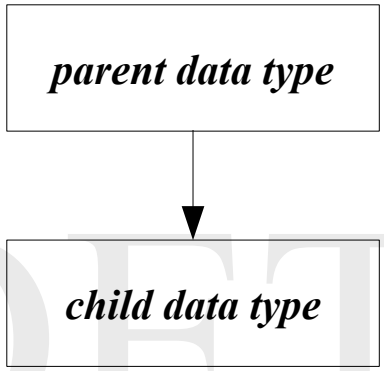
Programming methodologies - 16

Object-oriented programming

- the second problem may be solved by the application of the concept of *reuse* to the construction of data types
- indeed, if it were possible to define new data types from pre-existing data types, one can achieve their differentiation to make them suitable to new situations with a minimum of effort
- this mechanism, called *enheritance*, lets one extend in a hierarchical fashion to the new data type, *child data type* or *subtype*, the properties of the data type in which the definition is based, *parent data type* or *supertype*
- specifically, this means that
  - the *protected* fields of the internal data structure
  - the *public* methods

  of the *base* data type (parent data type) are directly accessible and, in the latter case, modifiable in the *derived* data type (child data type)

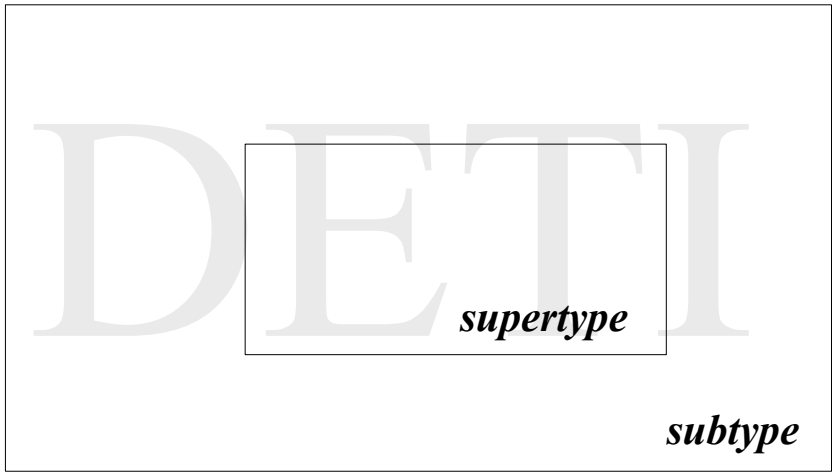
Object-oriented programming



- the creation mechanism of new data types, called *derivation* mechanism, allows one
  - to introduce new fields in the internal data structure
  - to introduce new public methods
  - to *override* methods: redefinition of [previously implemented] public methods of the base data type
  - to *implement virtual* methods: definition of public methods whose interface is in the base data type

Programming methodologies - 18

Object-oriented programming



- compatibility wise, the subtype is *compatible* with the supertype it is derived from, but the converse is not always true

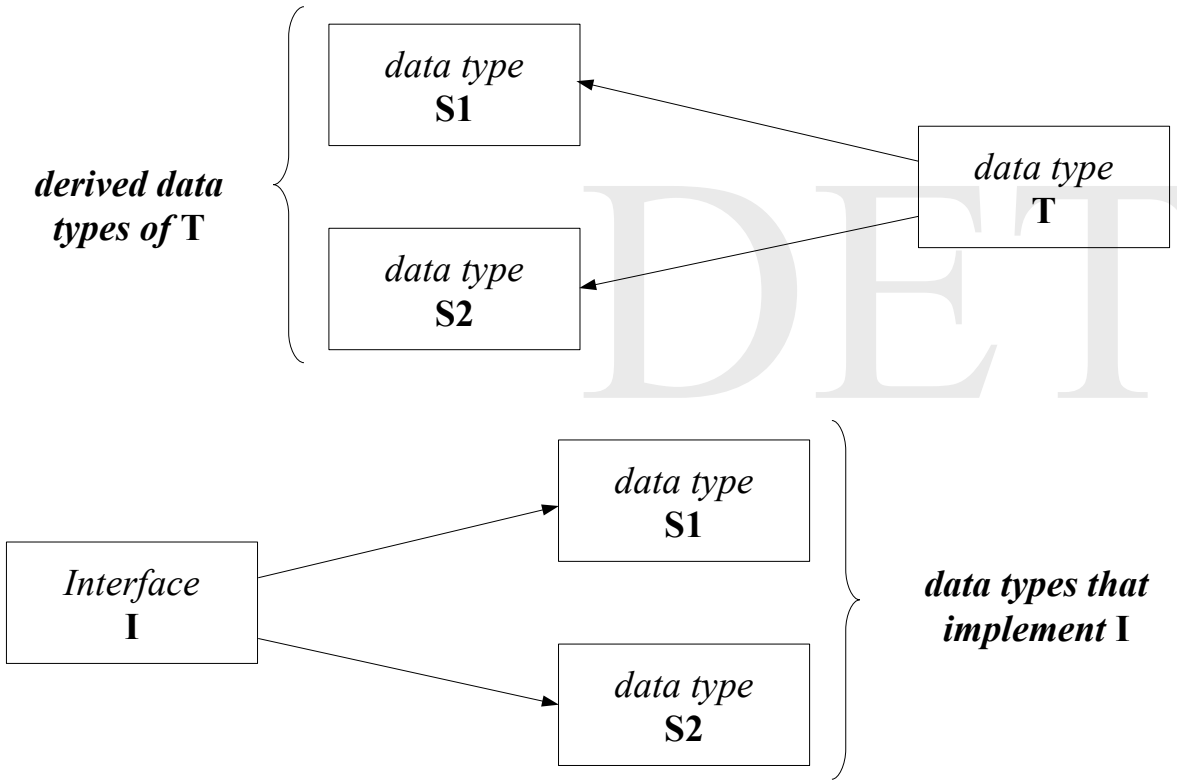
Programming methodologies - 19

Object-oriented programming

- the compatibility relation can be used to establish a very powerful principle of the object-oriented paradigm, *polymorphism*, which consists of the possibility of using the same variable to reference objects of different data types, derived from the same base type
- let `T` be a reference data type, `S1` and `S2` two distinct data types derived from the data type `T` and `t`, `s1` and `s2` variables of each one of these types where objects of the respective data types were assigned to the last two; consider now that one assigns any of these variables to the variable `t` (always a valid operation because `s1` and `s2` are compatible with `t`, although they are not mutually compatible) and one calls on it a method defined in `T`
- the association of the method suitable to the object being referenced is automatically selected and carried out
  - at the compilation stage, *static association*, when the above-referred method was not modified in the definition of any of the derived data types at play
  - in *runtime*, *dynamic association*, otherwise.

Programming methodologies - 20

Object-oriented programming




```
T t;
I i;
S1 s1 = new S1 (...);
S2 s2 = new S2 (...);

if (condition)
   t = s1;
   else t = s2;
   ...
t.m (...);
```

invocation of method m on variable t

```
   ...

if (condition)
   i = s1;
   else i = s2;
   ...
i.m (...);
```

invocation of method m on variable t

Programming methodologies - 21

Concurrent programming

- the functional decomposition of the solutions of complex problems leads quickly to the need of conceiving a group of activities that take place in a more or less autonomous manner and cooperate to achieve a common goal
- keeping a single thread of execution requires the integration in the user code of a *scheduler* which switches among the different activities
- this approach, being very complex and demanding, is also totally useless since present day operating systems allow multiprogramming, providing facilities for interprocess communication and synchronization
- furthermore, with the popularization of *multicore* processors, operating systems implement *thread* management at the *kernel* level, which means a speed up in the running of concurrent applications

Programming methodologies - 22

Concurrent programming

- two different approaches to solution design are in use
  - *event-driven approach*: the processes that implement the activities are normally blocked, waiting for an event which triggers their execution; one deals here with more or less independent processes, usually only one of them being active at a time; process communication takes place by writing and reading shared variables kept centrally; *visualization space managers*, which interact with the user through the mouse and the keyboard, are perhaps the most common example of this approach
  - *peer-to-peer approach*: the processes that implement the activities cooperate among themselves in a more or less specific manner; each is thought of as running a life cycle consisting of independent and interacting operations, the latter produce or collect information, block the process until certain conditions are attained, or wake up other processes when certain conditions are met
- whatever the case is, an always present issue is that, opposite to what happens in sequential programming, there is no guarantee of operations reproductivity, therefore, *debugging* is much more sensitive and much less effective here

Programming methodologies - 23

Concurrent programming

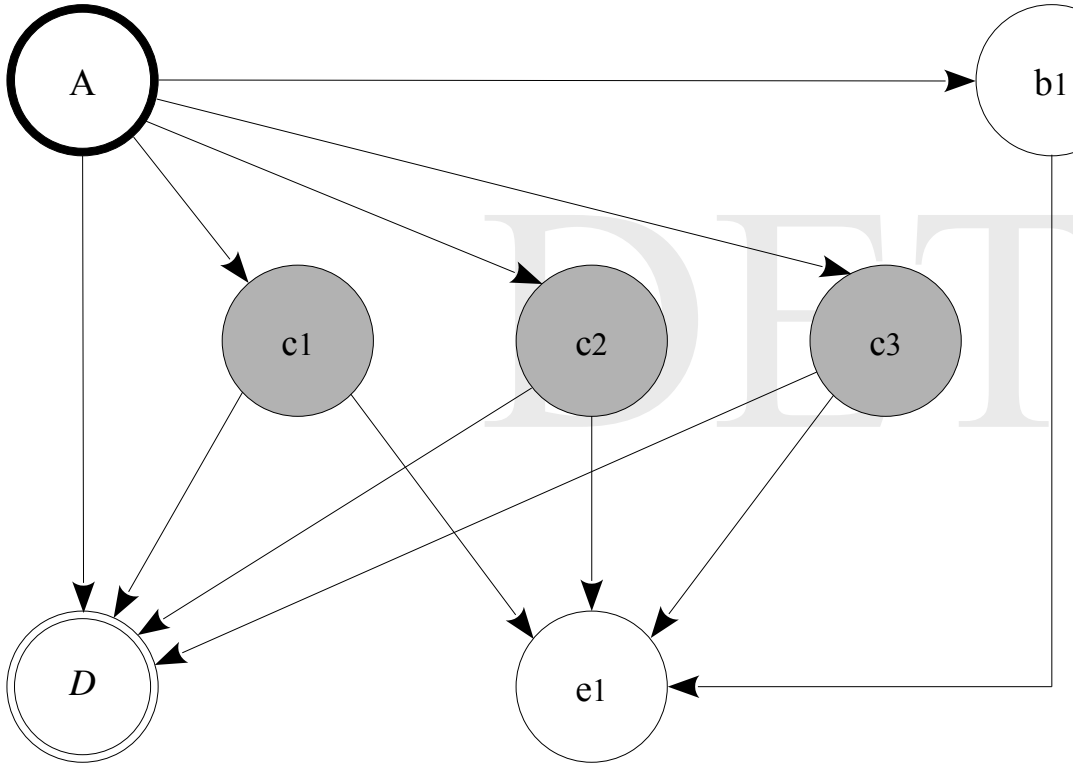
- there are two basic models for information sharing and communication
  - *shared variables*: it is the common model in *multithreaded* environments where the intervening processes write and read values stored in a centrally defined data structure; to prevent *racing conditions* that may lead to inconsistency of information, the access code to the shared region (*critical region*) has to run in mutual exclusion; there is also the need to have means for process synchronization
  - *message passing*: it is a model of universal application since it does not require the sharing of the processes addressing space; communication is carried out by message exchange between pairs of processes (*unicast*), between a process and all the others (*broadcast*), or between a process and all the others belonging to the same group (*multicast*); one assumes here that an infrastructure of communication channels, interconnecting all the processes, is available and that it is managed by an entity not belonging to the application, ensuring a mutual exclusive access to the channels and providing synchronization mechanisms

Programming methodologies - 24

Concurrent programming

- a program organized according to this methodology conforms to the principles of modular programming, or object-oriented programming, to describe the interaction and, therefore, also consists of multiple source files
- what is new here is that, by having multiple threads of execution, one needs to clearly distinguish between *active* and *passive* entities
- *active* entities represent the different intervening processes, while *passive* entities result from making explicit the different types of functionality present.

Concurrent programming



`A` – data type that contains the main program (active entity)
`b1` – instantiation of data type `B`
`c1` – instantiation of data type `C` (active entity)
`c2` – instantiation of data type `C` (active entity)
`c3` – instantiation of data type `C` (active entity)
`D` – data type that contains generic methods
`e1` – instantiation of data type `E`
→ – instantiation, access to a field or method invocation

sea of non-instantiated and instantiated data types and active and passive entities

Programming methodologies - 26

Distributed programming

- there are two main reasons that lead to moving from *concurrent programming* to *distributed programming*
  - *parallelization*: to take advantage of multiple processors and other *hardware* components of a parallel computer system to get a faster and more efficient execution of an application
  - *making a service available*: to supply a well defined functionality to an enlarged group of applications in a way that is consistent, reliable and safe
- the change in methodology entails, therefore, that one somehow finds a way to map over distinct computer systems the different processes and functionality centers a concurrent solution was previously divided

Programming methodologies - 27

Distributed programming

- the mapping can not be made in an automatic way, however; there are several issues in the concept of *parallel processing platform* which have to be carefully assessed for the migration to be possible
- some of them are
  - eventual heterogeneities among the nodes of the processing platform
  - there is no global clock to enable the chronological ordering of events
  - some of nodes of the processing platform and parts of the communication infra-structure may fail at any given time.

Suggested reading

- *On-line* support documentation for Java program developing environment by Oracle (Java Platform Standard Edition 8)

Sistemas Distribuídos

Concurrency 1

António Rui Borges

Summary

- *Program vs. Process*
  - *Characterization of a multiprogrammed environment*
- *Processes vs. Threads*
  - *Characterization of a multithreaded environment*
- *Execution environment*
- *Threads in Java*
- *Suggested readings*

Program vs. Process

Generally speaking, a *program* can be defined as a sequence of instructions which describes the execution of a certain task in a computer. However, for this task to be *in fact* carried out, the corresponding program must be executed.

A program execution is called a *process*.

Representing an activity that is taking place, a *process* is characterized by

- the *addressing space* – the code and the current value of all its associated variables
- the *processor context* – the current value of all the processor internal registers
- the *I/O context* – all the data that are being transferred to the input and from the output devices
- the *state* of the execution.

Modeling the processes - 1

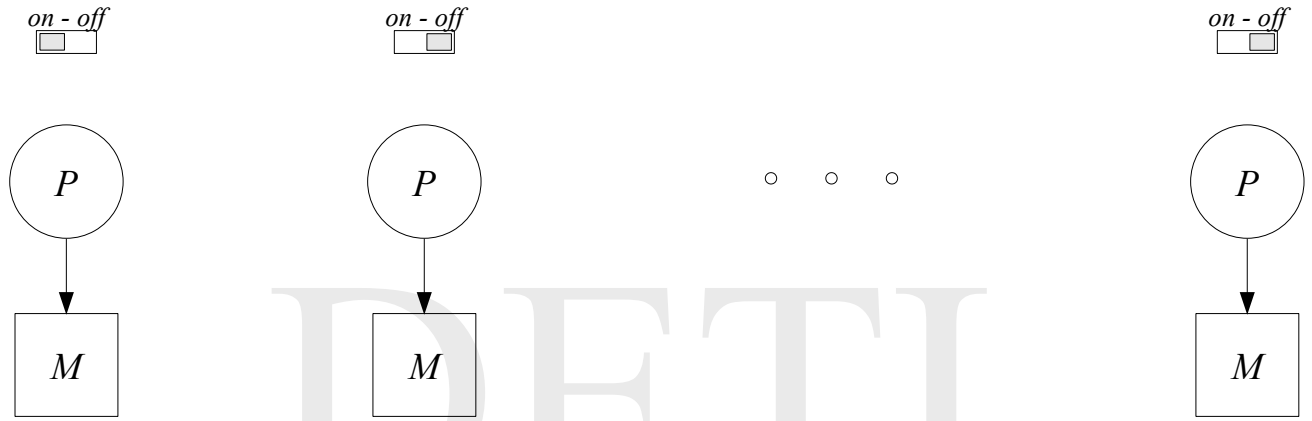
Multiprogramming, by creating an image of apparent simultaneity in the execution of different programs by the same processor, makes very difficult the perception of the activities that are taking place at the same time.

This image can be simplified if, instead of trying to follow the execution path undergone by the processor in its continuous meandering among processes, one supposes there are a set of virtual processors, one per process which concurrently coexists, and that the processes are run in parallel through the activation (*on*) and the deactivation (*off*) of the associated processors.

One further assumes in this model that

- process execution is not affected by the instant and the code location where the commutation takes place
- no restrictions are imposed to the total, or partial, execution time.

Modeling the processes - 2



- the *commutation* of the *process context* is simulated by the activation and the deactivation of the virtual processors and is controlled by its *state*
- in a *monoprocessor*, the number of active virtual processors at any given instant is one, at the maximum
- in a *multicore processor*, the number of active virtual processors at any given instant is equal to the number of processors in the core, at the maximum.

Process state diagram - 1

A process may be in different situations, called *states*, along its existence.

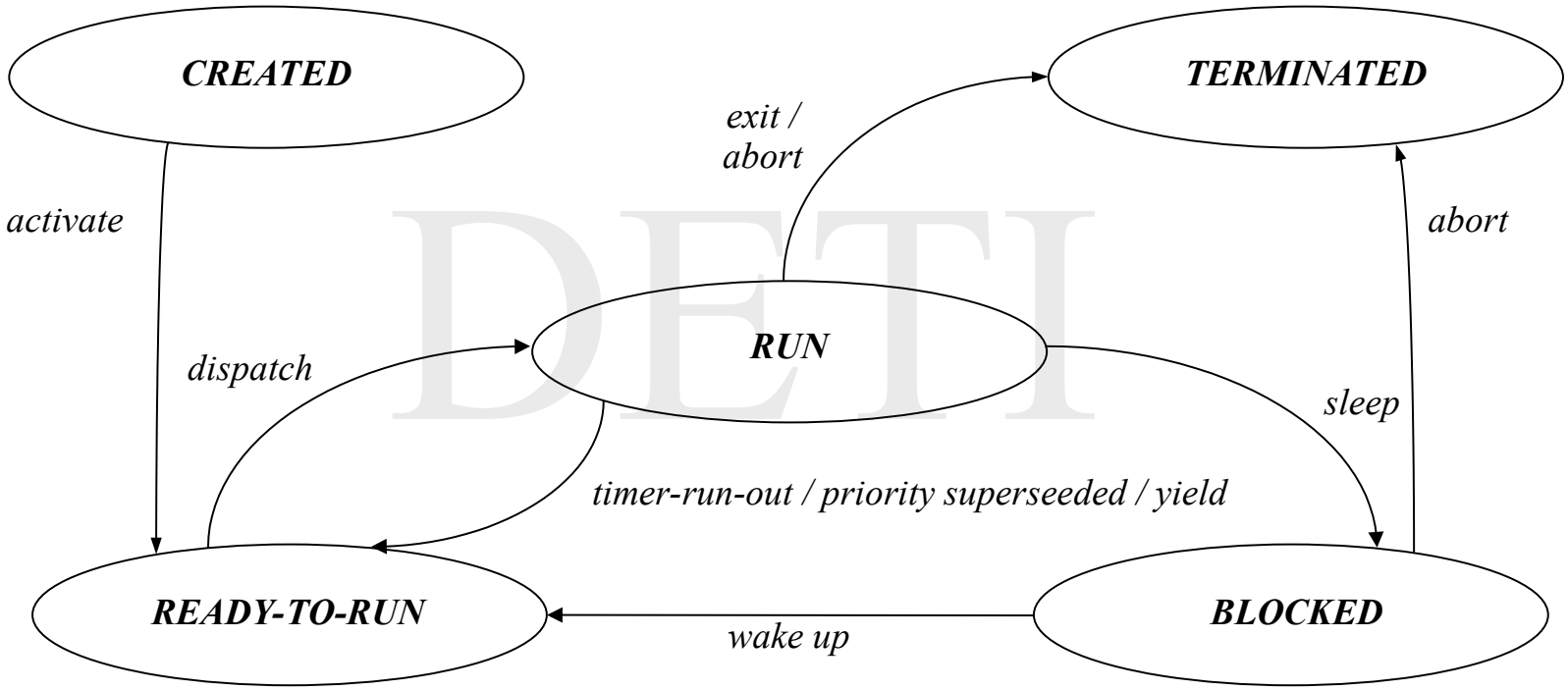
The most important states are the following

- *run* – when it holds the processor and is, therefore, in execution
- *ready-to-run* – when it waits for the assignment of the processor to start or resume execution
- *blocked* – when it is prevented to proceed until an external event occurs (access to a resource, completion of an input/output operation, etc).

State transitions are usually triggered by an external source, the operating system, but may be triggered by the process itself in some instances.

The part of the operating system which deals with [process] state transitions is called the *scheduler* (*processor scheduler*, in this case), and forms an integral portion of its nucleus, the *kernel*, which is responsible for exception handling and for scheduling the assignment of the processor and all other system resources to the processes.

Process state diagram - 2



Process state diagram - 3

activate – a process is created and placed in the *ready-to-run queue* waiting to be scheduled for execution

dispatch – one of the processes of the *ready-to-run queue* is selected by the scheduler for execution

timer-run-out – the process in execution exhausted the slot of processor time which was assigned to it (*preemptive scheduling*)

priority superseded – the process in execution looses the processor because the *ready-to-run queue* now contains a process of higher priority that requires the processor (*preemptive scheduling*)

yield – the process releases voluntarily the processor to allow other processes to be executed (*non-preemptive scheduling*)

sleep – the process is prevented to proceed and must wait for an external event to occur

wake up – the external event the process was waiting for has occurred

exit / abort – the process has terminated / is forced to terminate its execution and waits for the resources that were assigned to it to be released

Processes vs. Threads - 1

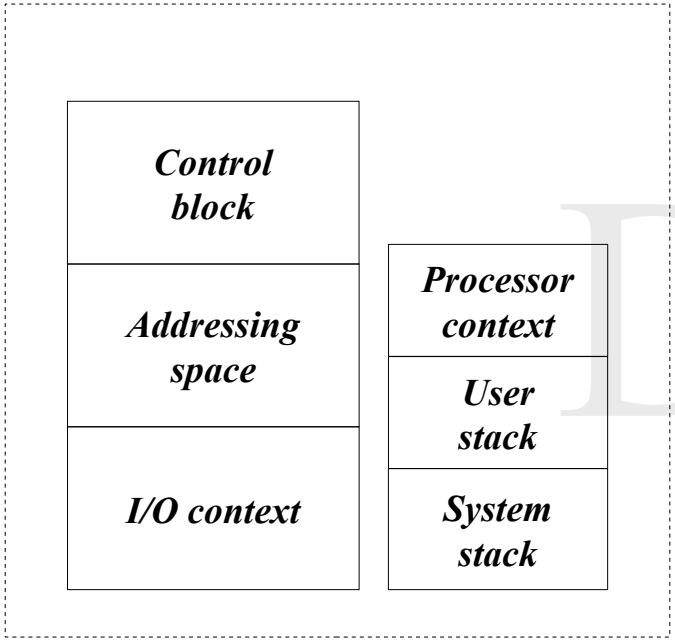
The concept of *process* embodies the following properties

- *resource ownership* – a private addressing space and a private set of communication channels with the input and output devices
- *thread of execution* – a *program counter* which points to the instruction that must be executed next, a set of *internal registers* which contain the current values of the variables being processed and a *stack* which keeps the history of execution (a *frame* for each routine that was called and has not yet returned).

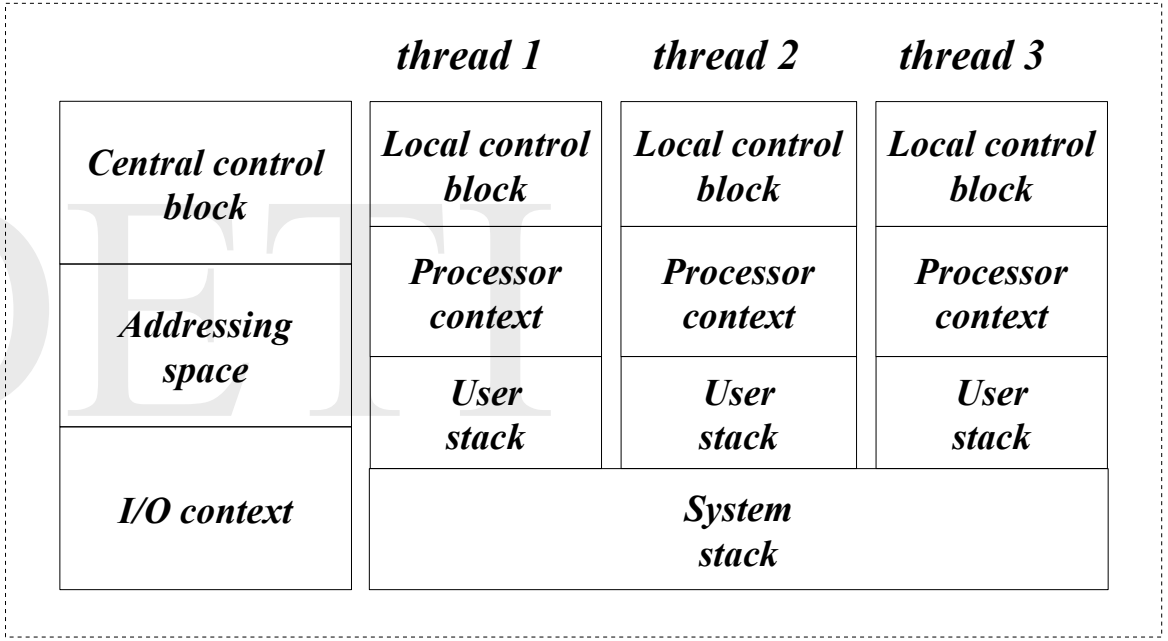
These properties, although taken together in a *process*, can be treated separately by the execution environment. When this happens, *processes* are envisaged as grouping a set of resources and *threads*, also known as *light weight processes*, represent runnable independent entities within the context of a single process.

Multithreading, then, means an environment where it is possible to create multiple *threads of execution* within the same process.

Processes vs. Threads - 2



Single threading

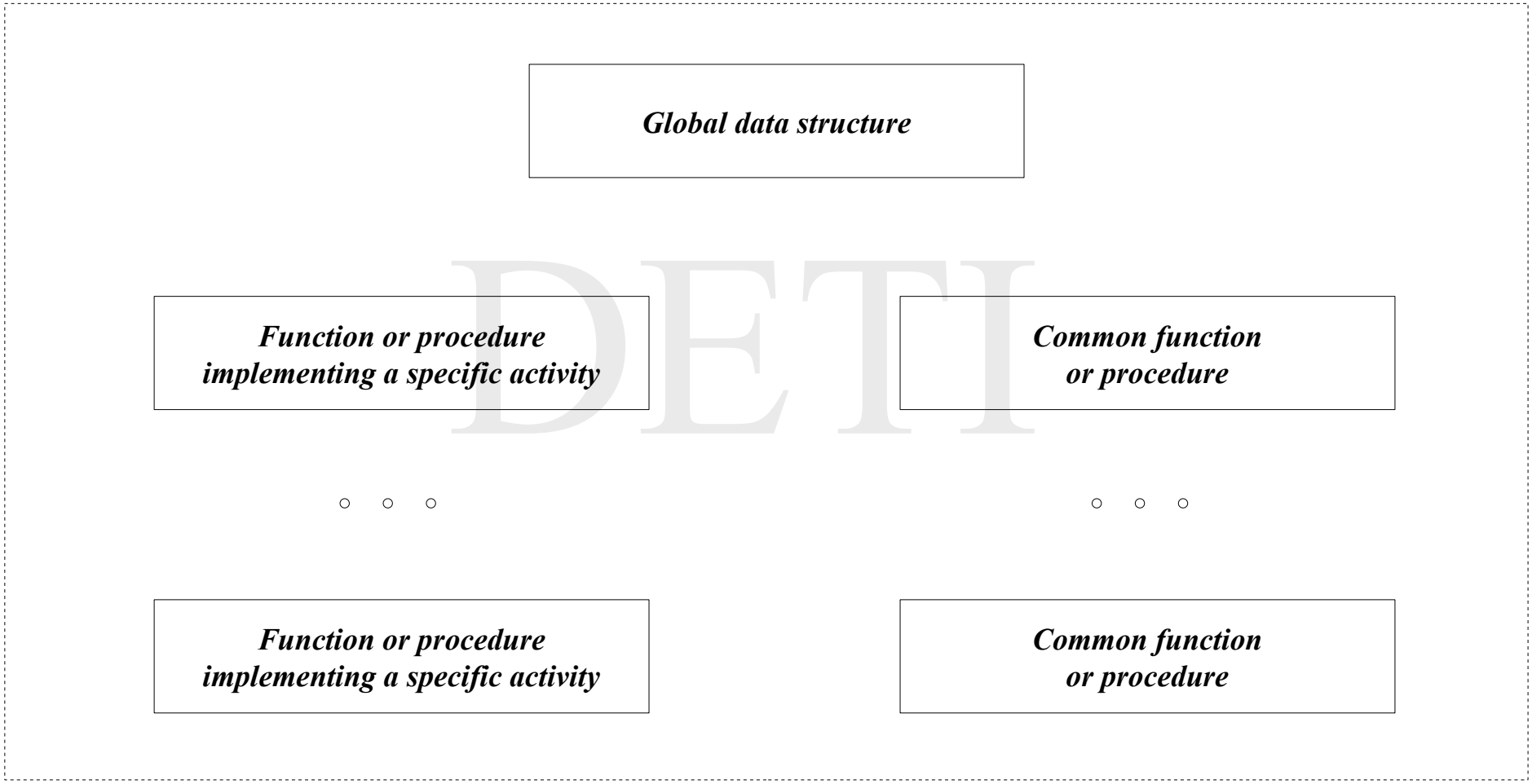


Multithreading

Advantages of a multithreaded environment

- *greater simplicity in solution decomposition and greater modularity in its implementation* – programs which involve multiple activities and service multiple requests are easier to design and implement in a concurrent perspective than in a pure sequential one
- *better management of computer system resources* – sharing the addressing space and the I/O context among the *threads* of an application results in decreasing the complexity of managing main memory occupation and access to the input / output devices
- *greater efficiency and speed of execution* – a solution decomposition based on *threads*, by opposition to one based on processes, requires less resources of the operating system, enabling that operations like process creation and termination and a context commutation to become less heavy and, thus, more efficient; furthermore, it becomes possible in symmetric multiprocessing to schedule for parallel execution multiple *threads* belonging to the same application, thus increasing the speed of execution

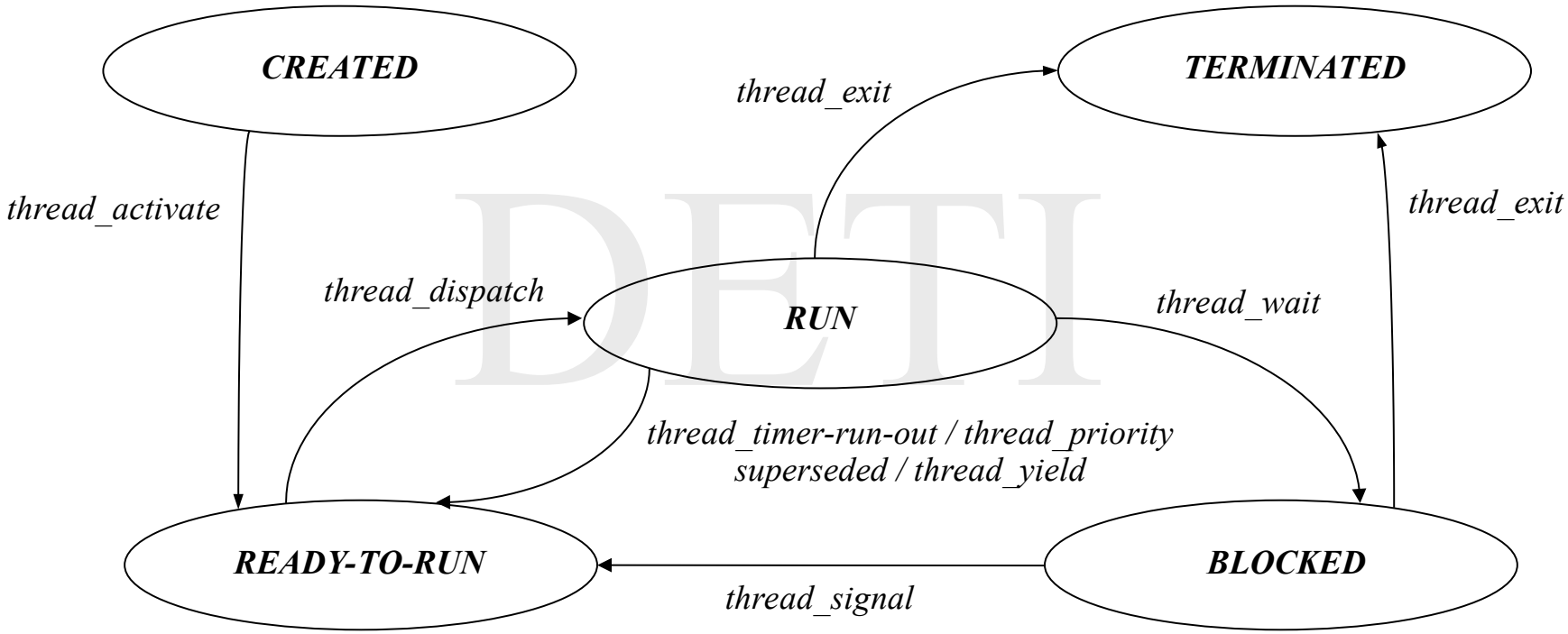
Organization of a multithreaded program - 1



Organization of a multithreaded program - 2

- each *thread* is typically associated with the execution of a *function or procedure implementing a specific activity*
- the *global data* structure forms an information sharing space, defined in terms of variables and communication channels with the input/output devices, to be accessed by the multiple *threads* that coexist at a given time for writing and for reading
- the *main program*, represented in the diagram by a *function or procedure implementing a specific activity*, constitutes the first *thread* to be created and, typically, the last *thread* to be concluded

Thread state diagram



Support to the implementation of a multithreaded environment

- *user level threads – threads* are implemented by a specific library at user level which brings support to the creation, management and scheduling of *threads without* kernel interference; this generates a very versatile and portable, but inefficient, implementation since, as the kernel perceives only processes, when a particular *thread* invokes a blocking *system call*, all the process is blocked, even if there were *threads* ready to be run
- *kernel level threads – threads* are implemented at kernel level by directly providing the operations for the creation, management and scheduling of *threads*; the implementation is operating system specific, but the blocking of a particular *thread* does not affect the dispatching of the remaining for execution and parallel execution in a multicore processor becomes possible

Execution environment - 1

Java virtual machine (JVM) constitutes the execution environment for a Java coded application. In principle, JVM runs on the top of the operating system of the hardware platform executing a Java program and establishes with it a very intimate connection. Access to information concerning the execution environment can be obtained through the invocation of methods on two reference data types of the Java base library, `java.lang`: `Runtime` and `System`.

Among the information that is provided, the following should be noticed

- number of processors and memory size available to run the code
- references to the streams associated with standard input, standard output and standard error devices
- access to a modifiable set of definitions, called *properties*, which characterize the execution environment
- read-only access to the definitions of the variables of the operating system user interface, the *shell*, where the java command was executed – called in this context *environment*.

environment - 1

```
[ruib@ruib-laptop environment]$ java CollectEnvironmentData

Characterization of Java Virtual Machine (JVM)
N. of available processors = 8
Size of dynamic memory presently free (in bytes) = 248250352
Size of total dynamic memory (in bytes) = 249561088
Maximum size of available main memory of the hardware platform where Java
virtual machine is installed (in bytes) = 367840460

Properties of the execution environment
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = /opt/jdk1.8.0_241/jre/lib/amd64
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
path.separator = :
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
user.dir =
     /home/ruib/Teaching/SD/2020_2021/aulas teóricas/exemplos demonstrativos/
         threadBasics/environment
java.runtime.version = 1.8.0_241-b07
java.io.tmpdir = /tmp
os.name = Linux
sun.jnu.encoding = UTF-8
os.version = 5.10.22-100.fc32.x86_64
user.home = /home/ruib
```

. . .

environment - 2

Variables of the execution environment

```
PATH   =   /opt/jdk1.8.0_241/bin:/opt/jdk1.8.0_241/jre/bin:/opt/mpich/bin:/home/
ruib/.local/bin:/home/ruib/bin:/usr/lib64/ccache:/usr/local/bin:/usr/bin:/
bin:/usr/local/sbin:/usr/sbin
LC_MEASUREMENT = pt_PT.UTF-8
LC_COLLATE = pt_PT.UTF-8
LOGNAME = ruib
PWD      =      /home/ruib/Teaching/SD/2020_2021/aulas      teóricas/exemplos
demonstrativos/threadBasics/environment
c=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36
:*.xspf=00;36:
XDG_SESSION_DESKTOP = KDE
SHLVL = 1
LC_MONETARY = pt_PT.UTF-8
DISPLAY = :0
LC_NUMERIC = pt_PT.UTF-8
HOME = /home/ruib
                                    .    .    .
```

Execution environment - 2

The execution environment also allows running commands directly on the underlying operating system user interface. Two reference data types of the Java base library, `java.lang`, are instrumental to fulfill this purpose: `ProcessBuilder` and `Process`.

Each `ProcessBuilder` object manages process attributes such as the *shell* command to be executed and the setting of the working directory and of the streams associated with standard input, standard output and standard error devices. Multiple processes can be created in succession from the same object, thus sharing the same attribute configuration.

Each process that is created is an instance of the `Process` data type. Means are provided here to check if it is still running, wait for its termination, get its termination status, kill it and obtain references to the streams associated with its standard input, standard output and standard error devices.

runCommand

```
[ruib@ruib-laptop runCommand]$ java ListWorkDir
Listing current working directory

----------------------------------------
total 16
drwxrwxr-x. 2 ruib ruib 4096 Mar 17 12:09 .
drwxrwxr-x. 8 ruib ruib 4096 Feb 26  2018 ..
-rw-rw-r--. 1 ruib ruib 1298 Mar 17 12:09 ListWorkDir.class
-rw-rw-r--. 1 ruib ruib 1626 Feb 28  2017 ListWorkDir.java
----------------------------------------
exit status = 0
```

Threads in Java - 1

Being Java a concurrent programming language, *threads* are supported by the language itself. Conceptually, the creation of a *thread* presupposes two entities to exist in the Java virtual machine: an object representing an autonomous thread of execution, the *thread* itself, and a non-instantiated reference data type, or an object instantiated from it, which defines the method executed by the *thread*, its life cycle.

Java virtual machine manages the *multithreaded* environment according to the following rules

- every running program consists of at least one *thread* which is implicitly created when the virtual machine, after initializing the execution environment, calls the method `main` on the initial data type
- the remaining *threads* are explicitly created by the *thread* `main`, or by any *thread* created in succession from the *thread* `main`
- the program terminates when all the created *threads* have finished executing their associated method.

Threads in Java - 2

Java base library, `java.lang`, provides two reference data types, one using the constructor *interface* and the other using the constructor *class*, which are instrumental in building a *multithreaded* environment.

```
public interface Runnable
{
  public void run ();
}


public class Thread
{
    . . .
  public void run ()
  public void start ()
    . . .
}
```

Threads in Java - 3

Each autonomous thread of execution is an instantiation of the reference data type `Thread`. It defines two methods which are operationally relevant in this context

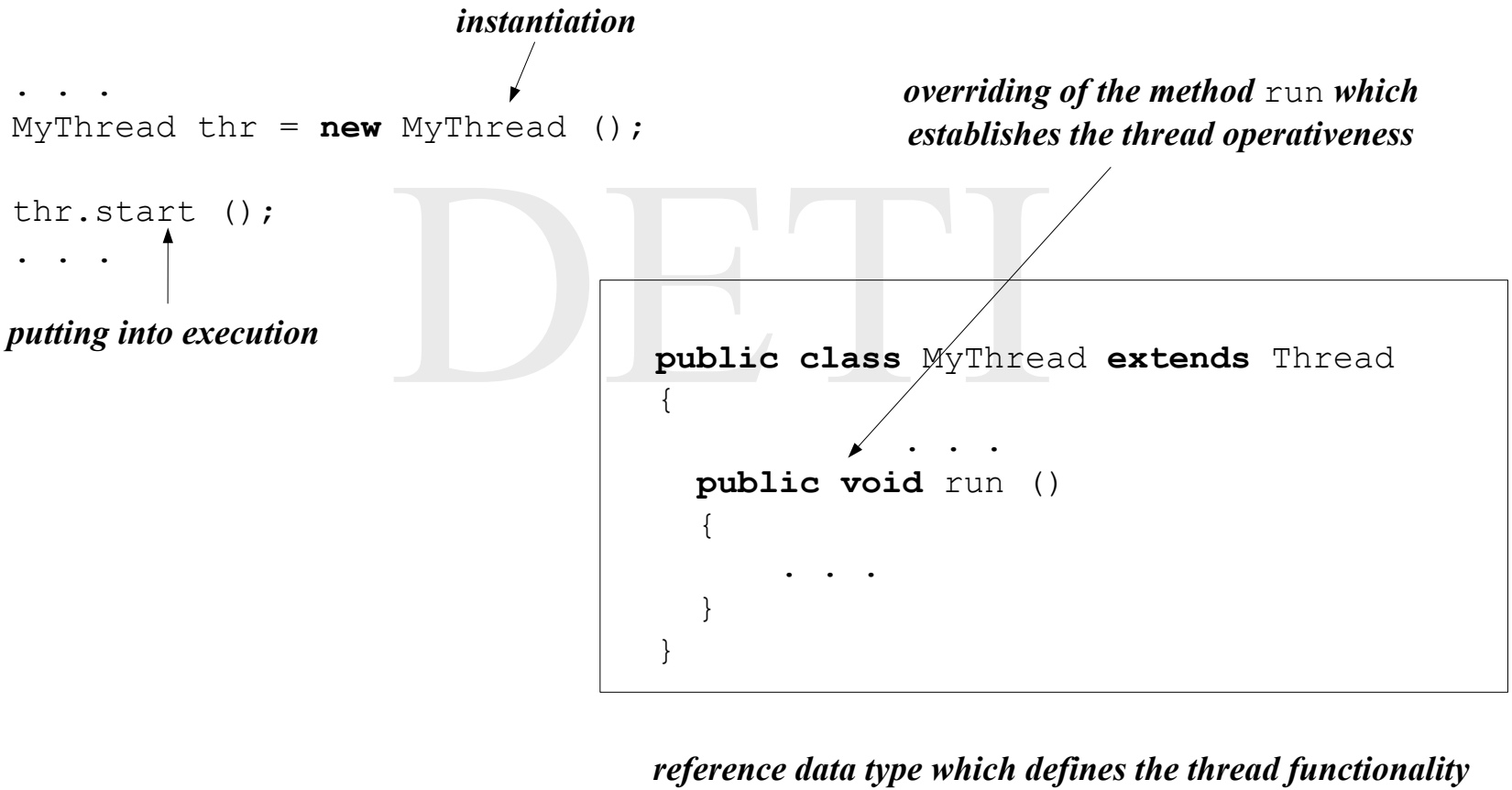
- `run` – which is called when the *thread* is put into execution (started) and which represents its life cycle
- `start` – which is called to *start* the *thread*.

It is not, however, strictly necessary to create new reference data types, derived from `Thread` and which *override* the method `run`, to ensure the execution of specific tasks. The same goal may alternatively be achieved by creating an independent reference data type which implements the interface `Runnable` and which, as a consequence, defines `run`.

The latter is usually reported as the preferred approach in Java related literature, but the former enables a Java solution to multiple inheritance which is quite useful in building servers that have a client service differentiation.

Threads in Java - 4

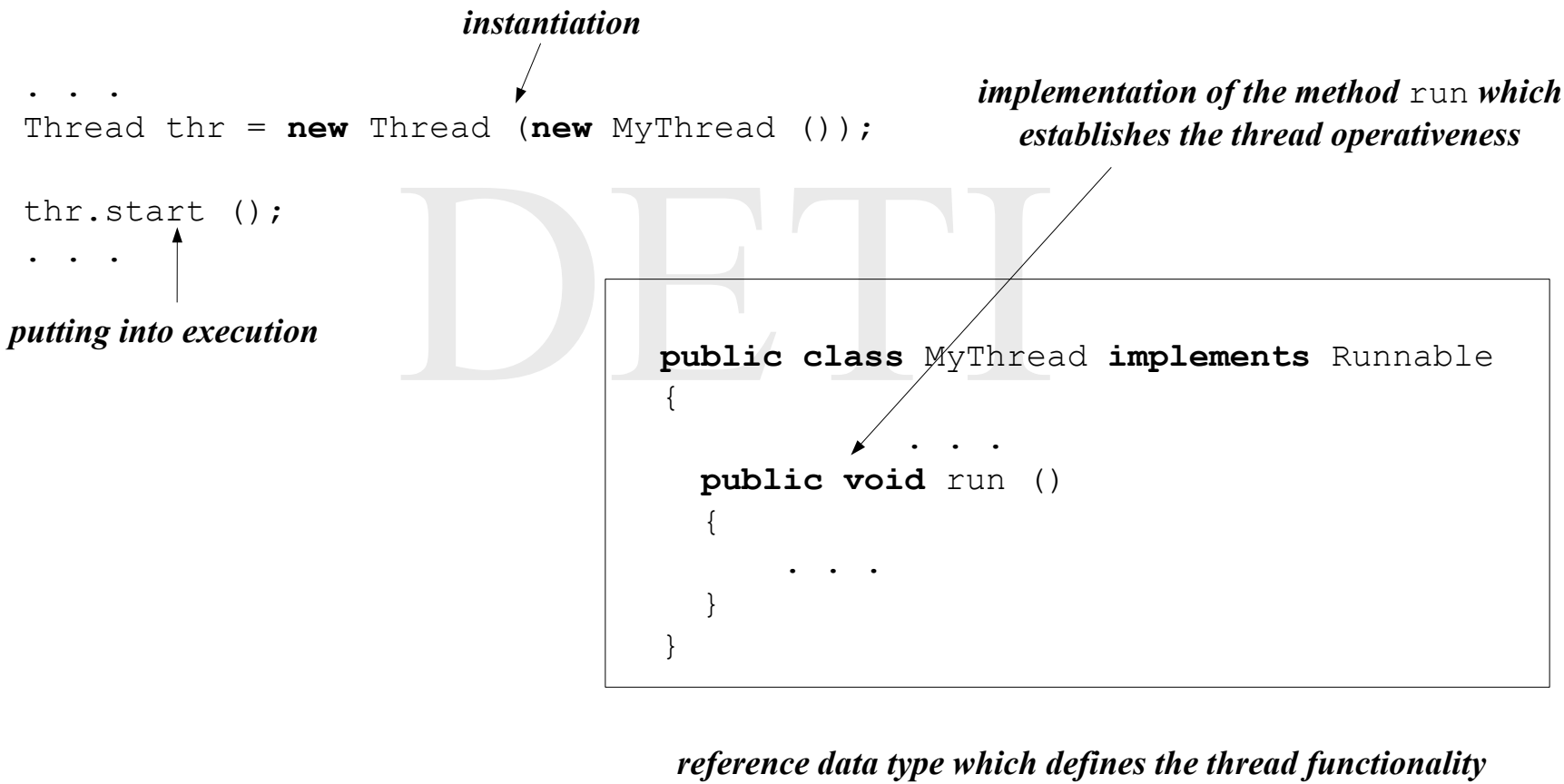
Thread creation (approach 1)



reference data type which defines the thread functionality

Threads in Java - 5

Thread creation (approach 2)



Threads in Java - 6

A *thread* has the following attributes

- *name* – assigned name (by default, the execution environment generates a *string* of format `Thread-#`, where `#` is the creation number successively incremented from zero)
- *internal identifier* – number of type `long` which unique and is kept unchanged during the *thread* lifetime
- *group* – group the *thread* belongs to (all the *threads* of the same application belong by default to the same group, the group `main`)
- *priority* – it may vary from 1 (*`MIN_PRIORITY`*) to 10 (*`MAX_PRIORITY`*), by default, the execution environment assigns the value of 5 (*`NORM_PRIORITY`*)
- *state* – *thread* current state
  - `NEW` (*CREATED*), after instantiation of a reference variable of data type `Thread`, or of a derived data type
  - `RUNNABLE` (*READY-TO-RUN* or *RUN*), when waits for execution or is in execution
  - `BLOCKED`, `WAITING` or `TIME_WAITING` (*BLOCKED*), when is blocked
  - `TERMINATED` (*TERMINATED*), after termination.

threadInfo

```
[ruib@ruib-laptop threadInfo]$ java CollectThreadData

Thread characterization

Name = main
Internal identifier = 1
Group = main
Priority = 5
Current state = RUNNABLE

Possible states:
   NEW
   RUNNABLE
   BLOCKED
   WAITING
   TIMED_WAITING
   TERMINATED
```

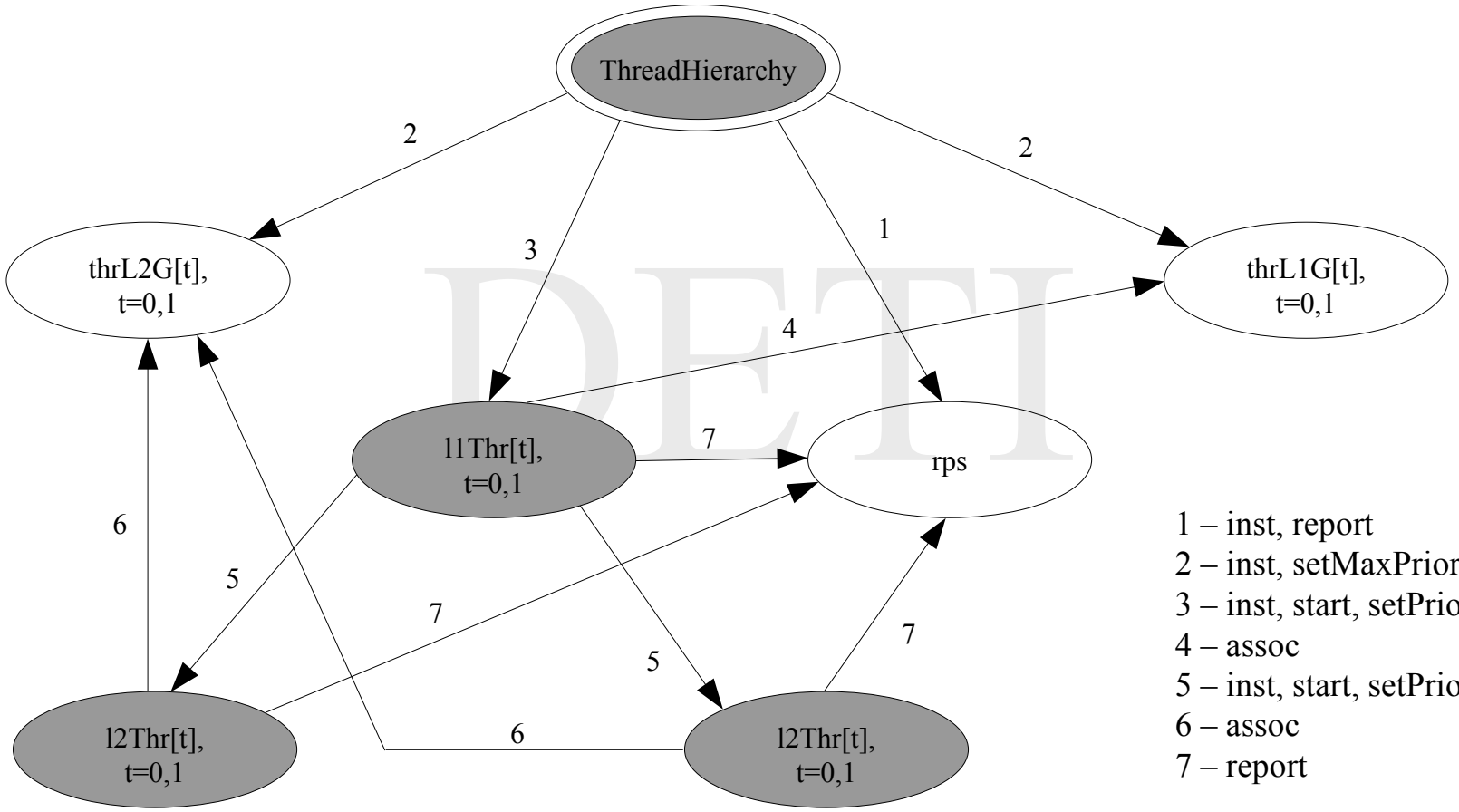
Threads in Java - 7

The partition in different groups of the *threads* which are instantiated in succession, makes possible to organize in a hierarchical and functional fashion an application, as well as taking advantage of the Java specificities in the definition of common properties and in the interaction with them. Java base library, `java.lang`, provides the reference data type `ThreadGroup` to fulfill this purpose.

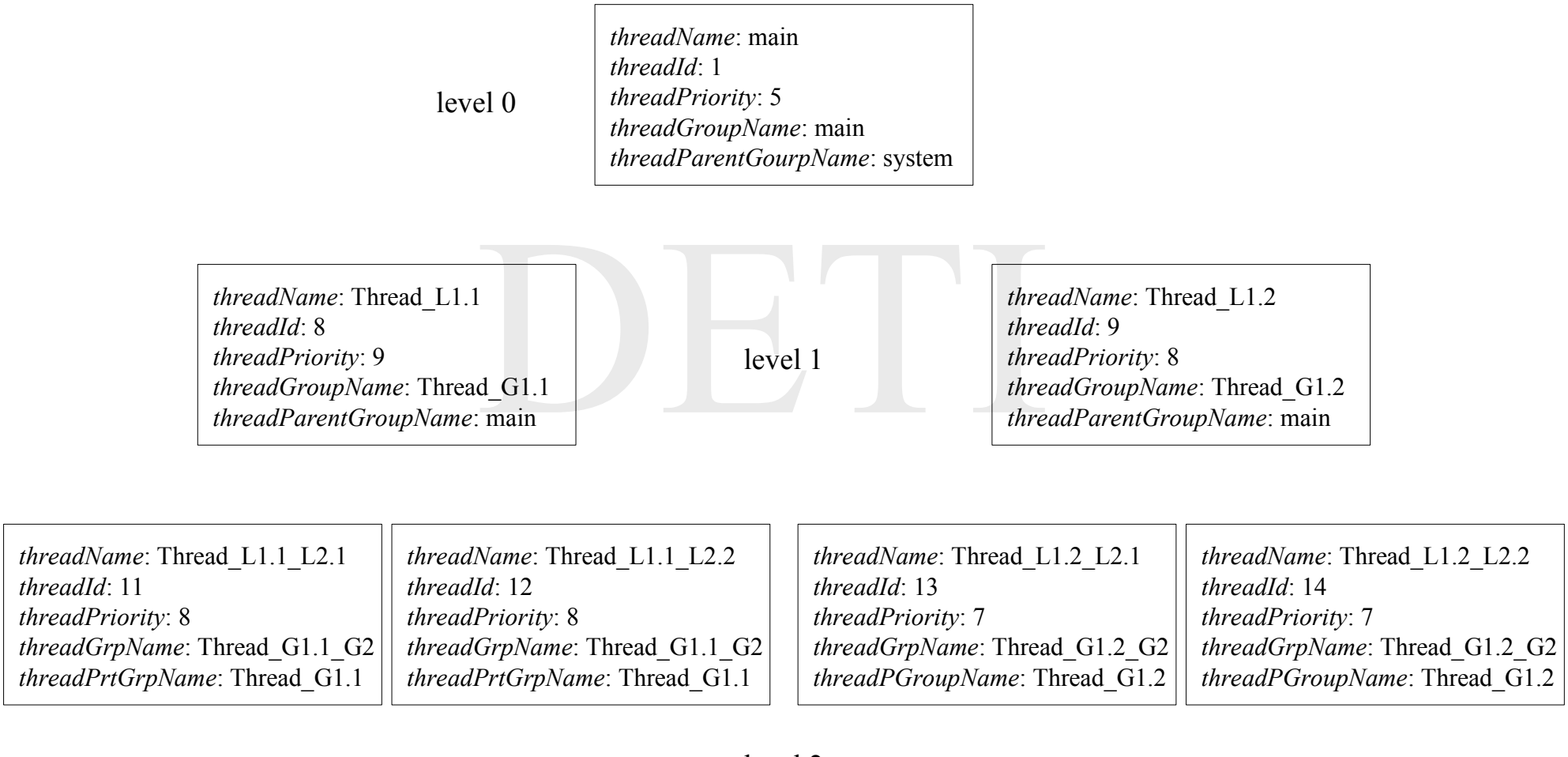
Thus, it turns out that it is possible

- to define at the very beginning the maximum priority and the property of being a *daemon* associated with a given group – all the *threads* later instantiated, and belonging to this group, will keep these properties
- to send an interrupt to *all* the *threads* belonging to the same group in an unique operation, instead of doing it separately to each member of the group.

hierarchy - 1



hierarchy - 2



hierarchy - 3

Printed values

```
[ruib@ruib-laptop hierarchy_1]$ java ThreadHierarchy1

Number of level 1 threads? 2
Number of level 2 threads per level 1 threads? 2

Thread name: main
Thread id: 1
Thread priority: 5
Name of the thread group: main
Name of the parent group of the thread group: system
N. of active threads in the thread group: 1
Name of active threads in the thread group: main
N. of active subgroups in the thread group: 0

Thread name: Thread_L1.1_L2.1
Thread id: 11
Thread priority: 8
Name of the thread group: Thread_G1.1_G2
Name of the parent group of the thread group: Thread_G1.1
N. of active threads in the thread group: 2
Name of active threads in the thread group: Thread_L1.1_L2.1 -
N. of active subgroups in the thread group: 0
                                        .   .   .
```

Threads in Java - 8

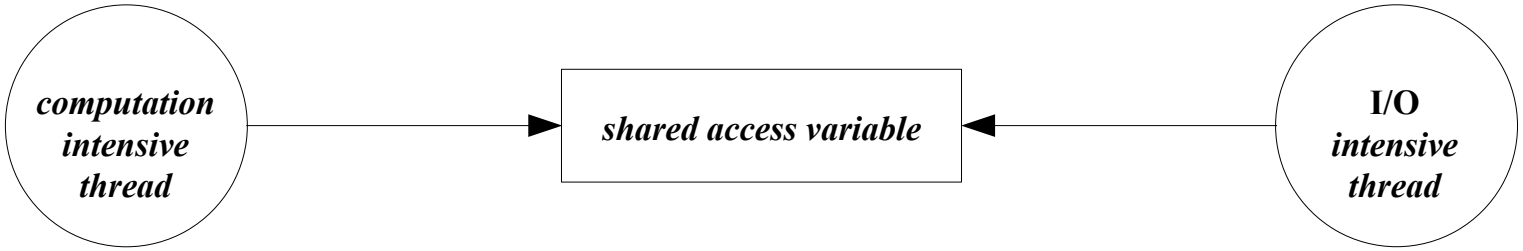
Java virtual machine supposes a policy of *non-preemptive scheduling* based on a system of static priorities with 10 levels

- the transitions between the state *RUN* and the state *READY-TO-RUN* are of type *thread_priority_superseded* and *thread_yield*
- the priority assigned to a *thread*, defined upon instantiation or modified before its creation, remains unaltered during its active life time.

Java virtual machine, however, does not enforce strictly the *scheduling* policy. The implementation has a lot of freedom on how it puts it to work. This is particularly true when Java virtual machine is run on the top of a general purpose multitasking operating system!

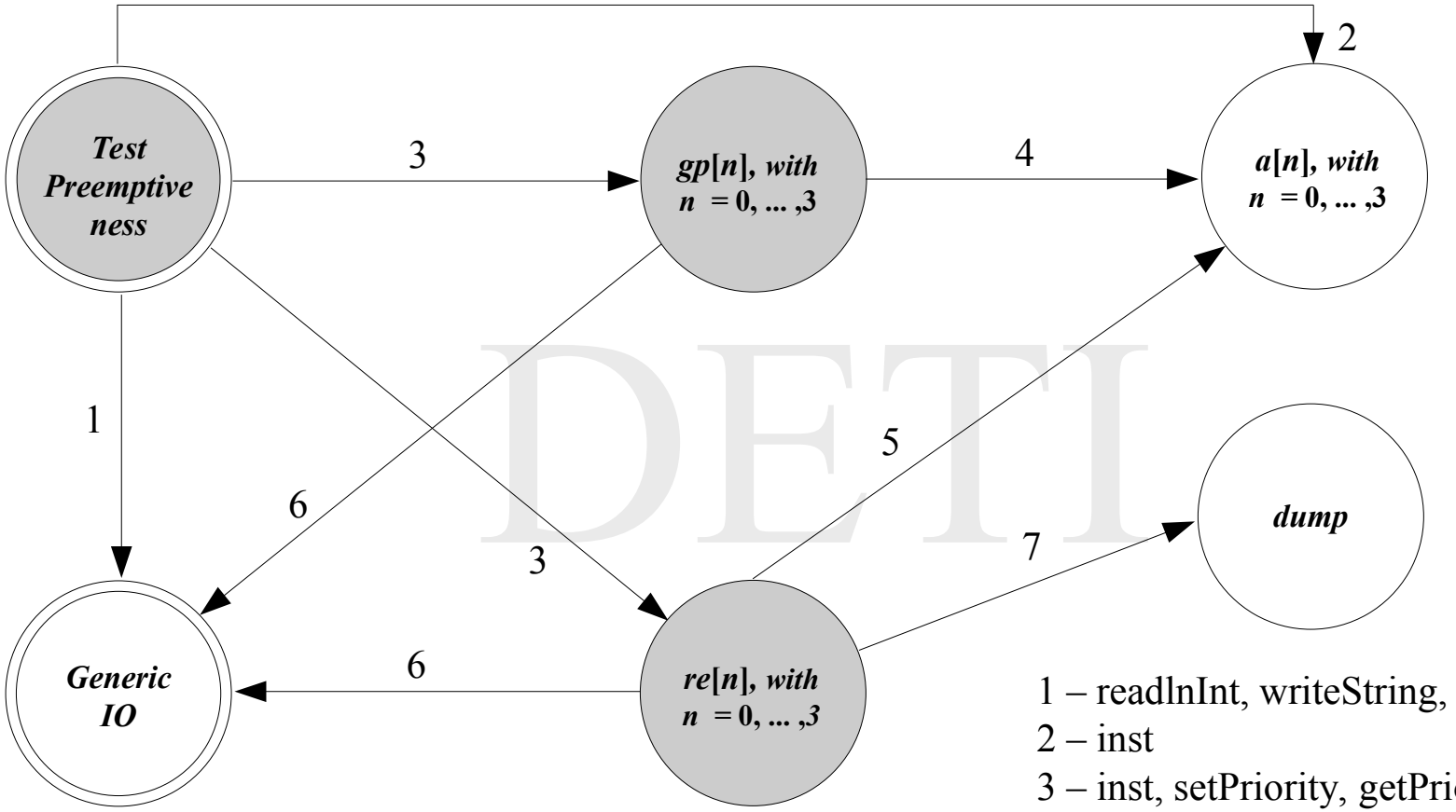
In Linux, for instance, Java threads are *kernel* level threads, taking advantage of *multicore* processors to enhance parallel execution, and the prevailing *scheduling* policy is the local one.

testPreemptiveness - 1



- there are four sets, consisting each of a computation intensive thread, an I/O intensive thread and a shared access variable
- the computation intensive thread successively increments by one the shared access variable 10 million times
- the I/O intensive thread successively reads and prints the shared access variable until its value reaches 10 million
- thread priorities can be changed and *yield* may be introduced after each increment operation of the computation intensive thread is performed
- the number of reads and prints carried out by the I/O intensive thread is used as a figure of merit to estimate the relative execution speed of both threads

testPreemptiveness - 2



1 – readlnInt, writeString, writelnString
2 – inst
3 – inst, setPriority, getPriority, start
4 – write / read variable
5 – read variable
6 – writelnString
7 – inst, write, flush

testPreemptiveness - 3

Without thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 1
Priority level of I/O intensive threads? 10
Priority of computation intensive threads = 1
Priority of I/O intensive threads = 10
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of A = 59702
A = 10000000
N. of iterations in printing values of D = 123791
D = 10000000
N. of iterations in printing values of C = 111804
C = 10000000
N. of iterations in printing values of B = 102411
B = 10000000
```

testPreemptiveness - 4

Without thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 10
Priority level of I/O intensive threads? 1
Priority of computation intensive threads = 10
Priority of I/O intensive threads = 1
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of C = 62180
C = 10000000
B = 10000000
N. of iterations in printing values of B = 103934
N. of iterations in printing values of D = 91698
D = 10000000
A = 10000000
N. of iterations in printing values of A = 119485
```

testPreemptiveness - 5

With thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 1
Priority level of I/O intensive threads? 10
Priority of computation intensive threads = 1
Priority of I/O intensive threads = 10
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of C = 4834998
C = 10000000
N. of iterations in printing values of D = 5085208
D = 10000000
N. of iterations in printing values of A = 4931095
A = 10000000
N. of iterations in printing values of B = 5193997
B = 10000000
```

testPreemptiveness - 6

With thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 10
Priority level of I/O intensive threads? 1
Priority of computation intensive threads = 10
Priority of I/O intensive threads = 1
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of D = 5053297
D = 10000000
A = 10000000
N. of iterations in printing values of A = 5151587
N. of iterations in printing values of B = 5461388
B = 10000000
N. of iterations in printing values of C = 5392911
C = 10000000
```

Threads in Java - 9

One should notice that the field `a` of the reference data type `Variable` includes the modifier `volatile`. Its precise meaning is to inform the Java compiler that the *threads*, during their execution, must permanently observe a *consistent* value in the variable `a`.

Consistency means in this sense that access to the variable `a` should always take place in the exact manner prescribed by the code of each *thread*.

This information is crucial here because Java memory model allows the compiler, when generating the *bytecode* of a given reference data type, as well as Java virtual machine, when interpreting this *bytecode*, to perform code optimization which, being totally consistent in a *singlethreaded* environment, may produce a paradoxical execution in a *multithreaded* environment.

Threads in Java - 10

A Java *multithreaded* application ends in principle when all its constituent *threads* terminate. In complex applications, where the number of support *threads* is very large, dealing with exceptional situations that may require aborting the operations, can become rather strenuous and demand the introduction of specific code whose practical utility is questionable.

To simplify the problem, Java presents two alternatives

- *calling the method* `void System.exit (int status)` – which forcibly terminates the Java virtual machine, returning the communicated *status* of operation
- turning the instantiated *threads*, directly or indirectly, created from the *thread* `main` into *daemons* – the Java virtual machine terminates as soon as all the remaining *threads* have this property.

Threads in Java - 11

Nevertheless, the usual termination of a *multithreaded* application is done by making the first or principal *thread* waiting for the termination of all the *threads* that may have been created from it.

In Java, one has a similar situation. The reference data type `Thread` has a method called `join` which, as it is tradititional in concurrent programming, and in a object oriented perspective, blocks the calling *thread* until the referenced *thread* ends.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 6: *Operating systems support*
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 3: *Processes*
- *On-line* support documentation for Java program developing environment by Oracle (Java Platform Standard Edition 8)

Sistemas Distribuídos

Concurrency 2

António Rui Borges

Summary

- *General principles of concurrency*
  - *Critical regions*
  - *Racing conditions*
  - *Deadlock and indefinite postponement*
- *Synchronization devices*
  - *Monitors*
  - *Semaphores*
- *Java concurrency library*
- *Suggested readings*

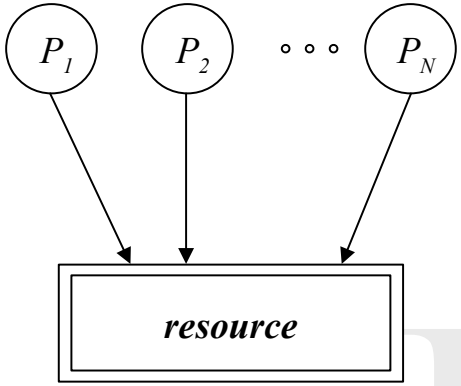
General principles of concurrency - 1

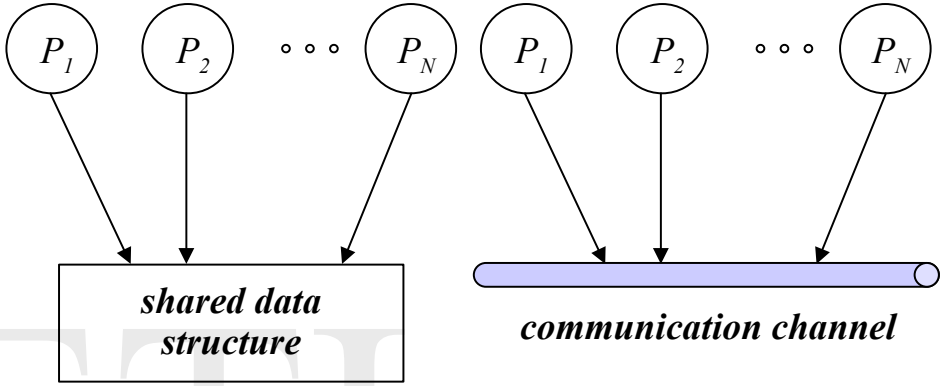
In a multiprogrammed environment, coexisting processes may present different behaviors interaction wise.

They may act as

- *independent processes* – when they are created, live and die without explicitly interacting among themselves; the underlying interaction is implicit and has its roots in the *competition* for the computational system resources; they are typically processes created by different users, or by the same user for different purposes, in an interactive environment, or processes which result from *job* processing in a *batch* environment
- *cooperating processes* – when they share information or communicate in an explicit manner; *sharing* presupposes a common addressing space, while *communication* can be carried out either by sharing the addressing space, or through a communication channel that connects the intervening processes.

General principles of concurrency - 2



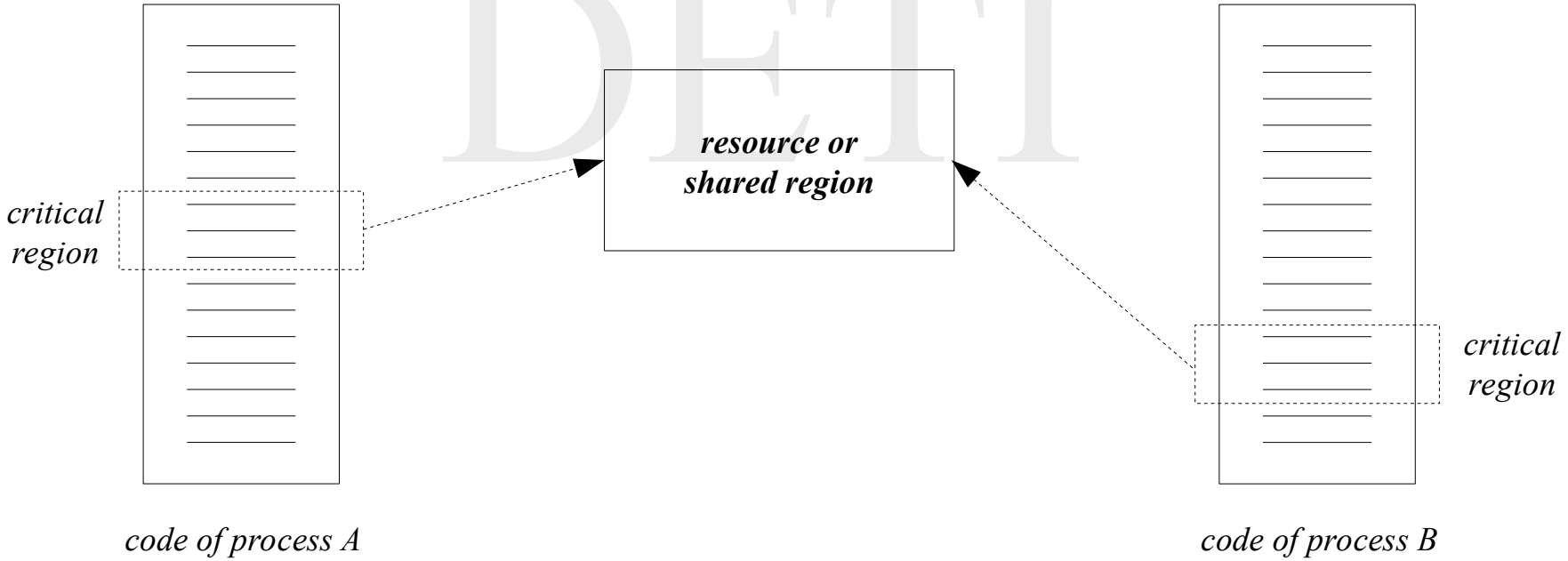


- *independent processes* that compete for access to a common resource of the computational system
- it is the *OS* responsibility to ensure that the resource assignment is carried out in a controlled fashion so that there is no loss of information
- this requires in general that only a single process may have access to the resource at a time (*mutual exclusion*)

- *cooperating processes* which share information or communicate among themselves
- it is the responsibility of the involved processes to ensure that access to the shared region is carried out in a controlled fashion so that there is no loss of information
- this requires in general that only a single process may have access at a time to the resource (*mutual exclusion*)
- the communication channel is typically a resource of the computational system; hence, access to it should be seen as *competition* for access to a common resource

General principles of concurrency - 3

Making the language precise, whenever one talks about *access by a process to a resource, or a shared region*, one is in reality talking about the processor executing the corresponding access code. This code, because it must be executed in a way that prevents the occurrence of *racing conditions*, which inevitably lead to loss of information, is usually called *critical region*.



General principles of concurrency - 4

Imposing mutual exclusion on access to a resource, or to a shared region, can have, by its restrictive character, two undesirable consequences

- *deadlock / livelock* – it happens when two or more processes are waiting forever (blocked / in *busy waiting*) for the access to the respective critical regions, being held back by events which, one may prove, will never occur; as a result the operations can not proceed
- *indefinite postponement* – it happens when one or more processes compete for the access to a critical region and, due to a conjunction of circumstances where new processes come up continuously and compete with the former for this goal, access is successively denied; one is here, therefore, facing a real obstacle to their continuation.

One aims, when designing a multithreaded application, to prevent these pathological consequences to occur and to produce code which has a *liveness* property.

Problem of access to a critical region with mutual exclusion

Desirable properties that a general solution to the problem must assume

- *effective assurance of mutual exclusion imposition* – access to the critical region associated to a given resource, or shared region, can only be allowed to a single process at a time, among all that are competing to the access concurrently
- *independence on the relative speed of execution of the intervening processes, or of their number* – nothing should be presumed about these factors
- *a process outside the critical region can not prevent another to enter*
- *the possibility of access to the critical region of any process that wishes to can not be postponed indefinitely*
- *the time a process is inside a critical region is necessarily finite*.

Resources

Generally speaking, a *resource* is something a process needs to access. Resources may either be *physical components of the computational system* (processors, regions of the main or mass memory, specific input / output devices, etc), or *common data structures* defined at the operating system level (process control table, communication channels, etc) or among processes of an application.

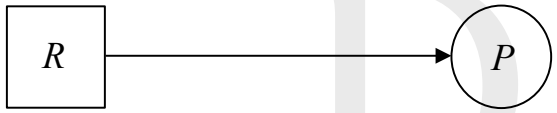
An essential property of resources is the kind of appropriation processes make of them. In this sense, resources are divided in

- *preemptable resources* – when they can be taken away from the processes that hold them, without any malfunction resulting from the fact; the processor and regions of the main memory where a process addressing space is stored, are examples of this class in multiprogrammed environments
- *non-preemptable resources* – when it is not possible; the printer or a shared data structure, requiring mutual exclusion for its manipulation, are examples of this class.

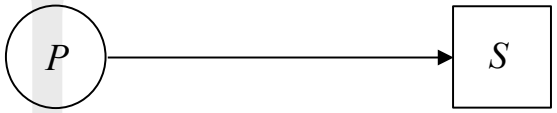
Schematic deadlock characterization

In a *deadlock* situation, only *non-preemptable* resources are relevant. The remaining can always be taken away, if necessary, from the processes that hold them and assigned to others to ensure that the latter may progress.

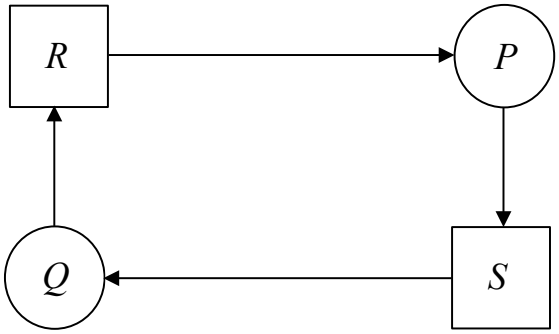
Thus, using this kind of framework, one may develop a schematic notation which represents a *deadlock* situation graphically.



process P holds the resource R



process P requires the resource S



typical deadlock situation
(*the most simple one*)

Necessary conditions to the occurrence of deadlock

It can be shown that, whenever *deadlock* occurs, there are four conditions which are necessarily present. They are the

- *condition of mutual exclusion* – each existing resource is either free, or was assigned to a single process (its holding can not be shared)
- *condition of waiting with retention* – each process, upon requesting a new resource, holds all other previously requested and assigned resources
- *condition of non-liberation* – nothing, but the process itself, can decide when a previously assigned resource is freed
- *condition of circular waiting* (or *vicious circle*) – a circular chain of processes and resources, where each process requests a resource which is being held by the next process in the chain, is formed.

Deadlock prevention - 1

The necessary conditions to the occurrence of *deadlock* lead to the statement

deadlock occurrence ⇒ *mutual exclusion on access to a resource* **and**
waiting with retention **and**
no liberation of resources **and**
circular waiting

which is equivalent to

no mutual exclusion on access to a resource **or**
no waiting with retention **or**
liberation of resources **or**
no circular waiting ⇒ *no deadlock occurrence* .

Thus, in order to make deadlock *impossible to happen*, one has only to deny one of the necessary conditions to the occurrence of deadlock. Policies which follow this strategy are called *deadlock prevention policies*.

Deadlock prevention - 2

The first, *mutual exclusion on access to a resource*, is too restrictive because it can only be denied for non-preemptable resources. Otherwise, *racing conditions* are introduced which lead, or may lead, to information inconsistency.

Reading access by multiple processes to a file is a typical example of denying this condition. One should point out that, in this case, it is also common to allow at the same time a single writing access. When this happens, however, *racing conditions*, with the consequent loss of information, can not be completely discarded. *Why*?

Therefore, only the last three conditions are usually object of denial.

Denying the condition of waiting with retention

It means that a *process must request at once all the resources it needs for continuation*. If it can get hold of them, the completion of the associated activity is ensured. Otherwise, it must wait.

One should notice that *indefinite postponement* is not precluded. The procedure must also ensure that sooner or later the necessary resources will always be assigned to any process which will be requesting them. The introduction of *aging* policies to increase the priority of a process is a very popular method used in this situation.

Imposing the condition of liberation of resources

It means that a *process, when it can not get hold of all the resources it requires for continuation, must release all the resources in its possession and start later on the whole request procedure from the very beginning*. Alternatively, it also means that a *process can only hold a resource at a time* (this, however, is a particular solution and is not applicable in most cases).

Care should be taken for the process not to enter a *busy waiting* procedure of request / acquire resources. In principle, the process must block after freeing the resources it holds and be waken up only when the resources it was requesting are released.

Nevertheless, *indefinite postponement* is not precluded. The procedure must also ensure that sooner or later the necessary resources will always be assigned to any process which will be requesting them. The introduction of *aging* policies to increase the priority of a process is a very popular method used in this situation.

Denying the condition of circular waiting

It means *to establish a linear ordering of the resources* and *to make the process, when it tries to get hold of the resources it needs for continuation, to request them in increasing order of the number associated to each of them*.

In this way, the possibility of formation of a circular chain of processes holding resources and requesting others is prevented.

One should notice that *indefinite postponement* is not precluded. The procedure must also ensure that sooner or later the necessary resources will always be assigned to any process which will be requesting them. The introduction of *aging* policies to increase the priority of a process is a very popular method used in this situation.

Monitors - 1

A *monitor* is a synchronization device, proposed independently by Hoare and Brinch Hansen, which can be thought of as a special module defined within the [concurrent] programming language and consisting of an internal data structure, initialization code and a set of access primitives.

```
monitor example
  (* internal data structure
     only accessible from the outside through the access primitives *)
  var
    val: DATA;                                    (* shared region *)
    c: condition;          (* condition variable for synchronization *)
  (* access primitives *)
  procedure pa1 (...);
  end (* pa1 *)
  function pa2 (...): real;
  end (* pa2 *)
  (* initialization *)
  begin
    ...
  end
end monitor;
```

Monitors - 2

An application written in a concurrent language, implementing the *shared variables paradigm*, is seen as a set of *threads* that compete for access to shared data structures. When the data structures are implemented as *monitors*, the programming language ensures that the execution of a *monitor* primitive is carried out following a mutual exclusion discipline. Thus, the compiler, on processing a *monitor*, generates the required code to impose this condition in a manner totally transparent to the applications programmer.

A *thread* enters a *monitor* by calling one of its primitives, which constitutes the only way to access the internal data structure. As primitive execution entails mutual exclusion, when another *thread* is presently inside the monitor, the *thread* is blocked at the entrance, waiting for its turn.

Monitors - 3

Synchronization among *threads* using monitors is managed by *condition variables*. *Condition variables* are special devices, defined inside a monitor, where a thread may be blocked, while waiting for an event that allows its continuation to occur. There are two atomic operations which can be executed on a *condition variable*

wait – the calling *thread* is blocked at the *condition variable* passed as argument and is placed *outside the monitor* to allow another *thread*, wanting to enter, to proceed

signal – if there are blocked *threads* in the *condition variable* passed as argument, one of then is waken up; otherwise, nothing happens.

Monitors - 4

To prevent the coexistence of multiple *threads* inside a *monitor*, a rule is needed which states how the contention arisen by a *signal* execution is resolved

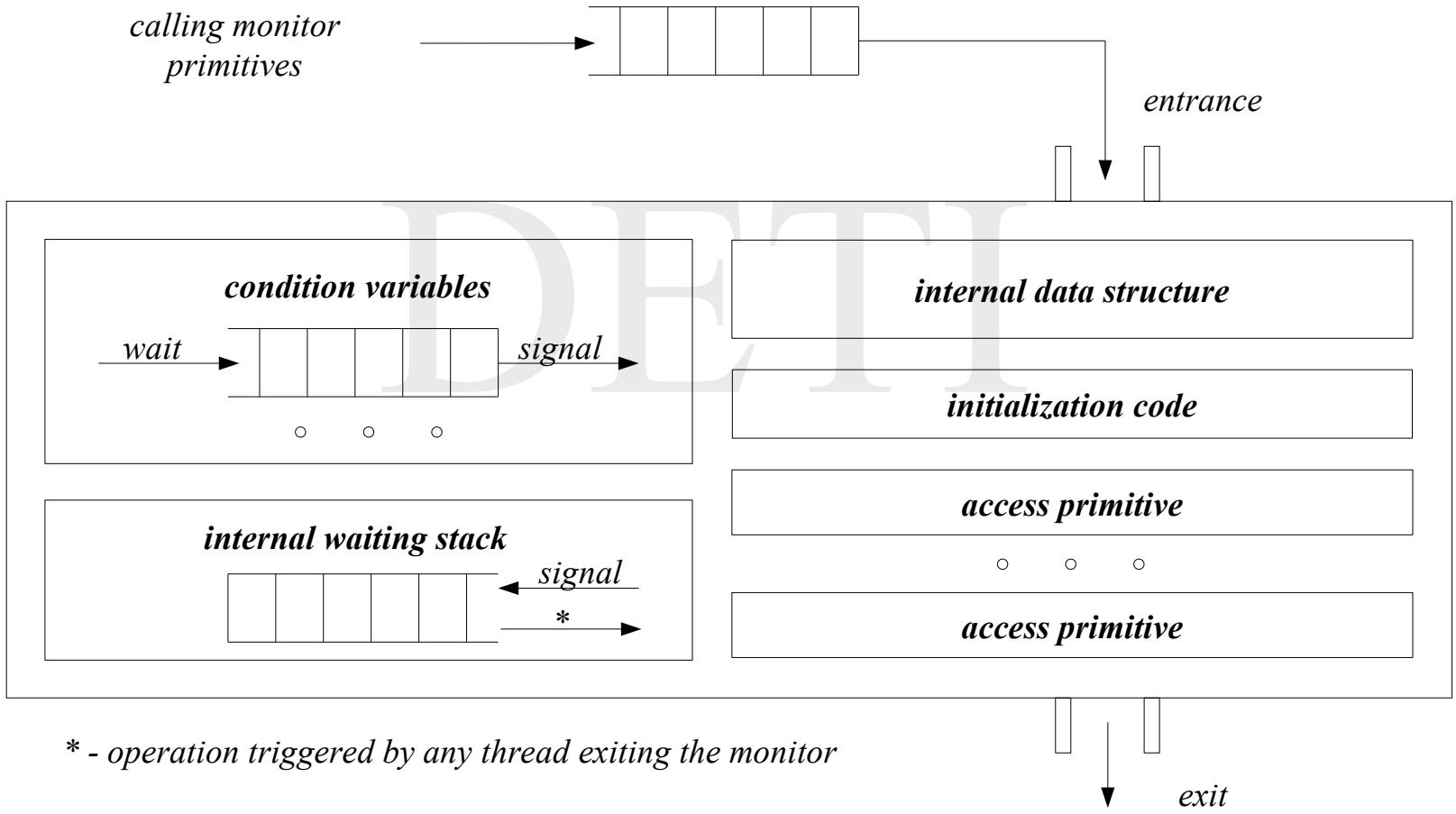
Hoare monitor – the *thread* which calls *signal* is placed *outside the monitor* so that the waken up *thread* may proceed; it is a very general solution, but its implementation requires a *stack*, where the *signal* calling *threads* are stored

Brinch Hansen monitor – the *thread* which calls *signal* must immediately exit the *monitor* (*signal* should be the very last executed instruction in any access primitive, except for a possible *return*); it is quite simple to implement, but it may become rather restrictive because it reduces the number of *signal* calls to one

Lampson / Redell monitor – the *thread* which calls *signal* proceeds, the waken up *thread* is kept *outside the monitor* and must compete for access to it again; it is still simple to implement, but it may give rise to *indefinite postponement* of some of the involved *threads*.

Monitors - 5

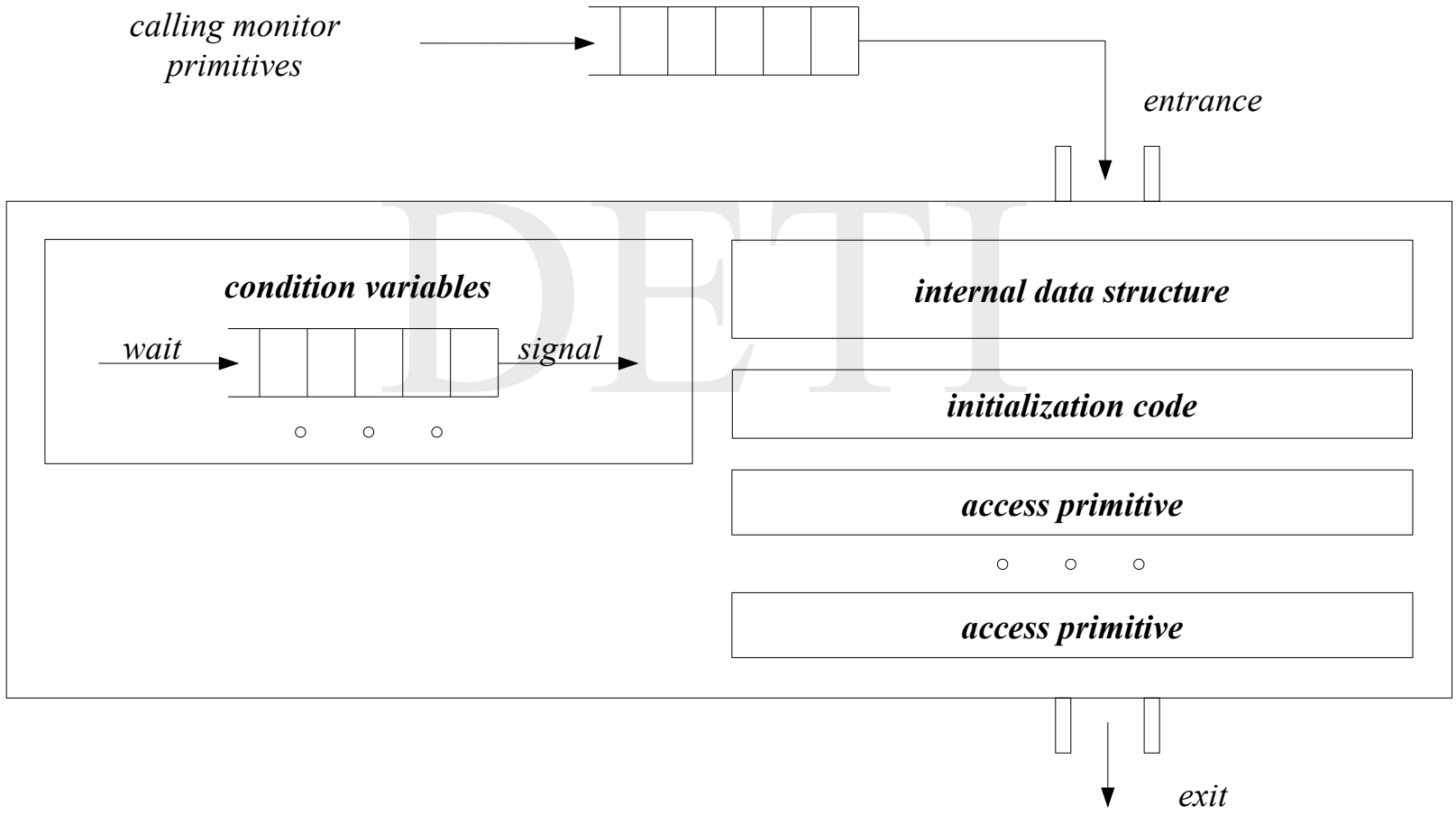
Hoare monitor



* - operation triggered by any thread exiting the monitor

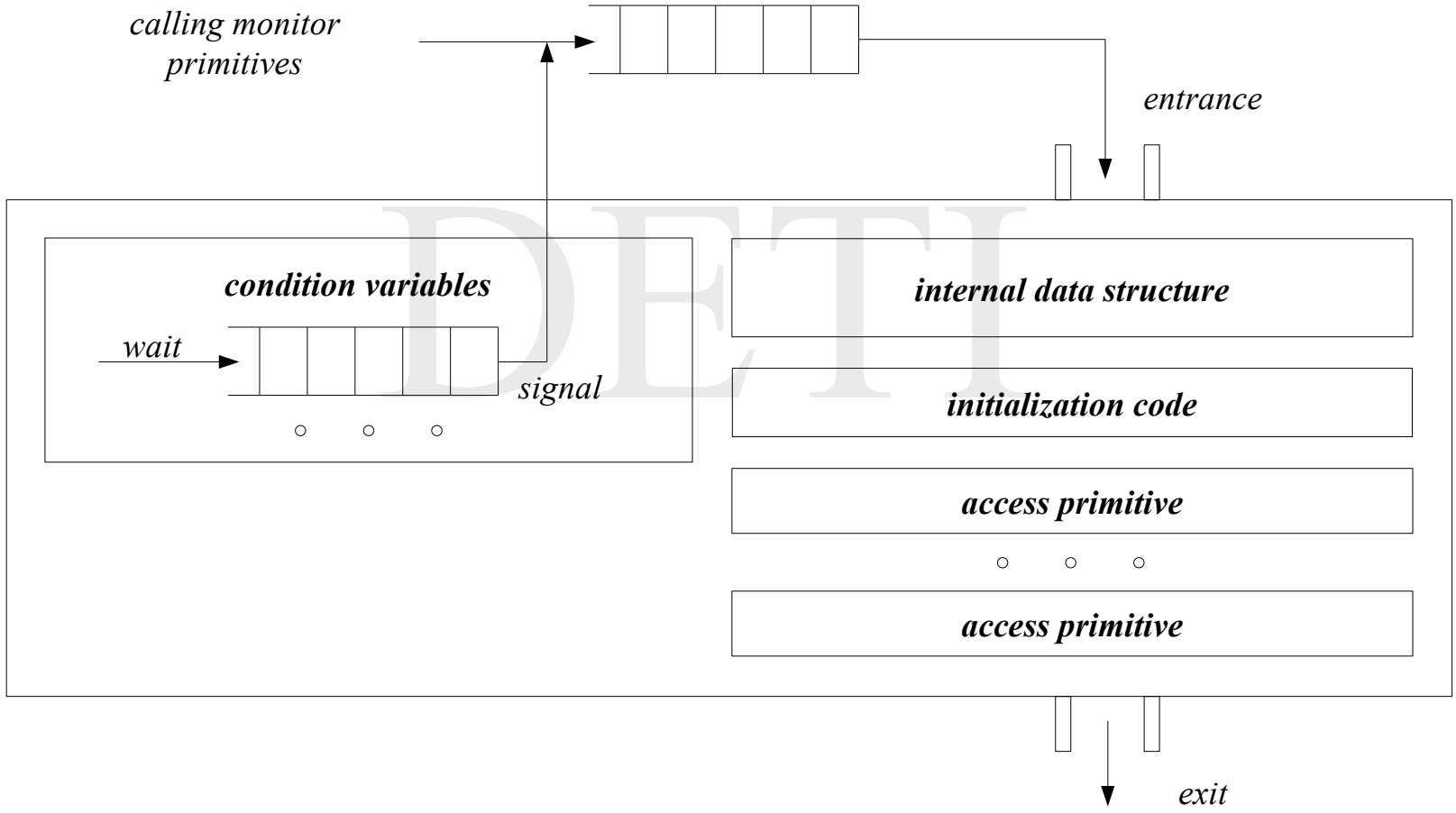
Monitors - 6

Brinch Hansen monitor



Monitors - 7

Lampson / Redell monitor



Monitors in Java - 1

Java supports Lampson / Redell monitors as its native synchronizing device

- each reference data type may be turned into a monitor, thus enabling to ensure mutual exclusion and *thread* synchronization when `static` methods are called upon it
- each instantiated object may be turned into a monitor, thus enabling to ensure mutual exclusion and *thread* synchronization when instantiation methods are called upon it.

In fact, and taking into account that Java is an object oriented language, each *thread*, being a Java object, can also be turned into a monitor. This property, if taken to the last consequences, enables a *thread* to block in its own monitor!

This, however, should never be done, since it introduces mechanisms of self-reference which are usually very difficult to understand due to the side effects they generate.

Monitors in Java - 2

The implementation in Java of a Lampson / Redell monitor has, however, some peculiarities

- the number of condition variables is limited to one, referenced in a implicit manner through the object which represents in *runtime* the reference data type, or any of its instantiations
- the traditional *signal* operation is named *notify* and there is a variant of this operation, *notifyAll*, the most commonly used, which enables the waking up of *all* the *threads* presently blocked in the condition variable
- furthermore, there is a method in data type `Thread`, named *interrupt*, which when called on a specific *thread*, aims to wake it up by throwing an *exception* if it is blocked on a condition variable.

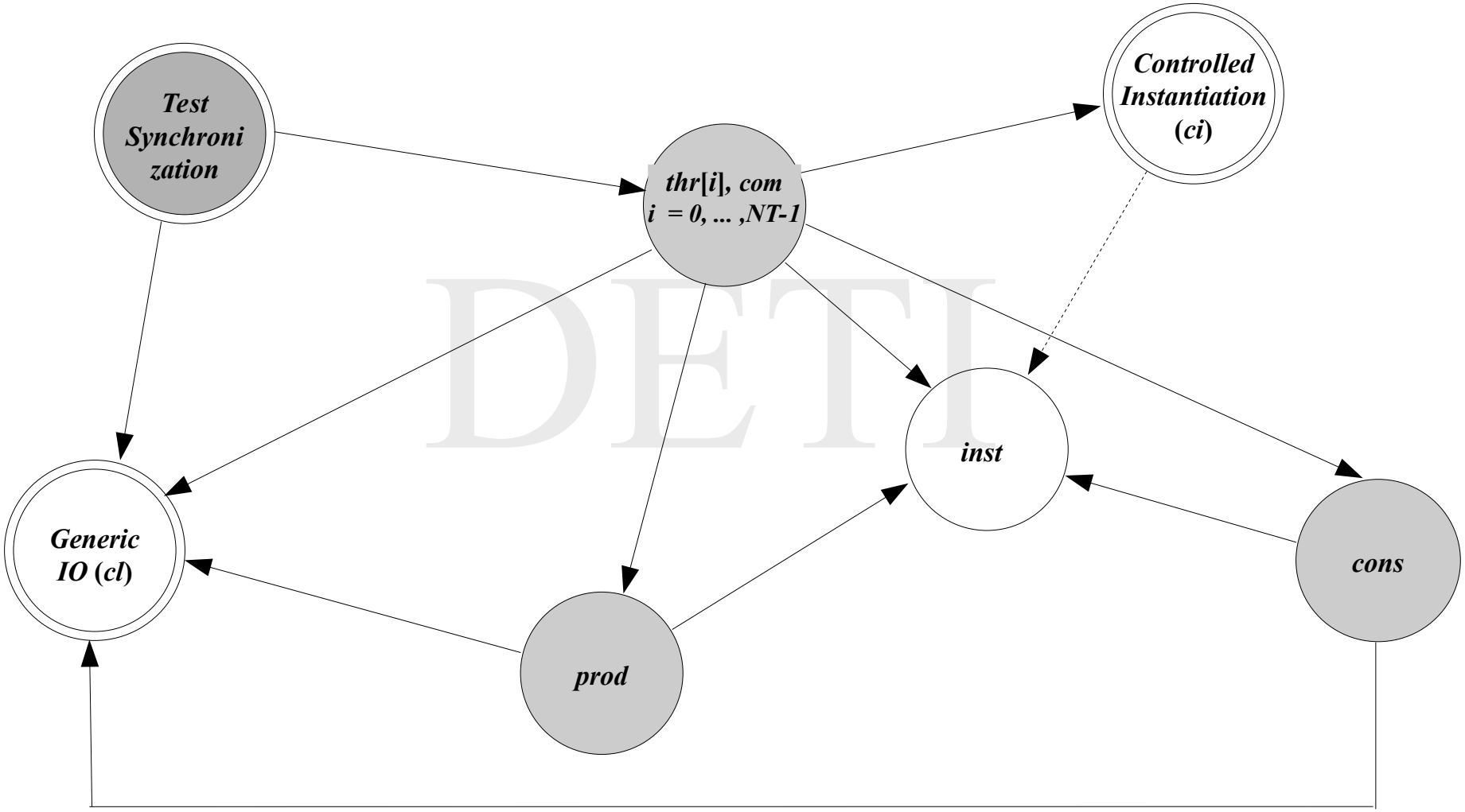
The need for the operation *notifyAll* becomes evident if one thinks that, by having a single condition variable per monitor, the only way to wake up a *thread* which is presently blocked waiting for a particular condition to be fulfilled, is by waking up *all* the blocked *threads* and then determine in a differential manner which is the one that can progress.

Controlled instantiation - 1

The example presented next illustrates the use of monitors in different situations

- *thread* `main` creates four *threads* of reference data type `TestThread` which try to instantiate objects of data type `ControlledInstantiation`, each a storage location for value transfer between *threads* of data types `Proc` and `Cons`
- the reference data type `ControlledInstantiation`, however, only allows the simultaneous instantiation of a maximum of two objects
- there are, thus, three types of monitors at play
  - the monitor associated with the object that represents in *runtime* the reference data type `ControlledInstantiation`
  - the monitor associated with each instantiation of data type `ControlledInstantiation`
  - the monitor associated with the object that represents in *runtime* the reference data type `genclass.GenericIO`
- the first controls the instantiation of `ControlledInstantiation` objects, the second the value transfer between threads of data type `Prod` and `Cons` and the third the printing of data in the *standard* output device.

Controlled instantiation - 2



Controlled instantiation - 3

Operation `wait` on a monitor defined at the reference data type level

```
public static synchronized ControlledInstantiation generateInst ()
{
  while (n >= NMAX)
  { try
    { ( Class.forName ("ControlledInstantiation")).wait ();
    }
    catch (ClassNotFoundException e)
    { GenericIO.writelnString ("Data type ControlledInstantiation was not " +
                               "found(generation)!");
      e.printStackTrace ();
      System.exit (1);
    }
    catch (InterruptedException e)
    { GenericIO.writelnString ("Static method generateInst was interrupted!");
    }
  }
  n += 1;
  nInst += 1;
  return new ControlledInstantiation ();
}
```

Controlled instantiation - 4

Operation `notify` on a monitor defined at the reference data type level

```
public static synchronized void releaseInst ()
{
  n -= 1;
  try
  { (Class.forName ("ControlledInstantiation")).notify ();
  }
  catch (ClassNotFoundException e)
  { GenericIO.writelnString ("Data type ControlledInstantiation was not found" +
                             "(release)!");
    e.printStackTrace ();
    System.exit (1);
  }
}
```

Controlled instantiation - 5

Operation `wait` and `notify` on a monitor defined at the object level

```
public synchronized void putVal (int val)
{
  store = val;
  while (store != -1)
  { notify ();
    try
    { wait ();
    }
    catch (InterruptedException e)
    { GenericIO.writelnString ("Method putVal was interrupted!");
    }
  }
}
```

Controlled instantiation - 6

Access with mutual exclusion to a non-thread safe library

```
Class<?> cl = null;           // representation of the library data type in JVM

try
{ cl = Class.forName ("genclass.GenericIO");
}
catch (ClassNotFoundException e)
{ System.out.println ("Data type genclass.GenericIO was not found!");
  e.printStackTrace ();
  System.exit (1);
}

synchronized (cl)
{ GenericIO.writelnString ("I have already created the threads!");
}
```

Controlled instantiation - 7

```
I have already created the threads!
I, Thread_base_2, got the instantiation number 2 of data type
ControlledInstantiation!
I, Thread_base_1, got the instantiation number 1 of data type
ControlledInstantiation!
I, Thread_base_1, am going to create the threads that will exchange the value!
I, Thread_base_2, am going to create the threads that will exchange the value!
I, Thread_base_1_writer, am going to write the value 1 in instantiation number 1
of data type ControlledInstantiation!
I, Thread_base_2_writer, am going to write the value 2 in instantiation number 2
of data type ControlledInstantiation!
I, Thread_base_1_reader, read the value 1 in instantiation number 1 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_1_writer, has terminated.
My thread which reads the value, Thread_base_1_reader, has terminated.
I, Thread_base_1, am going to release the instantiation number 1 of data type
ControlledInstantiation!
I, Thread_base_0, got the instantiation number 3 of data type
ControlledInstantiation!
I, Thread_base_0, am going to create the threads that will exchange the value!
I, Thread_base_2_reader, read the value 2 in instantiation number 2 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_2_writer, has terminated.
My thread which reads the value, Thread_base_2_reader, has terminated.
I, Thread_base_2, am going to release the instantiation number 2 of data type
ControlledInstantiation!
```

Controlled instantiation - 8

```
I, Thread_base_3, got the instantiation number 4 of data type
ControlledInstantiation!
I, Thread_base_3, am going to create the threads that will exchange the value!
I, Thread_base_3_writer, am going to write the value 3 in instantiation number 4
of data type ControlledInstantiation!
I, Thread_base_0_writer, am going to write the value 0 in instantiation number 3
of data type ControlledInstantiation!
I, Thread_base_0_reader, read the value 0 in instantiation number 3 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_0_writer, has terminated.
My thread which reads the value, Thread_base_0_reader, has terminated.
I, Thread_base_0, am going to release the instantiation number 3 of data type
ControlledInstantiation!
The thread Thread_base_0 has terminated.
The thread Thread_base_1 has terminated.
The thread Thread_base_2 has terminated.
I, Thread_base_3_reader, read the value 3 in instantiation number 4 of data type
ControlledInstantiation!
My thread which writes the value, Thread_base_3_writer, has terminated.
My thread which reads the value, Thread_base_3_reader, has terminated.
I, Thread_base_3, am going to release the instantiation number 4 of data type
ControlledInstantiation!
The thread Thread_base_3 has terminated.
```

Semaphores - 1

A different approach to solve the problem of access with mutual exclusion to a critical region is based on the observation.

```
/* control data structure */
#define  R    ...          /* number of processes wishing to access a critical region,
                              pid = 0, 1, ..., R-1 */

shared unsigned int access = 1;

/* primitive for entering the critical region */
void enter_RC (unsigned int own_pid)
{
   if (access == 0) sleep (own_pid);
      else access -= 1;
}

/* primitive for exiting the critical region */
void exit_RC ()
{
   if (there are blocked processes) wake_up_one ();
      else access += 1;
}
```

atomic operation
(its execution can not be interrupted)

atomic operation
(its execution can not be interrupted)

Semaphores - 2

A *semaphore* is a synchronization device, originally invented by Dijkstra, which can be thought of as a variable of the type

```
typedef struct
{ unsigned int val;  /* non-negative counting value */
  NODE *queue;       /* blocked processes waiting queue */
} SEMAPHORE;
```

where two atomic operations may be performed

down – if `val` is different from zero (the semaphore has a shade of green), its value is decreased by one unit; otherwise, the process which called the operation, is blocked and its id is placed in `queue`

up – if there are blocked processes in `queue`, one of them is waken up (according to any previously prescribed discipline); otherwise, the value of `val` is increased by one unit.

Semaphores - 3

```
/* kernel defined semaphore array */
#define  R   ...            /* semaphore id - id = 0, 1, ..., R-1 */

static SEMAPHORE sem[R];

/* down operation */
void down (unsigned int id)
{
  disable interrupts / lock access flag;
  if (sem[id].val == 0) sleep_on_sem (getpid(), id);
     else sem[id].val -= 1;
  enable interrupts / unlock access flag;
}

/* up operation */
void up (unsigned int id)
{
  disable interrupts / lock access flag;
  if (there are blocked processes in sem[id]) wake_up_one_on_sem (id);
     else sem[id].val += 1;
  enable interrupts / unlock access flag;
}
```

Java Dijkstra semaphore

```
public class Semaphore
{
   private int val = 0,                        // green / red indicator
               numbBlockThreads = 0;           // number of the blocked threads
                                               //  in the monitor

   public synchronized void down ()
   {
     if (val == 0)
        { numbBlockThreads += 1;
          try
          { wait ();
          }
          catch (InterruptedException e) {}
        }
        else val -= 1;
   }

   public synchronized void up ()
   {
     if (numbBlockThreads != 0)
        { numbBlockThreads -= 1;
          notify ();
        }
        else val += 1;
   }
}
```

Java concurrency library - 1

Java `concurrency` library supplies the following relevant synchronization devices

- *barriers* – devices which lead to the blocking of set of *threads* until a condition for their continuation is fulfilled; when the barrier is raised, all the blocked processes are waken up
- *semaphores* – devices that provide a more general implementation of the semaphore concept than the one prescribed by Dijkstra, namely allowing the calling of *down* and *up* operations where the internal field `val` is decreased or increased by more than an unit at a time and having a non-blocking variant of *down* operation
- *exchanger* – devices that allow an exchange of values between *threads* pairs through a *rendez-vous* type synchronization.

Java concurrency library - 2

- *locks* – Lampson-Redell monitors of general character (in contrast to Java *built-in* monitor, it is possible to define here multiple condition variables and to get a similar functionality to that provided by the *pthread* library
- *atomic variable manipulation* – *read-modify-write* mechanisms which allow writing and reading the contents of several types of variables without the risk of *racing conditions*.

The library also provides basic thread blocking primitives for creating locks and other synchronization devices.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 6: *Operating systems support*
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 3: *Processes*
- *On-line* support documentation for Java program developing environment by Oracle (Java Platform Standard Edition 8)

Sistemas Distribuídos

Introductory Concepts

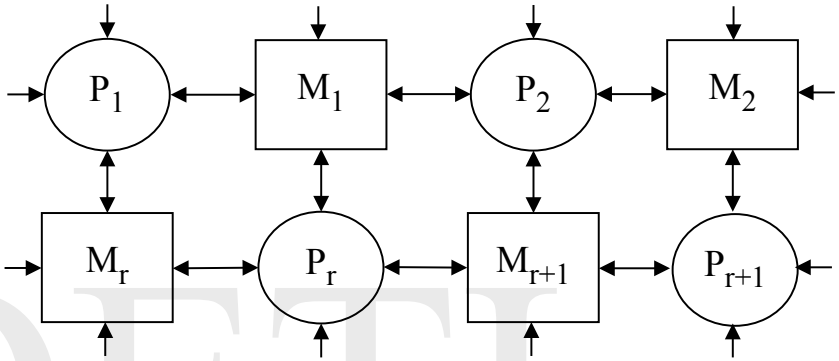
António Rui Borges

Summary

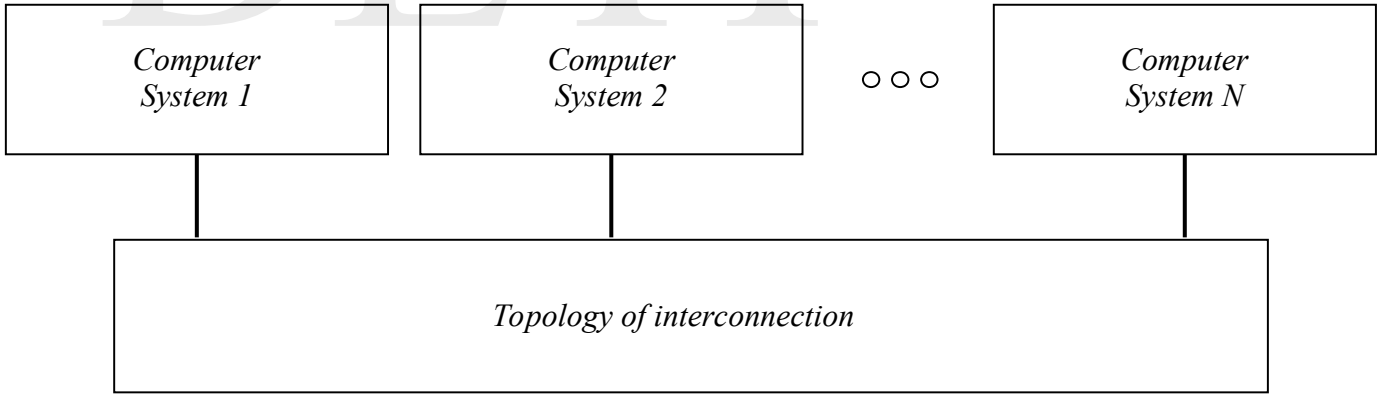
- *Parallel computer systems*
- *Characterization of distributed systems*
- *Issues to be taken into account in the project of distributed systems*
  - *Heterogeneity*
  - *Transparency*
  - *Openness*
  - *Security of the information flows*
  - *Scalability*
  - *Fault handling*
  - *Concurrency*
- *Ubiquitous computing* (*Internet Of Things*)
- *Cloud computing*
- *Suggested reading*

Parallel computer systems

Mesh multiprocessor



Computer network



Distributed systems - 1

The conventional definition of ***distributed system*** states that one is dealing with *an* [*operating*] *system* whose *components, located in the different processing nodes of a parallel computer system, communicate and coordinate their actions through message passing* [Coulouris et als.].

The main motivation underlying the construction and the use of *distributed systems* is resource sharing. *Resource* is to be understood here as an abstract entity which embodies something *material*, like processors, network infrastructures, storage devices and more or less specialized peripherals, or *immaterial*, like information taken in the most wide sense, or the functionalities that operate upon it.

This sharing is translated in two distinct ways

- *application parallelization* – by taking advantage of the multiple processors and other *hardware* components of the parallel computer system, one tries to ensure a faster and more efficient program execution
- *service availability* – by managing a set of somewhat related resources, one tries to organize them in such a way that an uniform communication interface based on a well-defined API may be provided.

Distributed systems - 2

An alternative definition of ***distributed system***, although essentially equivalent, is to say that one is dealing with *a collection of independent computing systems perceived by the users as a coherent system* [Tanenbaum et als.].

Several consequences stem from it

- *parallelism* – the existence of independent computer systems gives rise to simultaneous and autonomous threads of execution
- *global internal state* – the need to provide a coherent working image requires some form of activity coordination among the different processing nodes
- *communication through message passing* – the fact that no assumptions are made about how the computer systems are interconnected, entails a minimalist communication mechanism based on message passing
- *scalability* – the faculty of expanding the system by integration of more processing nodes is an immediate outcome of the prescribed organization
- *failure handling* – any of the components of the distributed system may fail at any time, leaving the remaining components in operation.

Types of distributed systems - 1

The way *distributed systems* are organized is strictly dependent on the kind of problems they are aimed to solve.

Four types of distributed systems are among the most common in practice

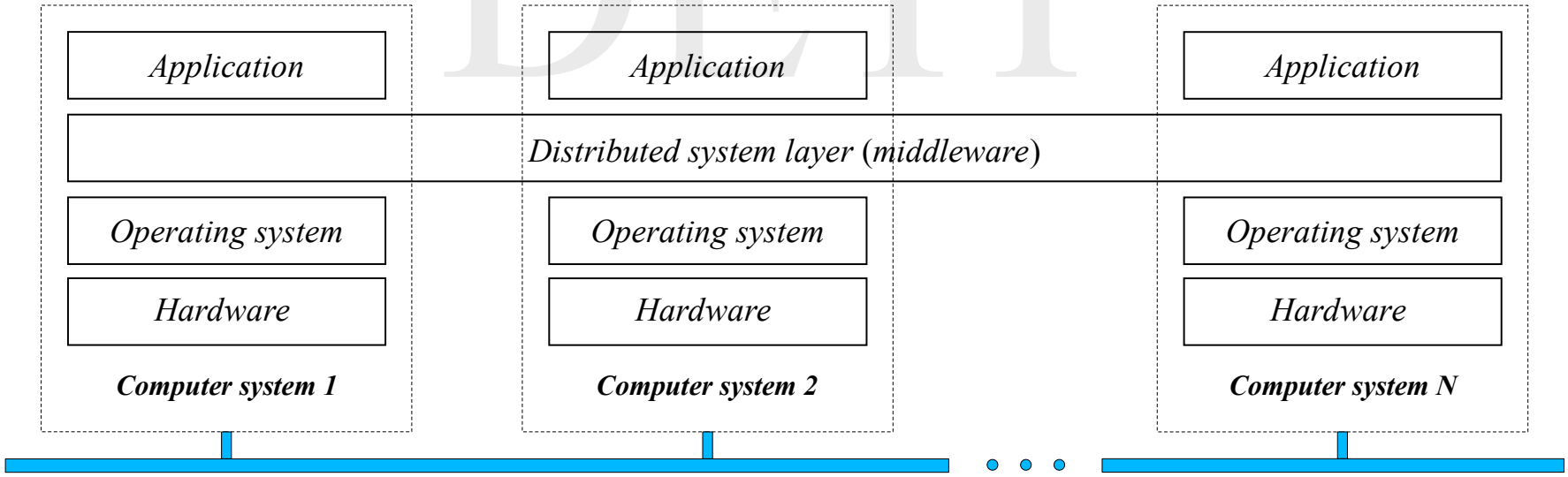
- *cluster systems* – they consist typically of a collection of computer systems interconnected by a high speed network; the aim is application parallelization; typically, the computer systems are identical, all run the same operating system and share the same network; there is a single master node that handles the allocation of processing nodes to a particular program, maintains a queue of submitted jobs and interfaces with the system users
- *grid systems* – no assumptions are made here concerning the hardware platforms which form the processing nodes, neither the operating systems they run, nor the interconnecting networks; the key issue is bringing together the resources from different organizations so that a community of people may cooperate; a virtual organization is thus formed, where only its members have access rights to the resources provided

Types of distributed systems - 2

- *transaction information systems* – the main processing nodes run applications whose primary goal is to manage information mostly organized in databases, called *servers*; this information is made available to remote-run programs, the *clients*, through a succession of operations of the type request / reply; integration at the lowest level allow the clients to wrap together a group of requests, possibly to different servers, into a single larger request and have it executed as a *transaction*, that is, all or none of the sub-requests are executed
- *enterprise integration systems* – as the applications become decoupled from the associated databases, it turns more and more apparent that means are needed for application integration independent of the databases; in particular, application components should be able to communicate directly with one another and not merely through the request / reply behavior supported by transaction information systems.

Distributed systems - 3

Assuming generic computer systems, as processing nodes, and diverse communication networks, as interconnection infrastructures, while simultaneously keeping an integrated view of its functionality, *distributed systems* can be understood as a software layer, located between the applications and the local operating systems, aiming to create a model of programmable virtual machine that hides all the underlying heterogeneity – the so called *middleware*.



Design of distributed systems / applications

The design of *distributed systems*, as well as the development applications run in them, requires that several issues whose importance is crucial to good performance to be considered.

The following are singled out

- the [possible] *heterogeneity* of the components that form the parallel computer system
- its *transparency*
- the degree of *openness*
- the *security* of the exchanged information flows and of the stored information
- the potential it presents for *scalability*
- the *failure handling*
- its potential for *concurrency*.

Heterogeneity - 1

The *heterogeneity* of a parallel computer system becomes apparent in multiple forms

- the interconnection of different types of computer networks
- the use of different hardware platforms as processing nodes
- different operating systems running simultaneously in different nodes
- different programming languages used in the coding of different software components of the same application
- the interconnection of functionalities provided by different suppliers.

In view of such diverse reality, it is mandatory to find solutions which provide a coherent approach.

Heterogeneity - 2

The solutions should be *localized*, introducing a greater or lesser degree of uniformity at a specific level of the communication chain

- specification and implementation of a network architecture
- conversion the data representation specificities presented by each processor into a common format
- harmonization of the *communication programming interface* offered by the different operating systems
- adjustment of the data structures implemented by different programming languages into a common format
- development of standards accepted and complied by different manufacturers.

Transparency - 1

The degree of *transparency* of a distributed system is a feature that expresses the greater or lesser success in masking the underlying complexity. Thus, making the operation of a distributed system *transparent* means describing its functionality in an integrated fashion, conceptually simple, instead of resulting from the interaction of a collection of independent components.

The *transparency* goal is, therefore, to hide the resources which are not directly relevant to the task being carried out, making them anonymous both to the user and/or the applications programmer.

The degree of *transparency* with which the different resources may be accessed and operated, portrays immediately the degree of abstraction and operativeness of the *middleware* layer.

Transparency - 2

Transparency modes

- *access transparency* – when the same operations are performed to access local and remote resources
- *position transparency* – when resource access is carried out without the knowledge of its precise physical or network location
- *network transparency* – when access and position transparency exist concomitantly
- *mobility transparency* – when the client access location to the resources may change within the system without affecting the operation being performed
- *migration transparency* – when the resources may be moved without affecting their access
- *relocation transparency* – when the resources may be moved without affecting their access, even when an access is taking place (strong version of *migration transparency*)

Transparency - 3

- *replication transparency* – when it is possible to instantiate multiple copies of the same resource without the fact of doing so becoming obvious
- *performance transparency* – when a dynamic system reconfiguration may occur to cope with load variations
- *scaling transparency* – when the system and applications may expand in scale without requiring any change in the system structure and on the application algorithms
- *concurrency transparency* – when access to shared resources is performed in parallel by multiple entities without being aware of one another (data should always remain consistent)
- *failure transparency* – when failures, occurring in the hardware and/or software system components, can be masked and, therefore, the tasks in execution be terminated.

Openness

The degree of *openness* of a computer system is a feature which determines its greater or lesser capability to be extended and re-implemented in different ways.

For *distributed systems*, *openness* is fundamentally related to the capability of incorporating new services and of making them available to a wide variety of applications.

It is, therefore, necessary

- to establish an uniform communication mechanism on access to shared resources
- to publish the main APIs
- to ensure a strict conformity, at the design and implementation levels, of each new component with the relevant standard.

Security of information - 1

The generalized service availability and the appetence demonstrated by parallel computer systems for the cooperative execution of applications raise the question of how safe are the exchanged information flows and of how safe is the data stored in the associated resources.

The *security of information* may be envisaged at three levels

- *confidentiality* – protection against its disclose to non-authorized entities
- *integrity* – protection against it modification or its corruption
- *availability* – protection against interference that disturb the access channels.

Security of information - 2

Some measures may be taken to reduce the risks

- *introduction of firewalls* – by creating a barrier around a local network which limits the input and output traffic to well-defined channels
- *message encryption* – by scrambling the contents of the information flows
- *use of electronic signatures* – for certification of the message author.

There are, however, situations of difficult solution

- *denial of service attacks* – when the service suppliers are flooded by a huge number of phony requests, which leads to service shutdown, preventing the access of the true users
- *security of mobile code* – executable files sent in attachment to messages, download of *plug-ins* and *applets*, for instance, may lead to undesirable system behavior, completely out of control of the local user.

Scalability

Distributed systems must allow *increases of scale*. That is, in view of a significant increase of the number and/or the size of the available resources and/or the number of the users which request them, must still keep an appropriate performance.

Some of the challenges one must face are

- *expansion of the physical resources* – increase of scale means in this sense that the amount of the required resources to service n users should not be larger than $O(n)$
- *performance losses* – increase of scale means in this sense that the performance loss resulting from the processing of a task should be about $O(log_2\ n)$ at the most, where n is some parameter measuring the task size
- *preventing future bottlenecks* – increase of scale means in this sense that an effective planning about the resources size and the access procedures must exist so that, taking into account the predictable future growth, prevents their exhaustion and the resulting performance losses.

Failure handling - 1

Distributed systems, being built out of multiple hardware and software components, are susceptible to partial failure of extreme varied sort. Dealing with it is, therefore, very difficult.

Some types of failure are easily detectable, but many others are masked by the complexity of the system. The great challenge is how to manage the system in an environment where some of the failures can not be detected, but whose existence one may merely suspect of.

Basic strategies for failure handling

- *masking the failures* – some of the detected failures can be hidden or, at least, their effect be made less severe
  - lost messages, if detected, can be transmitted again in many cases
  - resource replication can allow for the safeguarding of stored information when there is data corruption in some of the copies

Failure handling - 2

Basic strategies for failure handling (continuation)

- *recovering from failures* – in order to make it possible, it is necessary to design the software components in such a way that the internal states of the involved processes are periodically stored; when a failure occurs, processing is restarted from the last saved state

  in practice, this may also require code migration to other hardware resources and the use of updated copies of the data

- *tolerating failures* – some failures have to be tolerated; any attempt to solve them is pointless or impractical; in these cases, instead of leaving the user stranded, while successive access attempts are carried out, it is better to let him/her know of the fact.

Concurrency

Being resource sharing the main motivation which leads to the design and the implementation of *distributed systems*, it is paramount to ensure that the entity, or entities, managing access to the shared resource, impose mutual exclusion for the operation to be performed in a consistent fashion.

When access control is *centralized*, that is, dependent of a single entity, there is a simple solution based on standard devices for imposing mutual exclusion and synchronization, developed for multiprogramming environments in monoprocessors.

When, however, access control to the resource is *distributed*, dependent of several independent entities, the solution is a lot more complex because one has first to tackle with the problem of how to synchronize distinct entities not subject to a global clock.

Ubiquitous computing (*Internet Of Things*)

Principles

- *to make the communication natural* – to provide intelligent interface devices among computer systems and human beings, or other computer systems, in order to make information exchange spontaneous
- *to make computing sensitive to context* – to select the actions to be taken which are specific to current scenery interpretation.

Important application areas

- *intelligent houses* (*domotics* or *house automation*) – setting up house appliances, remote control and monitoring, establishing surveillance procedures, etc
- *autonomous vehicles* – vehicle to vehicle communication, enforcing of scenery dependent traffic rules, etc.

Cloud computing

Principles

- *resources on demand* – making available to potential users different kinds of service
  - specific applications, run remotely (*software as a service*)
  - specific hardware means for computation (*platform as a service*)
  - specific resources, such as mass storage or tailored running environments (*infrastructure as a service*)
- *indirect access* – resources are accessed remotely through standard protocols run from local computer systems, tablets or smart phones
- *scalability* – the system should have the ability to adapt to users needs, creating the illusion that the resources are unlimited.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 1: *Characterization of distributed systems*
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 1: *Introduction*

Sistemas Distribuídos

System Models

António Rui Borges

Summary

- *Models*
- *Types of models*
- *Architectural models*
  - *Client-server*
  - *Peer communication*
  - *Publisher-subscriber*
- *Communication primitives*
- *Fundamental models*
  - *Interaction*
  - *Failure*
  - *Security*
- *Suggested reading*

Models

Distributed systems are aimed to operate in the real world, which means that they must be designed to work properly even when subjected to a wide range of operating environments and/or facing difficult scenarios and predictable menaces.

One uses *description models* to enumerate the common properties and the design assumptions characteristic of a particular class of systems.

Thus, the goal of a model is to provide a simplified and abstract, but consistent, description of relevant features of the system under discussion.

Types of models

- *architectural models* – they define the way how the different components of the system are mapped into the nodes of the underlying parallel platform and how they interact among themselves
  - *mapping wise*: they aim to establish efficient patterns of data distribution and processing loads
  - *interaction wise*: they describe the functional role assigned to each component and their communication patterns
- *fundamental models* – they discuss the systemic characteristics affecting dependability
  - *interaction type*: they deal with issues such as communication bandwidth and latency
  - *failure handling*: they specify the kind of failures that may occur in the intervening processes and communication channels
  - *security*: they discuss possible menaces which are posed to the distributed system and which affect its performance

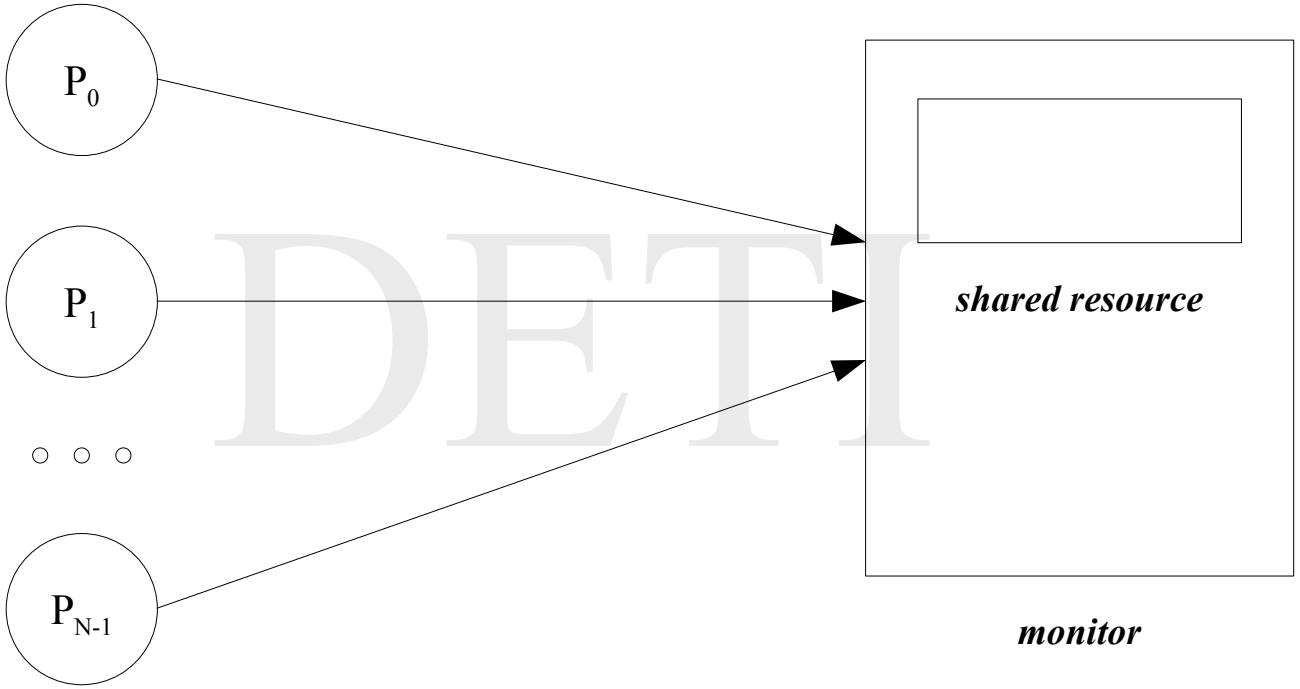
Architectural models

In a *distributed system*, the intervening processes cooperate among themselves in the undertaking of a common task. Responsibility division and mapping on specific processing nodes of the underlying parallel platform are among the most distinctive project issues.

A working approach consists in considering first a concurrent solution to the problem (valid in a monoprocessor computer system) and perform next a set of transformations leading to the migration of the solution to a computer network.

In designing the concurrent solution, any of the communication paradigms may be used, *shared variables* or *message passing*, as they are equivalent. The former, however, leads to more intuitive implementations.

Concurrent solution - 1



Concurrent solution - 2

- the shared resource belongs to the addressing space of all the intervening processes
- a *monitor* protects the access to the resource enforcing mutual exclusion
- process synchronization is done inside the monitor through *condition variables*
- the interaction model is *reactive*: each process runs until blocking or termination
- interaction itself is based on calling operations upon the monitor
- alternatively, if the programming language does not have a concurrency semantics, the shared resource may be protected by an *access* semaphore (or some similar device) and synchronization be carried out by semaphores as well, located now outside the critical region to prevent deadlock

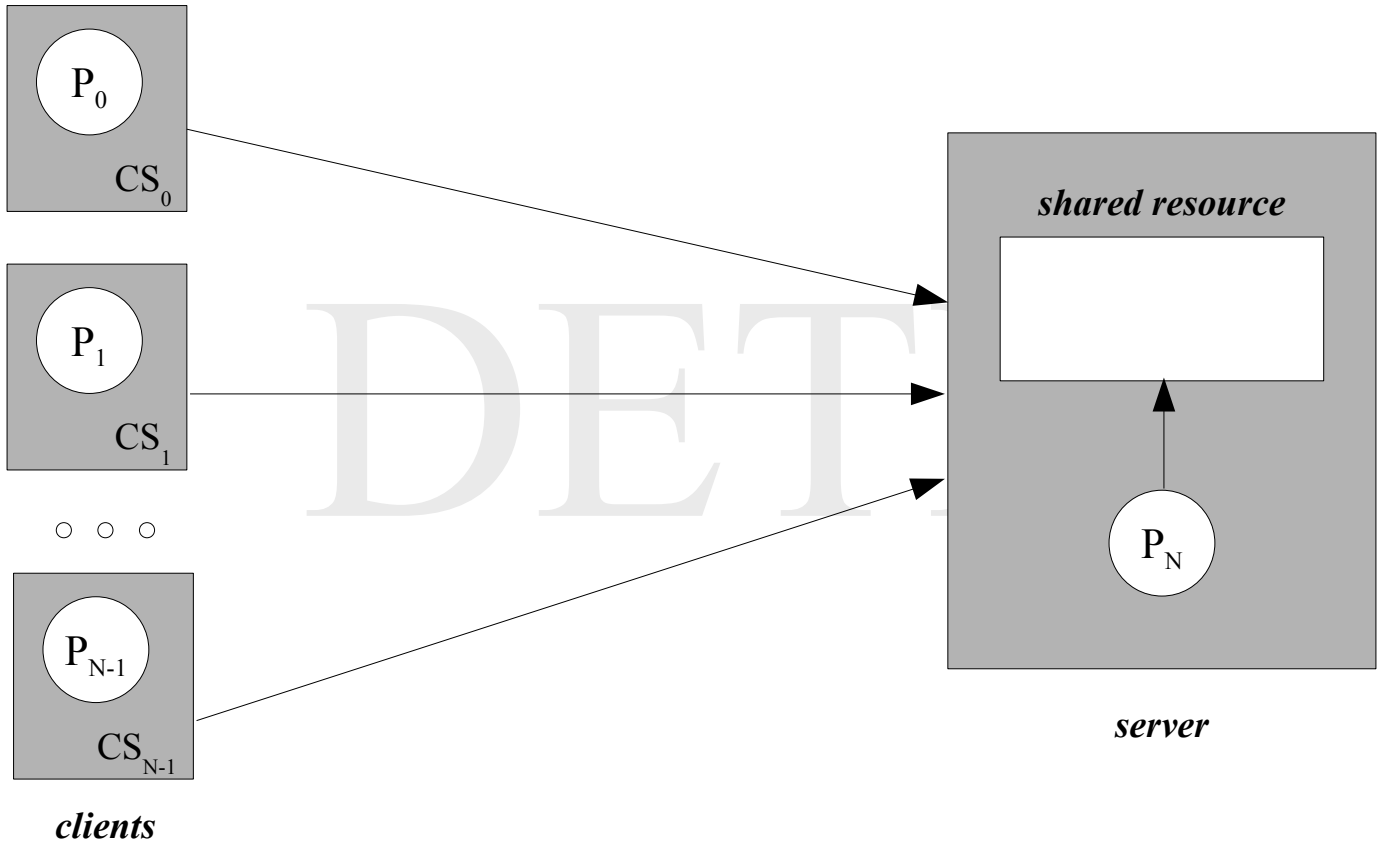
Client-server model – 1

On migrating processes and the shared resource to different computer systems, their addressing spaces become disjoint and communication must take place, in a implicit or explicit way, by message passing on a common communication channel.

A critical issue to consider is that, being the shared resource a passive entity, the access management to it requires the creation of a new process, whose duty is not only communicating with the other processes, but also executing locally the operations called on the shared resource.

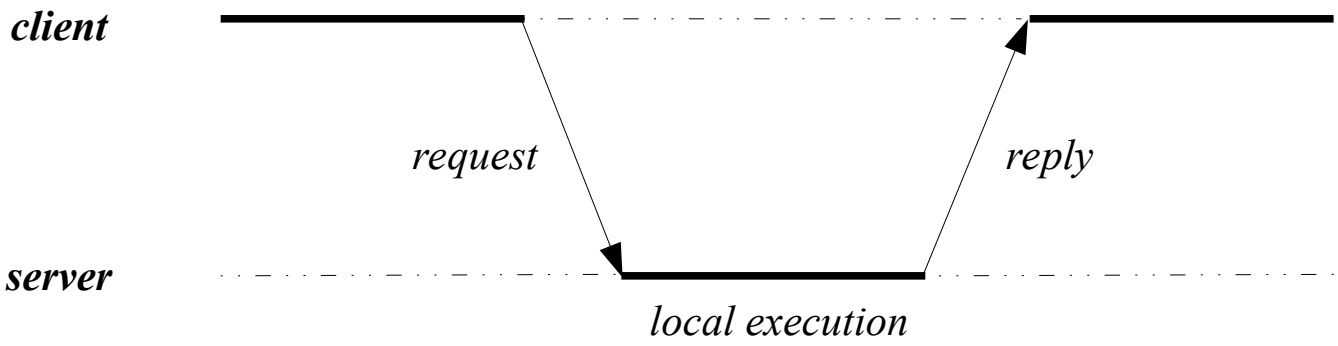
Therefore, an operational model where resource manipulation can be understood as *service rendering* becomes apparent. The process managing locally the shared resource is seen as the service render, usually called *server*. The remaining processes, which want to access the resource, are seen as the entities that request the service, called *clients*.

Client-server model – 2



Client-server model – 3

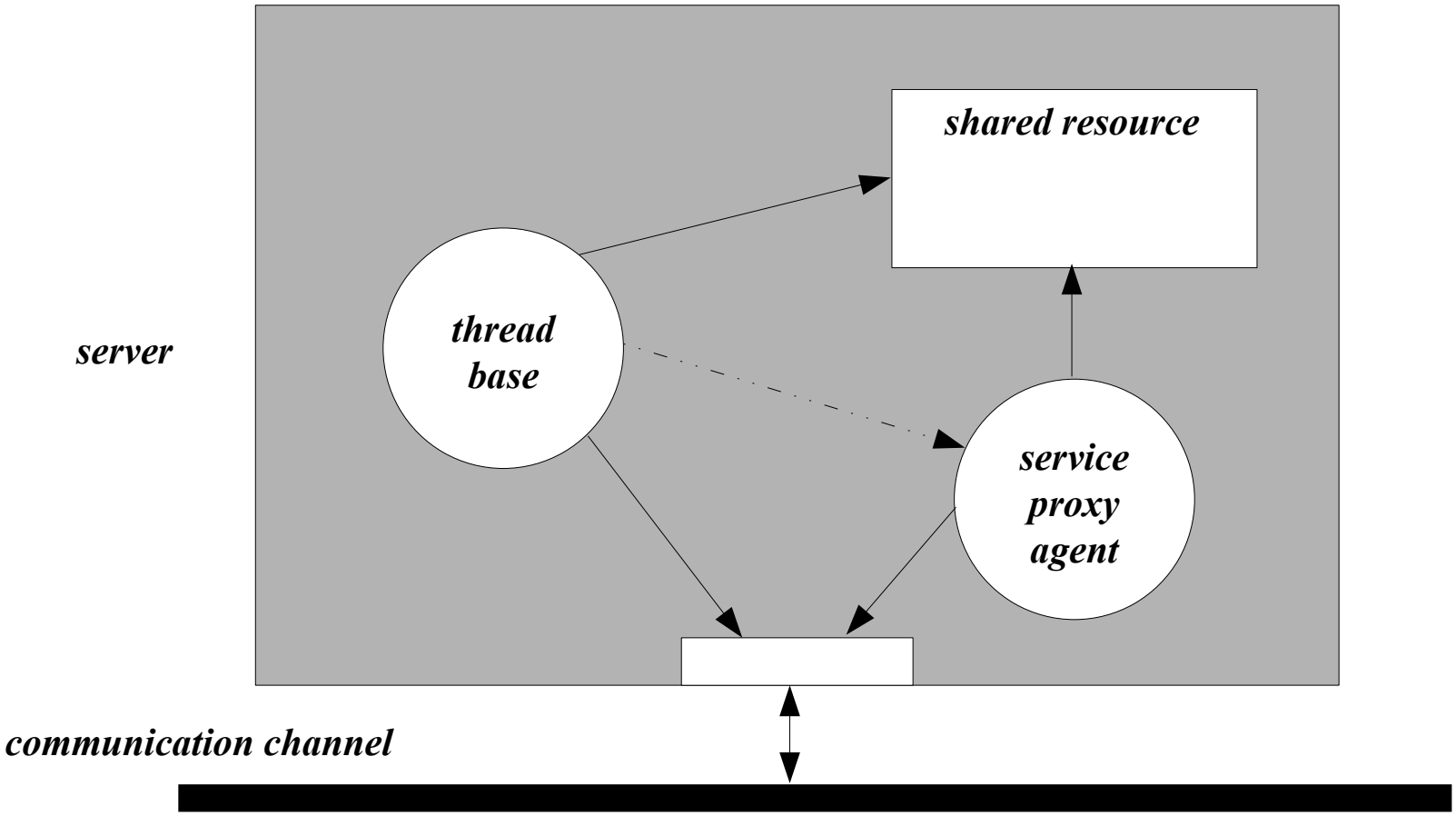
- operations called on the shared resource may be divided in
  - *request*: *client* process calls *server* process asking it to execute an operation on its behalf
  - *local execution*: execution by *server* process of the requested operation upon the shared resource
  - *reply*: *server* process answers to *client* process communicating the result of the operation



Client-server model – 4

- the *server* process has typically two roles, acting as a
  - *communication manager*: waits for service requests by *client* processes
  - *service proxy agent*: executes the operations on the shared resource as a proxy of the *client* processes
- the communication model is asymmetric
  - *server* → ***public*** / *clients* → ***private***: service operation requires that the service be known by all the interested parties, while service users do not need to be previously known by the service suppliers
  - *server* → ***eternal*** / *clients* → ***mortals***: service availability require it to be permanently operational, while its users only manifest themselves from time to time
  - *centralized control*: all service operations are centered in a single entity

Basic server architecture – 1

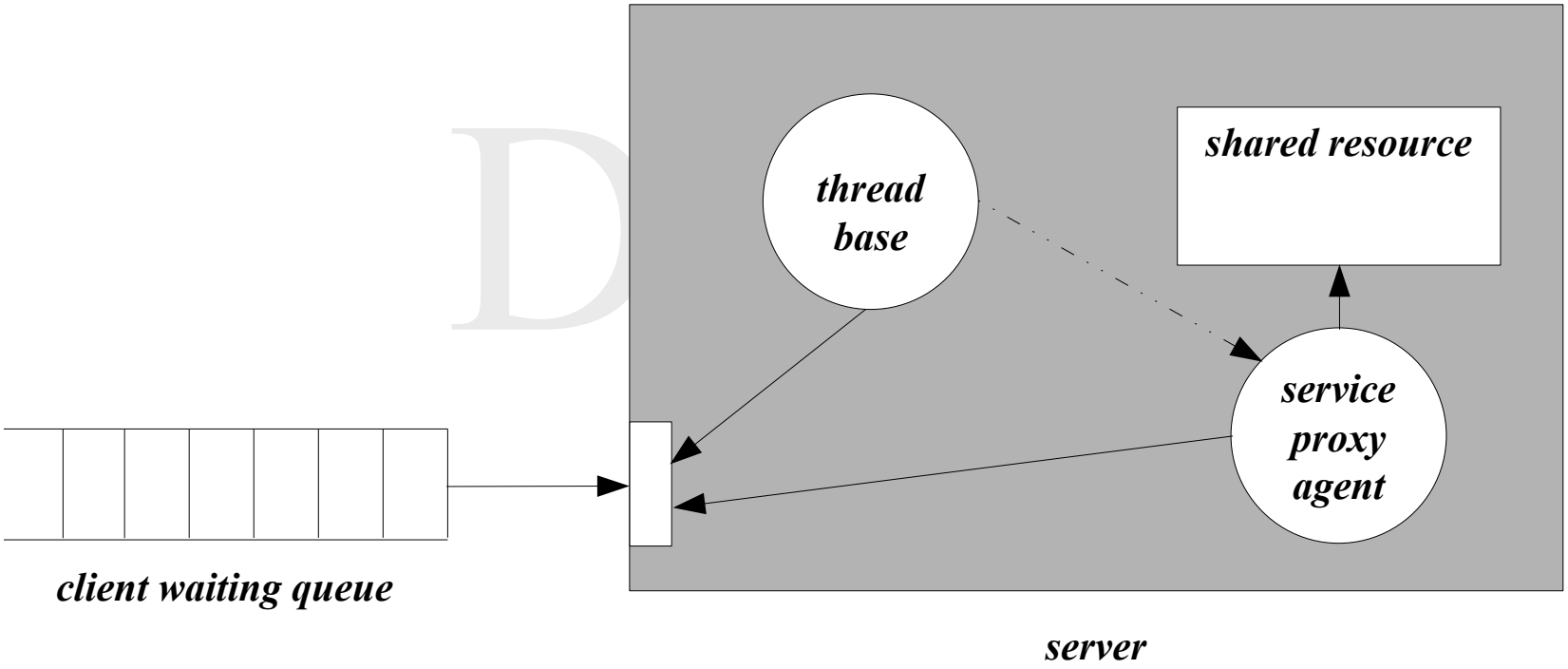


Basic server architecture – 2

- role of thread *base*
  - instantiate the shared resource
  - instantiate the communication channel and map it to a public known address
  - start listening at the communication channel
  - when a connection from a *client* process is established, create a *service proxy agent* thread to deal with the *client* request
- role of thread *service proxy agent*
  - determine the operation *client* process wishes to be performed in the shared resource (*request*)
  - execute the operation on its behalf (*local execution*)
  - communicate operation result to *client* process (*reply*)

Variants of the client-server model – 1

Request serialization



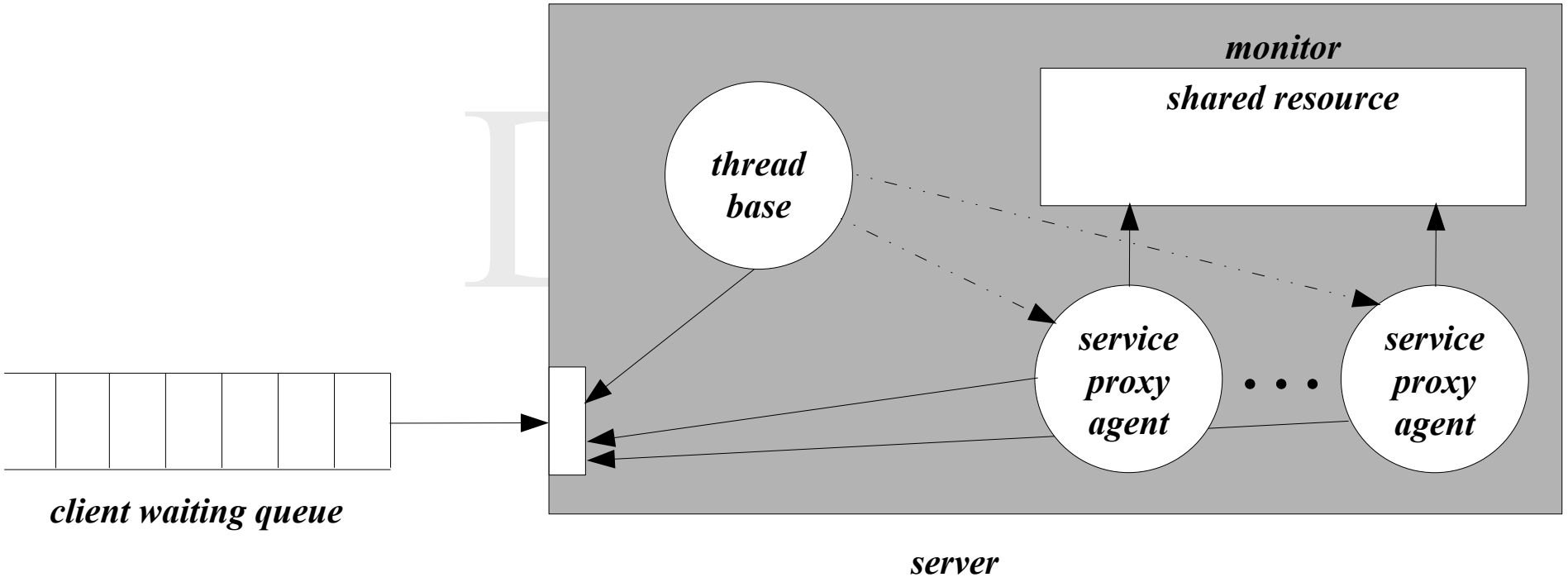
Variants of the client-server model – 2

Type 1 variant (***request serialization***)

- only a *client* process is serviced at a time: this means that the thread *base*, upon receiving a connection request, instantiates a *service proxy agent* and waits for its termination before start listening again
- the shared resource does not need any special protection to ensure mutual exclusion on access, since there is a single active *service proxy agent* thread
- it is a very simple, but rather inefficient, model since
  - the service time is not minimized, because one does not take advantage of the interaction dead times due to the lack of competition
  - it gives rise to *busy waiting* in the attempt to synchronize multiple client processes on the same shared resource.

Variants of the client-server model – 3

Server replication



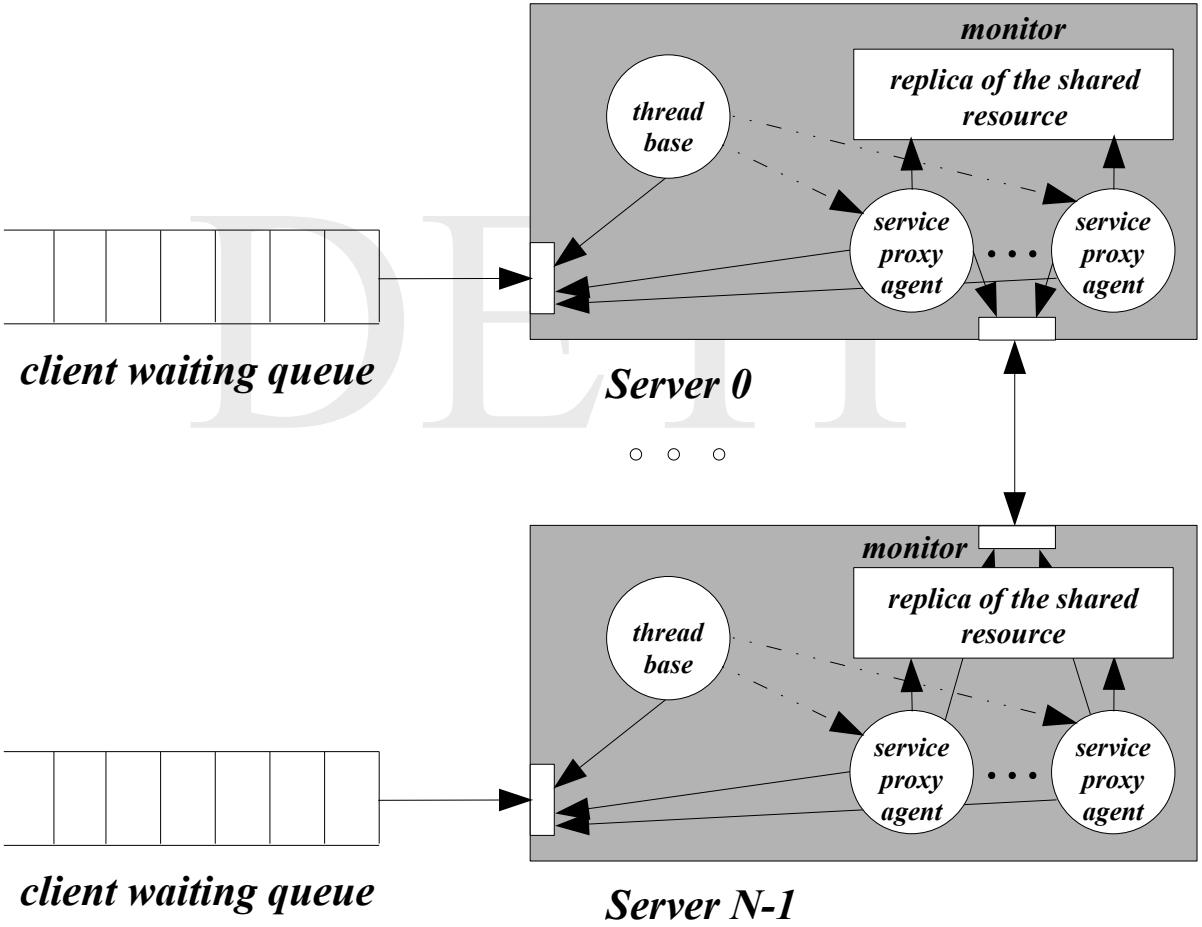
Variants of the client-server model – 4

Type 2 variant (***server replication***)

- *client* processes are serviced concurrently: this means that the thread *base*, upon receiving a connection request, instantiates a *service proxy agent* and starts listening again
- the shared resource is transformed into a monitor to ensure mutual exclusion on access, since there are now multiple active *service proxy agent* threads at the same time
- it is the traditional way *servers* are set up, as one tries to make the most of the resources of computer system where the server is located
  - the service time is minimized, because one takes advantage of the interaction dead times through concurrency
  - it enables the synchronization of different *client* processes on the same shared resource.

Variants of the client-server model – 5

Resource replication

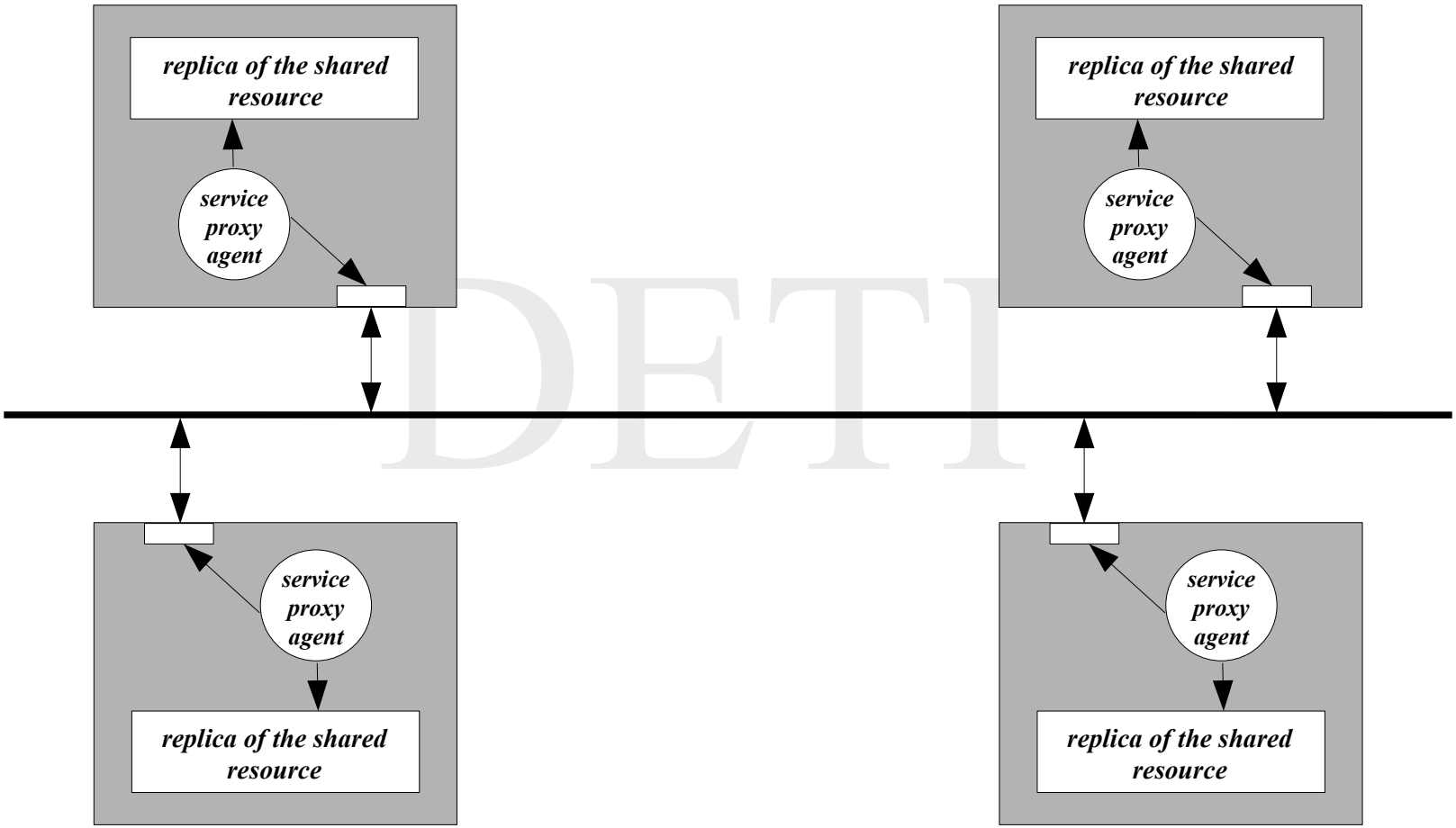


Variants of the client-server model – 6

Type 3 variant (***resource replication***)

- the service is simultaneously made available in several computer systems, each running a type 2 variant of the *server*
- the shared resource is, thus, replicated in each server, giving rise to multiple copies
- it is a sophisticated model which aims both to maximize service availability and to minimize the service time, even in load peak situations (thus, potentiating scalability)
  - the service is kept operational against the failure of particular *servers*
  - *client* requests are distributed among the available servers using a policy, provided by the DNS service, of geographical association for global requests and of rotational association for local requests
  - when there is an alteration of local data at one of the replicas of the shared resource, the need to keep the different replicas consistent arises and has to be dealt with.

Peer communication – 1



Peer communication – 2

- the service is simultaneously made available in several computer systems, each playing the same role; that is, as far as the service is concerned, there is no difference among the different processing nodes, called here *peers*
- the shared resource is, thus, replicated in each peer, giving rise to multiple copies
- it is a sophisticated model which aims both to maximize service availability and to minimize the service time, even in load peak situations (thus, potentiating scalability)
  - the service is kept operational against the failure of particular *peers*
  - in special situations, one of the peers has to assume a leading role so that the system as a whole may make a smooth transition from one stable state into another; the leader's choice is typically subjected to an election process where consensus must be reached.

Communication primitives - 1

Communication through *message passing* assumes two entities: the *forwarder*, which sends the message; and the *recipient*, that receives it.

The *send* primitive has at least two parameters: the destination address and a reference to a buffer in user space containing the data to be sent. Similarly, the *receive* primitive also has at least two parameters: the source address and a reference to a buffer in user space where the received data is to be stored.

Typically, the communication is *buffered* by the operating system: that is, on a send operation, data are first transferred into a kernel buffer before being properly delievered to the network; on a receive operation, data are first stored in a kernel buffer, upon reception, and are only transferred into the user buffer when the *receive* primitive is called.

Communication primitives - 2

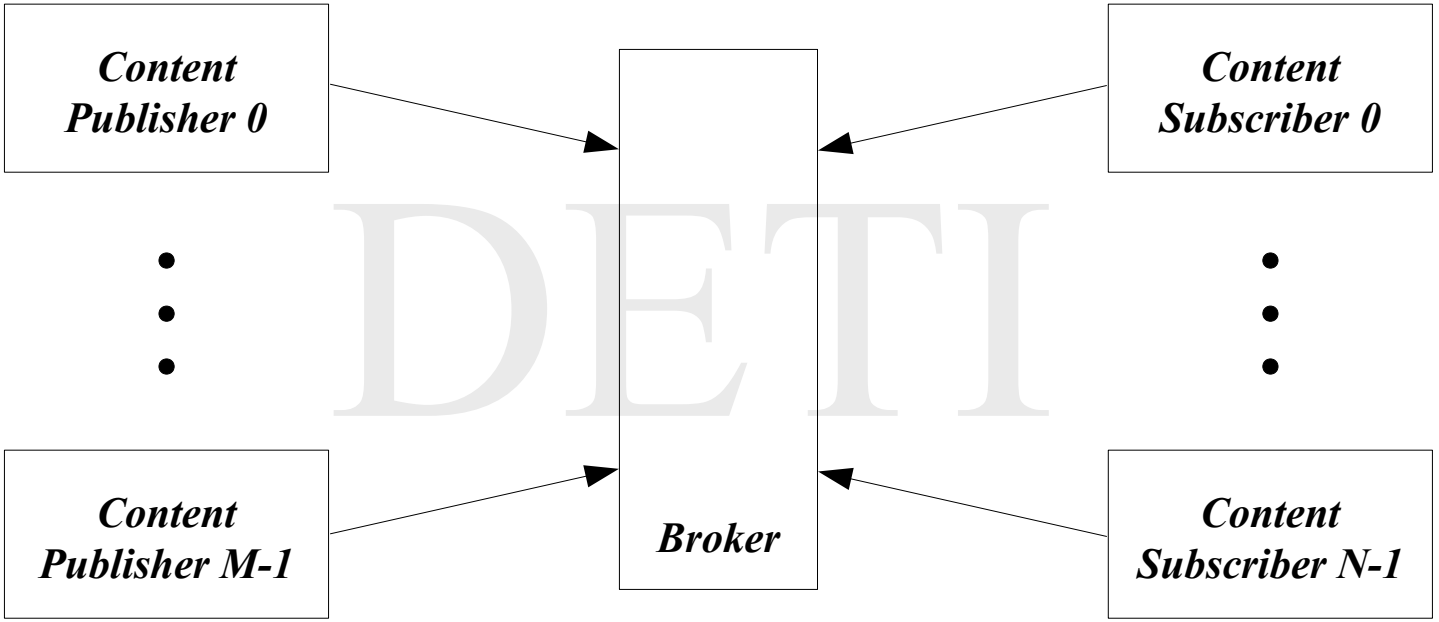
Communication primitives may be divided into

- *synchronous* – if *send* and *receive* primitives are coupled together: the send operation only completes when the forwarding processing node is aware that, on the recipient processing node, the receive operation has also been completed
- *asynchronous* – if *send* and *receive* primitives are totally uncoupled: the send operation completes as soon as data are transferred from the buffer in user space; there is no asynchronous receive primitive.

They may be further divided into

- *blocking* – if control only returns to the invoking process after the operation, whether synchronous or asynchronous, has completed
- *non-blocking* – if control returns to the invoking process immediately after the primitive is called, even if the respective operation has not yet been completed.

Publisher-subscriber model - 1

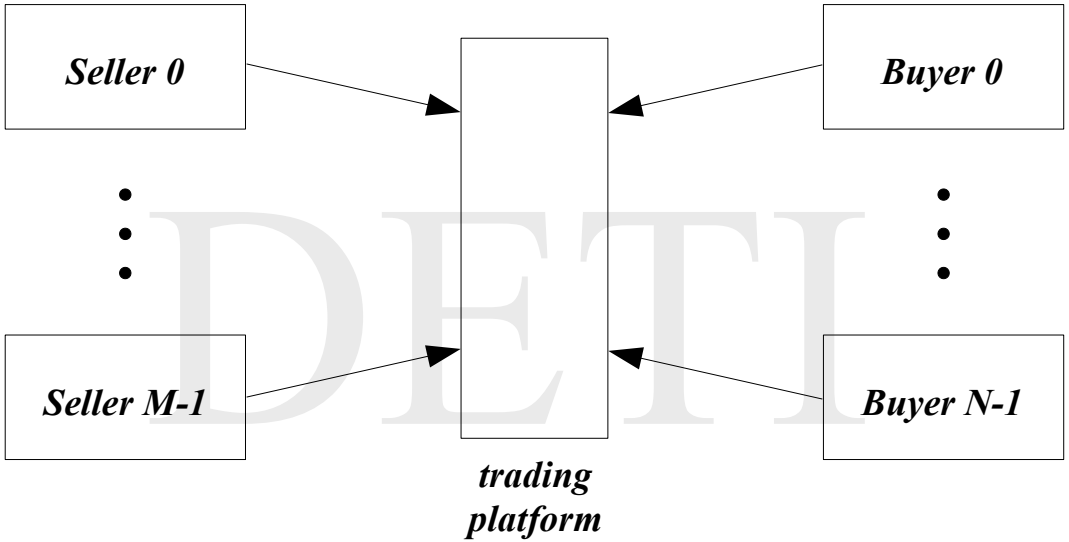


Publisher-subscriber model - 2

- there are multiple service providers, the *publishers*, and multiple service recipients, the *subscribers*, which are totally decoupled from one another through the mediation of an intermediary service, the *broker*
  - *publishers* produce information according to different topics and act as clients of the *broker* which store it and make it available to *subscriber* groups that have explicitly subscribed the topic it is associated with
  - *subscribers* act as clients both of the *publishers* and of the *broker*: the *publishers*, as they consume specific information produced by them; the *broker*, as they let it know of the topics they are interested in and that they should be alerted of when new data are made available
- the key feature is that, in contrast with the conventional *client-server model*, there is no synchronous interaction taking place here.

Publisher-subscriber model - 3

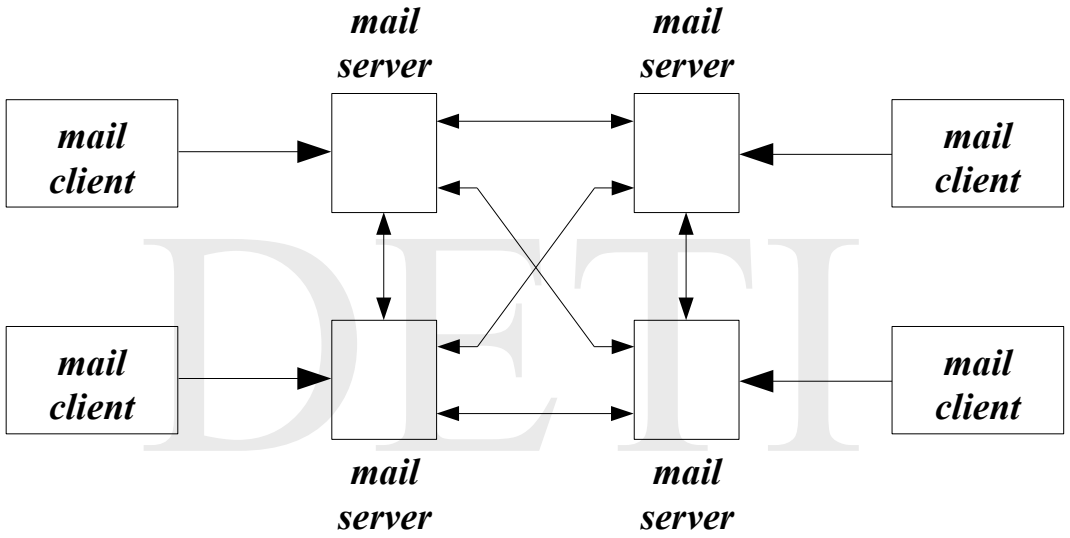
Electronic trading or **e-trading**



- the *sellers* act as *publishers* who post data about the products they want to sell at the trading platform
- the *buyers* act as *subscribers* who request the trading platform information about the products they want to buy
- the *trading platform* is the *broker* who manages possible transactions among them.

Publisher-subscriber model - 4

Electronic mail or **e-mail**



- the *mail servers* act as both *brokers* to local *mail clients* and other *mail servers*, for managing local messages, and as *publishers* to others *mail servers*, for forwarding remote messages
- the *mail clients* act as both *publishers* for sending messages and as *subscribers* for receiving messages.

Fundamental models

In a distributed system, where processes cooperate and communicate over a network, performance is directly related to the effectiveness of message exchange. When a remote service is part of it, the speed at which the exchange takes place is determined not only by the load and the performance of the server, but also by the routing and transferring capabilities of the interconnecting network and the delays in all the software components involved. To achieve short interactive response times, systems must be composed of relatively few software layers and the amount of data transferred per interaction should be small.

But fast response is not all that matters for a good *quality of service* to be attained. *Reliability*, *security* and *adaptability* to meet changing system configurations and resource availability are also considered important. In applications which deal with *time-critical* data, the availability of the necessary computing and network resources at the appropriate times is paramount.

Performance of a communication channel

Communication channels may be implemented either by data streams which connect the communication endpoints, or by simple message passing over the computer network.

The following properties are relevant

- *latency* – the delay between the times the sender process starts transmitting the message and the receiver process starts receiving it; it depends on the time the operating system services at both ends take to process the data, the delay to access the network and the routing overhead
- *bandwidth* – the total amount of information that can be transmitted over the computer network in a given time
- *jitter* – the time variation it takes to deliver a sequence of similar messages between the two endpoints.

Variants of the interaction model

In a distributed system, it is very difficult to set precise limits on the time taken for process execution, message delivery and local clock drift.

Tho opposing extreme situations are

- *synchronous distributed systems*
  - the time to execute each process step has known lower and upper bounds
  - each message transmitted over a communication channel is received within a known upper bound
  - each process has a local clock whose drift rate from *real time* has a known upper bound
- *asynchronous distributed systems*
  - the time to execute each process step has an arbitrary, but finite, upper bound
  - each message transmitted over a communication channel is received within an arbitrary, but finite, upper bound
  - each process has a local clock whose drift rate from *real time* is arbitrary.

Failure model

In a distributed system, both the intervening processes and the communication channels may *fail*. This means that their behavior may move apart from the pattern which was defined as desirable or correct.

Failures are usually classified as

- *omission failures* – when the actions prescribed to take place simply do not occur
- *timing failures* – when the actions prescribed to take place do not obey to the previously established time limits
- *arbitrary or bysantine failures* – when an unexpected error can temporally or permanently occur in result of a malfunction of any system component.

Failure classification

Failure Class	*Affects*	*Description*
fail-stop	process	a process halts and remain halted (it is assumed that processes either function correctly, or else stop; other processes may detect its state)
crash	process	a process halts and remain halted (other processes can not detect its state)
omission	channel	a message placed in the outgoing message buffer of the fowarder never arrives at the incoming message buffer of the recipient
send omission	process	a process completes a send operation, but no message is placed in the outgoing message buffer
receive omission	process	a message is placed in the incoming message buffer, but a receive operation by the recipient does not get it
clock	process	the process local clock exceeds the bounds on its drift rate from real time
performance	process	the process exceeds its bounds on the execution time between two operations
performance	channel	a message transmission time exceeds the stated bound
arbirary (or bysantine)	either process or channel	a process / channel exhibits an erratic behavior: it may transmit arbitrary messages at arbitrary times, commit omissions; or a process may stop or take an incorrect action

Security model - 1

The security of a distributed system can be achieved by securing the processes and the communication channels used in their interactions and by protecting against unauthorized access to the resources they encapsulate.

Resources are intended to be used in different ways by different users. They can either hold data private to specific users, accessible to special classes of users or shared by everybody who deals with the system. To support such an environment, *access rights* are defined and specify what operations are available and who is allowed in a differential way to perform them on the resource.

Users are, thus, included in the model as beneficiaries of resource access rights. An authority, called the *principal* and being either a user or a process, is associated with each operation invocation and operation result.

The *server* is responsible for verifying the identity of the principal behind each service request and checking if it has the required access rights so that the operation may be performed on its behalf; otherwise, reject the request. The *client*, on the other hand, must verify the identity of the principal behind the server to be sure that the reply comes from the intended one.

Security model - 2

To model security threats, an *enemy*, or an *adversary* as it is sometimes called, is postulated. The *enemy* is capable of sending any message to any process and/or reading any message exchanged by a pair of processes. The *enemy* may attack a distributed system from a computer platform that is either legitimately connected to the network, or that is connected in an authorized manner.

The threats may be divided in

- *threats to processes* – a process that is designed to handle incoming requests may receive a message which it can not necessary identify the forwarder
- *threats to communication channels* – an enemy may copy, alter and/or inject messages as they travel over the computer network, thus, putting at risk the privacy, the integrity and the availability of system information.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 2: *System models*
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 2: *Architectures*
    - Sections 2.1 to 2.3

Sistemas Distribuídos

Interprocess Communication and Synchronization

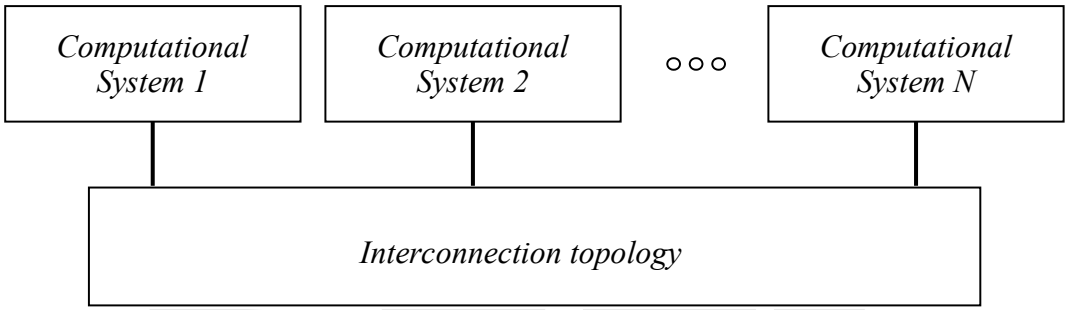
Message Passing

António Rui Borges

Summary

- *Communication system*
- *Programming interface*
  - *TCP protocol*
  - *UDP protocol*
- *Transformation of a concurrent solution into a distributed message passing solution*
  - *Transformation principles*
  - *Message structure*
  - *Client architecture*
  - *Server architecture*
- *Suggested reading*

Communication system - 1



Parameters affecting the performance of a communication system are

- *latency* – delay that occurs after the execution of a *send* operation and the beginning of data reception (it can be envisaged as the transfer of an empty message)
- *data transfer rate* – speed of data transmission between the sender and the receiver
- *bandwidth* – system *throughput* (volume of message traffic per unit of time).

$$message\ transmission\ time = latency + length / data\ transfer\ rate$$

Communication system - 2

When considering *real time multimedia* applications, there is another relevant property which plays an important role, *quality of service*. It describes the system capability to meet *deadline* constraints imposed by the transmission and the processing of data flows in continuum. In order for these operations to occur in a satisfactory manner, one requires an upper limit for latency and a lower limit for bandwidth of the associated data channels.

Present day communication systems are quite reliable. Failures are usually related to errors in the software at the sender or the receiver side than to network errors. Thus, it is common practice to transfer to the applications the responsibility of dealing with the detection and the correction of the remaining errors, a procedure that is known as the *end-to-end argument*.

Communication system - 3

From the application programmer point of view, the communication system must be viewed in an integrated and abstract manner masking the underlying complexity of the diverse physical networks it encompasses.

Thus, network software is arranged in a hierarchy of layers. Each layer presenting an operational interface to the layers above it that describes the properties of the communication system at this level in a logical fashion. A layer is represented by a software module present in every computer system attached to the network.

Each module appears to communicate directly with the corresponding module in another computer system, but in reality data is not transmitted directly between the modules at each level. Each layer of the network software communicates instead by local procedure calls with the levels above and below it.

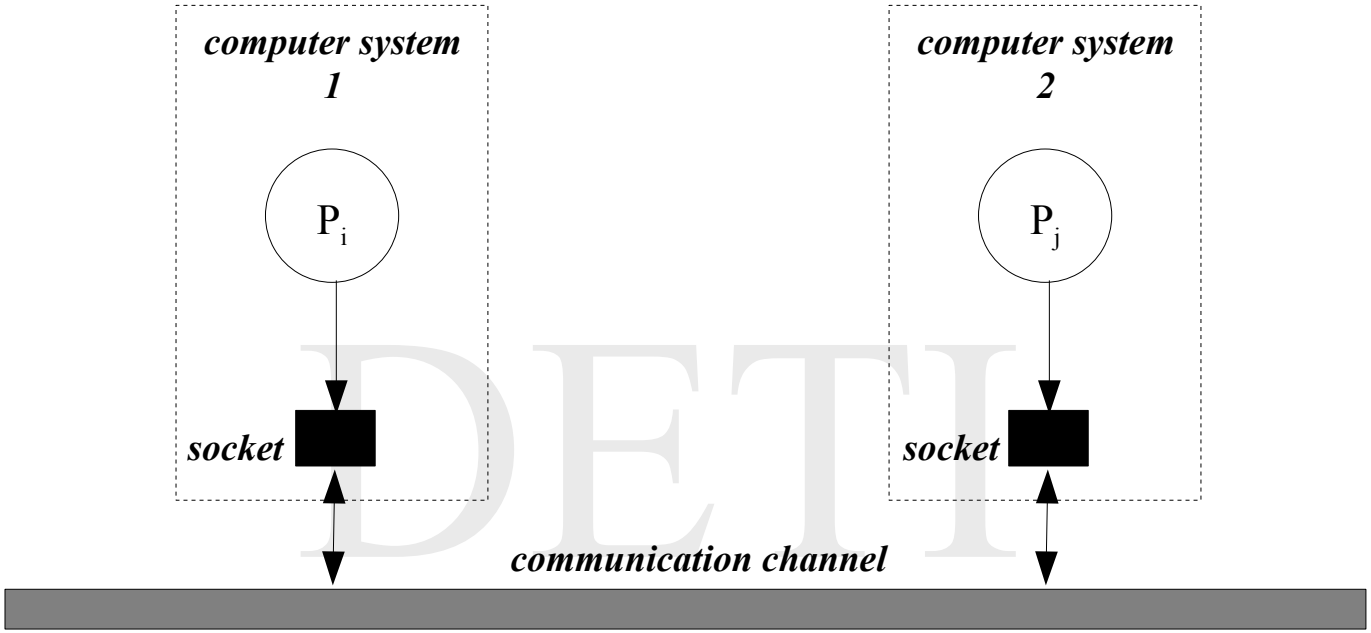
On the *sending side*, each layer, but the topmost, accepts data in a well-defined format from the layer above and applies a transformation procedure to encapsulate it in another well-defined format before passing it to the layer below. *On the receiving side*, the converse transformations are applied to data as it traverses the layers in the reverse direction.

Communication system - 4

OSI model

Application	protocols designed to meet the communication requirements of specific applications
Presentation	protocols to enable data transmission in a network specific representation which is hardware independent
Session	protocols for error detection and automatic recovery
Transport	protocols to enable message addressing to communication ports associated with processes
Network	protocols to enable the transfer of data packets between the network nodes
Data link (logical)	protocols to enable data transmission between network nodes connected by the same physical link
Physical	specification of the circuits and signals that drive a particular physical link

Programming interface – 1



Middleware presents to the application programmer a device, called *end point of communication* or *socket*, to enable message exchange among processes which do not share an addressing space.

Sockets are characterized by the *IP address* of the computer system and a *port* which defines within the computer system the end point of a specific communication channel.

Programming interface – 2

There are two main protocols for message exchange

- *TCP* – it is a *connection-oriented protocol*, meaning that a virtual communication channel must be established between the end points before any data exchange may take place
  it allows *bidirectional communication* because, once the channel has been established, a stream of data may flow from each end point
  it is *asymmetric*, since it was specifically designed for the client-server model, it assumes a different role for each end point
- *UDP* – it is a *connectionless* protocol, meaning that no virtual communication is required for data exchange to take place
  it only allows unidirectional communication because it assumes the transmission of a single message from one of the end points to the other
  it is *symmetric* because no different role is assigned to either of the end points.

TCP protocol – 1

TCP protocol requires two types of sockets

- *listening socket* – instantiated by the *server* and where it is listening for a connection request from a *client*
- *communication socket* – instantiated by the *client* when it needs a data exchange with the *server*, and by the *server* itself when it establishes a virtual communication channel with the *client* .

TCP protocol – 2

Client side

```
instantiateComSocket ();
connectToServer (serverPublicAdd);
openInputStream ();
openOutputStream ();
writeRequest ();
readReply ();
closeOutpoutStream();
closeInputStream();
closeComSocket ();
```

Server side

Thread base

```
instantiateListenSocket (serverPublicAdd);

while (true)
{ comSocket =
     listenToClientConnectionReq (serverPublicAdd);
  instantiateServiceProxyAgent (comSocket)
  startServiceProxyAgent ();
}
```

Service Proxy Agent

```
openInputStream ();
openOutputStream ();
readRequest ();
localExecution ();
writeReply ();
closeOutpoutStream();
closeInputStream();
closeComSocket ();
```

UDP protocol – 1

UDP protocol requires a single type of socket to transmit a message, called a *datagram packet*, from the source to the destination point

- *receiving socket* – instantiated by the *receiver* at a specific port for packet reception from different sources
- *sending socket* – instantiated by the *sender* for packet transmission to different destinations.

UDP protocol – 2

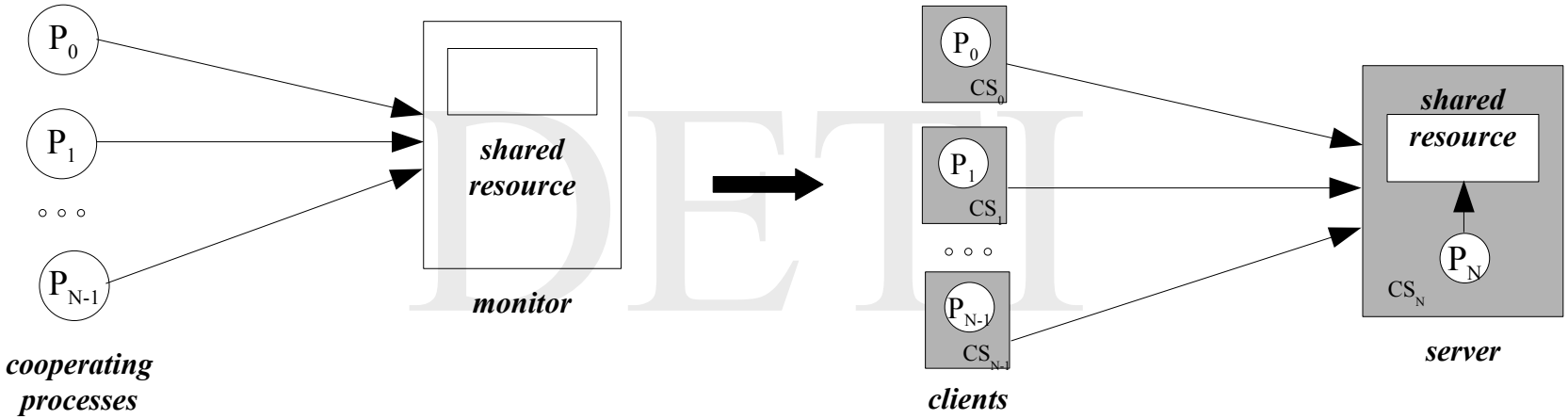
Source side

```
instantiateSendSocket ();
convMesgToByteArray
            (msg, byteArray);
instantiateDataPacket
     (byteArray, DestPublicAdd);
send (dataPacket);
```

Destination side

```
instantiateRecSocket
                (DestPublicAdd);
receive (dataPacket);
convByteArrayToMesg
                 (byteArray, msg);
```

Transformation principles – 1



Transformation principles – 2

A key feature of the transformation is that it must be carried out with minimal changes of the concurrent code, namely, the whole interaction mechanism among the different entities must be kept as it was previously set.

Since both the cooperating processes and the shared resource reside in different computer systems, there is no sharing of addressing space which entails that

- method invocation upon the shared resource must be carried out through message exchange: one message for the call and another for the call return
- besides the method parameters and the return value, messages should include those attributes of the caller process which are relevant for the execution of the method, or that are changed by its execution
- all message parameters are to be passed by value.

Message structure – 1

A message is transmitted through a communication channel and, at the lowest level, may be seen as an array of bytes. Since the client and the server processes are separate programs, it crucial for the receiver to know how to interpret this array of bytes and how to build from this data the values of the message parameters.

Thus, a message contents must include not only the values of the parameters, but also their type and how they are structured. The operation of building a message with these characteristics is called *information marshaling*, and the opposite operation of retrieving the values of the parameters from the array of bytes as *information unmarshaling*.

In Java, marshaling and unmarshaling of information is hidden from the programmer. It is only required that the message data type be defined as implementing the `Serializable` interface.

Message structure – 2

```
import java.io.Serializable;

public class Message implements Serializable
{
    private static final long serialVersionUID = <long literal>;

    /* definition of the message parameters */

    /* message instantiation */

    /* public methods for getting the values of the message parameters */
}
```


- if a message parameter is of a reference data type, it must also implement the `Serializable` interface
- this rule should be applied in a recursive way so that what remains are parameters of primitive data types

Message structure – 3

```
import java.io.Serializable;

public class Record implements Serializable
{
    private static final long serialVersionUID = 20140404L;

    public int nEmp;
    public String nome;

    public Record (int nEmp, String nome)
    {   this.nEmp = nEmp;
        this.nome = nome;
    }
}
```


serial representation of `new Record (105, "Ana Francisca")`

```
ACED0005737200065265636F726400000000013351740200024900046E456D704C00046E6F6D6574
00124C6A6176612F6C616E672F537472696E673B787000000006974000D416E61204672616E636973
6361
```

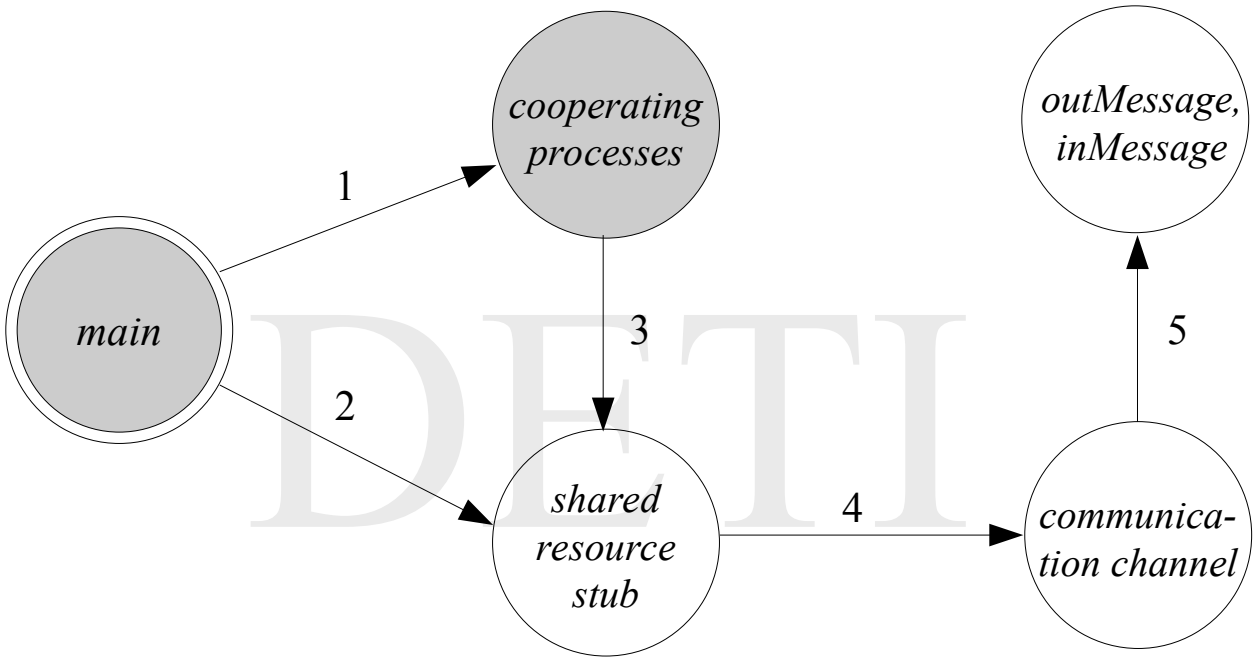
run `ShowMain` in the package `showSerialization` to see the interpretation of the array of bytes above

Client architecture – 1

In the concurrent solution, the `main` thread instantiated both the cooperating processes and the shared resource. Now, it can not any more instantiate the shared resource since it belongs to a different addressing space. However, to keep most of the code unchanged, a remote reference to the shared resource is to be instantiated. This remote reference, usually called the *stub* of the shared resource, has as instantiation parameters the internet address of the server where the shared resource is located and the number of the listening port. All other values used formerly for its instantiation must now be passed through the invocation of a new method on the stub.

It is the stub responsibility to convert all method invocations on the shared resource into an exchange of messages with the server where the shared resource is located.

Client architecture – 2



1 – instantiate, start, join

2 – instantiate, possible parameter communication for initialization, shutdown

3 – already defined methods

4 – instantiate, open, close, writeObject, readObject

5 – instantiate, get field values

Client architecture – 3

Data type which defines the main thread

It remains mostly unchanged. The modifications are the following

- the stub of the shared resource is instantiated instead of the shared resource itself
- if other values were formerly required for the instantiation of the shared resource, those values are now passed through the invocation of a new method on the stub
- if the server shutdown is required after the end of operations, the invocation of a new method on the stub is required.

Data type which defines the cooperating processes

It remains mostly unchanged. The modifications are the following

- a reference to the stub of the shared resource is passed upon instantiation instead of a reference to the shared resource itself.

Client architecture – 4

Data type which defines the stub of the shared resource

It is new and must be created. It is rather regular and, for the operations that are invoked on it, the following steps must be defined

- a communication channel to server is opened (instantiated)
- an outgoing message is instantiated based on the method identification, its parameters and the values of those attributes of the caller process which are relevant for the execution of the method
- the outgoing message (service request) is sent
- an incoming message (reply) is received and is checked for correctness
- those attributes of the caller process that were affected by the execution of the method are to be updated
- the communication channel is closed
- the method returns.

Client architecture – 5

Data type which defines the communication channel

It new and must be created. Its key feature is to encapsulate the operations carried out on sockets. The `ClientCom` data type, which is provided in the examples, can be used as it is or be modified according to specific requirements.

Data type which defines the message

It new and must be created. There may be a single data type that encompasses all the cases or multiple data types suitable for different situations.

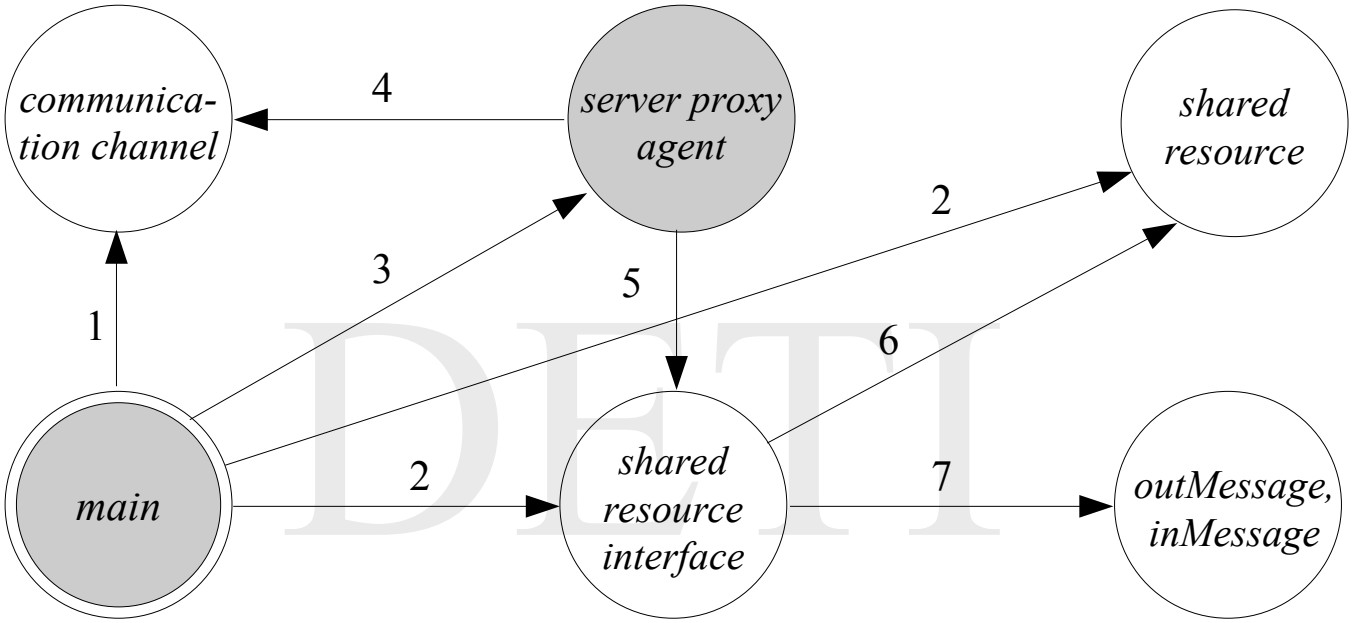
Server architecture – 1

The shared resource is a passive entity. So, in order to have it operational, the main thread, or base thread, of the server must instantiate it as well as a communication channel listening on the public address for service requests.

When a service request comes, the base thread has to instantiate and start a *service proxy agent* thread to deal with the request and returns next to the listening activity to take care of possible new incoming service requests (*server replication variant*).

The service proxy agent receives the incoming message, decodes it and sets itself as a clone of the client by incorporating the process attributes that are required, invokes the corresponding method on the shared resource, composes the outgoing message, sends it, closes the communication channel and terminates.

Server architecture – 2



1 – instantiate, start, end, accept

2 – instantiate

3 – instantiate, start

4 - readObject, writeObject, close

5 – processAndReply

6 – already defined methods

7 – instantiate, get field values

Server architecture – 3

Data type which defines the main thread

It is new and must be created. It is, however, almost invariant for all servers. The modifications are the following

- the public address for service requests depends on each server
- the shared resource and its interface which are instantiated, are specific of each server.

Data type which defines the service proxy agent thread

It is new and must be created. It is, however, almost invariant for all servers. The modifications are the following

- if one wants to have it run as a clone of the several classes of clients which access the server, it should implement the interfaces to each class related to the setting and the getting of the relevant attributes.

Server architecture – 4

Data type which defines the interface to the shared resource

It is new and must be created. Its organization, however, is invariant for all servers. The internal structure is the following

- there is only a public method, `processAndReply`, which decodes the incoming message (service request), processes it and generates the outgoing message (reply)
- the internal operation may be divided in two parts
  - incoming message validation with eventual incorporation of the client attributes
  - method invocation and outgoing message generation.

Data type which defines the shared resource

It remains mostly unchanged. The modifications are the following

- if there are specific references to the cooperating processes data types, they are to be changed to the service proxy agent data type.

Server architecture – 5

Data type which defines the communication channel

It new and must be created. Its key feature is to encapsulate the operations carried out on sockets. The `ServerCom` data type, which is provided in the examples, can be used as it is or be modified according to specific requirements.

Data type which defines the message

It is the same data type used in the client side.

Server architecture – 6

In general, in a given application, several servers are involved, some of them requesting services on others. Thus, one has a situation where some of the servers are simultaneously servers and clients.

This presents no conceptual difficulty. One has only to merge both functionalities producing a mixed architecture.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 4: *Interprocess communication*
    Sections 4.1 to 4.4
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 4: *Communication*
    Sections 4.1 to 4.3

Sistemas Distribuídos

Interprocess Communication and Synchronization

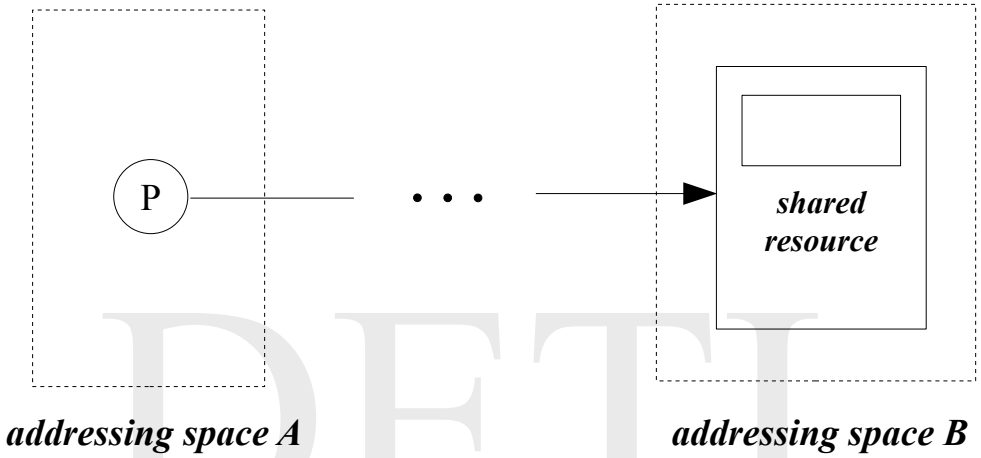
Remote Objects - 1

António Rui Borges

Summary

- *What is a remote procedure call*
- *Architecture of a remote procedure call environment*
- *Code migration*
- *Suggested reading*

What is a remote procedure call - 1



In a *remote procedure call*, the process which invokes a method on a shared resource, and the shared resource itself do not share the same addressing space. This means that

- in the addressing *B* the shared resource belongs to, a reference to it must be generated and made available to others
- in the addressing *A* where the calling process resides, a reference to the shared resource must be obtained before a method can be invoked on it.

What is a remote procedure call - 2

A *remote* procedure call does not behave exactly as a *local* procedure call. There are three distinctive features one should be aware of

- *the call may fail, even if the code has no errors*: this is due to the fact that either the remote resource, belonging to a different addressing space, is not presently instantiated, or the communication infrastructure connecting the two addressing spaces is not operating correctly
- *all procedure parameters and the return value, if it exists, must be passed by value*: since the invoking process and the shared resource reside in different addressing spaces, the sole available means of communication is through passing the relevant data itself, together with its format and structure; thus, information marshaling must take place at the source and information unmarshaling at the destination
- *remote procedure execution takes longer than local procedure execution*, a communication mechanism between the two addressing spaces is implicit, which requires that some kind message exchange between them exists.

What is a remote procedure call - 3

addressing space A

```
/* get a remote reference to the
   shared resource */
remRef = getRef (namingService);
            .  .  .
/* invoke the procedure xyz on the
   shared resource */
try
{ remRef.xyz ();
}
catch (error)
{ /* process error */
}
```

addressing space B

```
/* instantiate the shared resource */
locRef = new SharedResource ();

/* generate a remote reference to the
   shared resource */
remRef = generateRef (pubListenAdd);

/* register it in a naming service */
regRef (remRef);
```

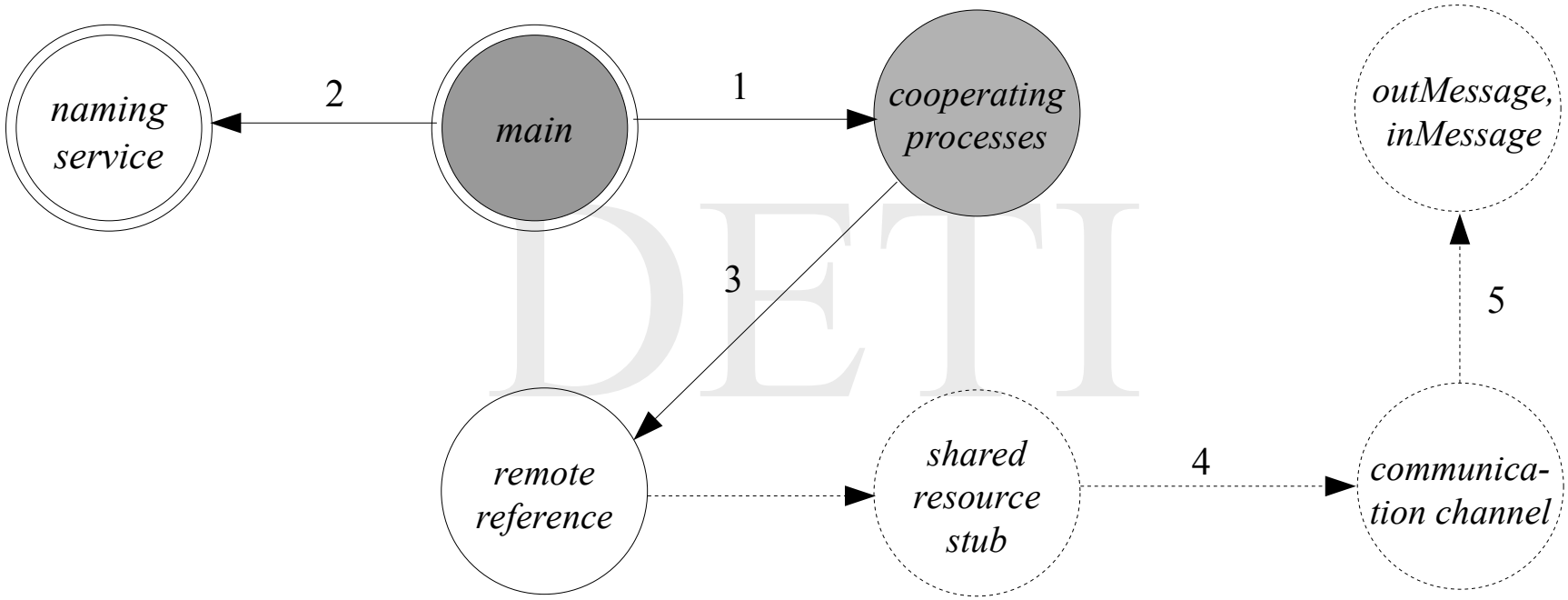
What is a remote procedure call - 4

An environment which implements remote procedure calls, must provide a *naming service* to register shared resources which are to be accessed remotely. On way of looking at it is by assuming it works like a dynamically organized telephone directory, mapping in this case the publicly known name of a shared resource to its location and access features. Thus, the application programmer in order to have network transparency, is only required to know the location of the naming service and the name of the entry to the shared resource in it.

On the other hand, the *remote* reference to the shared resource can be thought of containing the internet address of the platform where the shared resource is instantiated, the port number used to establish a communication with it, the signature of all the procedures which can be invoked on it and the name of the files which provide a description of the data types used as procedure parameters or return values.

Architecture of a remote procedure call environment – 1

Addressing space A



1 – instantiate, start, join
2 – get remote reference
3 – invocation of remote methods

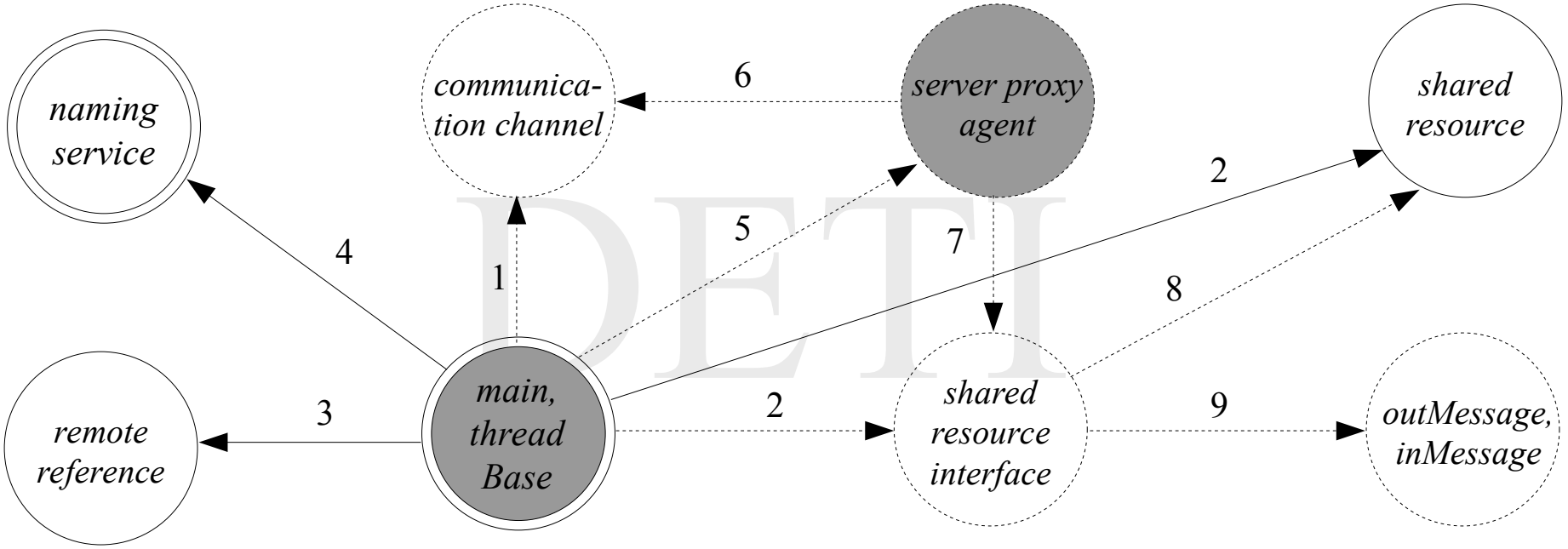
4 – instantiate, open, close, writeObject, readObject
5 – instantiate, get field values

Architecture of a remote procedure call environment – 2

- only the code associated with the entities described by continuous lines have to be written by the application programmer
- the code associated with the entities described by dashed lines is generated by the environment in a manner completely transparent to the programmer
- the programmer has only to provide the signature of all the procedures which can be invoked on the remote object and a description of the data types used as procedure parameters or return values

Architecture of a remote procedure call environment – 3

Addressing space B



1 – instantiate, start, end, accept
2 – instantiate
3 – generate
4 – register
5 – instantiate, start
6 – readObject, writeObject, close
7 – processAndReply
8 – method invocation
9 – instantiate, get field values

Architecture of a remote procedure call environment – 4

- only the code associated with the entities described by continuous lines have to be written by the application programmer
- the code associated with the entities described by dashed lines is generated by the environment in a manner completely transparent to the programmer
- the programmer has only to provide the signature of all the procedures which can be invoked on the now local object and a description of the data types used as procedure parameters or return values
- the aggregate of the entities described by dashed lines is usually called *skeleton* and has an execution which is independent of the *main thread* of the application
- thus, the *thread base* may be thought of as a thread which is instantiated and started when the remote reference is generated

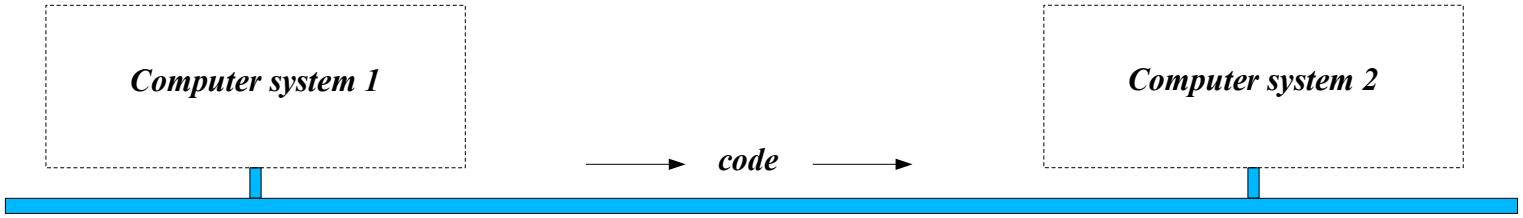
Code migration – 1

Code migration is a highly desired feature in a distributed system which enables some components of an application to be transferred from one processing node of the underlying parallel computer system to another during program execution.

Several reasons may lead to this

- *taking advantage of the computation power of specific processing nodes*: not all the nodes of a parallel computer system may be similar, therefore, parts of the computations to be carried out may be moved to specific nodes, on demand and in a dynamic way, to have the program run more efficiently
- *fault tolerance*: during execution, some of the processing nodes may fail, so it would be important, if one wants not to crash the application being executed, to be able to detect the malfunction and perform a dynamic reconfiguration of the software components previously assigned to the faulty nodes to new ones.

Code migration – 2



An issue one has to deal with is what form the code to be moved will take.

Several options are available

- *executable code*: this is the simpler form, but it requires that the original and the destination nodes are relatively similar so that the code will run without further processing
- *source code*: this is the most general form, no assumptions need to be made about the similarities of the original and the destination nodes
- *intermediate code*: this is typical of situations where the code to be moved is run through an interpreter.

Code migration – 3

Code migration, although being quite a desirable feature to have, has *security* problems in a general context. That is, one has to ensure that the code received at the destination node, being any, will not put at risk the integrity of the resources of the computer system where the node is located.

The way one usually deals with the issue, is to introduce a component in the code that manages the migration which will continuously monitor the access to the resources provided by the operating system, deciding on a case based manner whether the accesses should be allowed or denied.

This component is sometimes called the *security manager*.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 5: *Distributed objects and Remote invocation*
    Sections 5.1 to 5.3 and 5.5
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 10: *Distributed object-based systems*
    Sections 10.1 to 10.3.4

Sistemas Distribuídos

Interprocess Communication and Synchronization

Remote Objects - 2

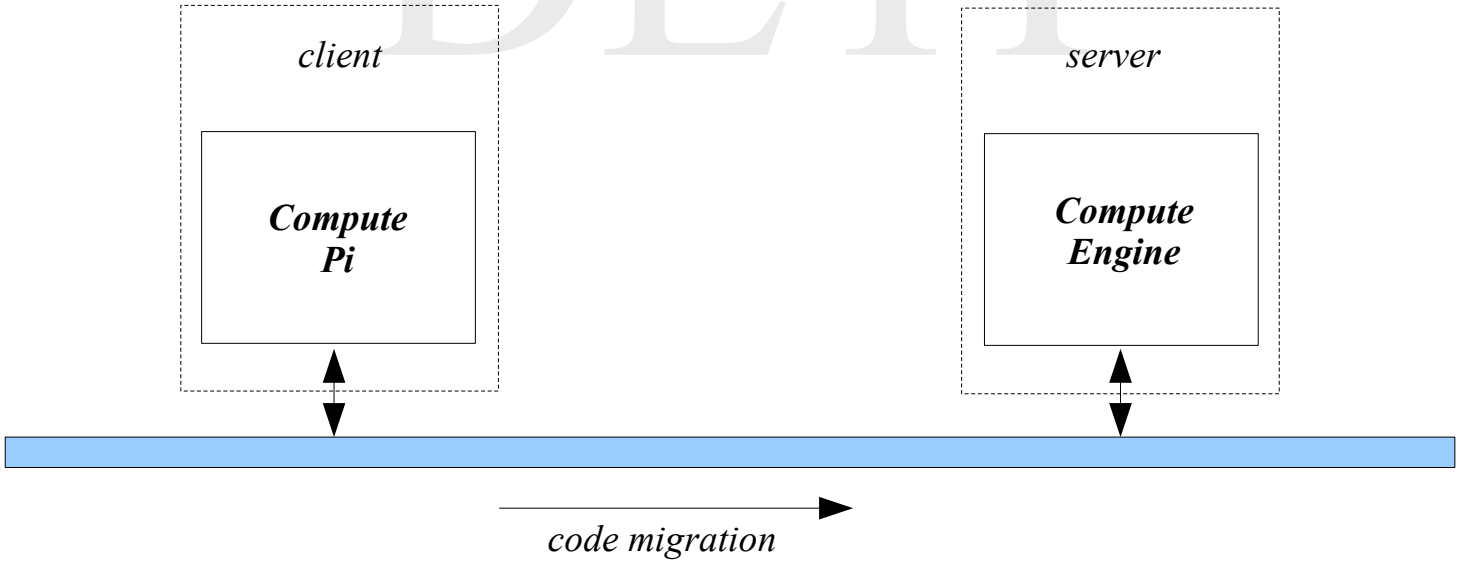
António Rui Borges

Summary

- *Characterization of the problem*
- *Code description*
- *Interaction diagrams*
- *Organization of the package BackEngine*
- *Local security policy*
- *Build and deploy*
- *Running the application*
- *Suggested reading*

Characterization of the problem - 1

- it is an example, adapted from the tutorial of Sun about *RMI*, to illustrate how code can be transferred between Java virtual machines, located on the same or on different hardware platforms, and be run in a JVM other than the one where it was formerly stored

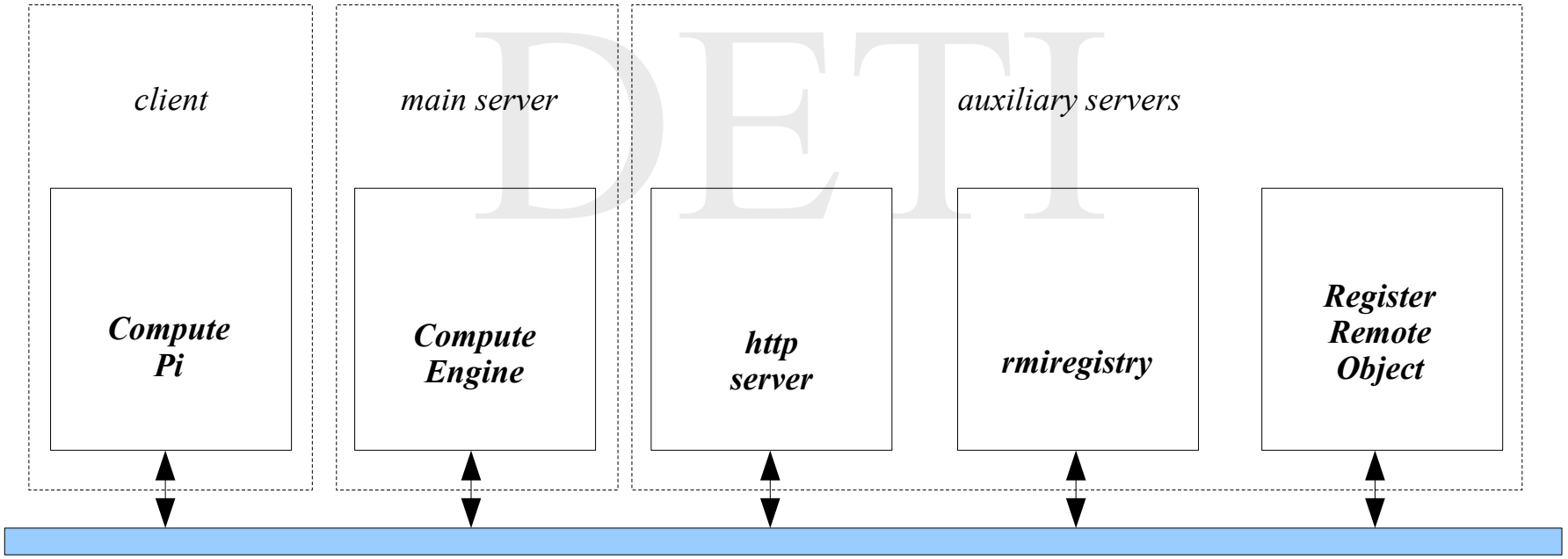


Characterization of the problem - 2

- two main classes of entities are considered
  - a *server object*, of type `ComputeEngine`, which provides a service for local execution of migrated code under remote control
  - several *client objects* to handle the local execution of migrated code under remote control (interfaces `Compute` and `Task`); in the present case, another object, of type `ComputePi`, transfers an object of type `Pi` (computation of π with a variable number of decimals) for remote execution
- and three auxiliary entities
  - a naming service for registering remote objects (`rmiregistry`)
  - a *server object*, of type `RegisterRemoteObject`, which provides support to register remote objects (those located in JVMs residing in hardware platforms other than the one the naming service is located)
  - a *http server* to assist the dynamic downloading of the data types used in remote calls.

Characterization of the problem - 3

Operational setup



Register Remote Object base thread - 1

```
public class ServerRegisterRemoteObject
{
   public static void main(String[] args)
   {
    /* get location of the registering service */

     String rmiRegHostName;
     int rmiRegPortNumb;

     GenericIO.writeString ("Name of the processing node where the registering service is located? ");
     rmiRegHostName = GenericIO.readlnString ();
     GenericIO.writeString ("Port number where the registering service is listening to? ");
     rmiRegPortNumb = GenericIO.readlnInt ();

    /* create and install the security manager */

     if (System.getSecurityManager () == null)
        System.setSecurityManager (new SecurityManager ());
     GenericIO.writelnString ("Security manager was installed!");

    /* instantiate a registration remote object and generate a stub for it */

     RegisterRemoteObject regEngine = new RegisterRemoteObject (rmiRegHostName, rmiRegPortNumb);
     Register regEngineStub = null;
     int listeningPort = 22001;                          /* it should be set accordingly in each case */

     try
     { regEngineStub = (Register) UnicastRemoteObject.exportObject (regEngine, listeningPort);
     }
     catch (RemoteException e)
     { GenericIO.writelnString ("RegisterRemoteObject stub generation exception: " + e.getMessage ());
       System.exit (1);
     }
     GenericIO.writelnString ("Stub was generated!");
```

Register Remote Object base thread - 2

```
    /* register it with the local registry service */

     String nameEntry = "RegisterHandler";
     Registry registry = null;

     try
     { registry = LocateRegistry.getRegistry (rmiRegHostName, rmiRegPortNumb);
     }
     catch (RemoteException e)
     { GenericIO.writelnString ("RMI registry creation exception: " + e.getMessage ());
       System.exit (1);
     }
     GenericIO.writelnString ("RMI registry was created!");

     try
     { registry.rebind (nameEntry, regEngineStub);
     }
     catch (RemoteException e)
     { GenericIO.writelnString ("RegisterRemoteObject remote exception on registration: " +
                                e.getMessage ());
       System.exit (1);
     }
     GenericIO.writelnString ("RegisterRemoteObject object was registered!");
   }
}
```

Register Remote Object and related interfaces - 1

```
public interface Register extends Remote
{
   public void bind (String name, Remote ref) throws RemoteException, AlreadyBoundException;

   public void unbind (String name) throws RemoteException, NotBoundException;

   public void rebind (String name, Remote ref) throws RemoteException;
}
```

Register Remote Object and related interfaces - 2

```
public class RegisterRemoteObject implements Register
{
   private String rmiRegHostName;
   private int rmiRegPortNumb = 1099;

   public RegisterRemoteObject (String rmiRegHostName, int rmiRegPortNumb)
   {
     if ((rmiRegHostName == null) || ("".equals (rmiRegHostName)))
        throw new NullPointerException ("RegisterRemoteObject: null pointer parameter on instantiation!");
     this.rmiRegHostName = rmiRegHostName;
     if ((rmiRegPortNumb >= 4000) && (rmiRegPortNumb <= 65535))
        this.rmiRegPortNumb = rmiRegPortNumb;
   }

   public void bind (String name, Remote ref) throws RemoteException, AlreadyBoundException
   {
     Registry registry;

     if ((name == null) || (ref == null))
        throw new NullPointerException ("RegisterRemoteObject: null pointer parameter(s) on on bind!");
     registry = LocateRegistry.getRegistry (rmiRegHostName, rmiRegPortNumb);
     registry.bind (name, ref);
   }
```

Register Remote Object and related interfaces - 3

```
    public void unbind (String name) throws RemoteException, NotBoundException
    {
      Registry registry;

      if ((name == null))
         throw new NullPointerException ("RegisterRemoteObject: null pointer parameter(s) on unbind!");
      registry = LocateRegistry.getRegistry (rmiRegHostName, rmiRegPortNumb);
      registry.unbind (name);
    }
    public void rebind (String name, Remote ref) throws RemoteException
    {
      Registry registry;

      if ((name == null) || (ref == null))
         throw new NullPointerException ("RegisterRemoteObject: null pointer parameter(s) on rebind!");
      registry = LocateRegistry.getRegistry (rmiRegHostName, rmiRegPortNumb);
      registry.rebind (name, ref);
    }
}
```

Compute Engine base thread - 1

```
public class ServerComputeEngine
{
   public static void main(String[] args)
   {
    /* get location of the registry service */

     String rmiRegHostName;
     int rmiRegPortNumb;

     GenericIO.writeString ("Name of the processing node where the registering service is located? ");
     rmiRegHostName = GenericIO.readlnString ();
     GenericIO.writeString ("Port number where the registering service is listening to? ");
     rmiRegPortNumb = GenericIO.readlnInt ();

    /* create and install the security manager */

     if (System.getSecurityManager () == null)
        System.setSecurityManager (new SecurityManager ());
     GenericIO.writelnString ("Security manager was installed!");

    /* instantiate a remote object that runs mobile code and generate a stub for it */

     ComputeEngine engine = new ComputeEngine ();
     Compute engineStub = null;
     int listeningPort = 22002;                          /* it should be set accordingly in each case */

     try
     { engineStub = (Compute) UnicastRemoteObject.exportObject (engine, listeningPort);
     }
     catch (RemoteException e)
     { GenericIO.writelnString ("ComputeEngine stub generation exception: " + e.getMessage ());
       e.printStackTrace ();
       System.exit (1);
     }
     GenericIO.writelnString ("Stub was generated!")
```

Compute Engine base thread - 2

```
/* register it with the general registry service */

 String nameEntryBase = "RegisterHandler";
 String nameEntryObject = "Compute";
 Registry registry = null;
 Register reg = null;

 try
 { registry = LocateRegistry.getRegistry (rmiRegHostName, rmiRegPortNumb);
 }
 catch (RemoteException e)
 { GenericIO.writelnString ("RMI registry creation exception: " + e.getMessage ());
   e.printStackTrace ();
   System.exit (1);
 }
 GenericIO.writelnString ("RMI registry was created!");

 try
 { reg = (Register) registry.lookup (nameEntryBase);
 }
 catch (RemoteException e)
 { GenericIO.writelnString ("RegisterRemoteObject lookup exception: " + e.getMessage ());
   e.printStackTrace ();
   System.exit (1);
 }
 catch (NotBoundException e)
 { GenericIO.writelnString ("RegisterRemoteObject not bound exception: " + e.getMessage ());
   e.printStackTrace ();
   System.exit (1);
 }
```

Compute Engine base thread - 3

```
      try
      { reg.bind (nameEntryObject, engineStub);
      }
      catch (RemoteException e)
      { GenericIO.writelnString ("ComputeEngine registration exception: " + e.getMessage ());
        e.printStackTrace ();
        System.exit (1);
      }
      catch (AlreadyBoundException e)
      { GenericIO.writelnString ("ComputeEngine already bound exception: " + e.getMessage ());
        e.printStackTrace ();
        System.exit (1);
      }
      GenericIO.writelnString ("ComputeEngine object was registered!");
  }
}
```

Compute Engine remote object and related interfaces

```
public interface Compute extends Remote
{
   Object executeTask (Task t) throws RemoteException;
}

public interface Task extends Serializable
{
   public static final long serialVersionUID = 2021L;

   Object execute();
}

public class ComputeEngine implements Compute
{
   public Object executeTask (Task t)
   {
     return t.execute ();
   }
}
```

Compute Pi thread base - 1

```
public class ComputePi
{
   public static void main(String args[])
   {
    /* get location of the generic registry service */

     String rmiRegHostName;
     int rmiRegPortNumb;

     GenericIO.writeString ("Name of the processing node where the registering service is located? ");
     rmiRegHostName = GenericIO.readlnString ();
     GenericIO.writeString ("Port number where the registering service is listening to? ");
     rmiRegPortNumb = GenericIO.readlnInt ();

    /* look for the remote object by name in the remote host registry */

     String nameEntry = "Compute";
     Compute comp = null;
     Registry registry = null;

     try
     { registry = LocateRegistry.getRegistry (rmiRegHostName, rmiRegPortNumb);
     }
     catch (RemoteException e)
     { GenericIO.writelnString ("RMI registry creation exception: " + e.getMessage ());
       e.printStackTrace ();
       System.exit (1);
     }
```

Compute Pi thread base - 2

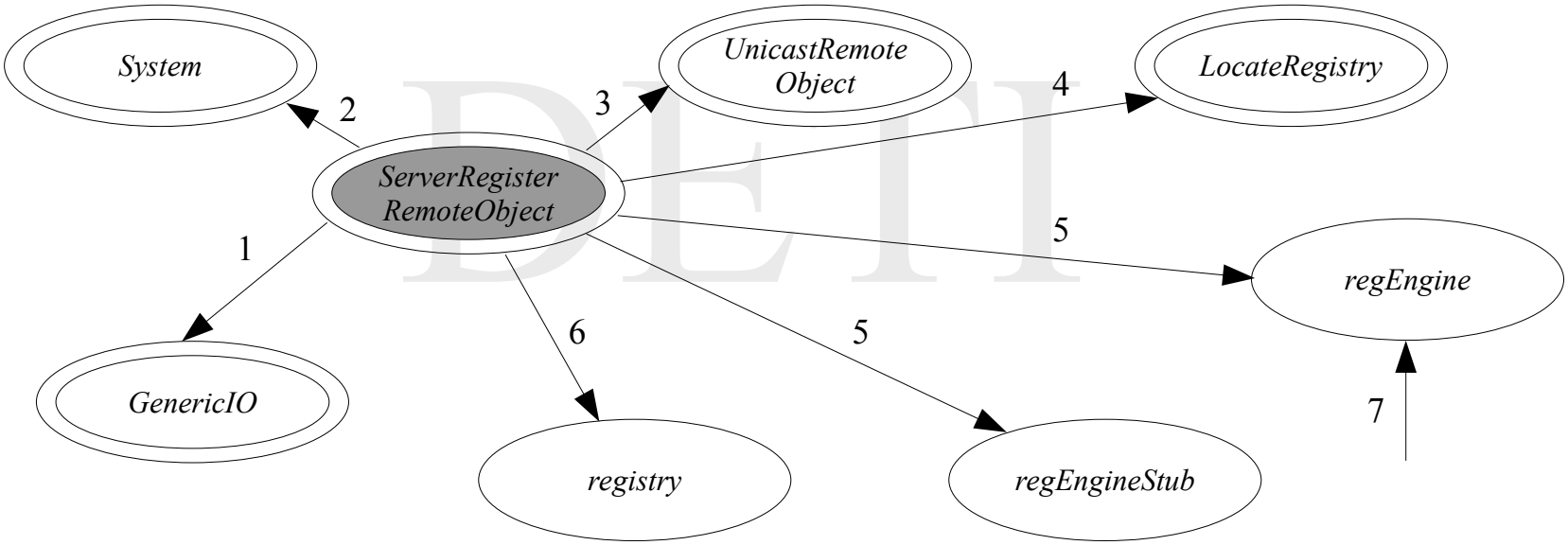
```
try
{ comp = (Compute) registry.lookup (nameEntry);
}
catch (RemoteException e)
{ GenericIO.writelnString ("ComputePi look up exception: " + e.getMessage ());
  e.printStackTrace ();
  System.exit (1);
}
catch (NotBoundException e)
{ GenericIO.writelnString ("ComputePi not bound exception: " + e.getMessage ());
  e.printStackTrace ();
  System.exit (1);
}

/* instantiate the mobile code object to be run remotely */

Pi task = null;
BigDecimal pi = null;

try
{ task = new Pi (Integer.parseInt (args[0]));
}
catch (NumberFormatException e)
{ GenericIO.writelnString ("Pi instantiation exception: " + e.getMessage ());
  e.printStackTrace ();
  System.exit (1);
}
```

Compute Pi thread base - 3

```
    /* invoke the remote method (run the code at a ComputeEngine remote object) */

     try
     { pi = (BigDecimal) (comp.executeTask (task));
     }
     catch (RemoteException e)
     { GenericIO.writelnString ("ComputePi remote invocation exception: " + e.getMessage ());
       e.printStackTrace ();
       System.exit (1);
     }

     /* print the result */
     GenericIO.writelnString (pi.toString ());
  }
}
```

Interaction diagrams - 1

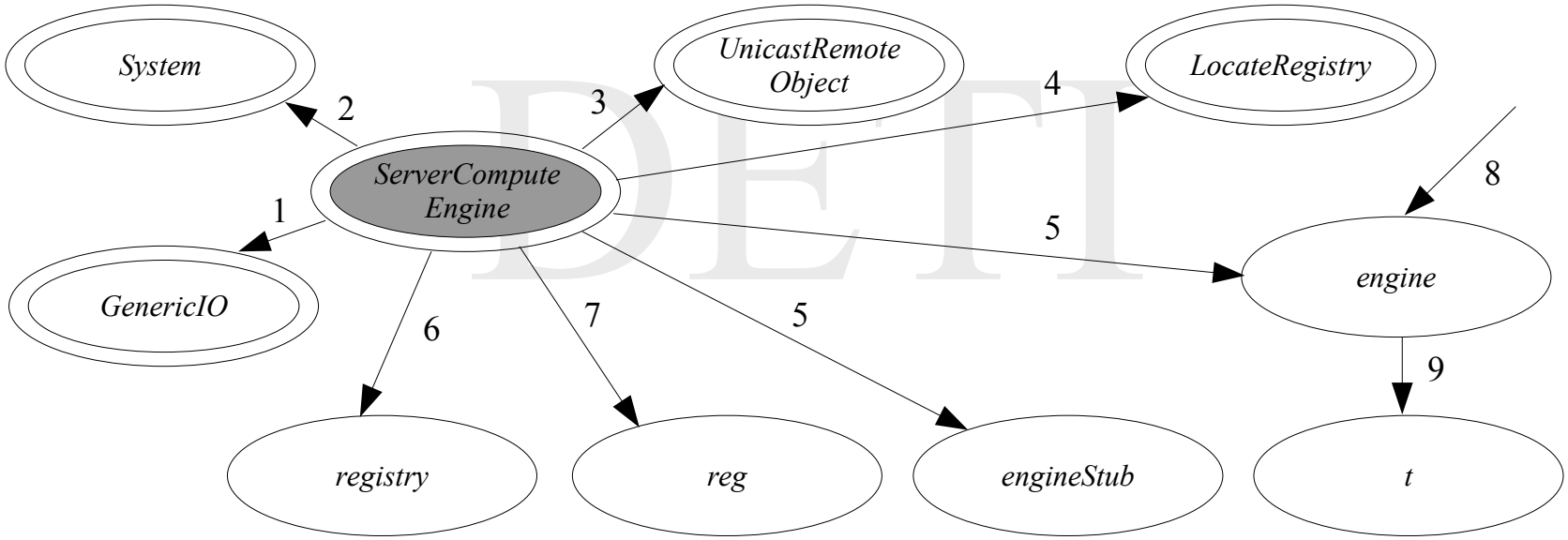
Remote object to support the registering of remote objects



1 – readlnInt, readlnString, writeString, writelnString
2 – getSecurityManager, setSecurityManager
3 – exportObject
4 – getRegistry
5 – instantiate
6 – instantiate, rebind
7 – bind, unbind, rebind

Interaction diagrams - 2

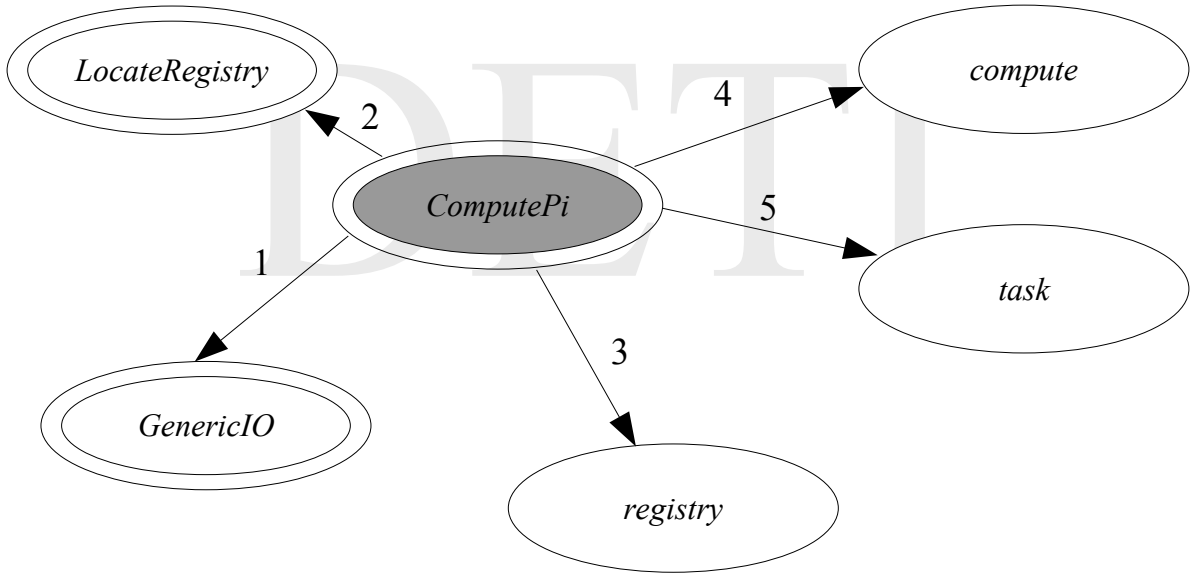
Remote object for local execution of migrated code under remote control



1 – readlnInt, readlnString, writeString, writelnString
2 – getSecurityManager, setSecurityManager
3 – exportObject
4 – getRegistry
5 – instantiate
6 – instantiate, locate
7 – instantiate, bind
8 – executeTask
9 – execute

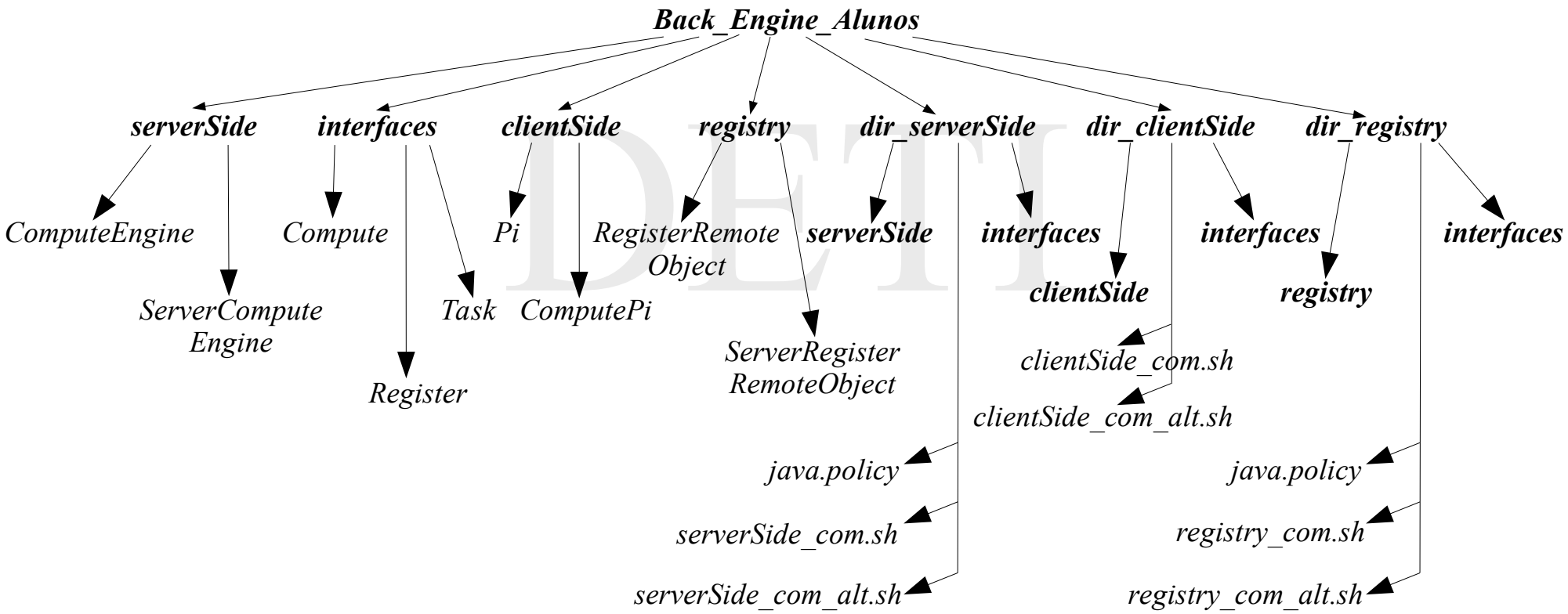
Interaction diagrams - 3

Client side



1 – readlnInt, readlnString, writeString, writelnString
2 – getRegistry
3 – instantiate, lookup
4 – instantiate, executeTask
5 – instantiate

Organization of the package BackEngine - 1



Organization of the package BackEngine - 2

Region for code development of the application

serverSide *– directory with the main server code*

- ***ComputeEngine***[.java] *– remote object which provides local execution of migrated code under remote control*
- ***ServerComputeEngine***[.java] *– instantiation and registering of the remote object* (*service provided*)

interfaces *– directory with the interfaces to the remote objects*

- ***Compute***[.java] *– interface for access to the object that provides local execution of migrated code under remote control*
- ***Task***[.java] *– interface for local execution of migrated code*
- ***Register***[.java] *– interface for access to the object that provides support for registering remote objects*

clientSide *– directory the client code*

- ***Pi***[.java] *– code to be migrated and that it is going to be executed remotely under local control*
- ***ComputePi***[.java] *– access to the remote object which provides local execution of migrated code under remote control*

registry *– directory with the auxiliary server code to support the registering of remote objects*

- ***RegisterRemoteObject***[.java] *– remote object to support the registering of remote objects*
- ***ServerRegisterRemoteObject***[.java] *– instantiation and registering of the remote object* (*service provided*)

Organization of the package BackEngine - 3

Region for deployment of the application

dir_serverSide *– directory for running the main server code*

- ***serverSide*** *– contains* ***ServerComputeEngine***[.class] *and* ***ComputeEngine***[.class]
- ***interfaces*** *– contains* ***Register***[.class], ***Compute***[.class] and ***Task***[.class]
- ***java.policy*** *– file specifying the local security policy*
- ***serverSide_com.sh*** *– shell script for the running code* (*data types are located with help of the http server*)
- ***serverSide_com_alt.sh*** *– shell script for the running code* (*data types are located through the file system*)

dir_clientSide *– directory for running the client code*

- ***clientSide*** *– contains* ***ComputePi***[.class] *and* ***Pi***[.class]
- ***interfaces*** *–contains* ***Compute***[.class] *and* ***Task***[.class]
- ***clientSide_com.sh*** *– shell script for the running code* (*data types are located with help of the http server*)
- ***clientSide_com_alt.sh*** *–shell script for the running code* (*data types are located through the file system*)

dir_registry *– directory for running the auxiliary server code*

- ***registry*** *– contains* ***ServerRegisterRemoteObject***[.class] *and* ***RegisterRemoteObject***[.class]
- ***interfaces*** *– contains* ***Register***[.class]
- ***java.policy*** *– file specifying the local security policy*
- ***registry_com.sh*** *– shell script for the running code* (*data types are located with help of the http server*)
- ***registry_com_alt.sh*** *– shell script for the running code* (*data types are located through the file system*)

Local security policy

- there is a single security rule in Java: *everything is forbidden, unless it explicitly allowed*


```
grant
{ permission java.util.PropertyPermission "user.dir", "read";
  permission java.net.SocketPermission "*:1024-65535",
                                "listen,resolve,connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
  permission java.io.FilePermission "/-", "read,write";
};
```

Local security policy

- there is a single security rule in Java: *everything is forbidden, unless it explicitly allowed*


```
grant
{ permission java.util.PropertyPermission "user.dir", "read";
  permission java.net.SocketPermission "*:1024-65535",
                                "listen,resolve,connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
  permission java.io.FilePermission "/-", "read,write";
};
```

Build and deploy - 1

- create a *shell* window
- position inside the directory `Back_Engine_Alunos`
- replace in `buildAndDeploy.sh` all instances of `ruib` by your login
- do the same in the files `set_rmiregistry.sh`, `clientSide_com_alt.sh`, `serverSide_com_alt.sh`, `serverSide_com.sh`, `registry_com_alt.sh`, `registry_com.sh`
- run the shell script `buildAndDeploy.sh`

```
[ruib@ruib-laptop Back_Engine_alunos]$ pwd
/home/ruib/Teaching/SD/exemplos demonstrativos/BackEngine/Back_Engine_alunos
[ruib@ruib-laptop Back_Engine_Alunos]$ ./buildAndDeploy.sh
Compiling source code.
Distributing intermediate code to the different execution environments.
Compressing execution environments.
Deploying and decompressing execution environments.
[ruib@ruib-laptop Back_Engine_alunos]$
```

Build and deploy - 2

```
[ruib@ruib-laptop Back_Engine_alunos]$ cat buildAndDeploy.sh
echo "Compiling source code."
javac interfaces/*.java registry/*.java serverSide/*.java clientSide/*.java
echo "Distributing intermediate code to the different execution environments."
cp interfaces/Register.class dir_registry/interfaces/
cp registry/*.class dir_registry/registry/
cp interfaces/*.class dir_serverSide/interfaces/
cp serverSide/*.class dir_serverSide/serverSide/
cp interfaces/Compute.class interfaces/Task.class dir_clientSide/interfaces/
cp clientSide/*.class dir_clientSide/clientSide/
mkdir -p /home/ruib/Public/classes
mkdir -p /home/ruib/Public/classes/interfaces
mkdir -p /home/ruib/Public/classes/clientSide
cp interfaces/*.class /home/ruib/Public/classes/interfaces/
cp clientSide/Pi.class /home/ruib/Public/classes/clientSide/
echo "Compressing execution environments."
rm -f dir_registry.zip dir_serverSide.zip dir_clientSide.zip
zip -rq dir_registry.zip dir_registry
zip -rq dir_serverSide.zip dir_serverSide
zip -rq dir_clientSide.zip dir_clientSide
echo "Deploying and decompressing execution environments."
cp set_rmiregistry_alt.sh /home/ruib
cp set_rmiregistry.sh /home/ruib
mkdir -p /home/ruib/test/BackEngine
rm -rf /home/ruib/test/BackEngine/*
cp dir_registry.zip dir_serverSide.zip dir_clientSide.zip /home/ruib/test/BackEngine
cd /home/ruib/test/BackEngine
unzip -q dir_registry.zip
unzip -q dir_serverSide.zip
unzip -q dir_clientSide.zip
[ruib@ruib-laptop Back_Engine_alunos]$
```

Running the application

- a *http* server is <u>not</u> required to run the version where the data types are located through the file system
- four shell windows are required
  - window 1: base directory, execute `set_rmiregistry[_alt].sh`
  - window 2: `dir_registry`, execute `registry_com[_alt].sh`
  - window 3: `dir_serverSide`, execute `serverSide_com[_alt].sh`
  - window 4: `dir_clientSide`, execute `clientSide_com[_alt].sh`

Suggested reading

- *On-line* support documentation for Java program developing environment by Oracle (Java Platform Standard Edition 8)

Sistemas Distribuídos

Synchronization

António Rui Borges

Summary

- *Time concepts*
  - *Global time*
  - *Local time*
  - *Logical time*
- *Adjustment of local time*
  - *Problem characterization*
  - *Cristian method*
  - *Berkeley algorithm*
  - *Network Time Protocol*
- *Logical clocks*
  - *Scalar logical clock*
    - *Total ordering of events*
  - *Vector logical clocks*
- *Suggested reading*

Time concepts

Physical phenomena, and in particular human activity, take place in space and time. *Time* is the key element on event characterization, on their ordering and on determining possible causal relations that may exist between pairs, or groups, of events.

Time itself is thought of in multiple ways vis a vis how the *observer* faces reality

- *global time* – time as perceived by an external observer
- *local time* – time as perceived by each of the observed entities
- *logical time* – time as perceived by the flow of information.

Global time - 1

Time can not be directly measured.

Its measurement is done by observing the periodical motion of well-defined objects, taking advantage of the intimate connection of *time* with *space*

- *astronomical time* – based on the motion of astral bodies in heavens: specifically, the circular-like motion of Earth around the Sun (each cycle represents a *solar year*) and the Earth rotating motion (each cycle represents a *solar day*); the *solar day* is successively divided into hours, minutes and seconds (one *day* comprehends 24 hours, one hour 60 minutes and one minute 60 seconds); the *solar second*, while standard unit for time measurement, is defined as the fraction of 1/86400 of the solar day; and the *solar year* as being approximately 365 days and 6 hours

Global time - 2

- *atomic time* – the periodic motion of astral bodies is not constant enough to be used as standard when one considers very long time intervals; research carried out in the past decades has shown that the Earth rotating motion has become slower as time goes by due to the friction produced by the tides and the dragging effect of the atmosphere; furthermore, the Earth rotating motion is prone to small oscillations in its angular speed due to, is thought, turbulence at the core; for all these reasons, the *standard second* was redefined when atomic clocks came into being; present day definition states the *second* to be equal to *the duration of 9 192 631 770 periods of the radiation corresponding to the transition between two hyperfine levels of the state of minimum energy of the atom cesium 133, at rest and at the temperature of 0 K*.

Global time - 3

International Atomic Time (TAI) represents the practical standard computed from the weighted average of time measured by more than 200 atomic clocks located in national institutions around the world. This task is coordinated by the International Institute of Weights and Measures (BIPM), through its International Bureau of the Hour (BIH).

The duration of the *standard second* was established in order to ensure its coincidence with the *solar second* at the time of its introduction, January 1, 1958.

In January 1, 1977, a correction was introduced to the convention, based on general relativity, to minimize the time dilation effect produced by variations of the Earth gravitational field (the atomic clocks, used to compute the average, are located at different altitudes and, thus, tick at slightly different rates).

Global time - 4

Coordinated Universal Time (UTC), based on the *international atomic time*, constitutes the main standard that sets the time to human activities. The system is based on *standard seconds* and is kept as close as possible to the *astronomical time* by adding or subtracting (the latter has not happen yet) individual seconds to compensate for the slowing down, and occasional irregularities, of the Earth rotating motion.

In the UTC time scale, the *second* and its submultiples are always constant; its multiples, *days*, *hours* and *minutes*, however, are not. From time to time, in order to keep it in pace with the *astronomical time*, a second is added or subtracted to the last minute of a day, typically in the last day of June or December.

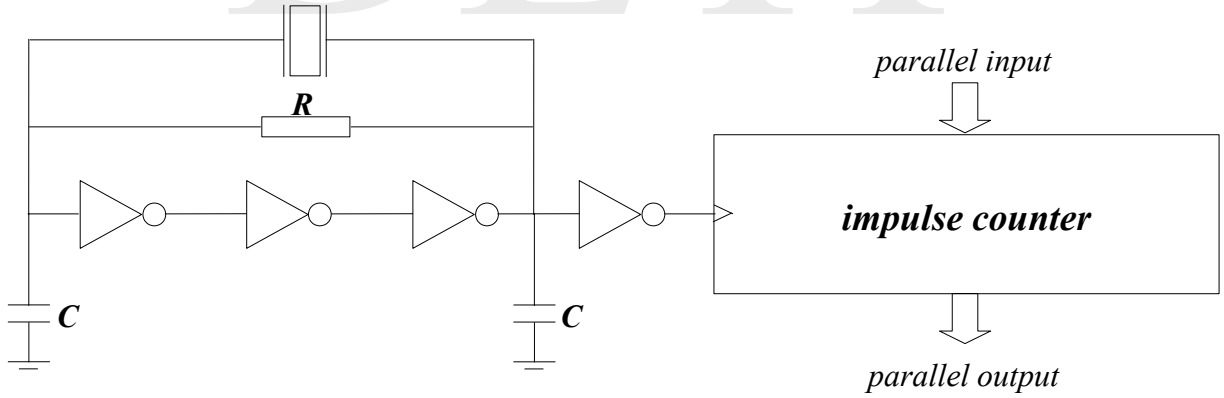
UTC time is made available from multiple sources, with different uncertainties. Among the most important, are

- short wave radio emitters: ±10 ms
- geostationary satellite systems (GEOS, GPS): ±0,5 ms
- internet (NTP): ±50 ms.

Local time – 1

The *clock* of a computer system consists of two elements: an oscillator circuit, controlled by a quartz crystal, and an impulse counter. The counting value at a given instant may be obtained by reading the *parallel output* and the counter may be set to a given state by feeding the *parallel input*.

The counting can be converted to a *time* provided an origin is defined (the convention in Unix, for instance, matches the origin to 0h 0m 0s of January 1, 1970).

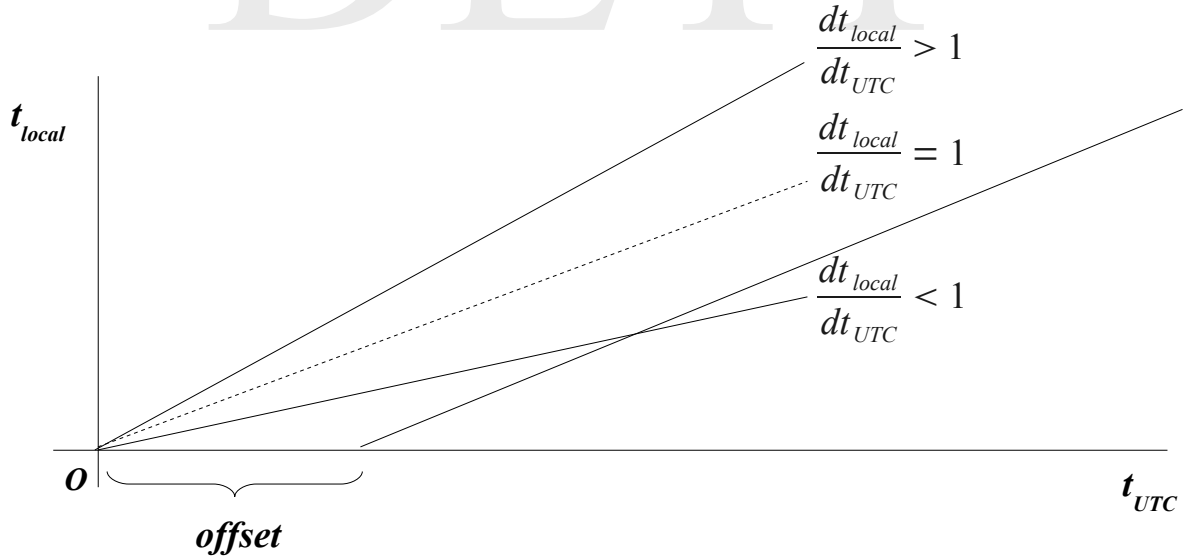


Local time – 2

The *clock* may display an incorrect time due to two different factors

- *offset* – the counting value differs from the correct value by a fixed number of impulses (error on defining the origin)
- *drift* – the oscillator frequency moves from the nominal one, changing its value erratically as time goes by due to the environment conditions, such as temperature and humidity (error on the counting rate).

Typically, an oscillator, controlled by a quartz crystal, has a *drift* of the order of magnitude of one part in 10^6 per second.



Time adjustment – 1

The problem of synchronizing the local clocks of the computer systems that make up the processing nodes of a parallel machine, can be thought of in two different ways

- *external synchronization* – given a known UTC source, $S(t)$, and a maximum interval of tolerated uncertainty, Δ, among the local clocks $Ck_i(t)$, with $i = 0,1,\ldots,$ N-1, and the UTC source $S(t)$, ensure for all time t that

$$\forall_{i \in \{0,1,\cdots,N-1\}} \; |S(t) - Ck_i(t)| < \Delta$$

- *internal synchronization* – given a maximum interval of tolerated uncertainty, Δ, among the local clocks $Ck_i(t)$, with $i = 0,1,\ldots,$ N-1, ensure for all time t that

$$\forall_{i,j \in \{0,1,\cdots,N-1\}} \; |Ck_i(t) - Ck_j(t)| < \Delta \quad .$$

Time adjustment – 2

One assumes that the processing nodes of the parallel machine are connected by some interconnection topology and that communication among them is carried out through message passing, with a finite transmission time, but without an upper limit, that is,

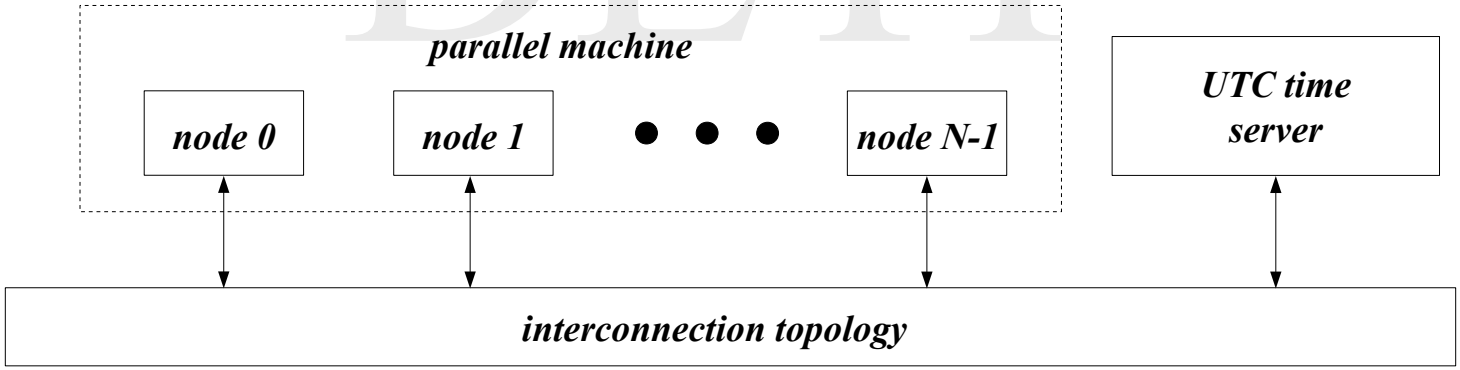
$$\forall_{L \in \mathbb{R}^+} \ \exists_{t_M} \ t_M > L$$

where t_M represents the message transmission time.

The adjustment itself should be made in a manner that the monotonicity of the local time is alwaays preserved. (***Why***?)

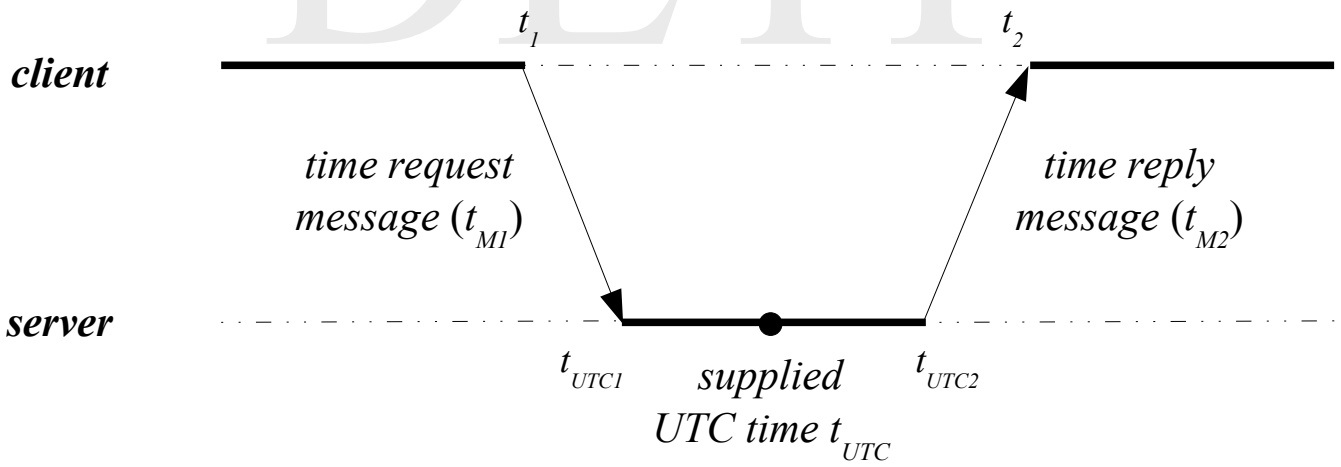
Cristian method – 1

It is an external synchronization method where one assumes the availability of a UTC time server. From time to time, in a proactive manner, in order to adjust its own local clock, each node (acting as a client) addresses the server through the sending of a message where the *right* time is requested. The server replies by sending it in a predefined format.



Cristian method – 2

- operation decomposition
  - the request is made at local time t_1 through the sending of a message with transmission time t_{M1}, the message is received at server UTC time t_{UTC1}
  - the time t_{UTC}, which will be returned, is adjusted to match approximately the middle of the server processing interval
  - the reply is sent at UTC time t_{UTC2} through a message with transmission time t_{M2}, the message is received at local time t_2.



Cristian method – 3

- the client has at its disposal the times t_1 , t_2 and t_{UTC} to adjust the local clock
- the client assumes that the *drift* of the local clock produces a negligible variation of the time interval $t_2 - t_1$
- expressing the *offset* between UTC time and local time by $off = S(t) - Ck(t)$, the offset estimation is given by

$$off_{est} = t_{UTC} - \frac{t_1 + t_2}{2}$$

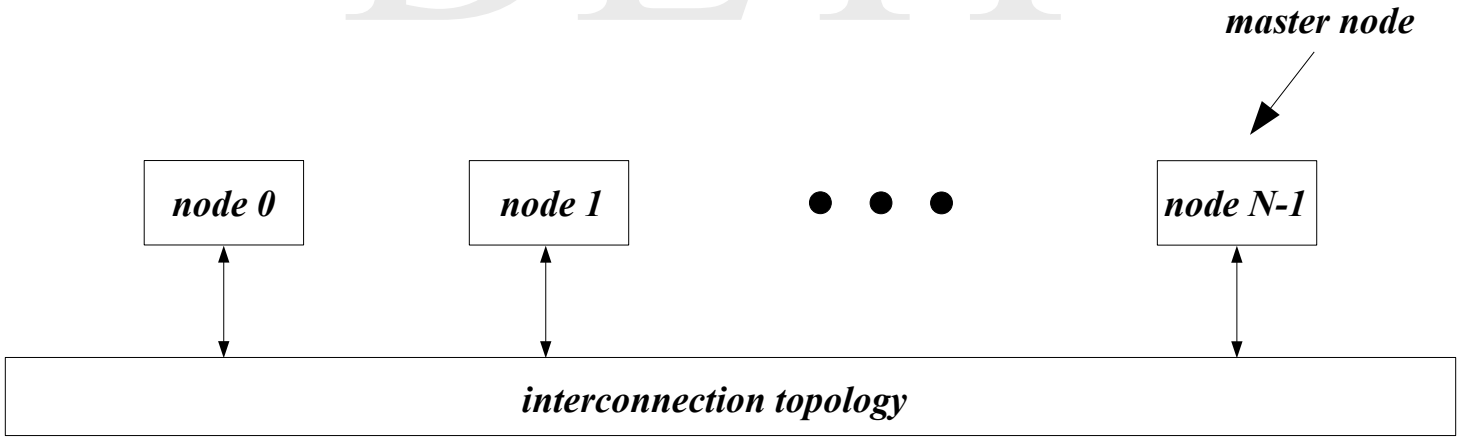
- the associated uncertainty to this value in the worst case, Δ_{est} , assumes that $t_2 - t_1 \approx t_{M1} + t_{MIN}$ (the processing time at the server is negligible when compared with message transmission time and one of them is transmitted in minimum time)

$$\Delta_{est} = \frac{t_2 - t_1}{2} - t_{MIN}$$

- if the estimated uncertainty is larger than the nominal accepted value, as in the case the communication channel endures random load variations or the server is very busy, the client may reject the *offset* estimation and try again later on.

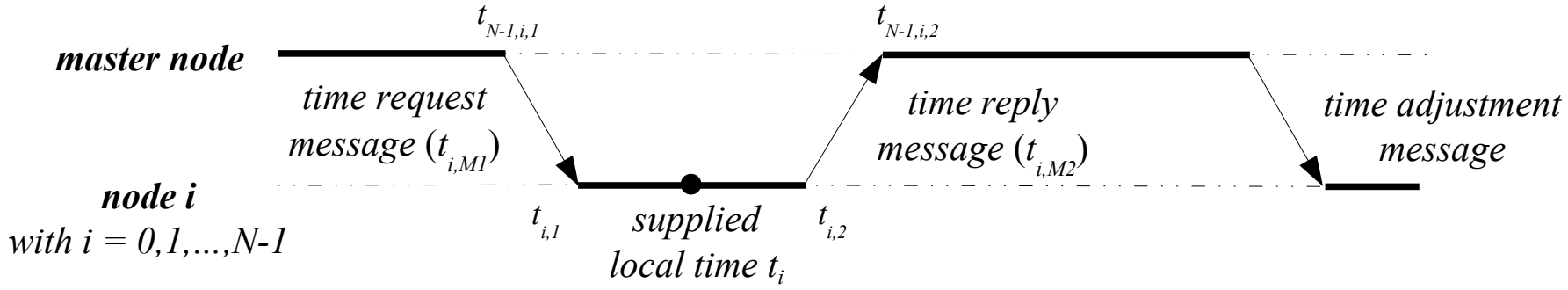
Berkeley algorithm – 1

It is an internal synchronization method to be applied when a UTC time server is not available. Thus, the sole goal is to keep synchronized among themselves the local clocks of the processing nodes of the parallel machine. From time to time, one of the nodes, called for this reason the *master* node, addresses in a proactive manner all the nodes of the parallel machine, itself included, for information about the local time. Then, it computes the average value of the offsets among the other local clocks and its own and sends to all the nodes, itself included, the correction that should be introduced to adjust the local time to a common value.



Berkeley algorithm – 2

- operation decomposition
  - the request is made by the master node at local time $t_{N-1,i,1}$ through the sending of a message to each node of the parallel machine with transmission time $t_{i,M1}$, a message is received by each node at local time $t_{i,1}$, with i = 0,1, … ,N-1
  - the time t_i, which will be sent by each node, is adjusted to match approximately the middle of the processing interval
  - the reply is sent by each node at local time $t_{i,2}$, with i = 0,1, … ,N-1, through a message with transmission time $t_{i,M2}$, the message is received by the master node at local time $t_{N-1,i,2}$
  - the master node computes the deviations to its local time and sends a message to all the nodes with the adjustment that must be introduced in each case.



Berkeley algorithm – 3

- the *master* node has at its disposal the times $t_{N-1,i,1}$, $t_{N-1,i,2}$ and t_i, with i = 0,1,…,N-1, to estimate the offset, and the associated uncertainty, of each of the local clocks
- the *master* node assumes that the *drift* of its local clock produces a negligible variation of the time intervals $t_{N-1,i,2} - t_{N-1,i,1}$, with i = 0,1,…,N-1
- the individual estimations are computed, according to the method of Cristian, by

$$off_{est}(i) = t_i - \frac{t_{N-1,i,1} + t_{N-1,i,2}}{2} \qquad \Delta_{est}(i) = \frac{t_{N-1,i,2} - t_{N-1,i,1}}{2} - t_{MIN}$$

com $i = 0,1,\cdots,N-1$

- the *master* node computes next the average of the estimations of the *offset*, $off_{est\ med}$, rejecting in the computations the cases where the estimated uncertainties are larger than the nominal accepted value
- the *master* node sends in the end to each of the nodes the adjustment which must be introduced in each case, $off_{est\ med} - off_{est}(i)$
- one should remark, since the adjustment is a differential value, the transmission time of this last message does not introduce any further uncertainty.

Network Time Protocol (NTP) - 1

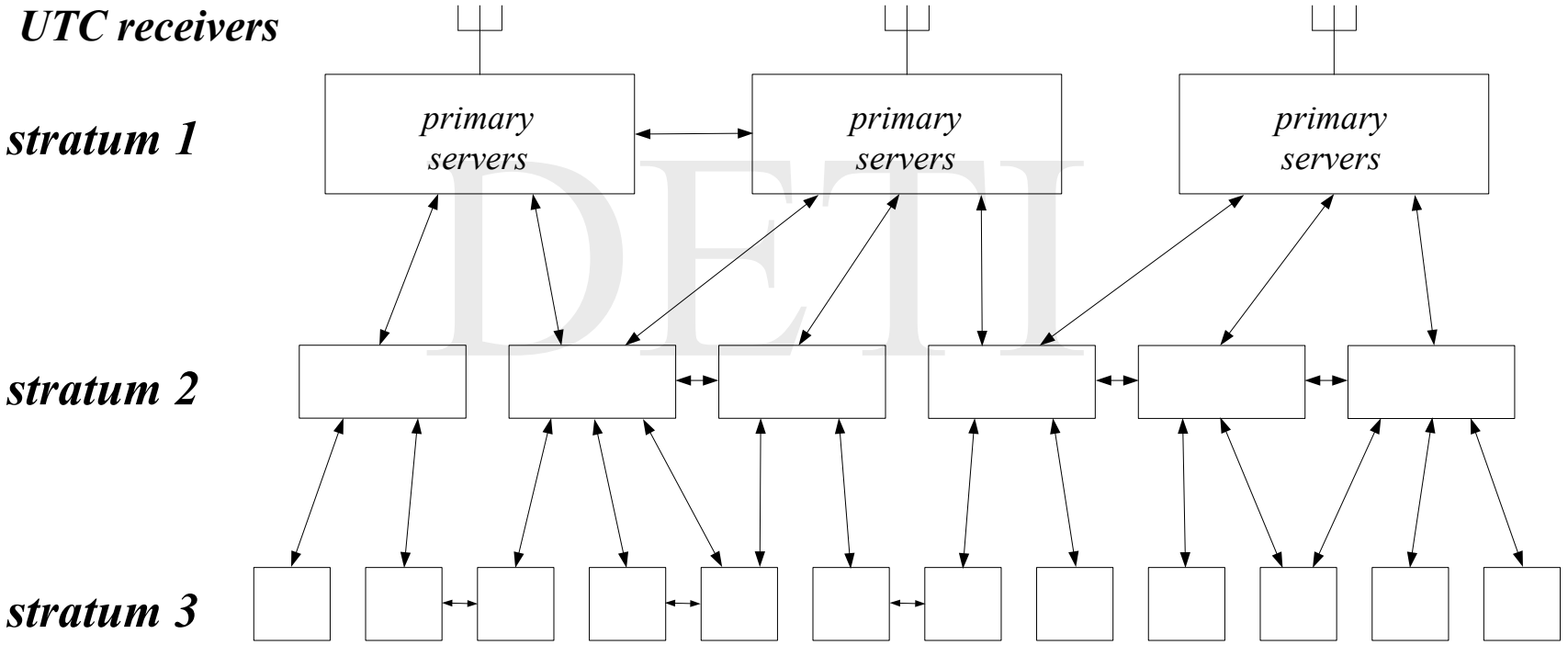
The methods just described aim the synchronization of the local clocks of computer systems integrated in local area networks. The *Network Time Protocol* (NTP), on the other hand, deals with the Internet itself.

The goal is

- to allow every computer system connected to the internet to adjust its local clock with reasonable accuracy and make the adjustment at a rate high enough to prevent serious time discrepancies due to *drift*
- to ensure that the provided service can outlive the more or less long losses of connectivity of specific servers, keeping a permanent availability
- to provide protection against particular interferences.

Network Time Protocol (NTP) – 2

Architecture of the service



Network Time Protocol (NTP) – 3

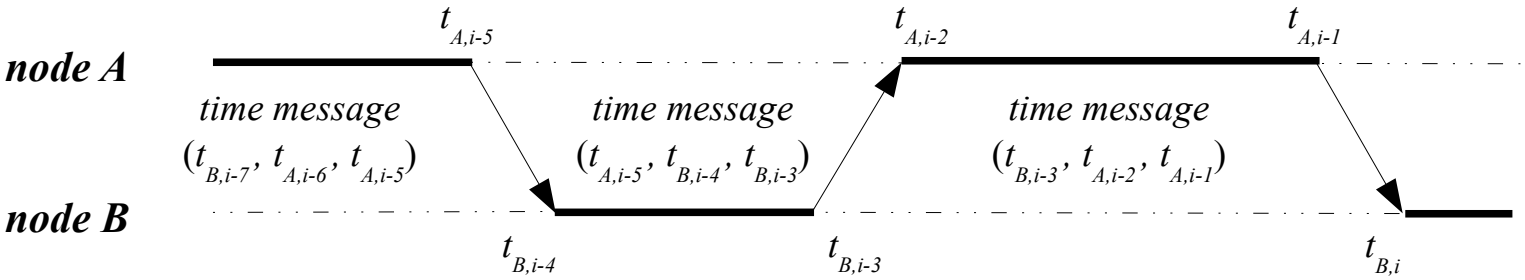
- the service assumes a hierarchical organization of computer systems in different levels, called *strata*
- the computer systems in the first level, *stratum* 1, are connected directly to a UTC source, so they are called *primary servers*
- the remaining computer systems, located in the other *strata*, are called *secondary servers* and adjust their local clocks with servers belonging to a *stratum* immediately above in the hierarchy
- the computer systems in each *stratum* can also coordinate its time information with other servers belonging to the same *stratum* to provide a globally more stable and robust information

Network Time Protocol (NTP) – 4

- as one goes down in the hiearchy, the degree of uncertainty of the produced time information increases because the errors introduced in each synchronization stage are cumulative
- any time adjustment sub-tree is dynamically reconfigured
  - whenever a *primary server* is unable to access its UTC source, it passes to the *stratum* 2, becoming a *secondary server*
  - whenever a server used to adjust the local clock of a computer system in a given *stratum* becomes unavailable, another server is looked for.

Network Time Protocol (NTP) – 5

- an exchange of time messages takes place continuously between pairs of nodes (A e B, for instance) for adjustment of the local clock of node B, if this belongs to a lower *stratum* in the hierarchy, or for mutual adjustment of the local clocks of both nodes, if they belong to the same *stratum*
  - each forward message includes three time stamps: the local times of transmission and reception of last received message, $t_{B,i-3}$ e $t_{A,i-2}$, and the local time of transmission of present message, $t_{A,i-1}$
  - upon message reception, the destination node saves the local reception time, $t_{B,i}$
  - the four times, $t_{B,i-3}$, $t_{A,i-2}$, $t_{A,i-1}$ e $t_{B,i}$, are then used to compute an estimation of the *offset* and its uncertainty.



Network Time Protocol (NTP) – 6

- the node B assumes that the *drifts* of its local clock and the local clock of node A produce negligible variations of the time intervals $t_{B,i} - t_{B,i-3}$ and $t_{A,i-1} - t_{A,i-2}$, respectively
- expressing the *offset* between the local times of both nodes as $off = Ck_A(t) - Ck_B(t)$, aits estimation is given by

$$off_{est} = \frac{t_{A,i-1} + t_{A,i-2}}{2} - \frac{t_{B,i} + t_{B,i-3}}{2}$$

- the associated uncertainty to this value in the worst case, Δ_{est}, assumes that the transmission time of one of two successive messages is minimum

$$\Delta_{est} = \frac{t_{A,i-2} - t_{B,i-3} + t_{B,i} - t_{A,i-1}}{2} - t_{MIN}$$

- the node B applies next an algorithm of statistical filtering to the successive pairs (off_{est}, Δ_{est}) which are computed to produce a final estimation
- the procedure is carried out with more than a server, the results are compared and may lead to a change of the server(s) being addressed.

Process synchronization

The variability introduced in the readings of the local clocks of the processing nodes of the parallel machine, by the time adjustment algorithms, makes impossible to use time information to synchronize the activities of the different processes which form a distributed application. The best one may hope to achieve is to generate a degree of uncertainty in the millisecond range between the time readings of pairs of local clocks, which means that about a million of instructions may be executed by each processor in a time interval of this magnitude.

On the other hand, Lamport (1978) has shown that if two processes residing in distinct nodes do not interact, it is not strictly necessary that their local clocks tick in pace: the mismatch is not observable and, therefore, conflicts will not arise. Thus, what really matters is not that all concerned processes agree on the value of *actual* time, but they order the relevant events that take place in the same manner, that is, those that are involved in some kind of interaction.

The exploitation of this concept leads to the so called *logic clocks* which only portray the flow of information that occurs.

Scalar logic clock - 1

One defines an *event* as any relevant activity which occurs during the execution of a process. Among all activities that take place, the communication activities are specially significant as far as process synchronization is concerned, that is, message *sending* and message *receiving*.

The ordering of events that occur in a distributed application, is based on two almost trivial observations

- when two events occur in the same process, they take place in the order perceived by the process
- when a message is exchanged between processes, the message *sending* event takes necessarily place *before* the *receiving* event of the same message.

Pairs of events are classified for ordering purposes as

- *sequential* – if it is possible to assert which one has occurred *before*
- *concurrent* – otherwise.

Scalar logic clock - 2

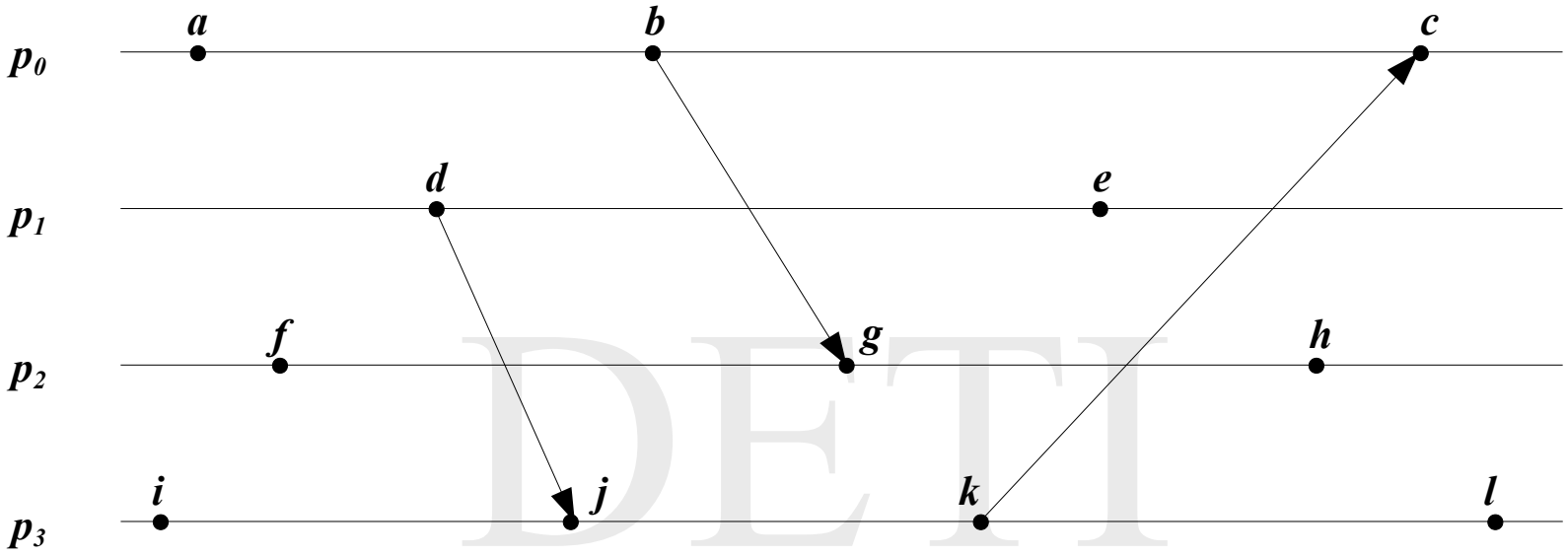
Lamport (1978) named *occurred before*, and denoted it by $\rightarrow$, the partial ordering of events which comes from generalizing the observations that were just made.

The relation *occurred before* can be formally defined in the following way

- $\exists_{p_i \in \{p_0, p_1, \cdots, p_{N-1}\}} \; e \rightarrow_i e' \Rightarrow e \rightarrow e'$
- $\forall_{message} \; \text{send}(m) \rightarrow \text{receive}(m)$
- $e \rightarrow e' \wedge e' \rightarrow e'' \Rightarrow e \rightarrow e''$

where p_i , with i = 0,1,…,N-1, are the processes that coexist in the distributed application and that are executed in distinct nodes of the parallel machine, and e, e' and e'' are generic events.

Scalar logic clock - 3



sequential events

$$f \rightarrow g \ \wedge \ g \rightarrow h \ \Rightarrow \ f \rightarrow h$$

$$d \rightarrow j \ \wedge \ j \rightarrow k \ \wedge \ k \rightarrow c \ \Rightarrow \ d \rightarrow c$$

concurrent events

$$\neg(f \rightarrow c) \ \wedge \ \neg(c \rightarrow f) \ \Rightarrow \ f \parallel c$$

$$\neg(i \rightarrow e) \ \wedge \ \neg(e \rightarrow i) \ \Rightarrow \ i \parallel e$$

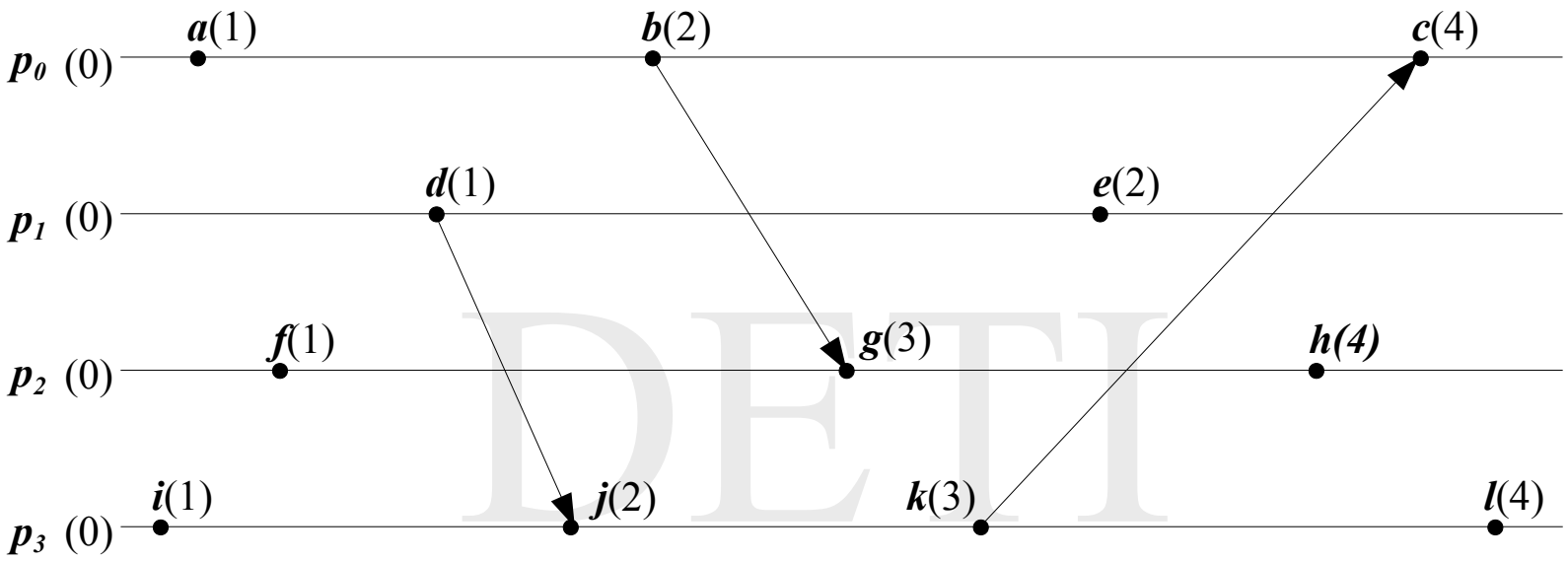
Scalar logic clock - 4

Lamport suggested a device, which he called *logic clock*, to explicit numerically the ordering resulting from the relation *occurred before*. A *scalar logic clock* is essentially a local counter of events, monotonically increasing, which has no direct association with the *actual* time.

In a distributed application, each process p_i, with $i = 0,1,\ldots,N-1$, uses a logical clock as its own local clock, Ck_i, and marks the events according to the following rules

- *initialization*: $Ck_i = 0$
- *occurrence of a locally relevant event*: updating of the local clock, $Ck_i = Ck_i + \alpha_i$, where α_i is a numeric constant usually equal to 1
- *message sending*: insertion of a time stamp *ts*, of value equal to Ck_i, in the message to be sent after the updating takes place
- *message reception*: adjustment of the local clock to the value *max* (Ck_i , *ts*) before updating its value with the reception event.

Scalar logic clock - 5



It is straightforward to prove by mathematical induction that

$$e \rightarrow e' \;\Rightarrow\; Ck_i(e) < Ck_j(e') \quad \text{, with } i, j = 0,1, \cdots, N-1 \;\; .$$

The converse, however, is not true.

Total ordering of groups of events - 1

Lamport has shown, however, that groups of related events may be subjected to an operation of *total ordering* and be perceived in the same order by processes residing in different nodes of a parallel machine, if logic clocks of the type he has prescribed are used in the generation of time stamps included in the exchanged messages.

Let e_j, with $j = 0,1,\ldots,K\text{-}1$, be events associated with the exchange of messages m_j among the processes p_i, with $i = 0,1,\ldots,N\text{-}1$, then the events e_j can be *totally ordered* if and only if one can establish a one-to-one correspondence between each event and a point in the numeric [straight] line through an associated property (the time stamp inserted in each message).

Total ordering of groups of events - 2

Since nothing prevents the time stamps associated with two distinct messages to be equal, $ts(m_p) = ts(m_q)$, with $p, q = 0,1,\ldots,K\text{-}1$ and $p \neq q$, Lamport defined a data structure he called *extended time stamp*, $(ts(m_p),id(m_p))$, which consists of the ordered pair formed by the message time stamp and the sender identification, and prescribed the following rule to order the *time stamps*

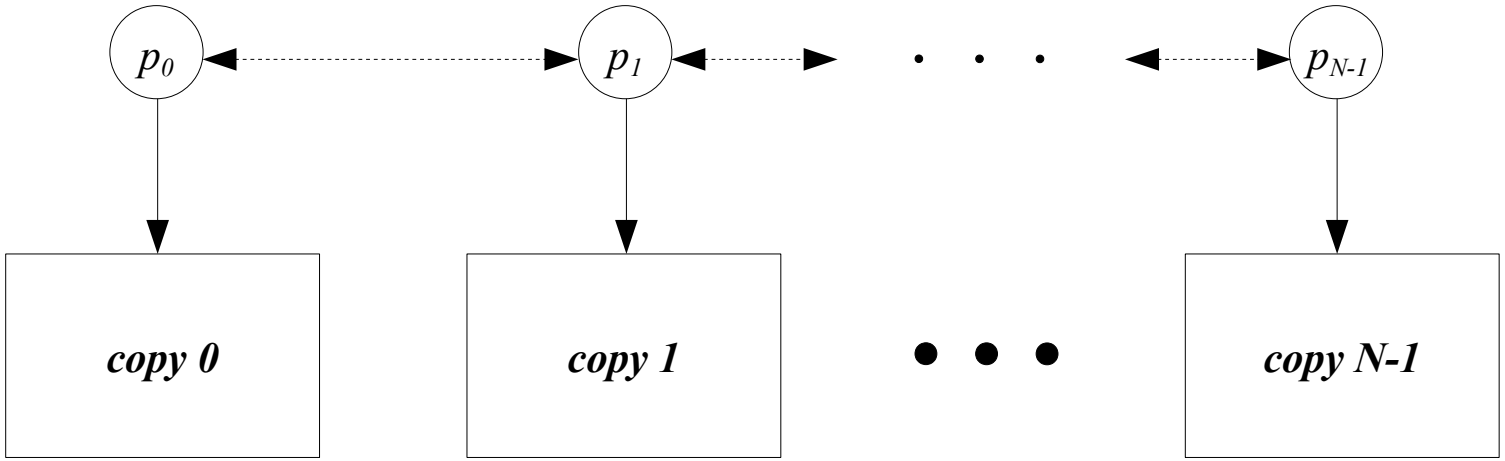
$$(ts(m_p), id(m_p)) < (ts(m_q), id(m_q)) \Leftrightarrow ts(m_p) < ts(m_q) \vee ts(m_p) = ts(m_q) \wedge id(m_p) < id(m_q) \ .$$

Total ordering of groups of events - 3

Assume that, in a given distributed application, there are N copies of the same data region located in geographically distinct places. Each copy is accessed by a specific process p_i, with i = 0,1,…,N-1. Each process p_i performs operations of *writing* and *reading* over the registers of its copy leading to the change of its contents.

Question

How to organize operations so that the different copies are kept permanently synchronized, that is, they always present the same value in all its registers?



Total ordering of groups of events - 4

The permanent synchronization of the copies requires that

- whenever a process wants to modify the value of a register of its local copy, the operation must be propagated first to the processes that manage the access to all the other copies
- the order of execution of these operations must be the same everywhere.

In order for this to be done successfully, one must assume that

- the processes p_i are kept in correct operation, that is, no *catastrophic failures* occur
- there is no message loss.

Total ordering of groups of events - 5

Lamport proposed the following algorithm to solve the problem

- each process p_i, upon asserting that next operation will modify the value of a register of its local copy, builds a message with all the data concerning the operation and attaches a time stamp with the value of its local clock marking the event
- the message is sent to all the group members, itself included
- upon receiving the message, each process p_i adjusts its local clock according to the rules prescribed by Lamport and inserts the message in a local queue in increasing order of its *extended time stamp*
- a message of *acknowledge* is sent to all the group members, itself included
- the operations described in the messages stored in each local queue are executed by each process p_i in the pre-established order when all group members have acknowledged the operation.

Vector logic clock - 1

Mattern (1989) and Fidge (1991) have introduced another type of logic clock with the goal of surpassing the limitation present in Lamport scalar logic clock

$$\neg[\forall_{e_i, e'_j} \; Ck_i(e_i) < Ck_j(e'_j) \;\Rightarrow\; e_i \rightarrow e'_j] \quad \text{, with } i, j = 0,1, \cdots, N-1 \text{ and } i \neq j \text{ ,}$$

that is, when two events e_i and e'_j occur in distinct processes, the fact that the value of the time stamp associated with the first be less than the value of the time stamp associated with the second does not mean the first event *has occurred before* the second.

Their idea was to keep information about events, not only retrieved from their own logic clock, but also all available information retrieved from the logic clocks of the other processes in the group, even if it were not updated. In this way, one can capture the *potential causality* that may exist between events occurring in processes residing in distinct nodes of the parallel machine.

Vector logic clock - 2

The suggested device operating in a N process system is basically a collection of monotonically increasing event counters, which again have no connection to the *actual* time.

The counter collection is organized as an array V, where each element represents a Lamport-like local clock.

In a distributed application, each process p_i, with $i = 0,1,\ ,N\text{-}1$, has its own vector logic clock V_i. Array V_i elements are interpreted in the following way

- $V_i[i] = Ck_i$, is the local clock of process p_i
- $V_i[j] = Ck_j$, with $j \neq i$, represents the perception process p_i has about the evolution of the local clock of process p_j (process p_j may meanwhile have marked more events, but process p_i has not yet received any information about them in the message time stamps so far received).

Vector logic clock - 3

The updating of the local vector clock is carried out according to the following rules

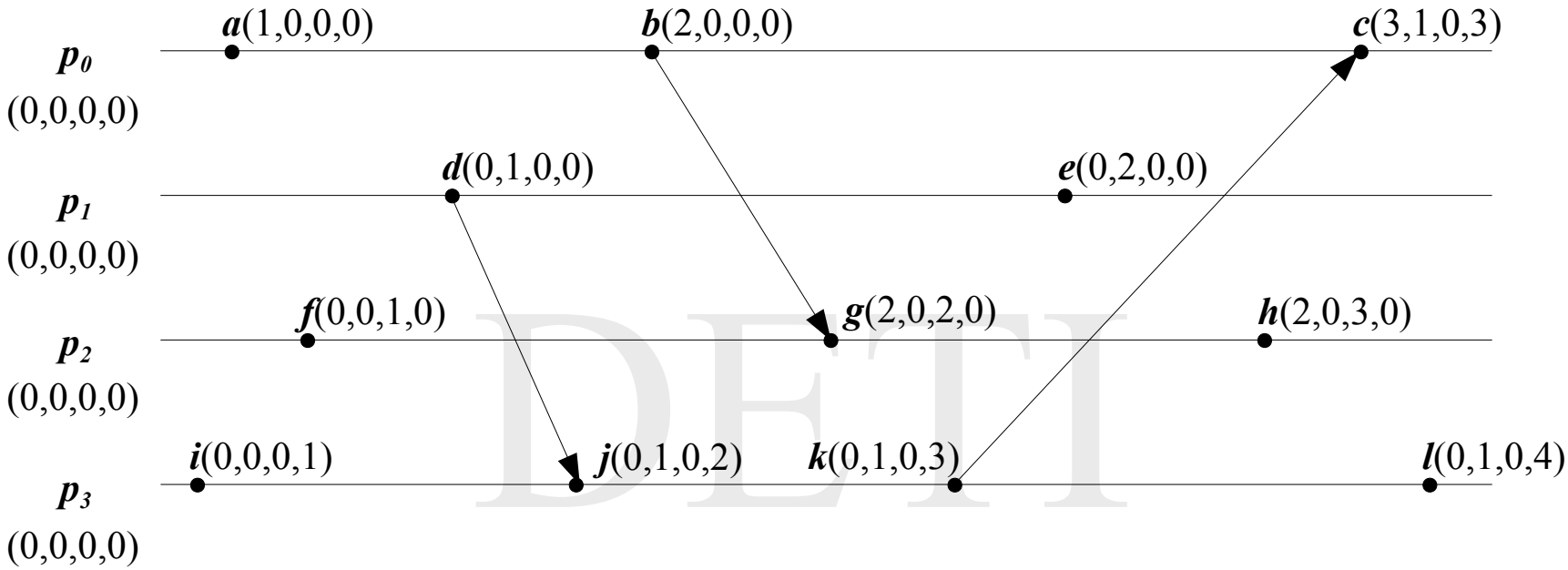
- *initialization*: $V_i[j] = 0$, with $j = 0,1,\ldots,N-1$
- *occurrence of a locally relevant event*: updating of the local clock $V_i[i] = V_i[i]+\alpha_i$, where α_i is a numeric constant usually equal to 1
- *message sending*: insertion of a time stamp *ts*, of value equal to V_i, in the message to be sent after the updating takes place
- *message receiving*: adjustment of each element of the local vector clock to the value *max* ($V_i[j]$, *ts*[*j*]), with $j = 0,1,\ldots,N-1$ and $i \neq j$, before updating its own value $V_i[i]$ with the reception event.

Vector logic clock - 4

Let V e V' be vector time stamps, then their values are compared according to the following rules

- $V = V' \Leftrightarrow \forall_{0 \le j < N} \; V[j] = V'[j]$
- $V \le V' \Leftrightarrow \forall_{0 \le j < N} \; V[j] \le V'[j]$
- $V < V' \Leftrightarrow V \le V' \wedge V \ne V'$.

Vector logic clock - 5



sequential events

$$f \rightarrow h \;\Rightarrow\; V_2(f) < V_2(h)$$

$$d \rightarrow c \;\Rightarrow\; V_1(d) < V_3(c)$$

concurrent events

$$f \parallel c \;\Rightarrow\; \neg[V_2(f) < V_0(c)] \;\wedge\; \neg[V_0(c) < V_2(f)]$$

$$i \parallel e \;\Rightarrow\; \neg[V_3(i) < V_1(e)] \;\wedge\; \neg[V_1(e) < V_3(i)]$$

Vector logic clock - 6

It is straightforward to prove by mathematical induction that

$$e \rightarrow e' \;\Rightarrow\; V_i(e) < V_j(e') \quad \text{, with } i, j = 0,1, \cdots, N-1 \;\; .$$

The converse is now also true.

$$V_i(e) < V_j(e') \;\Rightarrow\; e \rightarrow e' \quad \text{, with } i, j = 0,1, \cdots, N-1 \;\; .$$

Challenge

Prove it!

It is precisely this fact that allows, by comparing the associated time stamps to capture the *potential causality* which may exist between events that occurs in processes located in distinct nodes of a parallel machine and order them.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 11: *Time and global states*
    - Sections 11.1 to 11.4
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 6: *Synchronization*
    - Sections 6.1 and 6.2

Sistemas Distribuídos

Group Communication

António Rui Borges

Summary

- *Characterization of the problem*
- *Access to a shared object in mutual exclusion*
  - *Centralized access permission*
  - *Logic ring*
  - *Total ordering of events*
  - *Minimizing the numbers of messages*
- *Elective procedure*
  - *Election in a logic ring*
  - *Election in an unstructured group*
- *Suggested reading*

Characterization of the problem

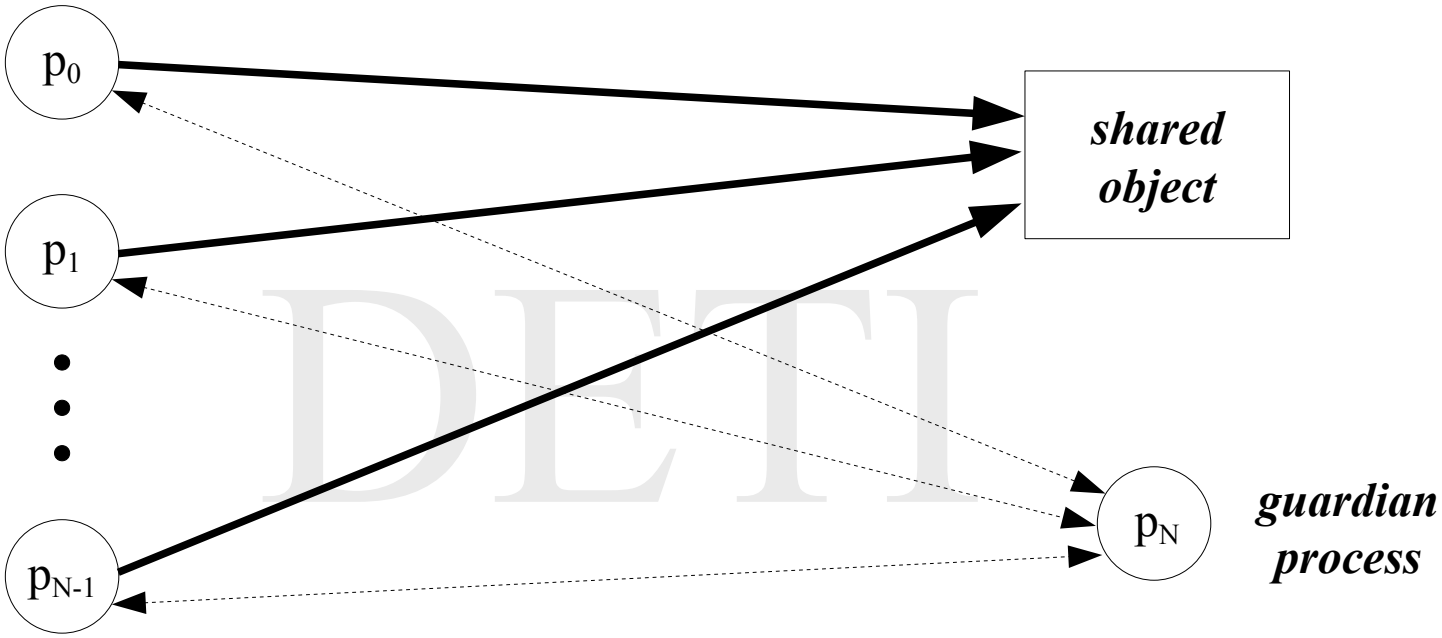
In *group communication*, all involved processes have the same standing. One may assume there is no relevant feature that distinguishes one from the others. We then say we are in a situation of *communication among equals*.

Thus, when they access a common resource, means have to be devised to prevent racing conditions which may lead to information inconsistency. Since they do not share in general an addressing space, synchronization among them must be carried out through message passing.

We may assume for the time being that

- the exchange messages have a finite transmission time, but without an upper limit
- there is no message loss.

Centralized access permission - 1



It is an almost straightforward adaptation of the client-server model with *request serialization*.

There is a special process, the *guardian process*, that keeps track of the accesses to the shared object and allows access to it on per request basis.

Centralized access permission - 2

Protocol

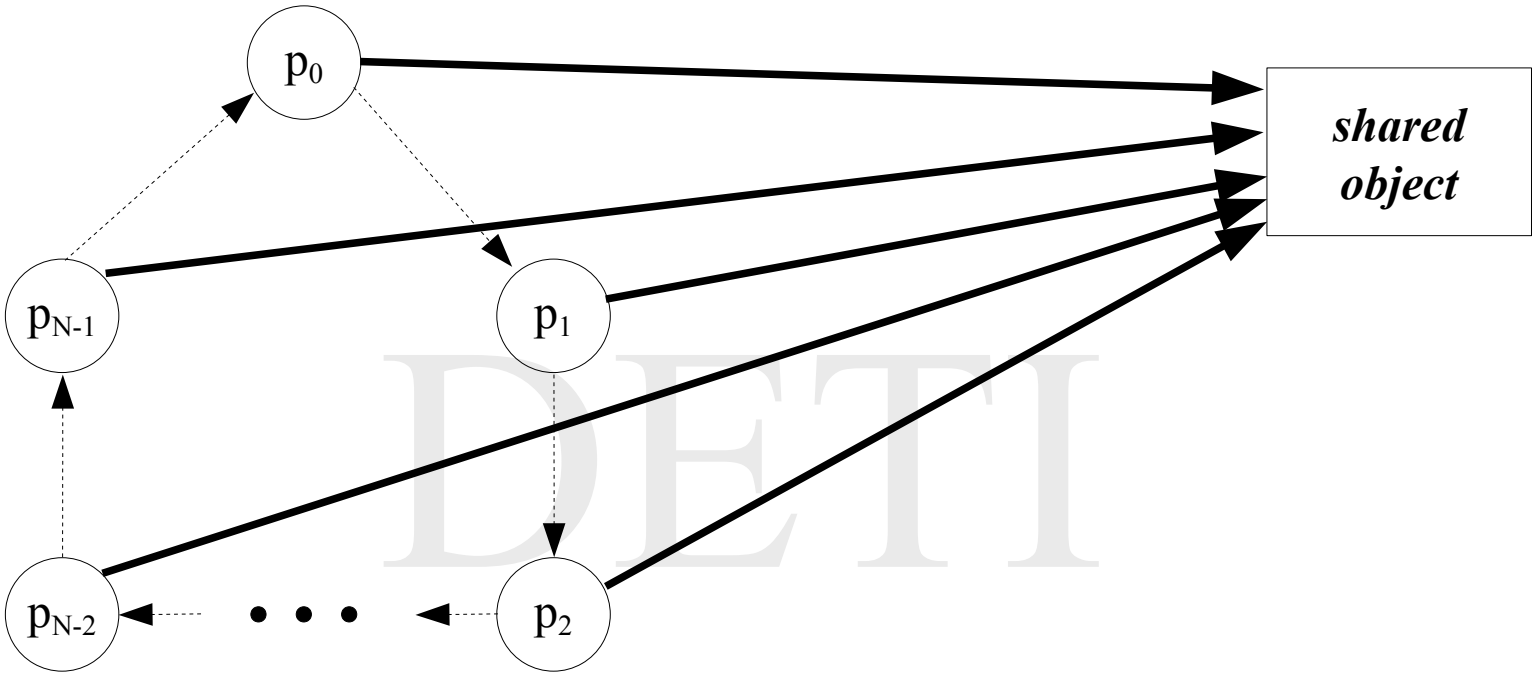
- whenever one of the peer processes p_i, with $i = 0, 1, \cdots, \text{N-1}$, wants access to the shared object, it sends a message of *request access* to p_N , requesting permission, and waits for the reception of a message of *grant access* from it
- if, at the time, there is no other process accessing the shared object, process p_N answers immediately; otherwise, it inserts the request in a waiting queue
- when the message of *grant access* is received, processes p_i may access the shared object
- when access to the shared object is terminated, process p_i sends a message of *release access* to p_N, signaling it no longer requires the object
- if there are pending requests in the waiting queue, process p_N retrieves the first and replies to the referenced process with *grant access*.

Centralized access permission - 3

Comments

- three messages are exchanged per access
- it is not really a true peer-to-peer solution, because it supposes the existence of a special process, the *guardian process*, to control the access to the shared object
- thus, it has a *single point* of failure, if there is a malfunction in process p_N, all the operation comes to a halt.

Logic ring - 1



A restriction is imposed on the processes that form the group: they are organized in a closed loop as far as communication is concerned. Process p_i, with $i = 0, 1, \cdots, N\text{-}1$, can only receive messages from process $p_{(i-1) \bmod N}$ and send messages to process $p_{(i+1) \bmod N}$.

A *token message* is continuously circulated among them. Access to the shared object can only be done by the process that has taken possession of the message.

Logic ring - 2

Protocol

- if the process p_i needs access to the shared object, it waits for the reception of the token message and once it holds it, it accesses the object; upon access termination, it sends the token message to the next process in the ring
- if the process p_i does not need access to the shared object, upon receiving the token message, it sends it immediately to the next process in the ring.

Logic ring - 3

Comments

- one message is always exchanged, either there is access, or not
- it is very efficient if the process group is small
- however, if it is very large, a given process may have to wait a long time to be able to access the object, even if no process is currently doing it.

Total ordering of events - 1

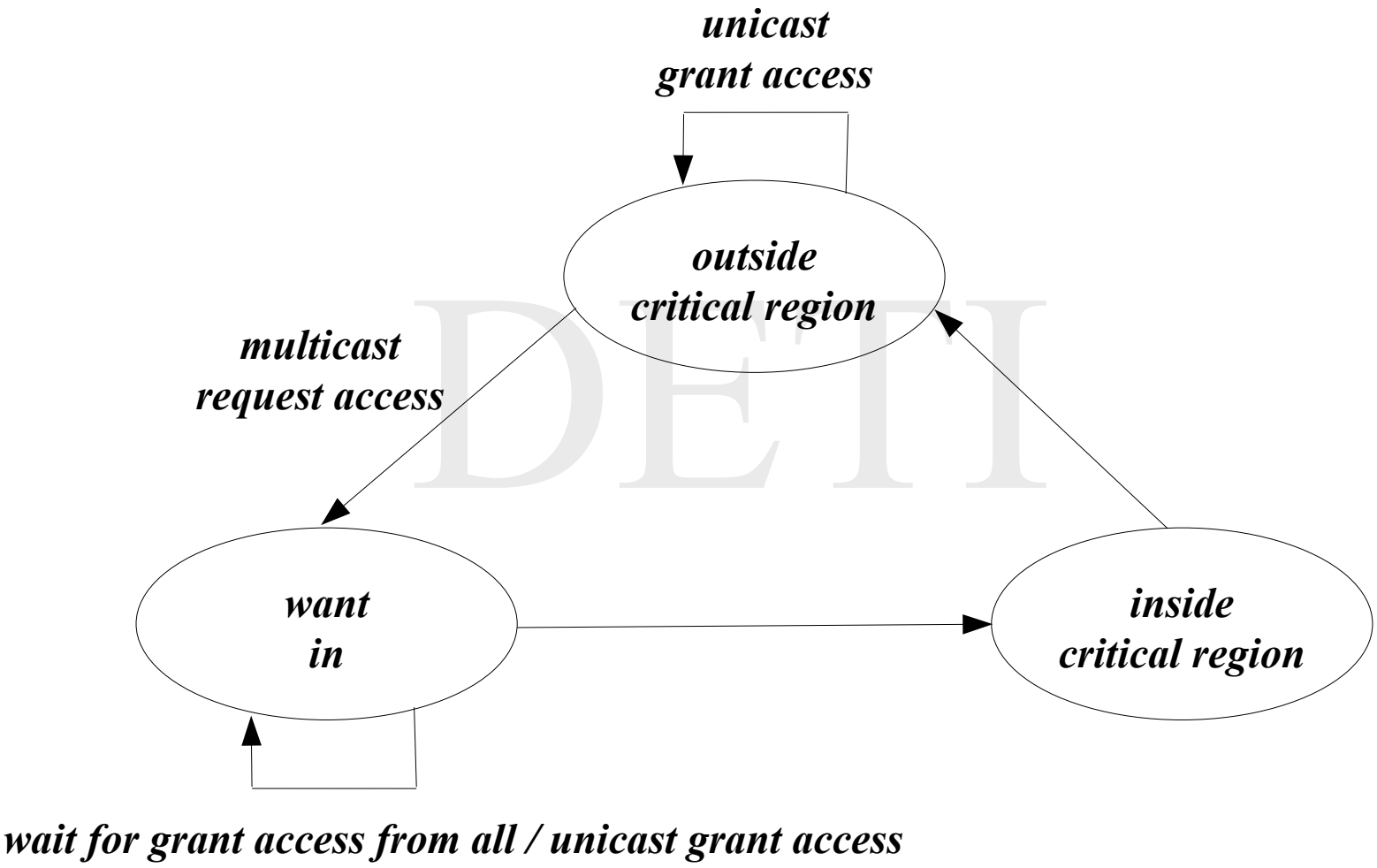
Ricart and Agrawala (1981) have proposed a general method to ensure that N processes access a shared object with mutual exclusion by total ordering their access requests through a Lamport logic clock.

Thus, all processes order in the same way the events associated with the access requests to the shared object and an overall consensus is attained.

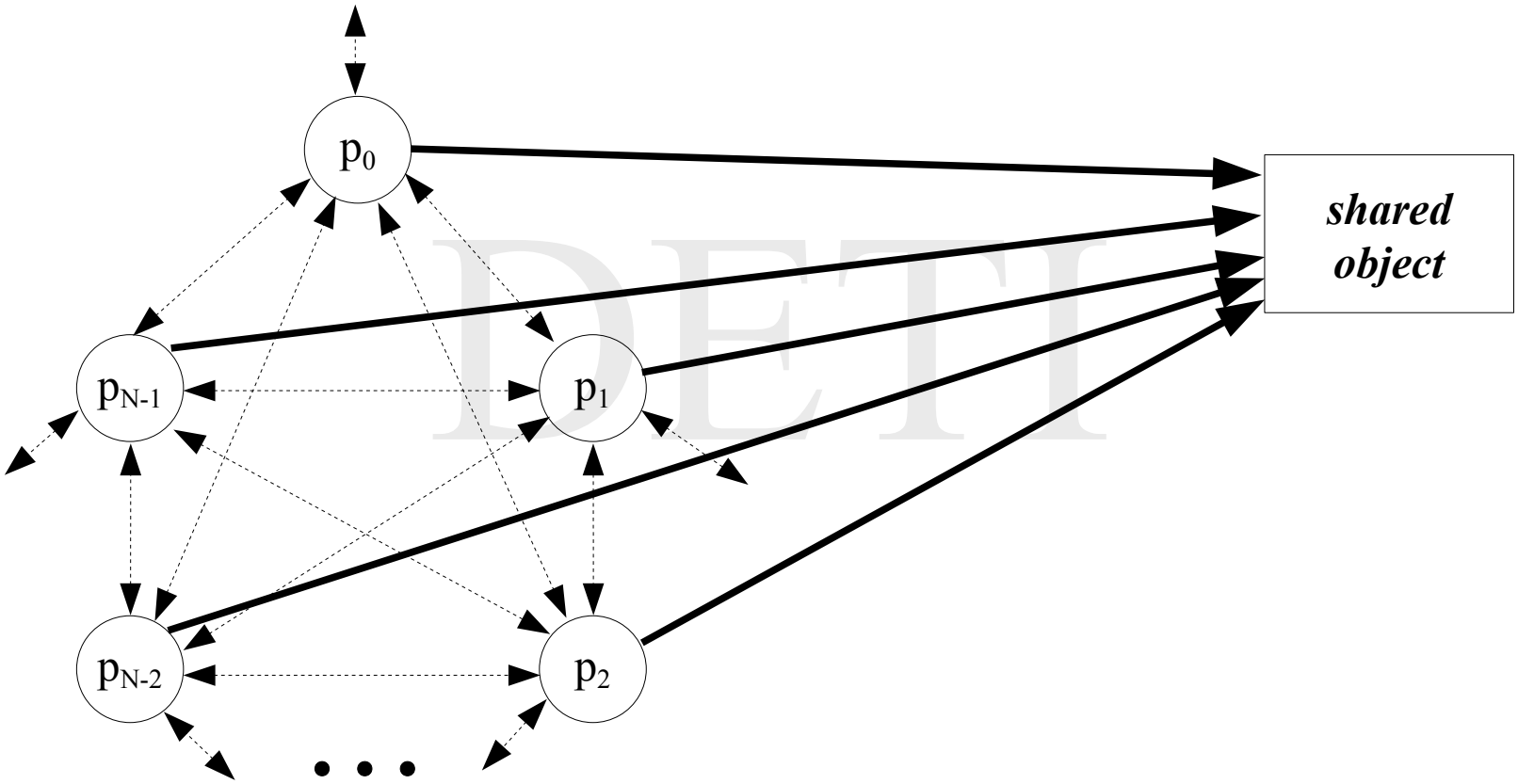
The exchanged messages include the time stamp associated with the sending event, which enables the receiving process, upon message reception, to adjust its local clock according to the rules prescribed by Lamport. Therefore, the potential causality between events can be made explicit.

To generate the total ordering of access request events, an extended time stamp containing the ordered pair $(ts(m_{ar}),id(m_{ar}))$, where $ts(m_{ar})$ is the message time stamp and $id(m_{ar})$ is the sender identification, is constructed.

Total ordering of events - 2



Total ordering of events - 3



Total ordering of events - 4

initialization

```
state = outsideCR;
requestMessageReady = false;
```

p_i enters the critical region

```
state = wantIn;
numberOfRequestsGranted = 0;
myRequestMessage = multicast (requestAccess);
requestMessageReady = true;
wait until (numberOfRequestsGranted == N-1);
state = insideCR;
```

p_i exits the critical region

```
state = outsideCR;
requestMessageReady = false;
while (!empty (requestQueue))
{ id = getId (queueOut (requestQueue));
  unicast (id, accessGranted);
}
```

Total ordering of events - 5

p_i receives an access request message from p_j

```
if (state == outsideCR)
   { id = getId (requestMessage));
     unicast (id, accessGranted);
   }
   else if (state == insideCR)
           queueIn (requestQueue, requestMessage);
           else { wait until (requestMessageReady);
                  if (getExtTimeStamp (myRequestMessage) <
                        getExtTimeStamp (requestMessage))
                     queueIn (requestQueue, requestMessage);
                     else { id = getId (requestMessage));
                            unicast (id, accessGranted);
                          }
                }
```

p_i processes access permissions

```
numberOfRequestsGranted += 1;
```

Total ordering of events - 6

Comments

- 2(N-1) messages are exchanged per access
- it is very efficient if the process group is small
- however, if it is very large, a lot of messages are exchanged.

Minimizing the number of messages – 1

Maekawa (1985) has shown that the access in mutual exclusion to a shared object by any of the peer processes that exist, does not require permission of all of them. The peer processes can be organized in partial groups and only get permission from all the processes belonging to their group.

Mutual exclusion and, therefore, the elimination of racing conditions on access to the shared object are ensured as long as the groups are not mutual exclusive.

The underlying principle is to impose that a process only accesses the shared object if and only if it has permission from all the processes belonging to its group. The permission is given through a voting process.

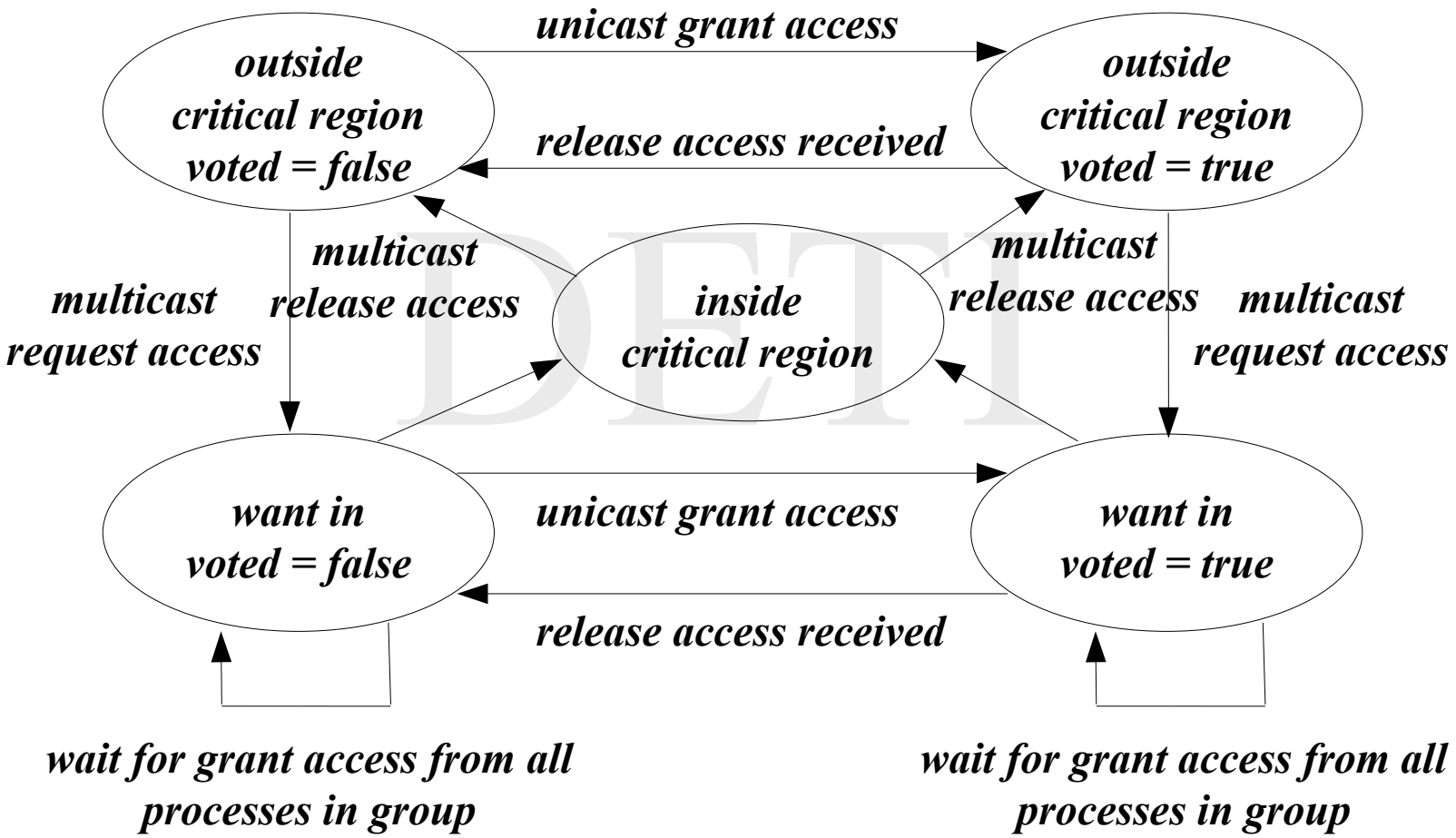
Minimizing the number of messages – 2

Each peer process p_i, with $i = 0, 1, \cdots, N-1$, belong to the *voting group* V_i.

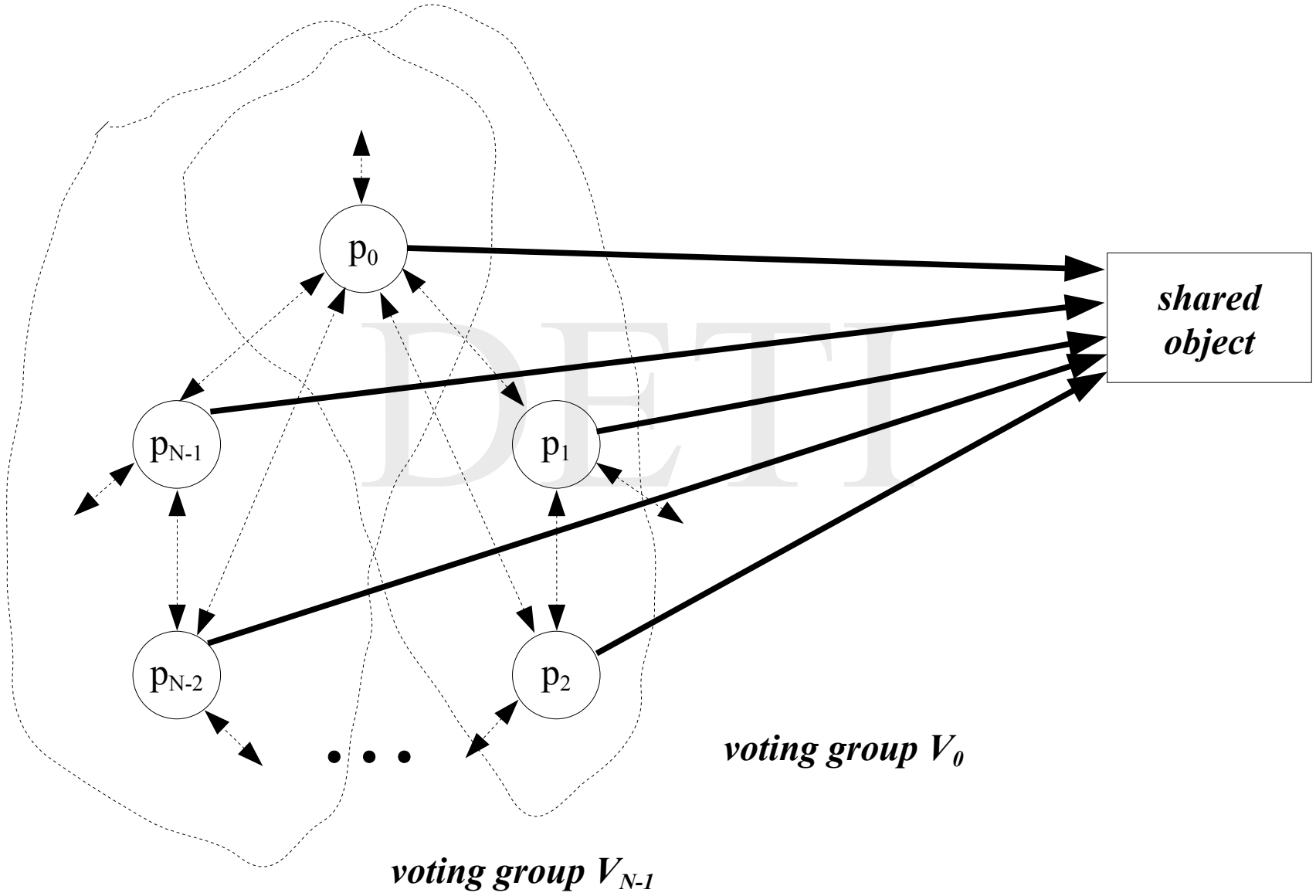
The elements of each voting group are chosen such that

- $\forall_{0 \le i < N} \quad V_i \subseteq \{p_0, p_1, \cdots, p_{N-1}\}$
- $\forall_{0 \le i < N} \quad p_i \in V_i$
- $\forall_{0 \le i, j < N} \quad V_i \cap V_j \neq \emptyset$
- $\forall_{0 \le i, j < N} \quad \#(V_i) \approx \#(V_j)$
- $\exists_{M \in \mathbb{N}} \ \forall_{0 \le i < N} \quad p_i$ belongs to M groups V_* .

Minimizing the number of messages – 3



Minimizing the number of messages – 4



Minimizing the number of messages – 5

The exact determination of the contents of the different voting groups V_i, with $i = 0, 1, \cdots, N-1$, is not a simple procedure. There is, however, an approximation which makes $\#(V_i) \simeq \#(V_j)$, $M \simeq O(\sqrt{N})$ and is trivial.

0	1	$\cdots$	$K-2$	$K-1$
K	$K+1$	$\cdots$	$2K-2$	$2K-1$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$(R-1)K$	$(R-1)K+1$	$\cdots$	$RK-1$	0

$$V_1 = \{0,\ 1,\ \cdots,\ K-1,\ K+1,\ \cdots,(R-1)K+1\}$$

Minimizing the number of messages – 6

initialization

```
state = outsideCR;
voted = false;
```

p_i enters the critical region

```
state = wantIn;
numberOfRequestsGranted = 0;
multicast (requestAccess)to all processes in V_i;
wait until (numberOfRequestsGranted == #(V_i));
state = insideCR;
```

p_i exits the critical region

```
state = outsideCR;
multicast (releaseAccess)to all processes in V_i;
```

Minimizing the number of messages – 7

p_i receives an access request message from p_j

```
if ((state != insideCR) && !voted)
   { id = getId (requestMessage));
     unicast (id, accessGranted);
     voted = true;
   }
   else queueIn (requestQueue, requestMessage);
```

p_i receives a release access message from p_j

```
if (!empty (requestQueue))
   { id = getId (queueOut (requestQueue));
     unicast (id, accessGranted);
     voted = true;
   }
   else voted = false;
```

p_i processes access permissions

```
numberOfRequestsGranted += 1;
```

Minimizing the number of messages – 8

Comments

- 3 $O(\sqrt{N})$ messages are exchanged per access
- it is very efficient when the process group is very large.

Minimizing the number of messages – 9

This algorithm is, however, incorrect! There are instances that, due to the prevailing racing conditions, *deadlock* may occur.

$$V_0 = \{0, 1\}$$
$$V_1 = \{1, 2\}$$
$$V_2 = \{2, 0\}$$

Processes p_0, p_1 e p_2 try to access the shared object about the same time. It may happen that, due to the moments of message arrival, p_0 votes for its own access, p_1 votes for its own access and p_2 votes for its own access. Thus, all processes receive a first permission, but never a second.

How to solve the problem?

Minimizing the number of messages – 10

Saunders (1987) has shown that a possible solution is to total order the access requests to prevent contention. Another one is to deny the condition of *circular waiting*, ordering the access requests by the process identification.

Adapt Maekawa algorithm so that deadlock is prevented!

Elective procedure - 1

There are cases which require the selection a process from a group to carry out a well-defined task at some specific moment. Since all processes are supposed to be conceptually similar and can not be distinguished one from the other, any of them can be in principle selected.

Important points to stress are

- the elective procedure has to be performed in a finite number of steps
- there can be no ambiguity, that is, only one process is selected in the end
- the decision must be consensual, that is, all the involved processes will accept the selection.

Elective procedure - 2

It will be assumed that

- the number of processes in the group is fixed and previously known
- their state at any given time is *in execution* or *catastrophic failure*
- the transmission time of the exchanged messages is finite, but has an upper bound, that is, *time-outs* may be defined
- messages may be lost.

Election in a logic ring - 1

- to begin with, no election is taking place and all processes are in the state of *no participant*
- any process may initiate the elective procedure; it changes its state to *participant* and sends to the next process in the ring a message of type *start election* with its own identification
- when a process receives a message of type *start election*, it carries out the following actions
  - if the process identification in the message is less than its own, it sends the message to the next process in the ring and changes its state to *participant*, if it was not yet changed
  - if the process identification in the message is greater than its own and it is not yet a participant, it replaces the process identification with its own, sends the message to the next process in the ring and changes its state to *participant*; however, if it is already in the state of *participant*, it discards the message to reduce the number of circulating messages
  - if the process identification in the message is equal to its own, then the elective procedure has terminated and the process itself was elected as *leader* (**why?**)

Election in a logic ring - 2

- when a process is elected as *leader*, it sends a message of type *elected* with its own identification
- any process receiving a message of type *elected* carries out the following actions
  - it changes its state to *no participant*
  - if the process identification in the message is different from its own, it saves the identification of the *leader* process and sends the message to the next process in the ring
  - if the process identification in the message is equal to its own, it discards the message leading to the end of the elective procedure.

Election in a logic ring - 3

This algorithm was proposed by Chang e Roberts (1979) and solves the problem if there are no failures.

What should be done if the logic ring requires a dynamic reconfiguration due to

- a process changing its state from *in execution* to *catastrophic failure*, or vice-versa?
- message loss?

Election in an unstructured group – 1

- to begin with, no election is taking place and all processes are in the state of *no participant*
- any process may initiate the elective procedure; it changes its state to *participant* and sends to all the processes having a lower identification number a message of type *start election* with its own identification
- when a process receives a message of type *start election*, it carries out the following actions
  - it replies to the sender process with a message of type *acknowledge*
  - if its state was *no participant*, it changes its state to *participant* and sends to all the processes having a lower identification number a message of type *start election* with its own identification
- when a process receives a message of type *acknowledge*, it waits for a message of type *elected* with the identification of the leader
- if, during a prescribed period of time, a process having the *participant* state does not receive any message of type *acknowledge*, it considers itself the process with the lower identification that is alive and assumes the role of *leader*, it then sends a message of type *elected* with its own identification to all the processes of the group to inform them of the fact.

Election in an unstructured group – 2

This algorithm was proposed by Garcia-Molina (1982) and solves the problem if there are no failures.

What should be done if the unstructured group requires a dynamic reconfiguration due to

- a process changing its state from *in execution* to *catastrophic failure*, or vice-versa?
- message loss?

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 12: *Coordination and agreement*
    - Sections 12.1 to 12.3
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 6: *Synchronization*
    - Sections 6.3 to 6.5

Sistemas Distribuídos

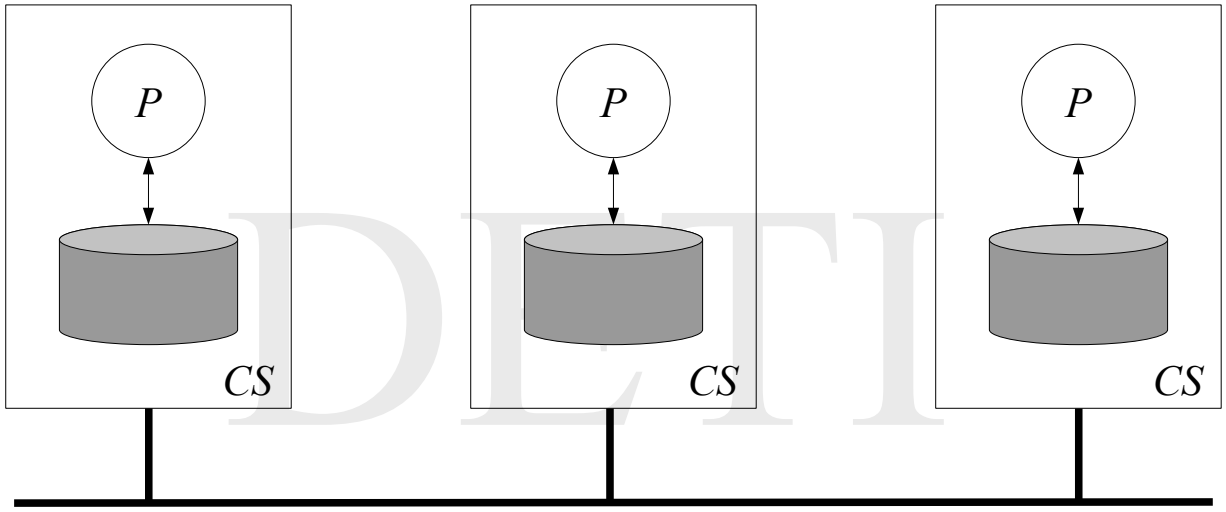
Consistency and Replication

António Rui Borges

Summary

- *Region of distributed storage*
  - *Characterization*
- *Consistency models*
  - *Criterion for 'correct operation'*
  - *Types*
    - *Strict consistency*
    - *Linearizability*
    - *Sequential consistency*
    - *Causal consistency*
    - *FIFO consistency*
- *Suggested reading*

Region of distributed storage - 1



A *region of distributed storage* can be thought of as a memory, a data base, or a file system, replicated over a set of processing nodes.

Region of distributed storage - 2

- it is assumed that
  - each processing node has a direct access to its local copy of the whole region
  - only two types of operations are relevant: *write*-like operations, which generate a modification of the contents of some register, and *read*-like operations, consisting of the remaining operations
  - when a *write*-like operation occurs, it must be propagated to all the copies so that an update takes place in each of them, *read*-like operations may, or may not, be propagated
  - access to the local copies of the region of distributed storage is thought to be carried out in parallel, that is, simultaneously, by the associated processes.

Consistency models

One defines *consistency model* as the set of rules, or contract, that if complied by the intervening processes, lead to the ‘correct operation’ of the region of distributed storage.

The issue, then, is defining with precision what one means by ‘correct operation’ and apply the concept in a systematic way.

Each particular model is characterized by specifying clearly the range of values a *read*-like operation may return vis a vis the *write*-like operations that have occurred previously.

When concurrency is assumed in access to data, the consistency models belong to a *data-centric* class.

Criterion for 'correct operation' - 1

Let $op_{i,0}$, $op_{i,1}$, $op_{i,2}$,…, with i = 0, 1,…, N-1, be any given sequence of *write*-like and *read*-like operations performed by the process i over the region of distributed storage.

Each operation is characterized by its type, arguments and returned value, and is supposed to be synchronous, that is, the next operation can only be executed when the one before has been terminated.

It is also assumed that the N processes, each performing its own sequence of operations, access in parallel the region of distributed storage. If the region were centralized, instead of distributed, the sequences of operations performed by the different processes would intermix in some manner, each time the accesses were instantiated, producing a global sequence of the type

$$op_{2,0}, op_{2,1}, op_{0,0}, op_{1,0}, op_{1,1}, op_{2,2}, op_{2,3}, op_{0,1}, op_{1,2}, \ldots .$$

Criterion for 'correct operation' - 2

The *criterion for 'correct operation'* of the region of distributed storage is now defined by referring to one or more global canonical sequences that establish the pattern of operation and which are produced by intermixing the partial sequences of the individual processes.

These canonical sequences are merely virtual, they do not have actually to occur. Their role is to serve as certification of what one calls '*correct operation*' to a given instantiation of parallel access to the region of distributed storage.

Based on what is perceived in the instantiation, the inexistence of a canonical sequence which obeys to the properties of the referenced consistency model, allows one to conclude that the model is not valid.

Strict consistency - 1

Any read-like operation performed on the register x returns the value produced by the most recent write-like operation on x.

It is the ideal situation which can only be achieved in monoprocessor systems.

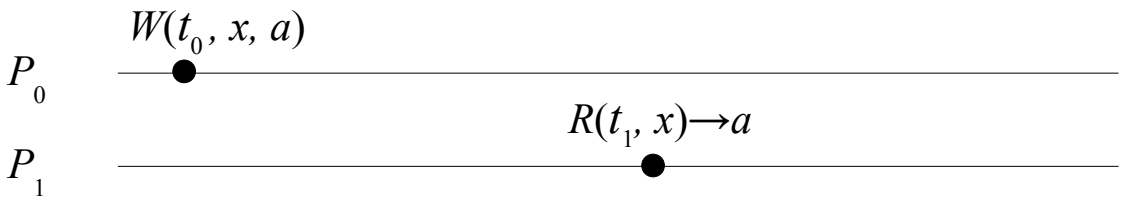
This model can not be implemented in parallel machines, because the notion of 'most recent' event is ambiguous in this context. The ambiguity stems from the fact the propagation speed of any physical signal is necessarily finite.

Therefore,

- the local clocks of the different processing nodes can not be synchronized with absolute accuracy, resulting in the impossibility of having a global clock to order chronologically the events
- the execution time of write like and read like operations is not null.

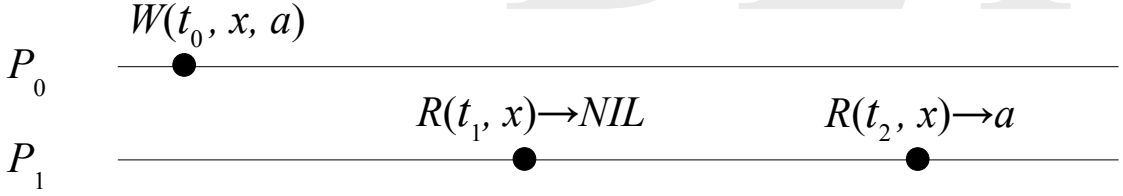
Hence, events can not be ordered in terms of an unique time standard.

Strict consistency - 2



strictly consistent storage region

canonical sequence: $t_0 < t_1 \Rightarrow W_0(t_0, x, a)$ - $R(t_1, x) \rightarrow a$



non strictly consistent storage region

there is no canonical sequence which mimics the results

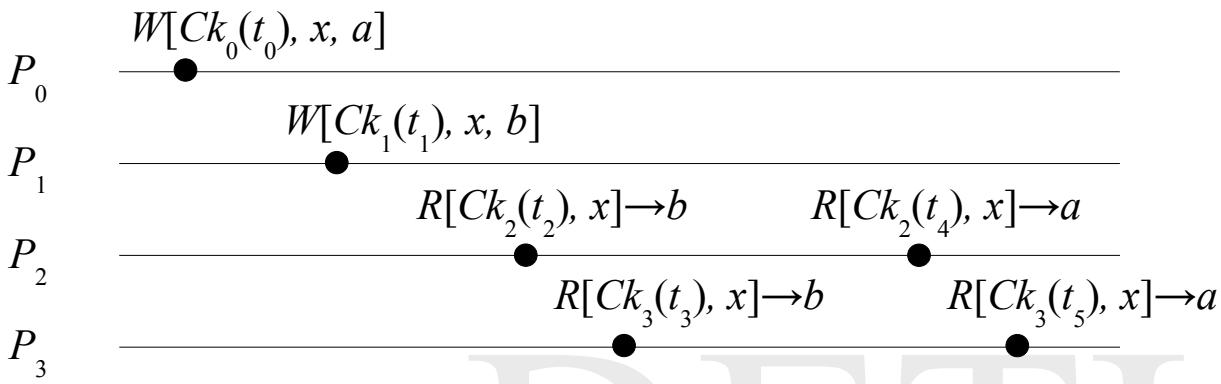
Linearizability - 1

Parallel access to a register x is seen by all involved processes as if the performed operations were ordered in an unique and well-defined sequence by keeping the chronological order that is locally perceived (Lamport / Herlihy & Wing).

This model supposes there is some synchronization procedure of the local clocks which *total orders* the events (*it is an approximation to the notion of global clock*). The goal is to ensure that the intervening processes always get the most updated data.

One aims to convey the idea that, for the sequences of operations carried out by the intervening processes in parallel, there is a canonical sequence of global execution of the same operations over a concentrated virtual region which transmits an image of unique execution, consistent with the chronological execution of the operations as they are locally perceived.

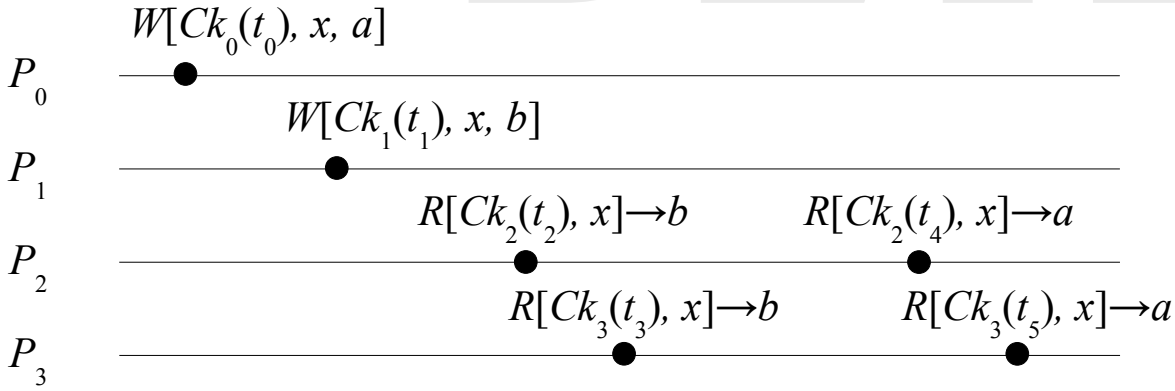
Linearizability - 2



linearizable storage region

canonical sequence: $Ck_1(t_1) < Ck_2(t_2)$, $Ck_3(t_3) < Ck_0(t_0) < Ck_2(t_4)$, $Ck_3(t_5)$ $\Rightarrow$

$\Rightarrow$ $W_1[Ck_1(t_1), x, b]$ - $R_{2,3}(-, x) \rightarrow b$ - $W_0[Ck_0(t_0), x, a]$ - $R_{2,3}(-, x) \rightarrow a$



non linearizable storage region

$Ck_0(t_0) < Ck_1(t_1)$ $\Rightarrow$ *there is no canonical sequence which mimics the results*

Linearizability - 3

Implementation

All *read*-like and *write*-like operations must be propagated and acknowledged by all the processes managing the local copies of the region of distributed storage, before the associated action takes place.

This can be achieved by *total ordering* the events associated with the operations through the use of a *logic scalar* clock, as the local clock, and adjusting it according to the rules prescribed by Lamport.

Possible application areas

All the cases where the most strict fairness is required. Government support services or financial transaction systems, for instance.

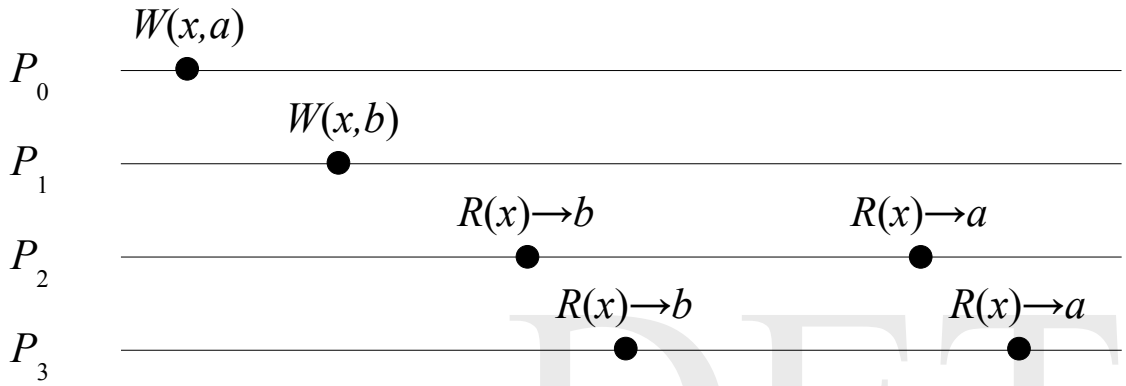
Sequential consistency - 1

Parallel access to a register x is seen by all involved processes as if the performed operations were ordered in an unique and well-defined sequence where the operations of each individual process keep the order of local execution (Lamport / Herlihy & Wing).

The first observation is that time, whether global, or local, does not play an explicit role in this framing. The second is that, because of this, it is not necessary to total order all the occurring events.

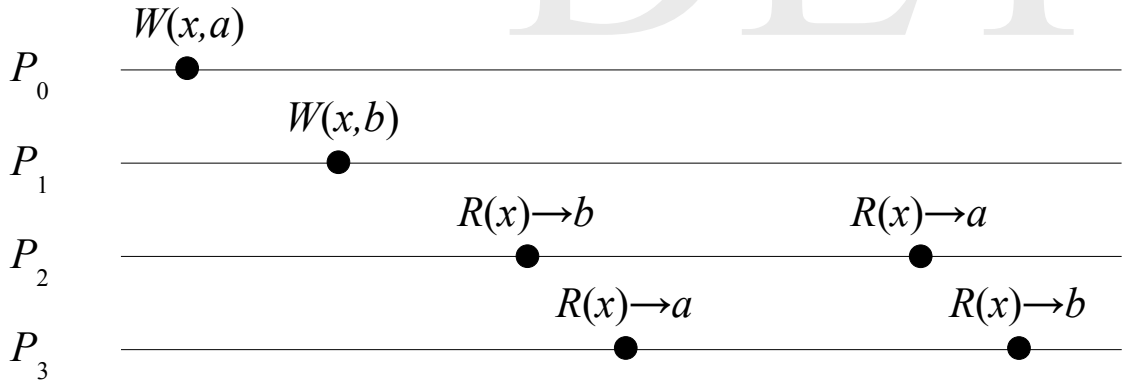
One aims to convey the idea that, for the sequences of operations carried out by the intervening processes in parallel, there is a canonical sequence of global execution of the same operations over a concentrated virtual region which transmits an image of unique execution, consistent with the successive execution of the local operations.

Sequential consistency - 2



sequentially consistent storage region

canonical sequence: $W_1(x,b)$ - $R_{2,3}(x)\rightarrow b$ - $W_0(x,a)\rightarrow a$ – $R_{2,3}(x)\rightarrow a$



non sequentially consistent storage region

there is no canonical sequence which mimics the results

Sequential consistency - 3

Implementation

Only *write*-like operations must be propagated and acknowledged by all the processes managing the local copies of the region of distributed storage, before the associated action takes place.

This can be achieved by *total ordering* the events associated with the *write*-like operations through the use of a *logic scalar* clock, as the local clock, and adjusting it according to the rules prescribed by Lamport.

Possible application areas

All the cases where fairness is in general required. Booking systems and e-commerce, for instance.

Causal consistency - 1

Parallel access to a register x is seen by all involved processes as if the performed operations, which keep between them a causal relation, were ordered in an unique and well-defined sequence. The remaining operations, said to be concurrent, may be perceived in any order by the different processes (Hutto / Ahamad).

In order to have a clear understanding of what this criterion states, it is necessary to define in detail the meaning of *causality* as it is used here.

Thus, one considers that

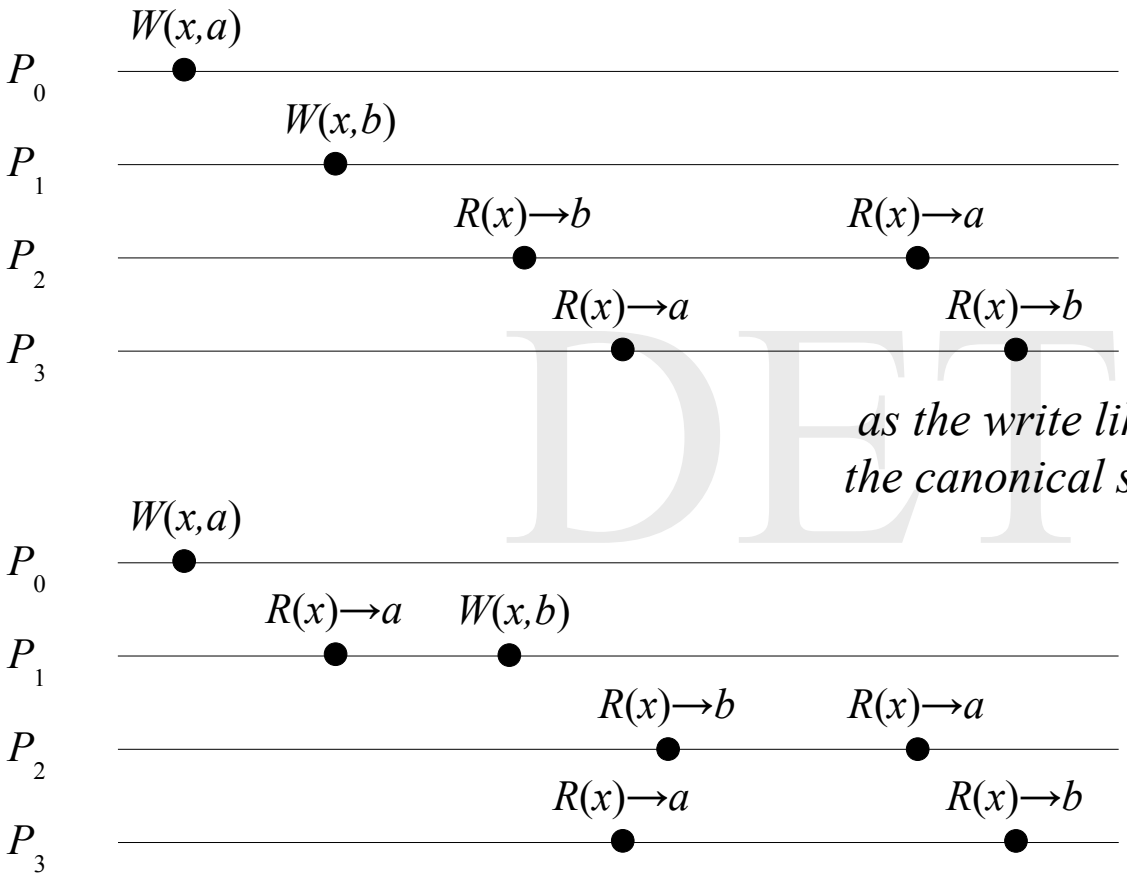
- the operations which are executed in succession by the same process are *causal* related among themselves
- a read like operation is always *causally* related to the write like operation which has supplied the value that was read, whether or not the process that executed each operation is the same.

Causal consistency - 2

The idea that is conveyed here is that, for the sequences of operations carried out by the intervening processes in parallel, there is a canonical sequence of global execution of the same operations over a concentrated virtual region which transmits an image of unique execution, consistent with the successive execution of the local operations, is restricted to the subset of operations that keep between them a causal relation.

Only for these operations the criterion of sequential consistency is imposed. All the remaining operations can be perceived in any order by the different intervening processes.

Causal consistency - 3



causally consistent storage region

as the write like operations are concurrent,
the canonical sequence is in this case empty

non causally consistent storage region

as the write like operations are causally related,
there is no canonical sequence which mimics the results

Causal consistency - 4

Implementation

Only *write*-like operations must be propagated and acknowledged by all the processes managing the local copies of the region of distributed storage.

This can be achieved by *partial ordering* the events associated with the *write*-like operations through the use of a *logic vector* clock, as the local clock, and adjusting it according to the rules prescribed for logic vector clocks.

Possible application areas

All the cases where causality is in general required. Chatting applications, for instance.

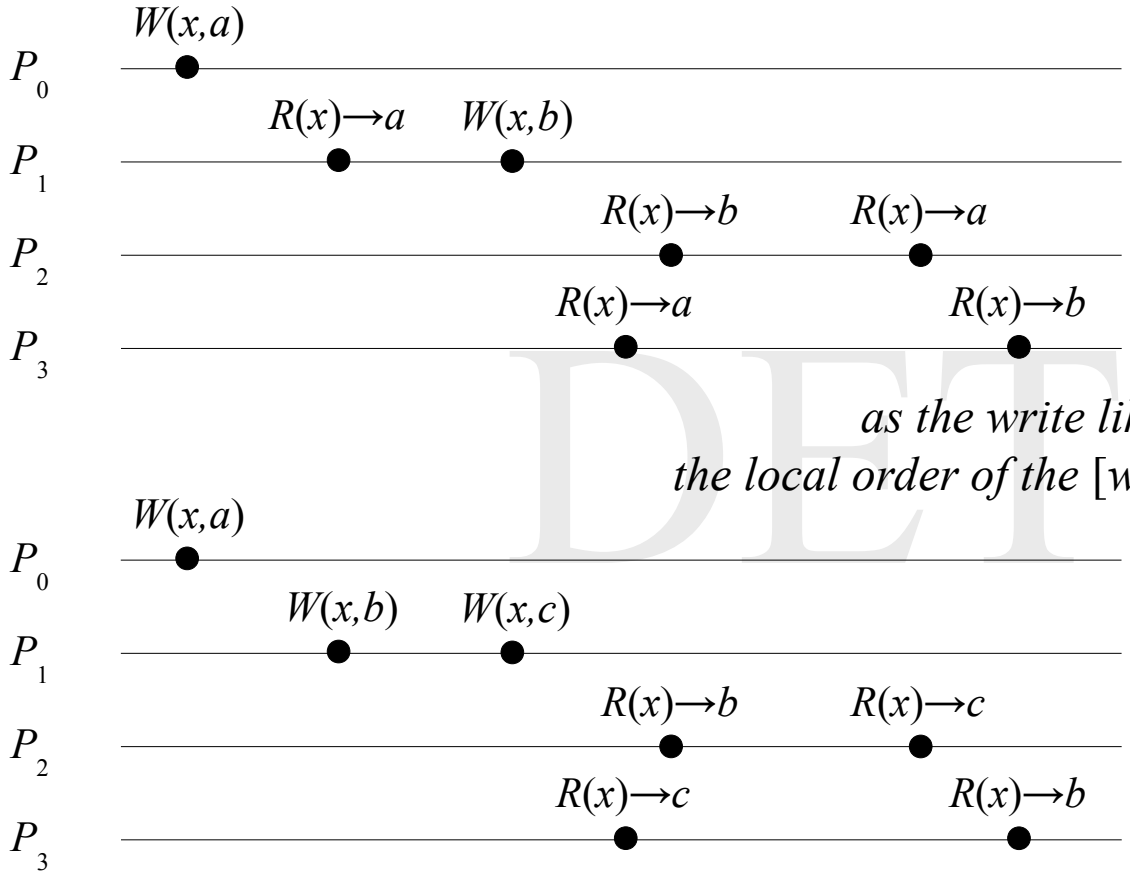
FIFO consistency - 1

Parallel access to a register x is seen by all involved processes as if the performed operations by any given process keep the order of local execution (Lipton / Sandberg).

One should note that in this case the existence of a canonical sequence of global execution of the same operations over a concentrated virtual region which transmits an image of unique execution, is irrelevant.

As long as the order of execution of the local operations is perceived as the same by all the intervening processes, and one is only referring to the write like operations, since the read like operations are not important, because they do not produce any effect, the intermixing of the partial sequences can be whatever and perceived in a different way by the intervening processes.

FIFO consistency - 2



FIFO consistent storage region

as the write like operations are concurrent, the local order of the [*write like*] *operations is trivial*

non FIFO consistent storage region

the local order of the [*write like*] *operations executed by process* P_1 *is not perceived as such by process* P_3

FIFO consistency - 3

Implementation

Only *write*-like operations must be propagated by all the processes managing the local copies of the region of distributed storage.

This can be achieved by *ordering* per process the events associated with the *write*-like operations through the use of a *message counter*.

Possible application areas

All the cases where a less degree of consistency is required. News organizations, weather forecasting, for instance.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 12: C*oordination and agreement*
    - Section 12.4
  - Chapter 15: *Replication*
    - Sections 15.1 and 15.2
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 7: *Consistency and replication*
    - Sections 7.1 and 7.2