Programming of Distributed Systems

Denis Caromel

University of Nice Sophia Antipolis, IUF

I3S-CNRS- INRIA Sophia Antipolis

OASIS Project

TC 4

Part: Distributed and Mobile Objects

D.E.A. RSD

RESEAUX ET SYSTEMES DISTRIBUES

Ecole Doctorale STIC,

Université de Nice - Sophia Antipolis

---

www.inria.fr/oasis/Denis.Caromel/

caromel@unice.fr

October 2005

Overview of 3 first Lectures

- 1. Introduction to Distributed Objects:

- Basic issues, and architecture
- RPC, language issues
- Client / Server, N tiers
- Grid (and Components)
- Peer 2 Peer
- Reminder to be looked at between Lecture 1 and 2:

  Threads and Java RMI principles

  N tiers Architecture

  B2G Components

- 2. Remote Objects vs. Active Objects

- Active Objects:
  - Principles
  - Asynchronous Communications
  - Futures
  - Group Communications
  - Meta Objects, and meta-Object Protocols

    Demo: C3D

- 3. Advance Features: Mobility and Formal Models

- Communicating Mobile Agents
- Demo: Mobile Penguin
- Localization issues and Performance modeling (cf. TC3)
- Formal Calculus for Objects and Asynchronous Objects:
  - Sigma calculus
  - ASP: Asynchronous Sequential Processes

- Hands-on (Practical Session): by yourself or Option 10:

Langages de programmation concurrente, parallèle et distribuée (M10 and Opt. 06)

- The ProActive environment
- Interactive monitoring and visualization: IC2D
- Programming a Communicating Mobile Agent

Table of Contents

CHAPITRE 1 Introduction 8

1.1 Definitions and architectures. 9
1.1.1 Definitions 9
1.1.2 Basic architectures 10

1.2 RPC: Remote Procedure Call 12
1.2.1 Principle 12
1.2.2 RPC variations 15
a) Forme, syntaxe. 15
b) Passage des paramètres 15
c) Représentation des données. 15
d) Sémantique de l'appel 16
e) Gestion des erreurs 16
f) Code sur machine distante 16
g) Découverte et nommage 16
h) Terminaison et GC distribué 17
i) Persistance des ‘‘objets‘‘ ou processus distants. 17
j) Sécurité. 17
k) Autres aspects et concepts. 18

1.3 N tiers applications in Business 20
1.3.1 Web Tiers and RPC 20
1.3.2 Middle Tiers and RPC 20

1.4 GRID computing. 21
1.4.1 Main ideas 21
1.4.2 Globes 23
1.4.3 WS GRID: OGSA -- OGSI from GGF 26

1.5 From Object to (Grid) Components 29
1.5.1 Fundamental of Components. 29
1.5.2 Business to Grid Components 29

1.6 Current hype (?): P2P 30
1.6.1 Data P2P, Principles: 30
1.6.2 A few attributes and Research Issues. 32
1.6.3 Computational P2P 34

1.7 Conclusion, Research Perspectives. 37

1.8 Waiting for structuring 39

CHAPITRE 2 Objects for Distribution: Basic Principles 41

2.1 Sequential features being usefull 42
2.1.1 Inheritance 42
2.1.2 Polymorphism 42
2.1.3 Dynamic Binding 43
2.1.4 Sequential system at execution 43

2.2 Objects and Activities 46
2.2.1 Parallelism orthogonal to objects 46
2.2.2 Parallelism per Active Objects 46

CHAPITRE 3 Reminder: Java Monitors 48

3.1 Basic ideas 49

3.2 Primitives 50
3.2.1 Thread Control Methods: 50
3.2.2 Synchronization 51

3.3 Example: Produser/Consumer in Java 55

3.4 Conclusions on monitors 58

CHAPITRE 4 Reminder: Java RMI 59

4.1 Basic Principles 60
4.1.1 Fonctionnalities 60
4.1.2 Architecture 61
4.1.3 How it works 62
4.1.4 Differences with RPC 63
4.1.5 Communication Model 63
4.1.6 Other characteristics 64

4.2 Classes and Interfaces 65
4.2.1 Interface Remote 66
4.2.2 Class UnicastRemoteObject 67
4.2.3 Global view 68
4.2.4 Binding of a RMI object 69
4.2.5 Lookup of an RMI object 70

4.3 Tools and compilers 71
4.3.1 rmic 71
4.3.2 rmiregistry 72

4.4 A typical developpement Method 73
4.4.1 Remote interfaces 73
4.4.2 Classes that implement the Remote Interfaces: server(s) 73
4.4.3 Clients using the remote objets 74
4.4.4 Compilation of sources 74
4.4.5 Code Generation: Stubs/Skeletons creation 74
4.4.6 Start the rmiregistry 74
4.4.7 Start the server 75

4.4.8 Start the clients . . . 75

4.5 Simple example: Remote Hello . . . 76
4.5.1 Remote Interface . . . 76
4.5.2 Server HelloWorld . . . 77
4.5.3 Registering in the rmiregistry . . . 78
4.5.4 A client to HelloWorld . . . 80
4.5.5 Compilation, generation, registry . . . 81
a) Server side . . . 81
b) Client side . . . 82

4.6 Implementation principle: ad hoc reification of calls . . . 83
4.6.1 Stub . . . 83
4.6.2 Skeleton . . . 85

CHAPITRE 5 Reminder: Advanced RMI . . . 86

5.1 Security and dynamic code loading . . . 87
5.1.1 Security Manager . . . 87
5.1.2 Security file . . . 88
5.1.3 Signed applets and RMI . . . 89
5.1.4 Codebase and RMI . . . 89
5.1.5 CLASSPATH and rmiregistry . . . 93

5.2 Applets and RMI . . . 95
5.2.1 Principles . . . 95
5.2.2 Code and principles of classes . . . 97
a) Remote interface Hello . . . 97
b) Server class (HelloImpl) and main . . . 98
c) Client Applet: HelloApplet . . . 99
d) Page HTML: hello.html . . . 100
5.2.3 Compilation and deployment . . . 101
a) File location, and HTTP servers . . . 101
b) Compilation . . . 102
c) Installation of the applet and CLASSPATH . . . 103
d) RMI registry, server and applet . . . 103
e) Applet execution . . . 105

5.3 Polymorphisme and RMI . . . 106
5.3.1 Principle and implications . . . 106
5.3.2 Result polymorphism . . . 107
5.3.3 Parameter Passing . . . 108

5.4 Example: RMI with polymorphisme . . . 109
5.4.1 Principle of the 2 interfaces . . . 109
5.4.2 Server implementation . . . 110
5.4.3 A client: compute Pi . . . 111

5.5 Callback, synchro, multithread, and RMI . . . 115
5.5.1 Model . . . 115
5.5.2 Asynchronous calls . . . 116
5.5.3 Call back . . . 117

CHAPITRE 6 Active Objects 119

CHAPITRE 7 Mobility and Formal Models 120

CHAPITRE 8 Hands-on Practical Session 121

CHAPITRE 1 Introduction

1.1 Definitions and architectures

1.1.1 Definitions

Concurrency:

Simultaneous access to a resource, physical or logical

examples:

memory, cache, disc sector, screen, printer, etc.

Parallelism:

Execution of several activities or precesses at the same time.

examples:

2 multiplications at the same time on 2 different processes.

Printing a file on two printer at the same time, saving on several discs (RAID: Redundant Arrays of Inexpensive Disks)

Distribution:

Several address spaces

examples:

2 PCs LAN, cluster of PC, WAN, GRID

1.1.2 Basic architectures

SISD: Single Instruction, Single Data

Classical sequential Machine,

Von Neumann

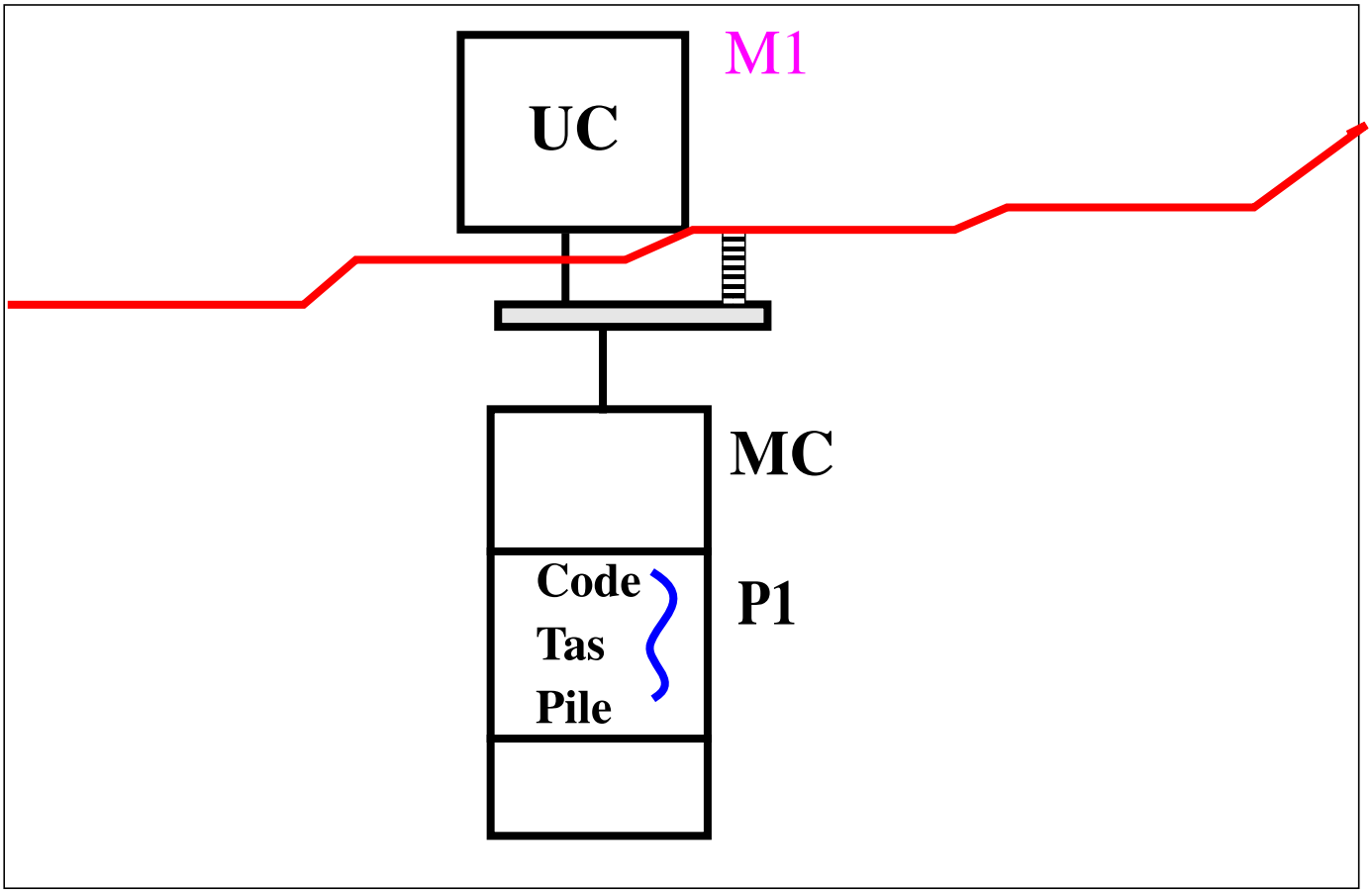


FIGURE 1 Machine séquentielle classique

Nowadays:

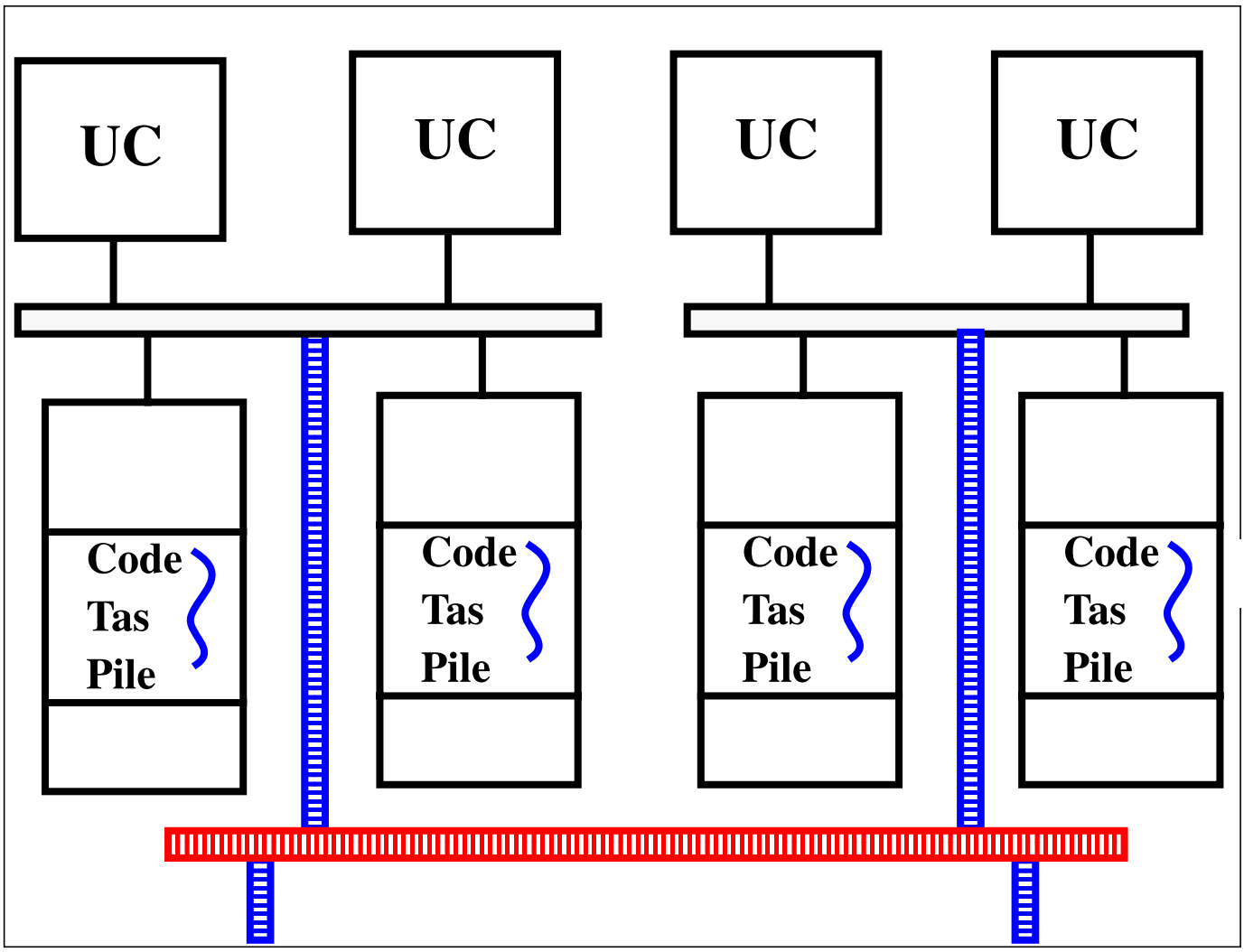


FIGURE 2 NUMA: Non Uniform Memory Access

Towards GRID computing:

Very large variation of bandwidth and latency

1.2 RPC: Remote Procedure Call

1.2.1 Principle

Using a procedure call to achieve an execution of some form on a remote machine

Passing of data, both way:

parameter,

result

Remote Procedure Call:

Depuis un processus (P1),
sur une machine (M1),

on va exécuter la procédure foo,
avec des paramètres param

sur une autre machine (M2),
(evt. dans un processus (P2))

on récupère éventuellement un résultat res

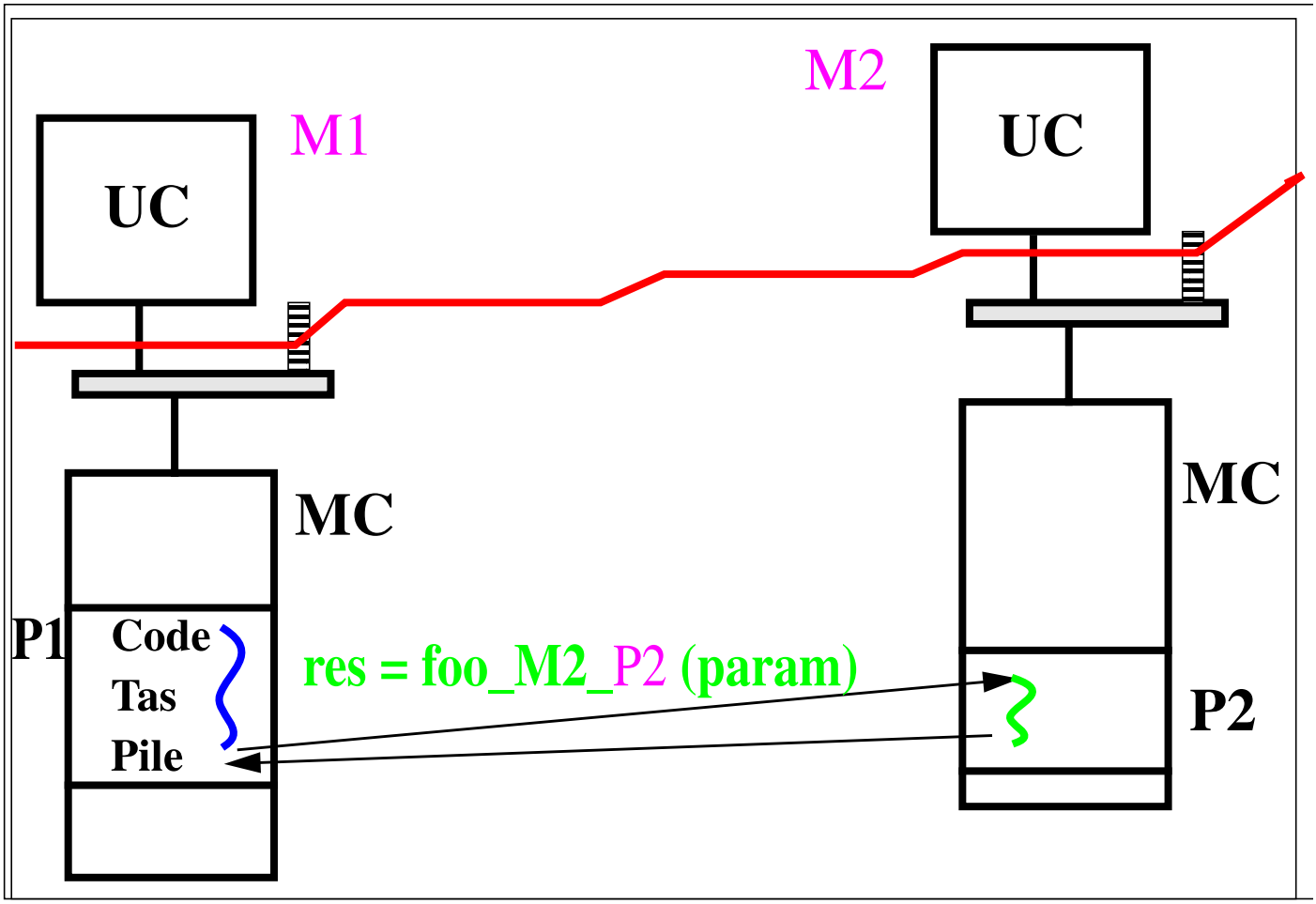


FIGURE 3 Principe d'un RPC

Plus structuré que simplement envoyer un message:

- On communique: paramètres
- On spécifie le code à exécuter: procédure
- On communique: résultats

Notes:

- NFS utilise pour son implémentation le RPC

- On utilise quelques fois le RPC même si il y a une mémoire unique:

  communications sans partage afin de protéger les espaces d'adressage (OS)

  Securité

1.2.2 RPC variations

a) Forme, syntaxe

Forme et syntaxe du RPC varient dans le code:

res = foo_M2 (param);

foo_M2_P2 (param, res);

res = P2.foo (param);

foo_ (Param, M2P2)

foo_ (Param, M2, P2)

Object:

Notation pointé pour l'appel distant:

remote_obj.foo (param1, param2);

Possibilité de retourner un résultat:

res = remote_obj.bar (param);

--> • - “Language centric"

b) Passage des paramètres

Sémantique par copie, par référence, etc.

c) Représentation des données

Intéropérabilité entres des machines incompatibles

Utilisation de XDR: External Data Representation

d) Sémantique de l'appel

Synchrone

Asynchrone

Futur

e) Gestion des erreurs

Panne ou congestion du réseau

Panne du client, du serveur

Avant ou pendant le traitement du RPC

Sémantique “at most one“:

Si OK, alors exécution 1 fois

Si erreur, alors 0 ou 1 fois (pas 2 ou plus)

Remontées des erreurs vers le client par des exceptions

f) Code sur machine distante

Comment est assurée la disponibilité du code sur la machine distante ?

--> • Code Mobile

--> • Java

--> • Serveurs d’applications

g) Découverte et nommage

Découverte et nommage :

des machines

des processus

des services (interfaces)

des objets accessibles

h) Terminaison et GC distribué

Comment savoir si un serveur peut être arrêté ?

Comment savoir si un objet accessible à distance n'est plus référencé par personne ?

i) Persistance des “objets“ ou processus distants

Le temps de la vie en mémoire:

résistence aux pannes

ou

Intrinsèque:

se trouvent sur un support de type disque, avec “activation“ (mise en mémoire) à la demande

j) Sécurité

Peut-on contrôler:

qu'un client donné a le droit d’appeler un serveur donné

qu'il a le droit d'accéder à un service donné

que personne n'intercepte les communications

On peut avoir:

du contrôle d'accès,

des protocoles d'authentification,

des protocoles de cryptage des communications : A, I, C,

--> • A : Authentification

--> • I : Intégrité

--> • C : Confidentialité

Une propriété souvent recherchée (pas simple à obtenir): NR

--> • Non Répudiation

Note: ces protocoles ne sont pas encore intégrés correctement avec des modèles généraux de RPC.

Recherches en cours (en particulier avec traitement de la mobilité du code et des calculs).

k) Autres aspects et concepts

Transactions

Migration du client sur la machine du serveur

Communications de groupe

Middleware à Messages (MOM, e.g. Java JMS):

Messages, queues de messages,

Messages vus comme des ressources transactionnelles

Communication événementielle:

événements, et réactions (traitement associé à l'occurrence d'un événement)

Anonymat: indépendance entre l'émetteur et les consommateurs d'un événement

...

1.3 N tiers applications in Business

See External material: N tiers Architecture

To be looked at quickly during the lecture,
to be studied by yourself between lecture 1 and 2.

1.3.1 Web Tiers and RPC

1.3.2 Middle Tiers and RPC

Summary and comparison to next section: GRID

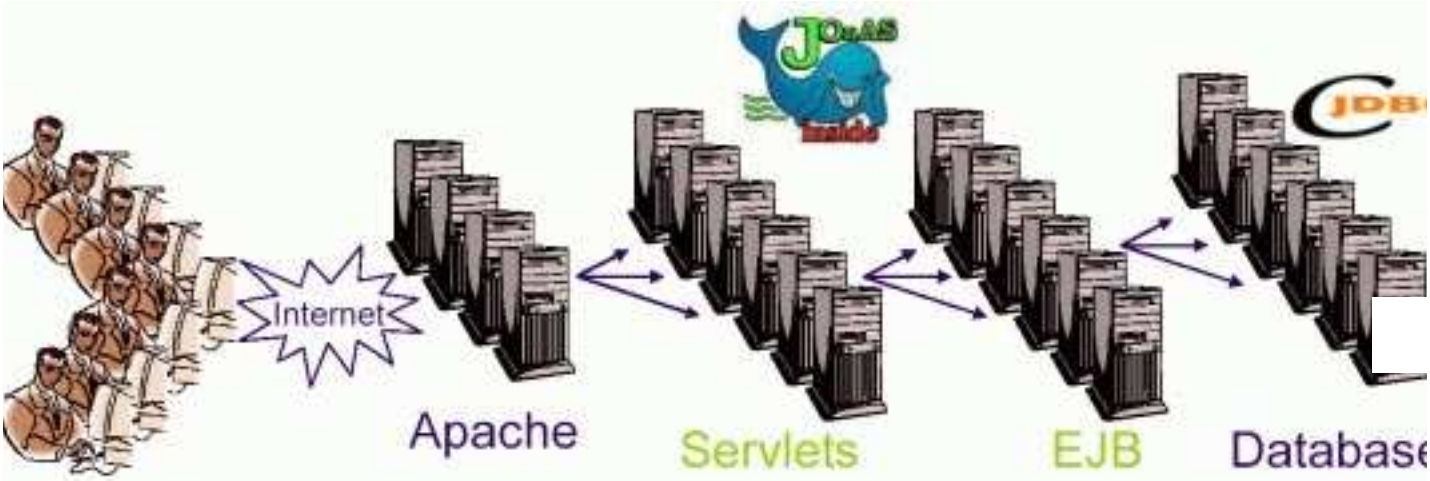


FIGURE 4 Clustering and Parallelism in N tiers

Courtesy of Emmanuel Cecchet <Emmanuel.Cecchet@inrialpes.fr>

1.4 GRID computing

1.4.1 Main ideas

GRID = electric network in the US

A gripping idea:

like electricity, computer cycles cannot be stored, if not used they are lost

So, the research community tries to put together:

--> .techniques
--> .architectures
--> .infrastructures
--> .operational services

allowing to use all those "lot cycles"

Not limited to cycles:

- Computational GRID
- Data GRID

A definition:

Grid is a parallel and distributed system that enables the use, sharing, selection, and aggregation of resources across multiple administrative domains based on their availability and capability.

An important idea:

on- demand access to computing, data, and services.

Social, administrative, and policy issues, stating that Grid computing is concerned with

- coordinated resource sharing and problem

- solving in dynamic, multi-institutional virtual organizations

Access to computers, software, data, but also other resource:

particull accelerator, visualization engine, network links, telescopes,...

Cost, and users' quality-of-service requirements, should be able to be taken into account...

1.4.2 Globes

tap://www.globus.org/

The Globes Toolkit

The toolkit addresses issues of

- --> .security,
- --> .information discovery,
- --> .resource management,
- --> .data management,
- --> .communication,
- --> .fault detection,
- --> .and portability.

A needed definition:

Soft state protocols:

use **periodic refresh messages** to keep distributed state alive while conditions change

This has raised concerns regarding the scalability of protocols that use the soft-state approach.
In existing soft state protocols, the values of the timers that control the sending of these messages, and the timers for age-ing out state, are very important.

Most of time determined empirically, sometimes with performance modeling and evaluation.

The main toolkit components are:

Grid Resource Allocation and Management (GRAM) protocol:

a gatekeeper service provides for secure, reliable, service creation and management

Meta Directory Service (MDS-2):

for information discovery through soft state registration data modeling, and a local registry (GRAM reporter)

Grid Security Infrastructure (GSI),

supports single sign on, delegation, and credential mapping.

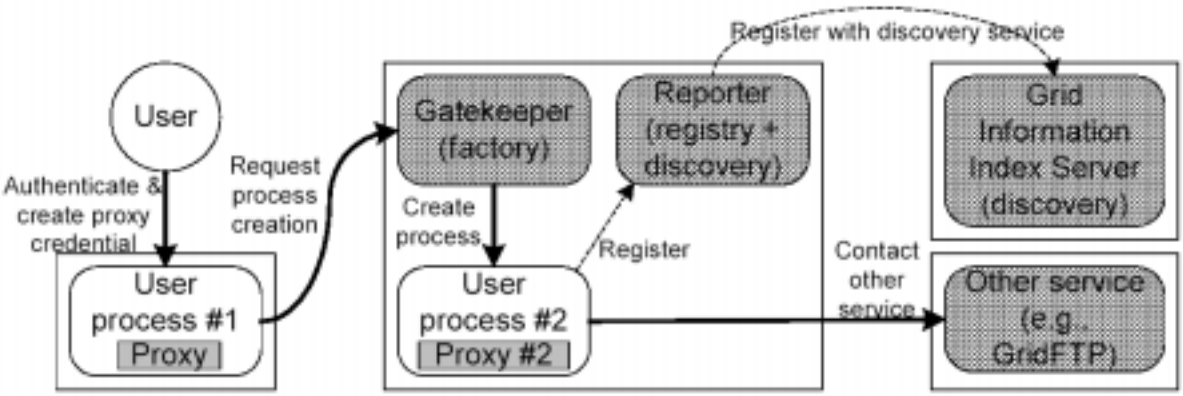


Figure 1: Selected Globus Toolkit mechanisms, showing initial creation of a proxy credential and subsequent authenticated requests to a remote gatekeeper service, resulting in the creation of user process #2, with associated (potentially restricted) proxy credential, followed by a request to another remote service. Also shown is soft-state service registration via MDS-2.

FIGURE 5 Globus Toolkit GRAM and security

Also: MPI version for Globus.

Initial work by:

Gropp and E. Lusk and N. Doss and A. Skjellum

and Argonne group

A high-performance, portable implementation of the MPI message passing interface standard.

MPICH is a freely available, portable implementation of MPI, a standard for message-passing libraries.

MPICH-G2 is a grid-enabled implementation of the MPI v1.1 standard. That is, using Globus services (e.g., job startup, security).

MPICH-G2 allows you to couple multiple machines, potentially of different architectures, to run MPI applications.

MPICH-G2:

automatically converts data in messages sent between machines of different architectures,

supports multiprotocol communication by automatically selecting TCP for intermachine messaging and (where available) vendor-supplied MPI for intramachine messaging.

1.4.3 WS GRID: OGSA -- OGSI from GGF

Global Grid Forum:

http://www.gridforum.org/

The Global Grid Forum (GGF) is a community-initiated forum of individual researchers and practitioners working on distributed computing "grid" technologies.

GGF is the result of a merger of the Grid Forum, the eGrid European Grid Forum, and the Grid community in Asia-Pacific.

Global Grid Forum have a major effort underway to define the :

Open Grid Services Architecture (OGSA)

Open Grid Services Infrastructure (OGSI)

http://www.ggf.org/ogsa-wg/

http://www.ggf.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf

http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf

Managing the Grid only with (extended) Web Services

Recent News (end Sept. 03):

Oracle consortium for Business Grid

GSH and GSR:

Managing Grid WS:

OGSA services can be created and destroyed dynamically. Services may be destroyed explicitly, or may be destroyed or become inaccessible as a result of some system failure.

Because Grid services are dynamic and stateful.

GSH:

> A Grid service instance is assigned a globally unique name, the Grid service handle (GSH),

GSH carries no protocol- or instance-specific information such as network address and supported protocol bindings. This information is encapsulated within a GSR:

> Grid service reference (GSR).

Unlike a GSH, which is invariant, the GSR(s) for a Grid service instance can change over that services lifetime. A GSR has an explicit expiration time, or may become invalid at any time during a services lifetime.

Allows to deal with failure, installation of new version of the same service, restart of a service, etc.

Needed: Fct: GSH --> GSR

Open Grid Service Infrastructure (OGSI)

The objective of the OGSI Working Group is to review and refine the Grid Service Specification and other documents that derive from this specification, including Open Grid Service Architecture (OGSA) infrastructure.

http://www.gridforum.org/ogsi-wg/drafts/GS_Spec_draft03_2002-07-17.pdf

http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf

... on going ..

References with many references on GRID research:

http://www.gridcomputing.com/

1.5 From Object to (Grid) Components

External material: B2G Components

To be looked at quickly during the lecture,
to be studied by yourself between lecture 1 and 2.

1.5.1 Fundamental of Components

1.5.2 Business to Grid Components

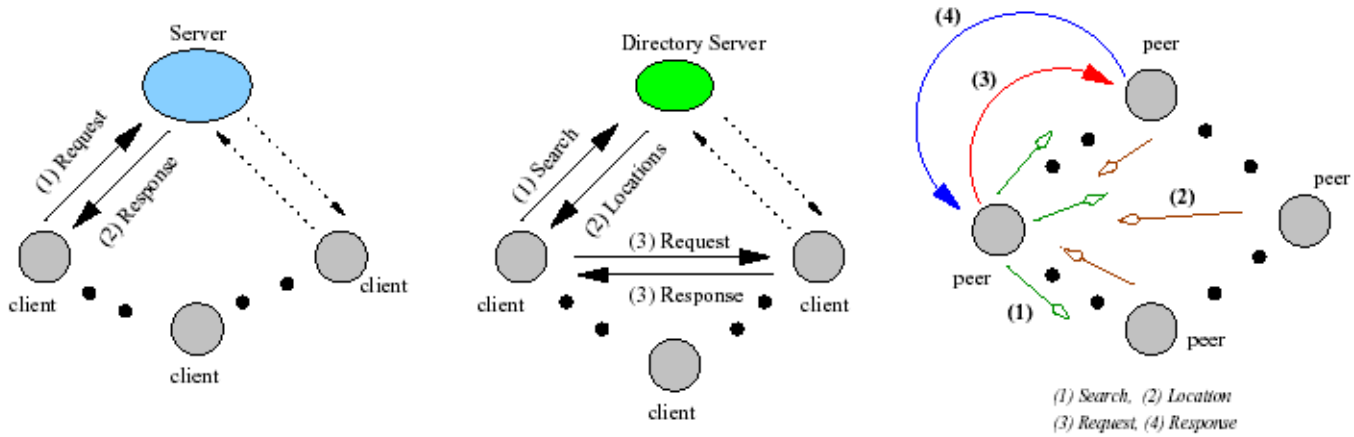
1.6 Current hype (?): P2P

1.6.1 Data P2P, Principles:

Definition:

a distributed system where each node or executing task is in a Peer relation to others

Equal, match, a person who is of equal standing with another in a group



(a) Traditional Client–Server Computing (b) Napster–style Computing (c) Gnutella/Freenet–style Computing

FIGURE 6 Example of File Sharing approaches

Mainly use for storing data in a distributed manner:

Napster (MP3), Gnutella (Files, pure distributed), Freenet (sharing of storage, security, anonymity), ...

In a pure form, no servers, or dedicated routers

From: On the Potential of Peer-to-Peer Computing

Krishna Kant, Ravi Iyer, http://www.comp.nus.edu.sg/~ngws/p2p/On%20the%20Potential%20of%20Peer-to-Peer%20Computing.pdf

Three important characteristics of Peer-to-Peer group members.

1. Have an operational computer of server quality.
2. Have an addressing system that is independent of the DNS.
3. Able to cope with variable connectivity.

HYBRID: Some nodes are Router-terminals that facilitate the interconnections between Peers.

PURE: All nodes are Peers, and each Peer may function as router, client, or server, according to the status of the query.

Note that the common telephone connections are close to this model (P2P), but not pure P2P:

A usual telephone communication requires a central exchange, or set of central exchanges to make and maintain the connection between (dumb) terminals.

1.6.2 A few attributes and Research Issues

Environment Attributes:

1. Network latency:

Ranges from uniformly low (e.g., for a high-speed LAN) to highly variable (e.g., for general WAN).

2. Security concerns:

Ranges from low (e.g., corporate intranet) to high (e.g., public WAN).

3. Scope of failures:

Ranges from occasional isolated failures (e.g., a laboratory network of workstations) to frequent failures, possibly including massive failures that result in network partitions.

4. Connectivity:

Ranges from always-on (e.g., nodes in a business LAN) to occasional-on (e.g., laptops and other mobile devices).

5. Heterogeneity:

Ranges from complete homogeneity to complete heterogeneity (in hardware, O/S, protocol stack, services and application interfaces).

6. Stability:

Ranges from highly stable (i.e., occasional changes/upgrades that are known in advance) to unstable (i.e., sudden upgrades/changes to hardware/software that the P2P application is unaware of).

Research issues in P2P:

1. *Location and addressing* mechanisms for finding desired agent(s).

2. Schemes for *multiparty communication and synchronization*.

3. Protocol for setting up appropriate streaming multimedia channels.

4. QoS aware resource allocation and scheduling.

5. Mechanisms for handling migration of resources and the performance impact of migration.

6. Mechanisms for *tolerance against failures* and intermittent connectivity.

7. *Security mechanisms* in a mutually suspicious environment and secure multiparty collaboration.

8. Application interfaces that work robustly in a heterogeneous environment.

1.6.3 Computational P2P

Idea:

use 100 000 PCs to compute something

Example:

Seti@Home

SETI@home is a scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI).

Participate by running a free program that downloads and analyzes radio telescope data.

Example of architecture:

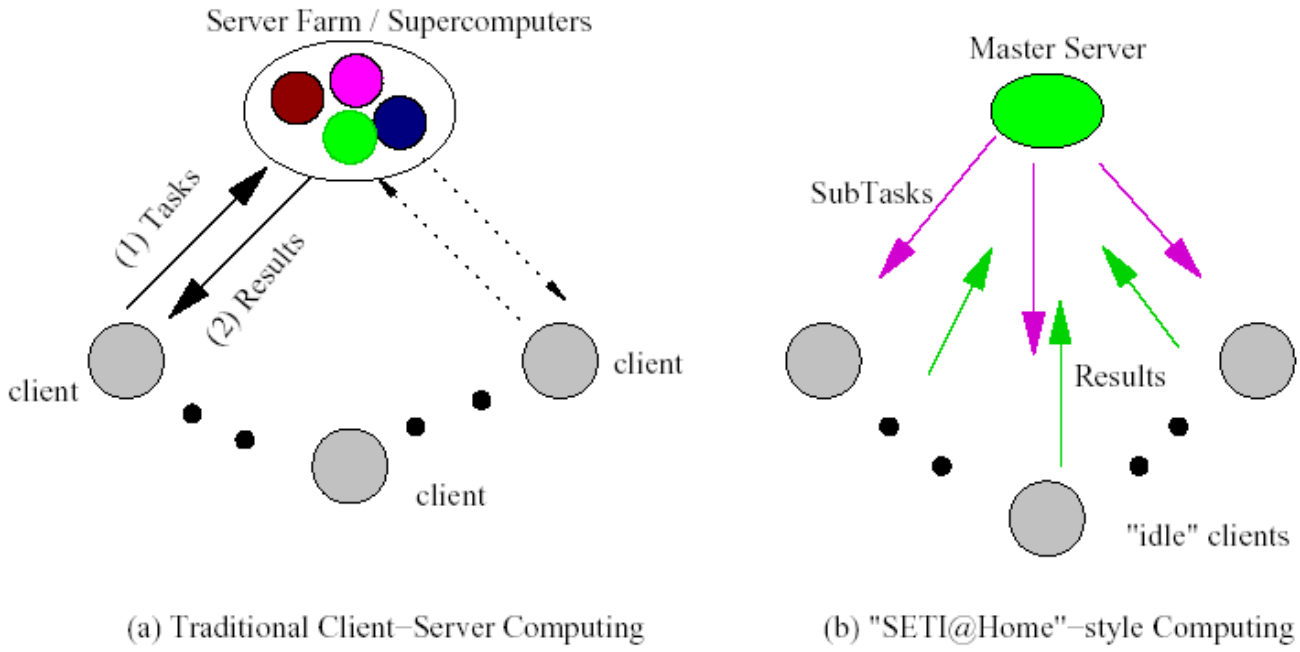


FIGURE 7 Compute Service Approaches

In my opinion, 2 crucial problems:

For Data P2P

The addressing function and system

For Computational P2P

Pure P2P system where there is no centralized server, and peers can really communicate with each other.

What programming abstraction for those systems ?

Imagine:

One could start a computation that ... nobody could stop ...

Definition: Intranet P2P:

Computational P2P running within an institution to tap from desktop cycles when not in use.

Companies try to do it but only Master/Slave so far.

E.g. Entropia, PC GRID Computing, http://www.entropia.com/,

A few references:

A Definition of Peer-to-Peer Networking for the Classification of Peer-to- Peer Architectures and Applications

Rüdiger Schollmeier, Institute of Communication Networks, Technische Universität München, 80333 München, Germany, Schollmeier@ei.tum.de

http://www.computer.org/proceedings/p2p/1503/15030101.pdf

A Framework for Classifying Peer to Peer Technologies,

Krishna Kant, Ravi Iyer (Enterprise Architecture Lab, Intel Corporation), Vijay Tewari (E-Business Solutions Lab, Intel Corporation)

http://kkant.ccwebhost.com/papers/taxonomy.pdf

An interesting presentation of P2P:

http://www.she.de/images/SHE/Internet/P2P_Computing_beyond_Napster.pdf

1.7 Conclusion, Research Perspectives

Open issues:

- GRID, a lot of infrastructure,
  concepts or theorem .... ?
- Middleware: adaptability, auto-adaptability
- (Static) verification of (partly generated, and adaptable) distributed systems.
- Components that compose (semantics, and performance)
- Computational P2P,
  Transaction and checkpointing
- Behavioral specification of components
  Model Checking (of asynchronous systems)

... more to come ...

La programmation distribuée est en train de devenir de plus en plus une composante incontournable des logiciels.

Commerce Electronique, Portails de Calculs, Cartes à Puces et Terminaux, Applications collaboratives, Jeux distribués, etc.

C'est également en passe de devenir bien plus sophistiqué qu'une simple interaction Client-Serveur.

De moins en moins de Clients, et de Serveurs purs, mais des processus, voire des objets distribués, qui communiquent, inter-agissent en s'échangeant des références, font des callbacks, etc.

Certains aspects vont prendre de l'importance:

- *Intégration plus fine au modèle N 1/3*
- *Interaction entre la programmation distribuée et les Composants*
- *Sécurité*

1.8 Waiting for structuring ...

Asynchronous Model:

Parallel Execution,

Communication,

Synchronization

Sharing/serialization, DSM, Model of computation, properties.

Dynamic Code generation, MOP, AOP

Transport Layer,

Infrastructure:
Discovery, Registry, Code up-and-Downloading,

WSDL, WSFL, SOAP,

Jini

Pattern for Distributed Programming:

Acceptor et Connector

Thread par session

Thread par requête

Objets actifs vs. Moniteur

Services asynchrones

Migration (mobilité)

Design and conception: MDA , UML, etc.

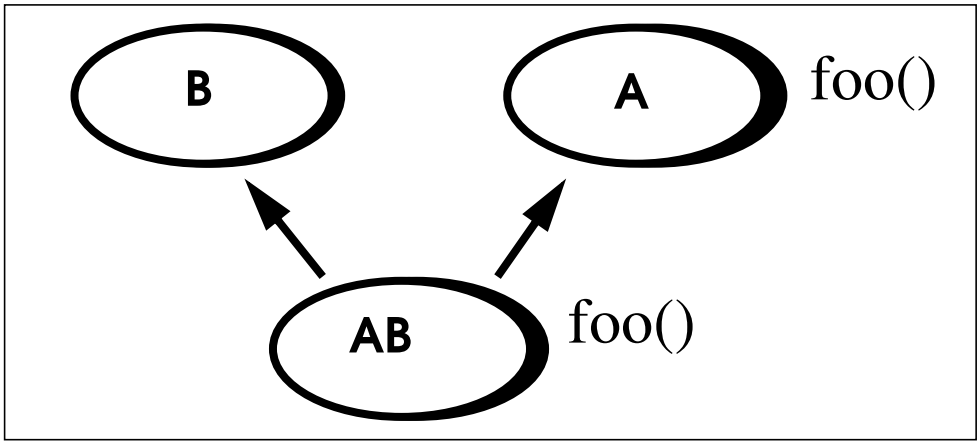
CHAPITRE 2 Objects for

Distribution: Basic Principles

2.1 Sequential features being usefull

Class Language, typed

2.1.1 Inheritance



Simgle, multiple, repeated

Interface (Java), Delegation

Redefinition (overriding ou overwriting),

Overloading, Renaming

Co-variance, No-variance, contra-variance

2.1.2 Polymorphism

Static type, Dynamic Type (entity, variable)

Object type (neither static, neither ni dynamic)

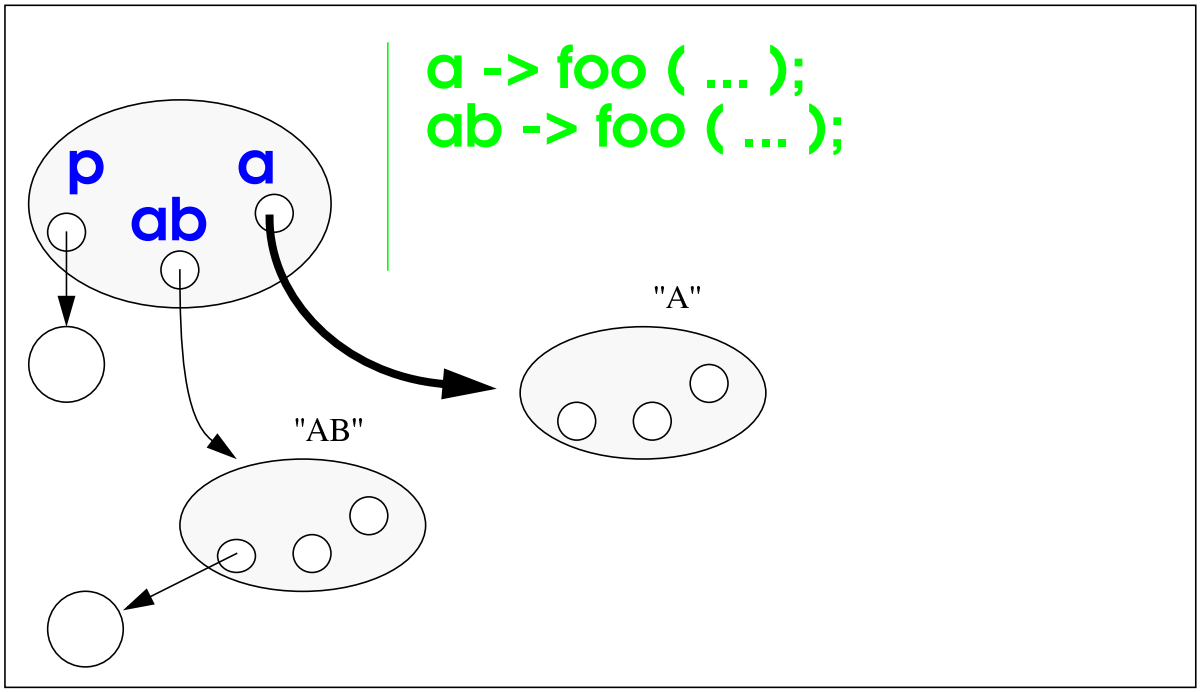
Polymorphic assignment

Reverse Polymorphic Assignment (?=),

dynamic cast

2.1.3 Dynamic Binding

The routine body is decided dynamically, depending on the dynamic type of the object.

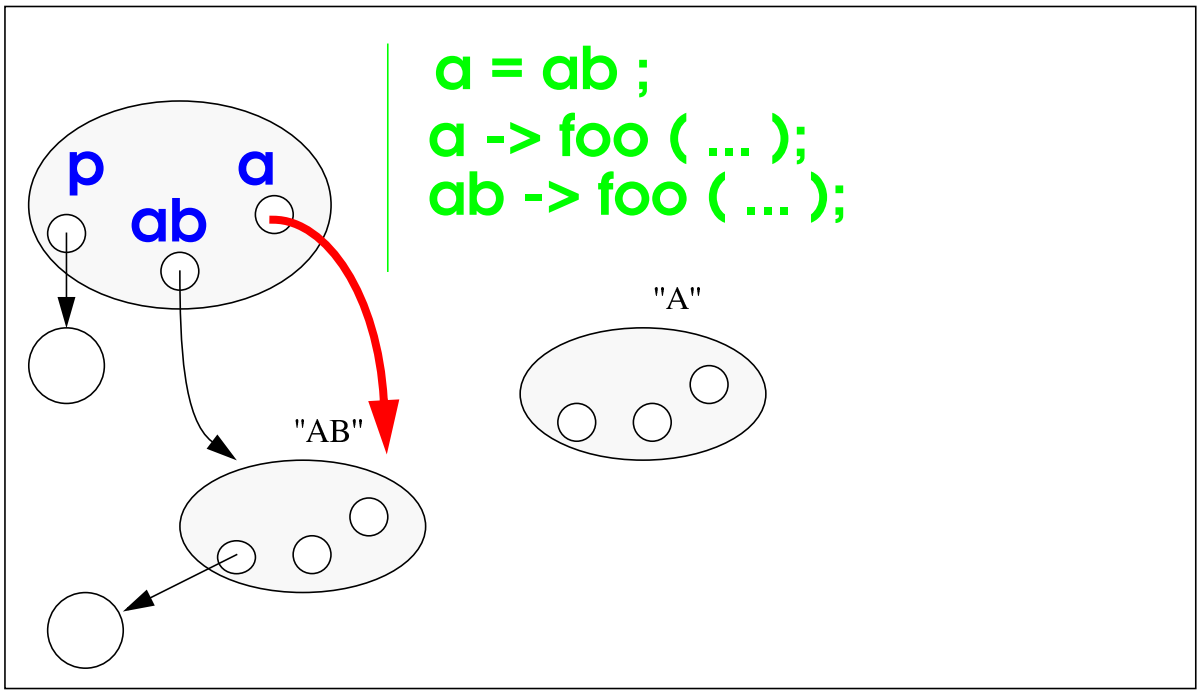


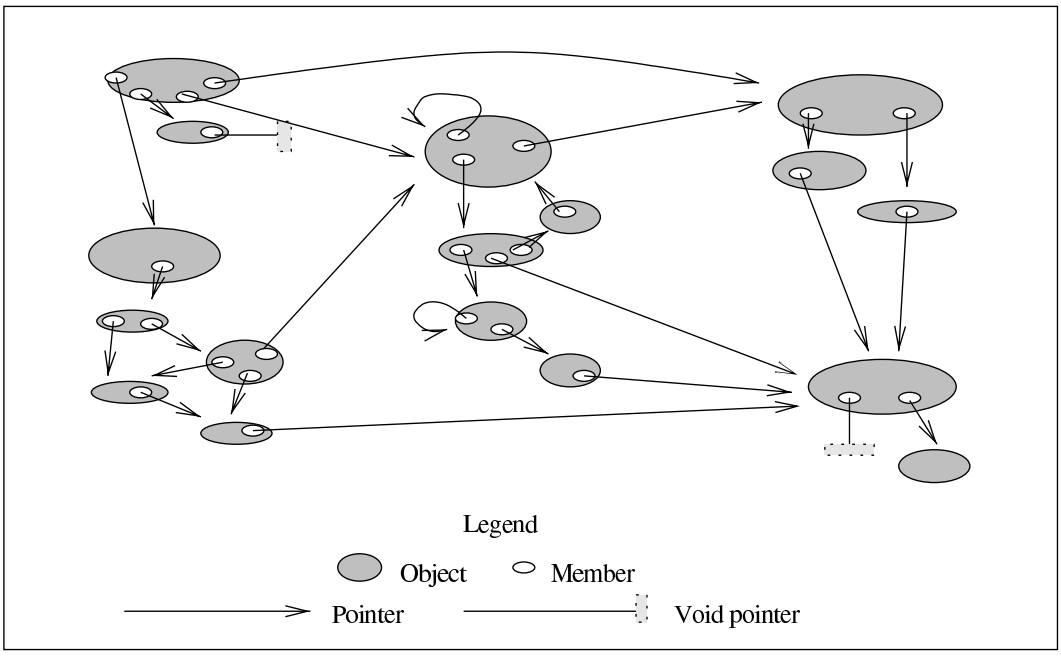
After a Polymorphic assignment

2.1.4 Sequential system at execution

A single activity (“processus”, “thread”):

the root object of the system

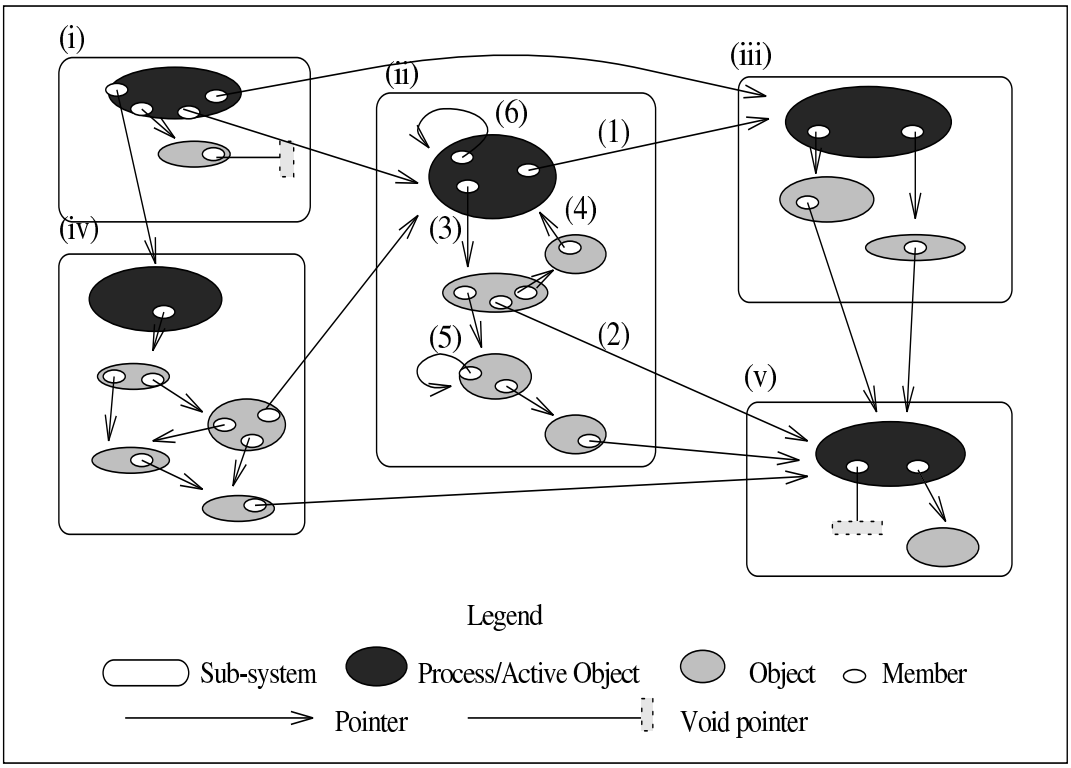




Wih method calls: the activity “go through” one object to another (execution stack)

Sometimes, what is just a method call, is named a “message sending” (e.g. Smalltalk), but the semantics is still a standard sequantial execution.

Question: how to get several activities ??



2.2 Objects and Activities

A first solution that is close to thread and not really object-oriented.

2.2.1 Parallelism orthogonal to objects

Parallel activities are orthogonales to objects).

Usually very close to the Monitor + Threads

This is the case for Java threads, or other languages such as Hybrid, ConcurrentSmalltalk

2.2.2 Parallelism per Active Objects

Each activity is associated to an active object.
An activity is an objet (active)

True object-oriented distribution and parallelism.

Two main categories of active object models :

Uniforme Models, or Homogeneous:

All objects are active

e.g. Actors

Non-uniform Models:

Only some objects are active

Parallelism:

2.2.1 orthogonal to objects

2.2.2 per Active Objects

Classification:

- **Java Thread:** 2.2.1

- **Java RMI:** in between 2.2.1 and 2.2.2

- **ProActive:** 2.2.2, Non-uniform Models

CHAPITRE 3 Reminder: Java

Monitors

3.1 Basic ideas

Threads + Moniteur (objets)

Principes:

1 lock / objet

1 lock / classe

ou 1 lock par block

D'après:

Concurrent Programming in Java,
© 1996 Doug Lea, Addison-Wesley,
October 1996.

http://gee.cs.oswego.edu/dl/

3.2 Primitives

3.2.1 Thread Control Methods:

suspend();

(itself or other, but deprecated)

temporarily halts a thread in a way that will continue normally after a (non-suspended) thread calls resume on that thread.

T.resume();

(other, but deprecated)

Resume a suspended thread.

sleep (ms);

(itself)

causes the thread to suspend for a given time (specified in milliseconds) and then automatically resume.

The thread might not continue immediately after the given time if there are other active threads.

thread.join()

suspends the caller until the target thread completes (that is, it returns when isAlive is false).

A version with a (millisecond) time argument returns control even if the thread has not completed within the specified time limit.

thread.interrupt()

causes a sleep, wait, or join to abort with an InterruptedException, which can be caught and dealt with in an application-specific way.

(deprecated)

3.2.2 Synchronization

Synchronization is implemented by exclusively accessing the underlying and otherwise inaccessible internal lock:

sometimes called a mutex, monitor

that is associated with each Java Object (including Class objects for statics).

Each lock acts as a counter.

If the count value is not zero on entry to a synchronized method or block because another thread holds the lock, the current thread is delayed (blocked) until the count is zero.

On entry, the count value is incremented. The count is decremented on exit from each :

synchronized method or

block,

even if it is terminated via an exception.

Waiting and Notification

The methods:

wait, notify, and notifyAll

may be invoked only when the synchronization lock is held on their targets.

This is normally ensured by using them only within methods or code blocks synchronized on their targets. Compliance cannot usually be verified at compile time.

Failure to comply results in an IllegalMonitorStateException at run time.

A wait invocation results in the following actions:

- The current thread is suspended.
- The Java run-time system places the thread in an internal and otherwise inaccessible wait queue associated with the target object:

  Dormant

- The synchronization lock for the target object is released, but all other locks held by the thread are retained. (In contrast, suspended threads retain all their locks.)

A notify invocation results in the following actions:

- If one exists, an arbitrarily chosen thread, say T, is removed by the Java run-time system from the internal wait queue associated with the target object (Dormant).
- T must re-obtain the synchronization lock for the target object, which will always cause it to

block at least until the thread calling notify releases the lock.

Dormant ---> blocked

It will continue to block if some other thread obtains the lock first.

- T is then resumed at the point of its wait.

A notifyAll invocation:

works in the same way as notify except that the steps occur for all threads waiting in the wait queue for the target object.

Two alternative versions of the wait method take arguments specifying the maximum time to wait in the wait queue. If a timed wait has not resumed before its time bound, notify is invoked automatically.

If an interrupt occurs during a wait, the same notify mechanics apply except that control returns to the catch clause associated with the wait invocation.

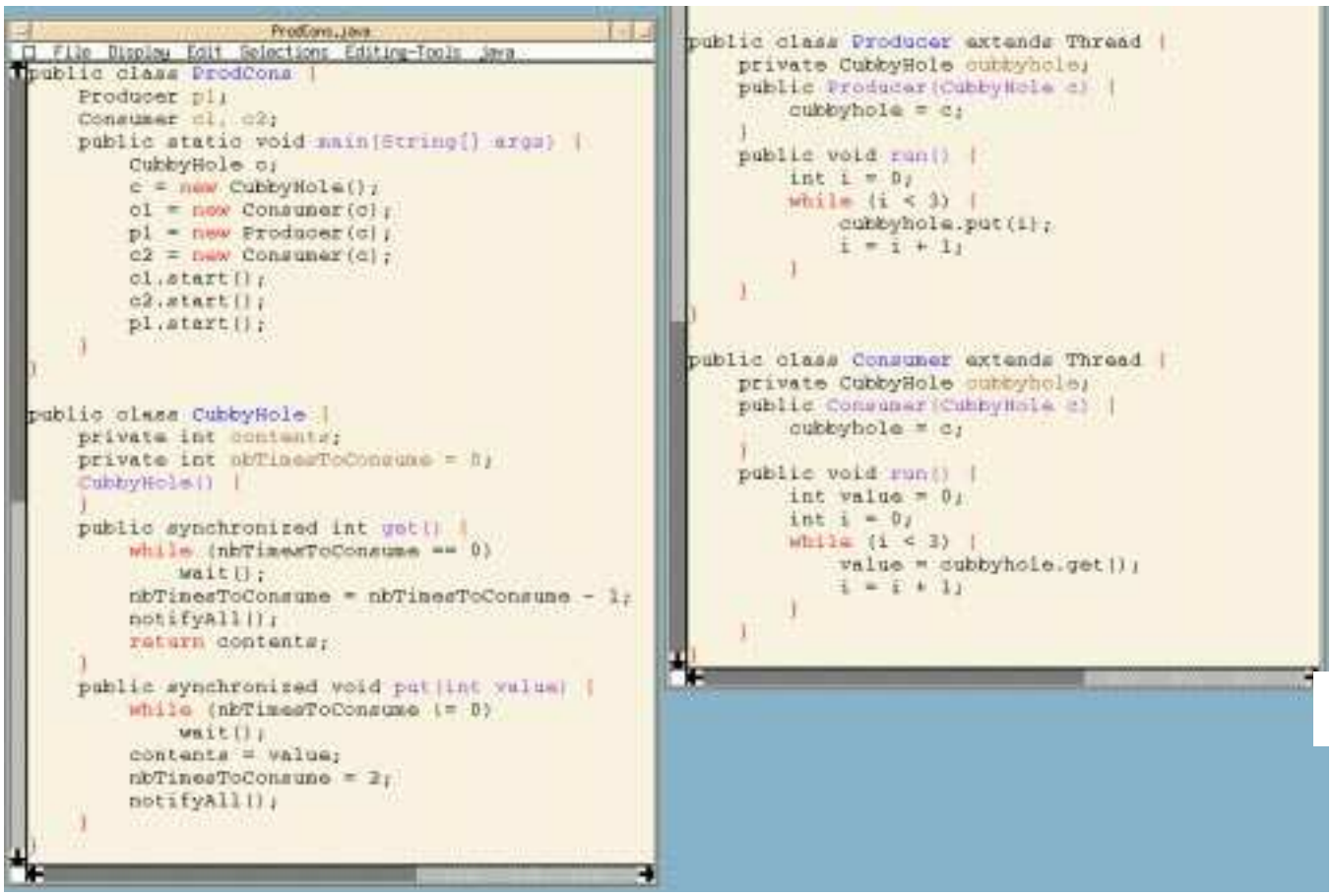
Récapitulatif:

Wait --> Dormant, perte du lock

notify --> choisi un Dormant -> blocked
le lock n'est pas perdu

example:

3.3 Example: Produser/Consumer in Java



```
ProdCons.java
File Display Edit Selections Editing-Tools Java
public class ProdCons {
    Producer p1;
    Consumer c1, c2;
    public static void main(String[] args) {
        CubbyHole c;
        c = new CubbyHole();
        c1 = new Consumer(c);
        p1 = new Producer(c);
        c2 = new Consumer(c);
        c1.start();
        c2.start();
        p1.start();
    }
}

public class CubbyHole {
    private int contents;
    private int nbTimesToConsume = 0;
    CubbyHole() {
    }
    public synchronized int get() {
        while (nbTimesToConsume == 0)
            wait();
        nbTimesToConsume = nbTimesToConsume - 1;
        notifyAll();
        return contents;
    }
    public synchronized void put(int value) {
        while (nbTimesToConsume != 0)
            wait();
        contents = value;
        nbTimesToConsume = 2;
        notifyAll();
    }
}
```

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    public Producer(CubbyHole c) {
        cubbyhole = c;
    }
    public void run() {
        int i = 0;
        while (i < 3) {
            cubbyhole.put(i);
            i = i + 1;
        }
    }
}

public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    public Consumer(CubbyHole c) {
        cubbyhole = c;
    }
    public void run() {
        int value = 0;
        int i = 0;
        while (i < 3) {
            value = cubbyhole.get();
            i = i + 1;
        }
    }
}
```

FIGURE 8 Produser / Consumer in Java

Cubby Hole

DLProdCons.java

File Display Edit Selections Editing-Tools java

```
public class DLProdCons {
    public static void main(String[] args) {
        Producer prod1, prod2;
        InterConsumer ic1, ic2;
        Consumer cons1, cons2;
        CubbyHole ch1, ch2;
        ch1 = new CubbyHole(2);
        ch2 = new CubbyHole(2);
        ic1 = new InterConsumer(ch2, ch2);
        cons1 = new Consumer(ch2);
        cons2 = new Consumer(ch2);
        ic2 = new InterConsumer(ch2, ch2);
        prod1 = new Producer(ch1);
        prod2 = new Producer(ch1);
        ic1.start();
        cons1.start();
        cons2.start();
        ic2.start();
        prod1.start();
        prod2.start();
    }
}

public class InterConsumer extends Thread {
    private CubbyHole consCubbyhole;
    private CubbyHole prodCubbyhole;
    public InterConsumer(CubbyHole consC, CubbyHole prodC)
        consCubbyhole = consC;
        prodCubbyhole = prodC;
    }
    public void run() {
        int value = 0;
        int i = 0;
        while (i < 3) {
            value = consCubbyhole.get();
            prodCubbyhole.put(value);
            i = i + 1;
        }
    }
}
```

FIGURE 9 Cubby Hole program

Cubby Hole execution: deadlock

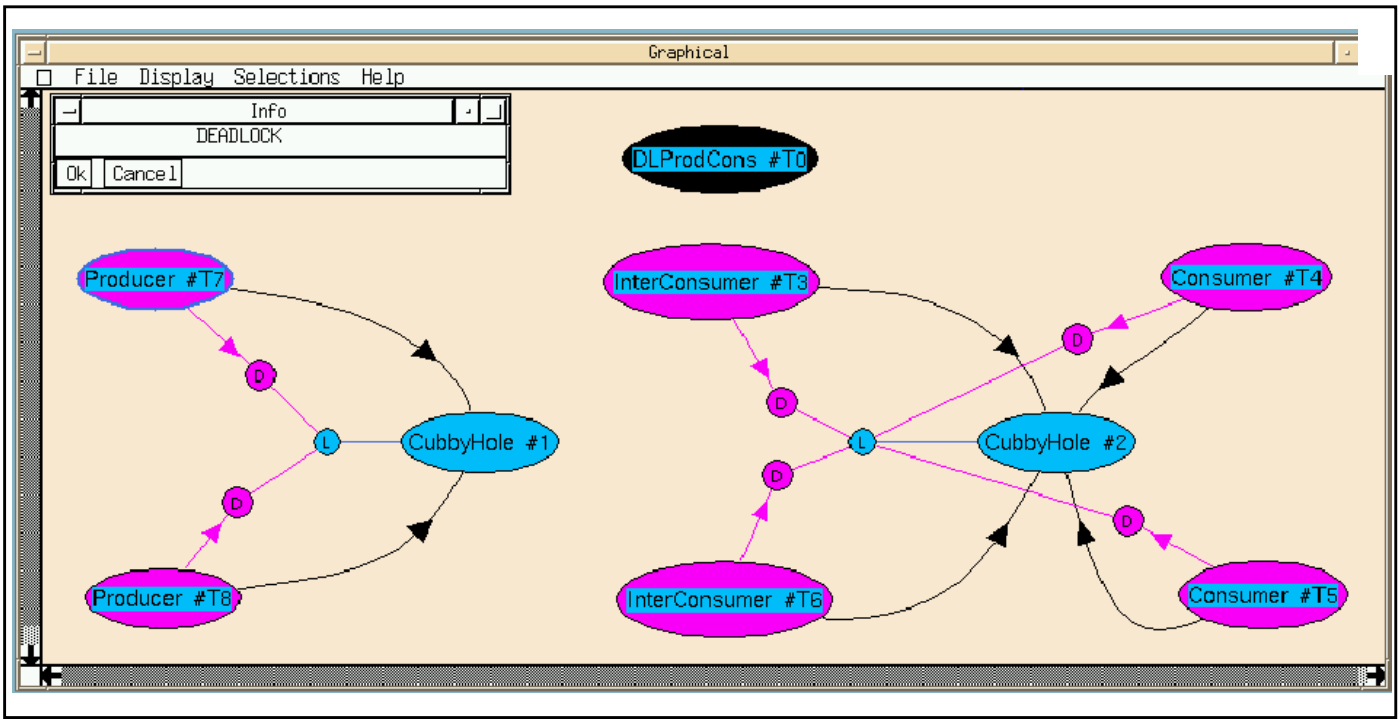


FIGURE 10 Cubby Hole execution: deadlock

Cubby Hole normal execution

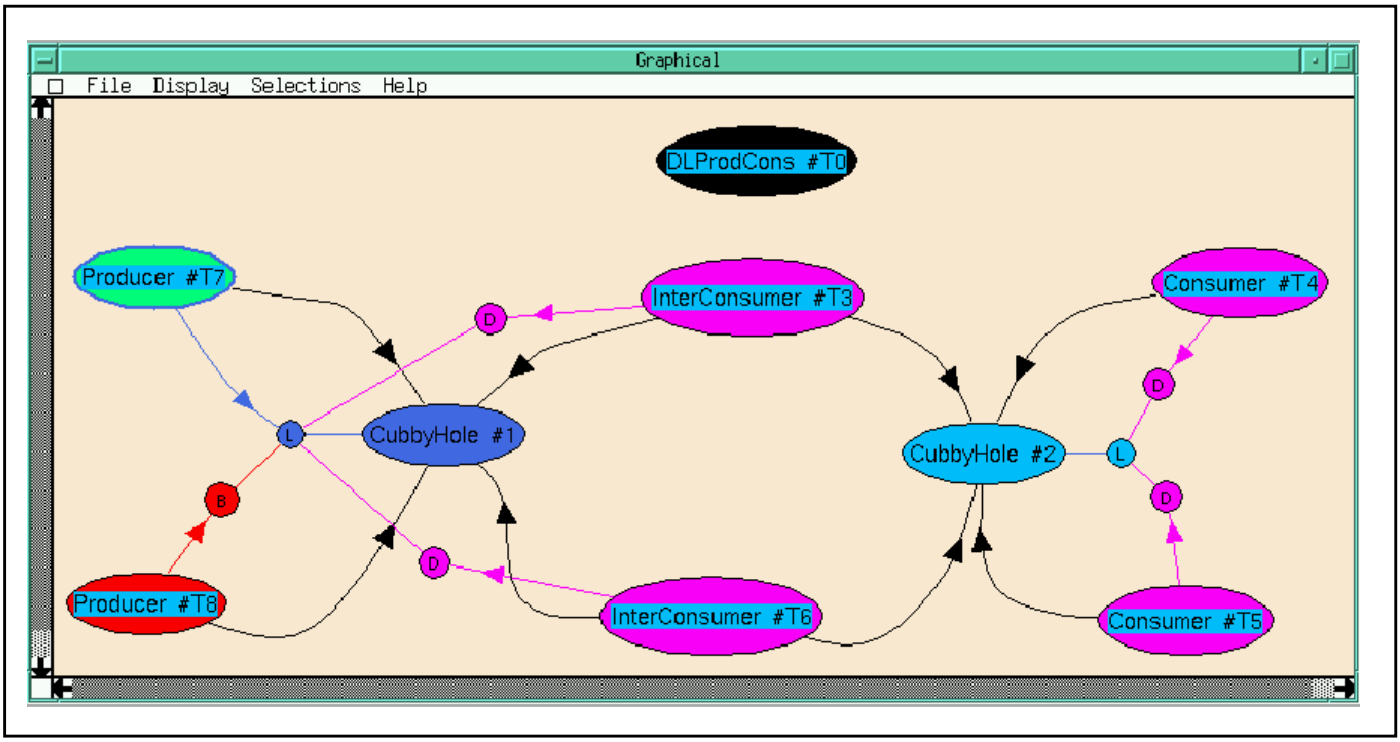


FIGURE 11 Cubby Hole normal execution

3.4 Conclusions on monitors

Bas niveau :

- Sémantique de Signal assez difficile à saisir, de plus des variations existent.
- Signal is quelquefois utilisé à des fins d'optimisation.
- Les programmes réalisés with des moniteurs deviennent très rapidement très complexes.

mais utiles:

- pas vraiment mieux quand on doit gérer explicitement de la mémoire partagée: Système d'Exploitation (multi-tâches, multi-utilisateurs, multi-processeurs, multi-threading)

Extensions quelquefois utilisées:

cond.*Queue* ;

- Retourne vrai s'il existe des processus en attente sur la condition

Nécessaire pour un lecteur-rédacteurs si l'on veut contrôler (un peu) les priorités

CHAPITRE 4 Reminder: Java RMI

4.1 Basic Principles

RPC within Java

4.1.1 Fonctionnalities

synchronous RPC

Dot notation for remote calls:

remote_obj.foo (param1, param2);

Returning a result:

res = remote_obj.bar (param);

--> • - “Language centric"

Marker interface

Copy of objects (DAG, cycles):

--> • Serialization, deep copy

Security, Applets, and
Persistant remote objects (activable)

Generation of stubs and skeletons

Transport: TCP,
but other tranports can be used (UDP, etc. with Socket Factory).

4.1.2 Architecture

Cassical diagram :

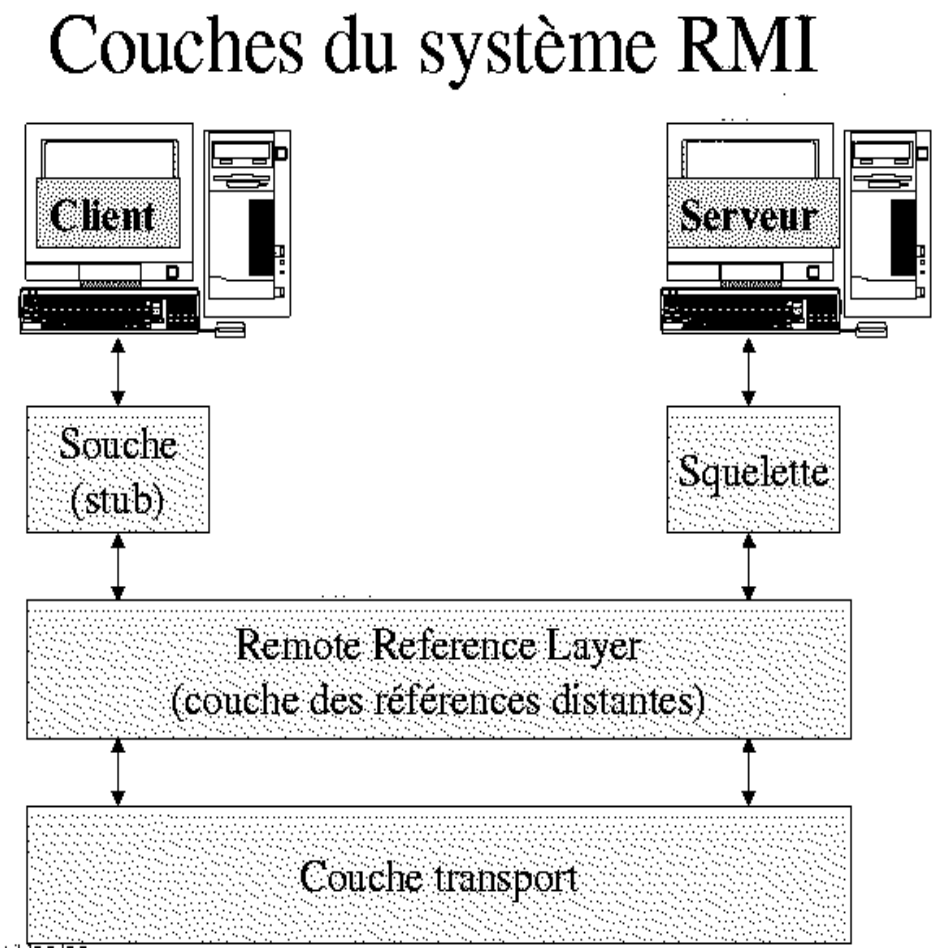


FIGURE 12 RMI Layers

Actually using :

- Java serialization
- Byte code for the interoperability

RMI has its own local "directory"(rpcbind)

rmiregistry

RMI as its own "directory" global (LDAP, CDS):

Jini (Note: +/- also usable : JNDI)

4.1.3 How it works

Principles at execution:

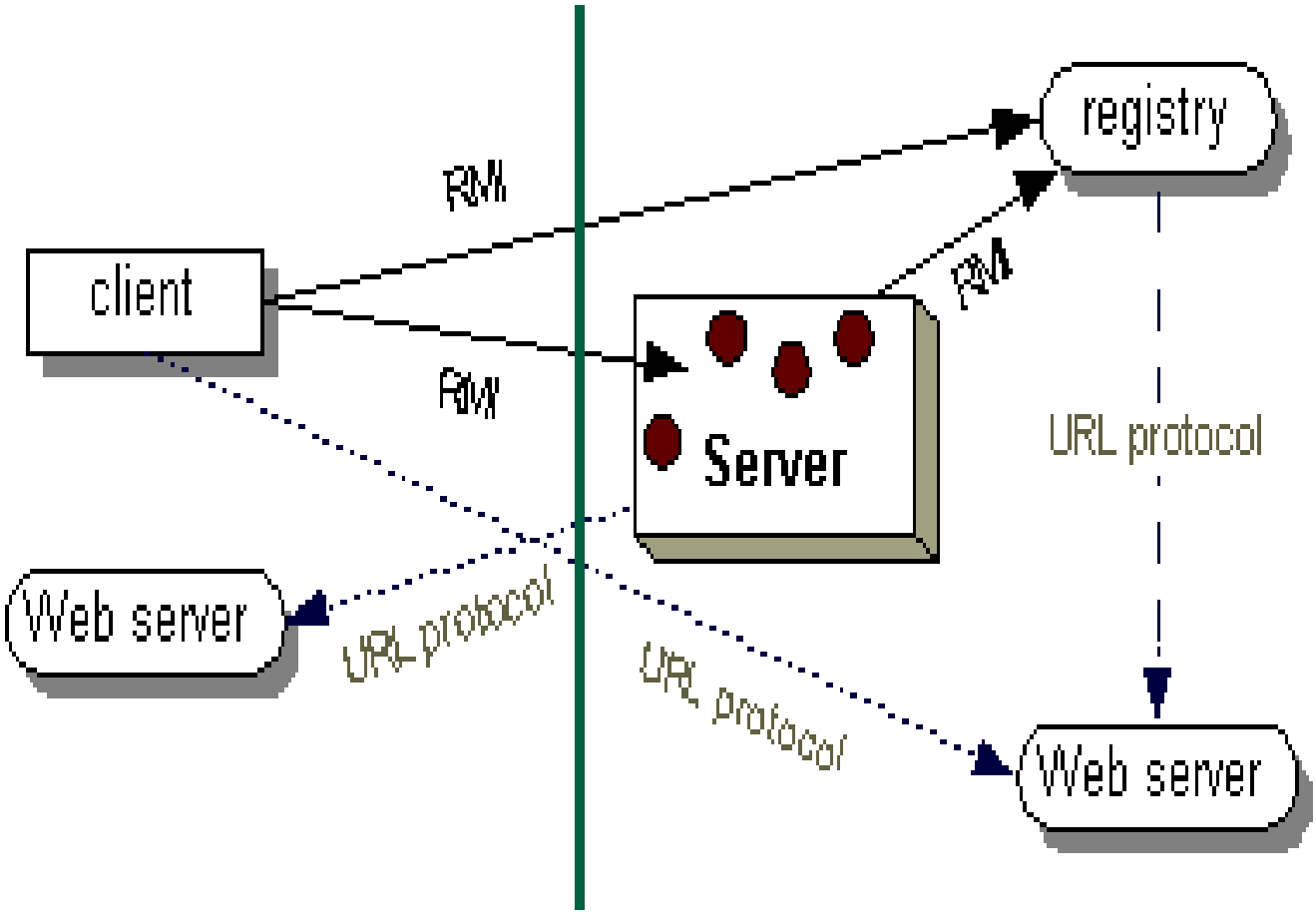


FIGURE 13 RMI: principles at exécution

Using a "registry" to associate a name to a reference towards a remote object

Web server (HTTP protocol) to dynamically upload, or download byte code if necessary:

from client to server, or server to client.

4.1.4 Differences with RPC

Main differences:

- syntax (dot notationvs. procedure with “handle“
- Only TCP (not really UDP for now)
- but one can used the “secured sockets“

4.1.5 Communication Model

Semantics of parameter passing:

- primitif types (int, double, ...) are always sent by value
- objects, that are not “Remote“, are passed by deep copy
  Serialization: trees, DAGs, any graphes even with cycles
- “Remote“ object are passed by reference
  a stub on the remote object is actually sent.

--> example on board ...

4.1.6 Other characteristics

+

Distributed GC

Some integration with Exceptions
(only functionnal exceptions, not with what I call the “non-functionnal exceptions”)

Dynamic code loading

-

Non-integration, unification with threads

No service policy that can be expressed for a server object (FIFO, buffer, reader-writer, etc.)

Il One has to deal with synchronize routine if needed.

4.2 Classes and Interfaces

For the standard user, the API mainly offers:

- an interface: java.rmi.Remote
- a classe: java.rmi.server.UnicastRemoteObject

Packages:

java.rmi

Basic package with Interface Remote, classe Naming, RMISecurityManager, and RemoteException

java.rmi.server

Implementation: server side, stub and skeletons, Transport and HTTP tunneling

java.rmi.registry

Repository. Mapping Name --> remote ref.

java.rmi.dgc

Some access to the distributed GC

java.rmi.activation

Persistence and activation (wake up) automatic for remote objects

4.2.1 Interface Remote

public interface Remote {}

A marqueur interface

Only to identy interfaces of type Remote:

implement directly or indirectly this interface

A "Remote Interface " has to respect the following constraints:

- implements the interface java.rmi.Remote
- Each method must declares the exception java.rmi.RemoteException in its throw part (or an ancestor exception: java.io.IOException, java.lang.Exception)
- Arguments and results of methods must be Serializable (implement the interface java.io.Serializable).
  Recursively, except static or transient

Example:

```
public interface BankAccount extends java.rmi.Remote {
        public void deposit(float amount)
              throws java.rmi.RemoteException;
        public void withdraw(float amount)
              throws OverdrawnException, java.rmi.RemoteException;
        public float getBalance()
              throws java.rmi.RemoteException;      }
```

4.2.2 Class UnicastRemoteObject

public class UnicastRemoteObject

extends RemoteServer

Allows to implement a class that is remotely acceessible

A remotely accessible class implements one or several interfaces that extend Remote

Example:

```
public class BankAccountImp
  extends UnicastRemoteObject
  implements BankAccount


  public BankAccountImp () throws RemoteException {
            super();
        }
public void deposit(float amount) throws RemoteException {
...
}
...
```

Actually, the call to the default constructor of the super class (`super();`) is automatically achieved. So, it is not necessary.

The class, of course, implements the methods defines in the remote interfaces.

Such a class can have public methods besides those of the remote interfaces: those are accessible only from the locale VM.

4.2.3 Global view

Global diagram for classes and interfaces RMI in the package java.rmi :

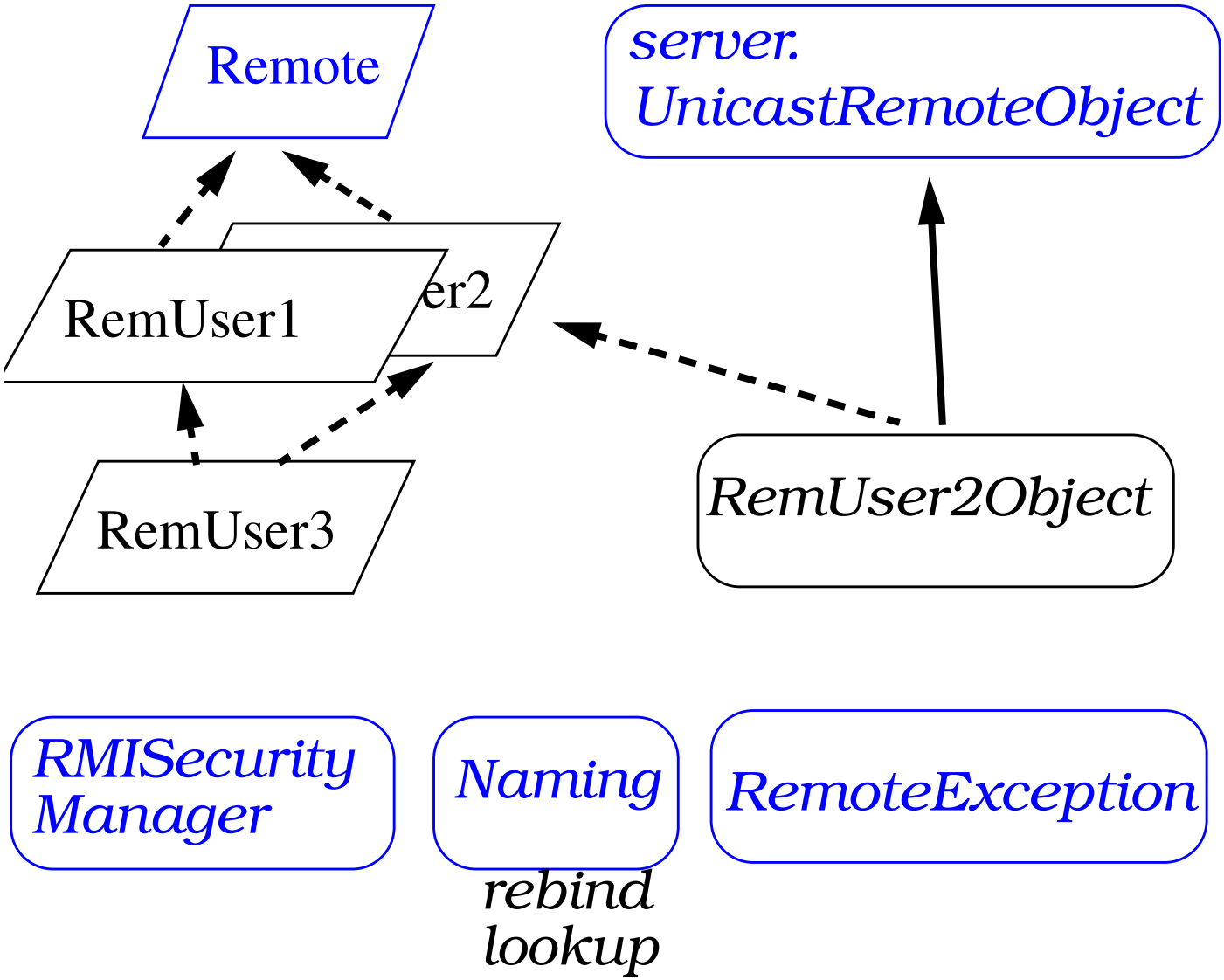


FIGURE 14 classes and interfaces RMI

4.2.4 Binding of a RMI object

public class Naming

method rebind

ou

method bind

Allows to register a RO in a naming server (directory) :

rmiregistry.

Example:

```
{
    // Create and register the server object
    HelloServer serverObject = new BankAccountImp ();
    Naming.rebind("//clio.unice.fr/BankAccount",
                  serverObject );
    // Signal successful registration
    System.out.println("BankAccount Server bound in registry");
    }
```

Such operation can be achieved only locally!

4.2.5 Lookup of an RMI object

public class Naming

method lookup

Allows, from a distant machine in that case, to get a remote reference towards an RMI object

Provided, of course, that it was registered in a rmiregistry.

Example:

```
BankAccount remoteA = (BankAccount) Naming.lookup(
                        "//clio.unice.fr/BankAccount");
                    float r= remoteA.getBalance ();
}
     catch (NotBoundException error)
       {
       error.printStackTrace();
       }
{
```

Note:

One must know the machine where the RMI object is!

4.3 Tools and compilers

rmic, rmiregistry (rmid : later on)

4.3.1 rmic

Allows to generate Stubs andSkeletons for the classes implementing the Remote interface.

One must give the complete name, with the package.

Example:

% rmic BankAccountImp

Create the .class:

BankAccountImp_Skel.class
BankAccountImp_Stub.class

NAME
rmic - Java RMI stub compiler
SYNOPSIS
rmic [-classpath path] [-d directory] [-depend] [-g]
[-keepgenerated] [-nowarn] [-O] [-show]
[-verbose] [-v1.2] package-qualified-class-names

- keepgenerated: keep the sources

-v1.2
Version 1.2 of Java ndoes not need any longer the skeletons, but by default keep them by compatibility with Java 1. option -v1.2 not to keep them.

4.3.2 rmiregistry

Start the registry (naming service) for Java RMI objects, on the current machine, with a given certain port number

By default: port is 1099.
Use a port above 1024

Example:

```
% rmiregistry &
```

or

```
% rmiregistry 2001 &
```

NAME

rmiregistry **- Java remote object registry**

SYNOPSIS

rmiregistry [port]

DESCRIPTION

The Java rmiregistry command creates and starts a remote object registry on the specified port on the current host. If port number is omitted, the registry is started on port 1099. The rmiregistry command produces no output and is typically run in the background. For example:

example% rmiregistry &

4.4 A typical developpement Method

Generally speaking, one follows these steps:

4.4.1 Remote interfaces

Chose the information that have to be remotely accessed

Structure them into objet: interface without direct access to data (only accessor methods)

Define interfaces with these methods and the exceptions (RemoteException + others: functionnal exceptions)

4.4.2 Classes that implement the Remote Interfaces: server(s)

Define the classes that implement :

Remote Interfaces

UnicastRemoteObject (in general)

Register at least one object in the rmiregistry

4.4.3 Clients using the remote objets

Declaration of instances for Remote Interfaces

Get a remote object from the rmiregistry

Classical call of methods, with exception handling

4.4.4 Compilation of sources

For Clients and Servers: the standard javac

4.4.5 Code Generation: Stubs/Skeletons creation

Use rmic to generate Stub/Skeletons for all classes that implementent Remote

Check first that“Remote“ classes compile (javac).

4.4.6 Start the rmiregistry

Chose a port (default: 1099)

(See later on: unset CLASSPATH)

Start the rmiregistry

4.4.7 Start the server

Launch one or several servers

(See later on:

java.rmi.server.codebase property

security policy)

4.4.8 Start the clients

Start one or several clients

Generally, there will be a way to give (e.g. on the command line) :

--> • where the server is (what machine),

--> • and on what port

the rmiregistry is listening.

4.5 Simple example: Remote Hello

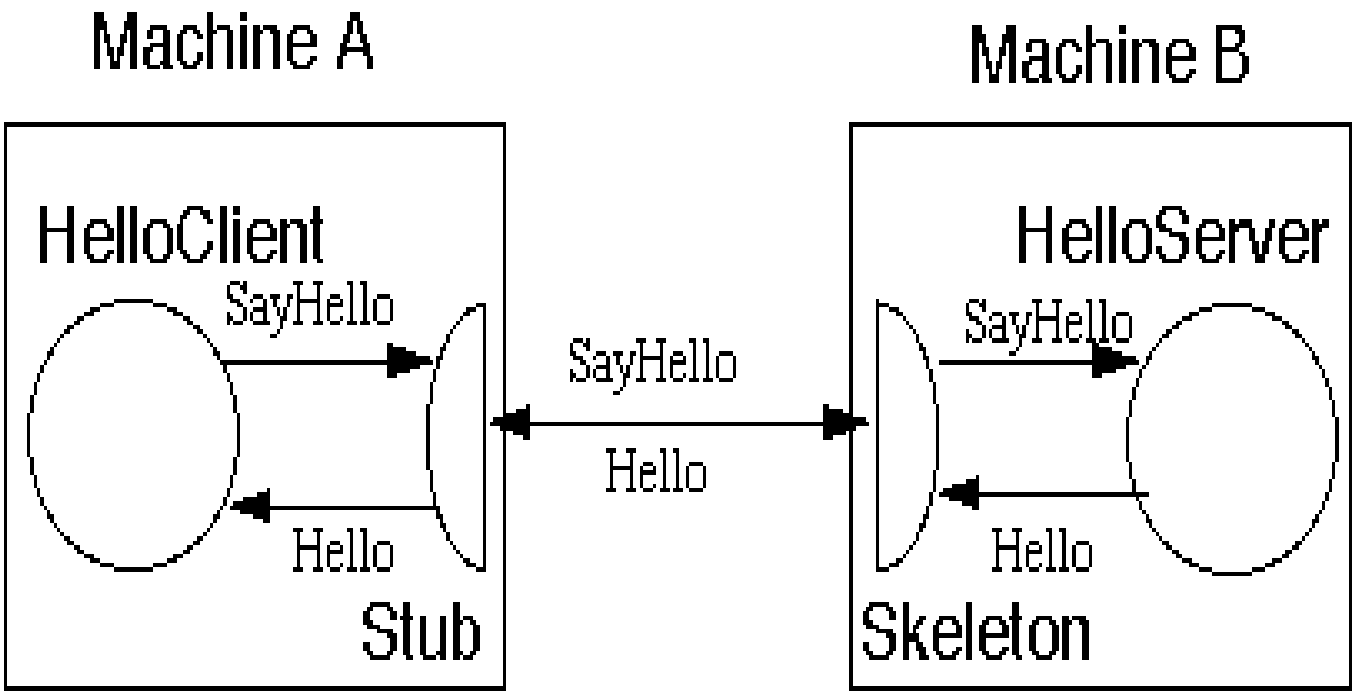


FIGURE 15 Remote Hello world

4.5.1 Remote Interface

```
public interface Hello extends java.rmi.Remote
  {
  String sayHello() throws java.rmi.RemoteException;
  }
```

4.5.2 Server HelloWorld

```
import java.net.InetAddress;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloServer
     extends UnicastRemoteObject
     implements Hello
  {
  public HelloServer() throws RemoteException {  }

  public String sayHello()
    {
    return  "Hello World from " + getHostName();
    }

  protected static String getHostName()
    {
    try
      {
      return InetAddress.getLocalHost().getHostName();
      }
    catch (java.net.UnknownHostException who)
      {
      return "Unknown";
      }
    }
```

4.5.3 Registering in the rmiregistry

Creation of the remotely accessible object,

and registering in the rmiregistry
(HelloServerMain.java)

```
public static void main(String args[])
    {
    // Create and install a security manager
    System.setSecurityManager(new RMISecurityManager());

    try
      {
      // Create and register the server object
      HelloServer serverObject = new HelloServer();
      Naming.rebind("rmi://clio.unice.fr/HelloServer",
                    serverObject );
      // Signal successful registration
      System.out.println("HelloServer bound in registry");
      }
    catch (Exception e)
      {
      System.out.println("HelloServer err: ");
      e.printStackTrace();
      } } }
```

Naming :

```
Naming.rebind("rmi://clio.unice.fr/HelloServer", serverObject );
```

The full syntax for the Naming

(rebind and lookup)

is the following:

"rmi://host:port/ServerLabel"

“host”, “rmi:“ and “port" (if 1099) are optionnal for a rebind

```
Hello remote = (Hello) Naming.rebind( "rmi://clio.unice.fr:1099/HelloServer",
                                                              serverObject);
```

At least “host” for a lookup:

```
Hello remote = (Hello) Naming.lookup( "//clio.unice.fr:1099/HelloServer");
Hello remote = (Hello) Naming.lookup( "//clio.unice.fr/HelloServer");
```

One can bind or unbind on a rmiregistry,

only from the JVM that is on the same host (security).

lookup of course can be done from any host.

4.5.4 A client to HelloWorld

```
import java.rmi.*;
import java.net.MalformedURLException;
public class HelloClient
  {
  public static void main(String args[])
    {
    try {
      Hello remote = (Hello) Naming.lookup(
                        "rmi://clio.unice.fr/HelloServer");
      String message = remote.sayHello();
      System.out.println( message );
      }
    catch ( NotBoundException error)
      {
      error.printStackTrace();
      }
    catch ( MalformedURLException error)
      {
      error.printStackTrace();
      }
    catch ( UnknownHostException error)
      {
      error.printStackTrace();
      }
    catch ( RemoteException error)
      {
      error.printStackTrace();
      }
    }
  }
```

The multiple catch allow to detect what exception is thrown.

4.5.5 Compilation, generation, registry

a) Server side

For Java 2, place the following files in your home directory:

.java.policy

```
grant {
      // Allow everything for now
      permission java.security.AllPermission;
};
// Do not use it systematically!!
```

Compile the sources:

javac Hello.java HelloServer.java

Generate the Stubs and Skeletons

rmic HelloServer

Produces:

HelloServer_Skel.class

HelloServer_Stub.class

Start the rmiregistry

rmiregistry 2001 &

Start the server:

java HelloServerMain

b) Client side

Compile the sources:

javac Hello.java HelloClient.java

One needs the remote interface (Hello.java)

Classe HelloServer_Stub.class must be available on the client side (ad hoc solution):

Copy the file HelloServer_Stub.class *from the server to he client machine (or NFS),*

Start the client:

java HelloClient

The server location is, in that example, in the source code. Of course in a real application it should be given as a parameter or a configuration file.

Note:

With Java RMI, one can creates locally some remotelly accessible objects

but one cannot

remotely creates remotely accessible objects

4.6 Implementation principle: ad hoc reification of calls

Version 1.1.x

4.6.1 Stub

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class HelloServer_Stub
    extends java.rmi.server.RemoteStub
    implements Hello, java.rmi.Remote
{
    private static java.rmi.server.Operation[] operations = {
        new java.rmi.server.Operation("java.lang.String sayHello()")
    };

    private static final long interfaceHash = 6486744599627128933L;

    // Constructors
    public HelloServer_Stub() {
        super();
    }
    public HelloServer_Stub(java.rmi.server.RemoteRef rep) {
        super(rep);
    }
    // Methods from remote interfaces

    // Implementation of sayHello
    public java.lang.String sayHello() throws java.rmi.RemoteException {
        int opnum = 0;
        java.rmi.server.RemoteRef sub = ref;
        java.rmi.server.RemoteCall call =
sub.newCall((java.rmi.server.RemoteObject)this, operations, opnum, interfaceHash);
        try {
            sub.invoke(call);
        } catch (java.rmi.RemoteException ex) {
```

```
            throw ex;
        } catch (java.lang.Exception ex) {
            throw new java.rmi.UnexpectedException("Unexpected exception",
ex);
        };
        java.lang.String $result;
        try {
            java.io.ObjectInput in = call.getInputStream();
            $result = (java.lang.String)in.readObject();
        } catch (java.io.IOException ex) {
            throw new java.rmi.UnmarshalException("Error unmarshaling return",
ex);
        } catch (java.lang.ClassNotFoundException ex) {
            throw new java.rmi.UnmarshalException("Return value class not
found", ex);
        } catch (Exception ex) {
            throw new java.rmi.UnexpectedException("Unexpected exception",
ex);
        } finally {
            sub.done(call);
        }
        return $result;
}
}
```

4.6.2 Skeleton

```
// Skeleton class generated by rmic, do not edit.
// Contents subject to change without notice.
public final class HelloServer_Skel
    extends java.lang.Object
    implements java.rmi.server.Skeleton
{
    private static java.rmi.server.Operation[] operations = {
        new java.rmi.server.Operation("java.lang.String sayHello()")
    };

    private static final long interfaceHash = 6486744599627128933L;

    public java.rmi.server.Operation[] getOperations() {
        return operations;
    }

   public void dispatch(java.rmi.Remote obj, java.rmi.server.RemoteCall call, int
opnum, long hash) throws java.rmi.RemoteExcepti
on, Exception {
        // Exceptions pass through, to be caught, identified and marshalled

        if (hash != interfaceHash)
            throw new java.rmi.server.SkeletonMismatchException("Hash
mismatch");
        HelloServer server = (HelloServer) obj;
        switch (opnum) {
        case 0: { // sayHello
            call.releaseInputStream();
            java.lang.String $result = server.sayHello();
            try {
                java.io.ObjectOutput out = call.getResultStream(true);
                out.writeObject($result);
            } catch (java.io.IOException ex) {
                throw new java.rmi.MarshalException("Error marshaling return", ex);
            };
            break;
            }
        default:
            throw new java.rmi.RemoteException("Method number out of range");
        }}}
```

CHAPITRE 5 Reminder: Advanced

RMI

5.1 Security and dynamic code loading

5.1.1 Security Manager

A SecurityManager must be used with RMI, otherwise RMI clients and servers cannot dynamically load code other than from the CLASSPATH.

Principle of a Security Manager:

> if no security manager is installed, then only the classes accessible from the CLASSPATH can be loaded
>
> A security manager checks various aspects on the classes at loading.

One way to load a SecurityManager only if there is none:

```
if (System.getSecurityManager() == null) {
  System.setSecurityManager(
        newRMISecurityManager());
}
```

In the general case (not with NFS, not with an applet), a SecurityManager is needed in both the client and the server.

All JVM that must download some code need a SM (Java 2).

RMI clients download code: the stubs

For an applet:

it is possible to have the SM only in the server, for the client already has one SM: there is one in the browser that loaded the applet

5.1.2 Security file

To get the basic securities needed for RMI

.java.policy

```
grant
  {
      permission java.net.SocketPermission "*:1024-65535",
            "connect,accept,resolve";
      permission java.net.SocketPermission "*:1-1023",
            "connect,resolve";
  };
```

The second part is for reserved ports

Or of course all permissions:

```
grant {
      // Allow everything for now
      permission java.security.AllPermission;
};
```

5.1.3 Signed applets and RMI

By défaut, an applet can communicate (tcp, so RMI) only with the host from which it was loaded

To get his name: getCodeBase().getHost()

If on wants to communicate with an RMI object being on another host: one needs a signed applet

See security lectures

An applet cannot either by default communicate through RMI with the host that loaded it..

5.1.4 Codebase and RMI

A codebase is a place from which code can be loaded in a VM.

CLASSPATH is just a local codebase

An applet codebase is always relatively given from the URL from which it was loaded.

With RMI:

java.rmi.server.codebase

At client start (also sever if needed, see later on) one must specifies the java rmi server codebase, so that dynamic loading of code (byte code from Java classes) is possible if necessary.

For instance: the implementation code of the stub.

codebase is a property that represents one or several URL from which classes can be loaded

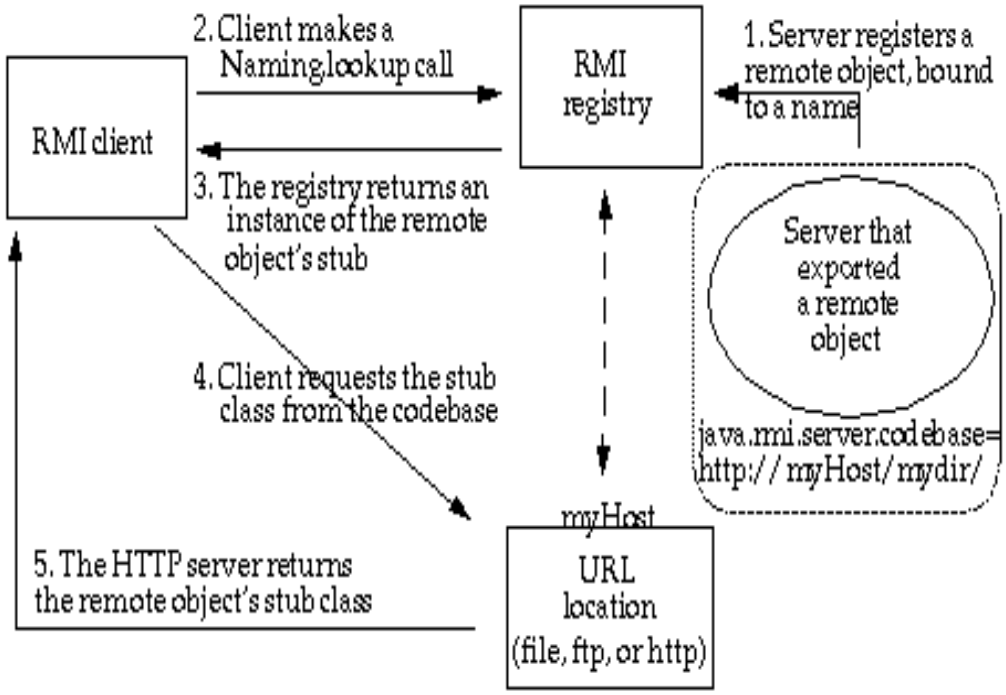


FIGURE 16 Dynamic loading of stubs with RMI

example:

java -Djava.rmi.server.codebase=http://myhost/ ~myusrname/myclasses/ examples.hello.HelloImpl

Or several codebase with a space separator:

-Djava.rmi.server.codebase="http://webfront/myStuff.jar http://webwave/myOtherStuff.jar http://webwave/dir/"

A / is needed if it is a directory and not if it is a file!

One can use an URL towards a file:

file:/myDirectory/location

The technique is useful not only for stubs, but also to get the code (.class) of polymorphic parameters:

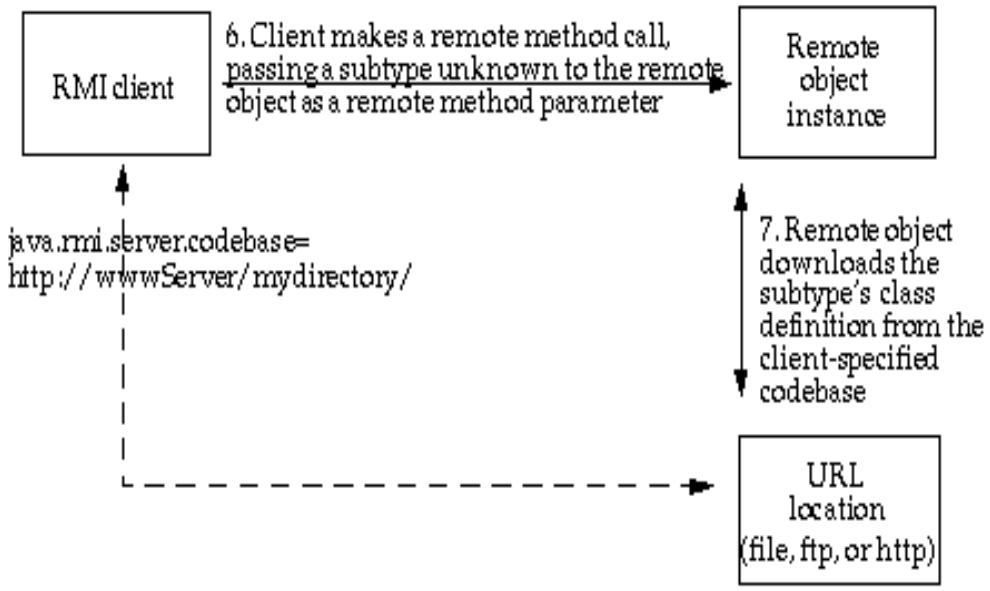


FIGURE 17 Dynamic loading from polymorphism

It is useful for client-server infrastructures that must be fully generic.

See for instance:

http://java.sun.com/docs/books/tutorial/rmi/

Notes:

- **codebase is** a kind of pointer towards code, it is also a kind of annotation that is added to an object

- serialized objects and code are using different paths

- Serialized objects and bytecode are taking different path:

  - objects: sockets (TCP/IP)
  - .class: HTTP (over TCP/IP)

5.1.5 CLASSPATH and rmiregistry

Before launching the rmiregistry, one must make sure not to have a CLASSPATH that would allows to load classes that one wants to dynamically load to the client

Especially the stub for the remote object!

If the rmiregistry can access some classes with the CLASSPATH, it will be used rather than the codebase,

and the client won't be able to load the stub, or other classes

See the :

java.rmi.server.codebase property

So, to be on the safe side:

unset CLASSPATH

before launching the rmiregistry

Notes:

an application can bind or rebind an object only from the same host than the rmiregistry

With Java RMI, there are no remote creation dof remote objects ... from a distant VM.

All remotely accessible objects are created and registered locally.

Jini allows to alleviate this constraint with other features:

- lookup over an interface
- broadcast to find a sever without knowing on what machine it lives
- “lease” notion: a proxy with a limited validity (in time), need to be renewed
-

5.2 Applets and RMI

5.2.1 Principles

From an applet, we will achieve an RMI call towards a server that is running on the host from which the applet was loaded.

The resultat of the call is send back in the browser, and display by the applet.

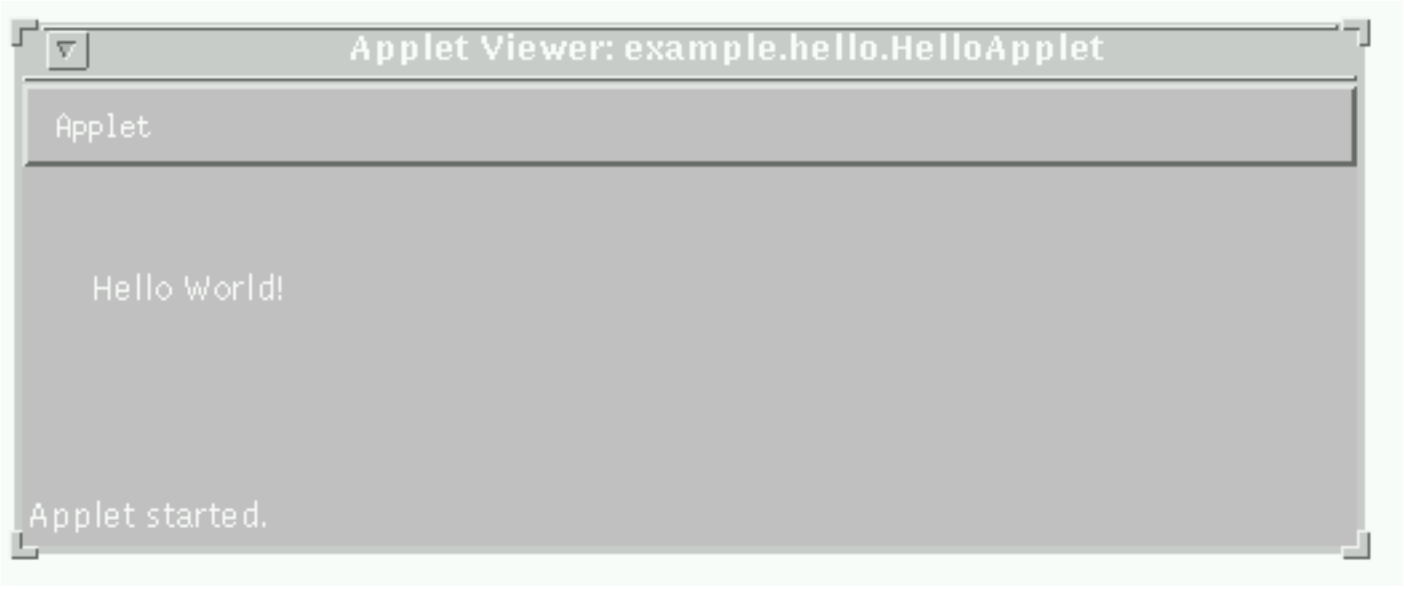


FIGURE 18 Execution of the applet Hello World

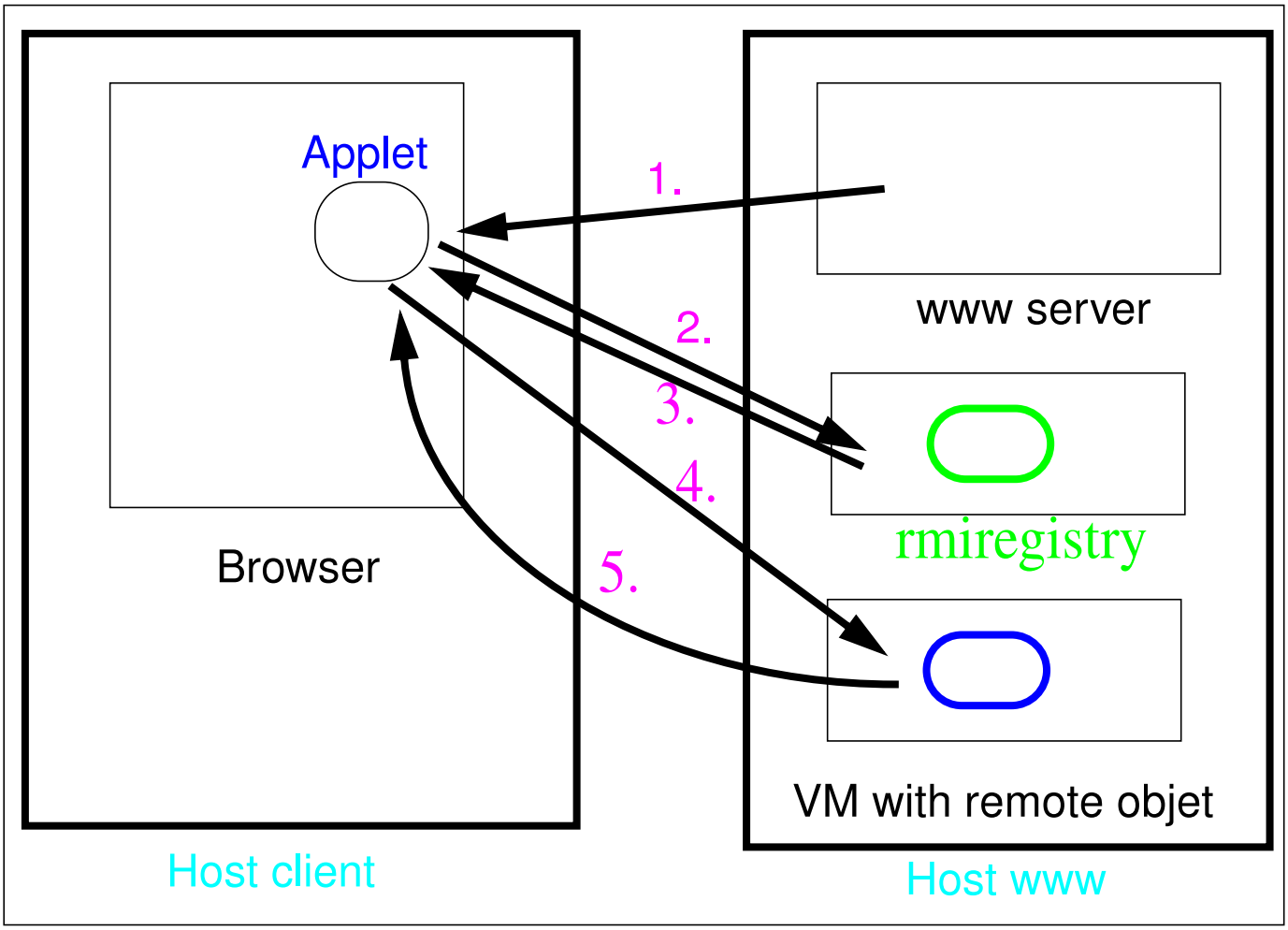


FIGURE 19 Architecture of an RPC with an RMI applet

1. Applet Loading
2. Lookup in rmiregistry
3. Return of the stub
4. Call to remote object
5. Result return

5.2.2 Code and principles of classes

Be careful with package usage:

In Java there is a mapping between

the full package name of a class

and

the path to find this class

That allows the compiler to find the classes

In the example:

- package examples.hello
- sources are in $HOME/mysrc/examples/hello

a) Remote interface Hello

Classical Remote interface,

No changes from the previous example.

```
package examples.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
   String sayHello() throws RemoteException;
}
```

b) Server class (HelloImpl) and main

```
package examples.hello;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject
                                    implements Hello {

   public HelloImpl() throws RemoteException {
      super();
   }
public String sayHello() {
      return  "Hello World!";
   }
public static void main(String args[]) {
// Create and install a security manager
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      try {
         HelloImpl obj = new HelloImpl();
         // Bind this object instance to the name "HelloServer"
         Naming.rebind("HelloServer", obj); // eventuellement clio.unice.fr:2001
         System.out.println("HelloServer bound in registry");
      } catch (Exception e) {
         System.out.println("HelloImpl err: " + e.getMessage());
         e.printStackTrace();
} } }
```

UnicastRemoteObject and Hello

Implementation of methods

A security manager is needed

Instantiation of the remote object

Registering in the miregistry

c) Client Applet: HelloApplet

```
package examples.hello;

import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {

   String message = "blank";

   // "obj" is the identifier that we'll use to refer
   // to the remote object that implements the "Hello"
   // interface
   Hello obj = null;

   public void init() {
      try {
         obj = (Hello)Naming.lookup("//" +
                  getCodeBase().getHost() + "/HelloServer");
         message = obj.sayHello();
      } catch (Exception e) {
         System.out.println("HelloApplet exception: " +
                           e.getMessage());
         e.printStackTrace();
    } }
public void paint(Graphics g) {
      g.drawString(message, 25, 50);
} }
```

getCodeBase().getHost() to find the host

no securityManager: we have the applet one

The lookup send the stub object, and stub code will also be send to the client with the http protocol.

d) Page HTML: hello.html

```
<HTML>
<title>Hello World</title>
<center> <h1>Hello World</h1> </center>

The message from the HelloServer is:
<p>
<applet codebase="myclasses/"
        code="examples.hello.HelloApplet"
        width=500 height=120>
</applet>
</HTML>
```

Notes:

- A serveur HTTP is needed on the machine from which one wants to get (download) the classes.

- codebase is a directory where to find the .class of the applet.
It specifies a directory under the one used to load the applet:

relative path

in general a good solution
(../ if the codebase was above the HTML dir)

- The code attribute of the applet specifies the package full name: examples.hello.HelloApplet

5.2.3 Compilation and deployment

Be carefull, a few specific points

a) File location, and HTTP servers

Remainder: the source files:

- Hello.java
- HelloImpl.java
- HelloApplet.java
- hello.html

Compilation with the specification of where the .class must be placed:

here $HOME/public_html/myclasses

Very often, the web servers allow to access to the user directory with :

http://host/~username/ , *and*

public_html is egal to www

otherwise, and in all cases where the applet host and server host have access to the same file system, the one can use file URL:

file:/home/username/public_html

Other solution: a minimal HTTP server given by par Sun:

http://java.sun.com/products/jdk/rmi/class-server.zip

b) Compilation

Let say we place the .class in

$HOME/public_html/myclasses

the we will compile in the following way:

javac -d $HOME/public_html/myclasses examples/hello/Hello.java examples/hello/HelloImpl.java examples/hello/HelloApplet.java

This command:

- create the directory examples/hello (if it does not already exist) dans le répertoire $HOME/public_html/myclasses
  The directory myclasses must already exist
- Compile and create the 3 files:

  Hello.class
  HelloImpl.class
  HelloApplet.class

Then, one must generate the stubs and skeletons:

rmic -d $HOME/public_html/myclasses examples.hello.HelloImpl

The command is executed on the class that implement the remote objects (HelloImpl) and it generate:

HelloImpl_Stub.class and HelloImpl_Skel.clas

in the same dir than the other .class

c) Installation of the applet and CLASSPATH

Put the HTML file under the dir accessible from the browser:

mv $HOME/mysrc/examples/hello/hello.html $HOME/public_html/

To start the server (HelloImpl), the directory $HOME/public_html/myclasses must be in the CLASSPATH.

d) RMI registry, server and applet

Start the RMI registry:

No CLASSPATH

unset CLASSPATH
rmiregistry & ..... *ou* ..... rmiregistry 2001 &

cbecause we want the client using the rmiregistry to download the .class with the HTTP server

If the rmiregistry find in its CLASSPATH the .class, the it will ignore the server codebase that tells it how to find the classes, and the client won't work.

Bug ou feature ?

Start the server:

The Java property

java.rmi.server.codebase

allows to specify how to transmit the stub from the server host to the client

Fist from the serveur to the rmiregistry:

if the rmiregistry need the stub, it wont find it locally, for PATH was set to void, so it will lookup in the codebase:

and go through the http server,

this loading source is memorised in the object, later on, the client will use the same mean to load the .class stub object being received).

```
java
    -Djava.rmi.server.codebase=http://myhost/~myusrname/
myclasses/
    -Djava.security.policy=$HOME/mysrc/policy
    examples.hello.HelloImpl
```

Note: the codebase last / is needed!

In that case, a specific policy file is also given: $HOME/mysrc/policy

If it works, you should see: “HelloServer bound in registry"

e) Applet execution

http://myhost/~myusrname/hello.html

From a browser or the applet viewer:

appletviewer http://myhost/~myusrname/hello.html

5.3 Polymorphisme and RMI

Powerful mechanism for distributed programming

Allow to achieve generic client-server, and also to manage the software evolution without stopping complex systems.

Open architectures

5.3.1 Principle and implications

Sending a sub-type of the (expected) static type

Classical polymorphism .

Here, we must dynamically manage one thing:

the object that is sent might not be known in the process (JVM) where it arrives

the object is unknown=
we do not have the byte code for it

Alike for the stubs of the previous example,

it will be neededto dynamically load the code from JVM to another

5.3.2 Result polymorphism

From Server to client:

The client calls a method of this type:

```
Result1 res = serveur.foo ( ...)
```

but the server retourns an object of type Result2 that of course derives from Result1

The client will have to download the Result2 code so it can use the object received from the remote call (call methods, acces to attributes).

Note:

The client will also need to download all classes (or interfaces) used by Result2:

attributes,
parameters of methods,
etc.

5.3.3 Parameter Passing

From Client to Server:

The client calls a method of the type:

Result1 res = serveur.foo (ParmA1, ParamB1)

but does not give parameters of types ParmA1, ParamB, but rather of types ParmA2, ParamB2 sub-type of the (expected) static type

The server will have to download the code ParmA2, ParamB2 so it can execute the service

Note:

The server will also need to download all classes (or interfaces) used by the two new classes.

attributes,
parameters of methods,
etc.

5.4 Example: RMI with polymorphisme

5.4.1 Principle of the 2 interfaces

A generic server:

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;

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

A task to execute:

```
package compute;
import java.io.Serializable;

public interface Task extends Serializable {
        Object execute();
}
```

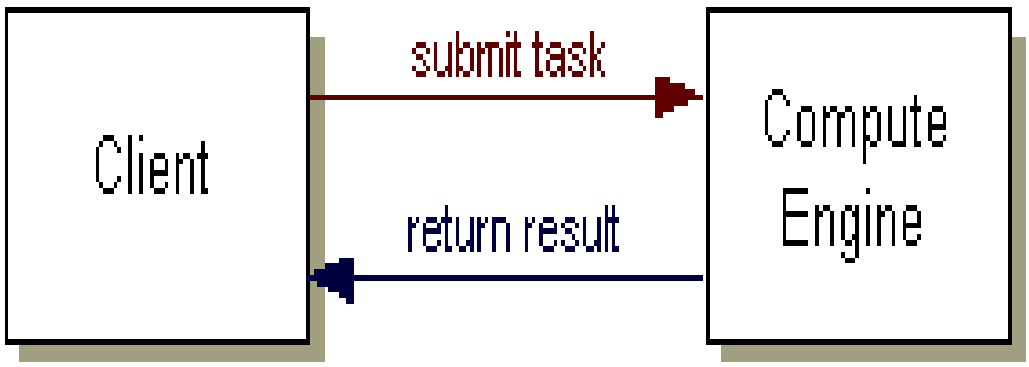


FIGURE 20 Generic Client-Server

5.4.2 Server implementation

The server only executes the tasks given to it:

executeTask calls execute on the objects of type Task that are transmitted to it.

```
package engine;

    import java.rmi.*;
    import java.rmi.server.*;
    import compute.*;

    public class ComputeEngine extends UnicastRemoteObject
                        implements Compute
    {
       public ComputeEngine() throws RemoteException {
          super();
       }

       public Object executeTask(Task t) {
          return t.execute(); // HERE: Polymorphism, ST vs. DT
       }

       public static void main(String[] args) {
          if (System.getSecurityManager() == null) {
             System.setSecurityManager(new RMISecurityManager());
          }
          String name = "//host/Compute";
          try {
             Compute engine = new ComputeEngine();
             Naming.rebind(name, engine);
             System.out.println("ComputeEngine bound");
          } catch (Exception e) {
             System.err.println("ComputeEngine exception: " +
                                e.getMessage());
             e.printStackTrace();
          }
       }
    }
```

5.4.3 A client: compute Pi

Classic principle of server connection:

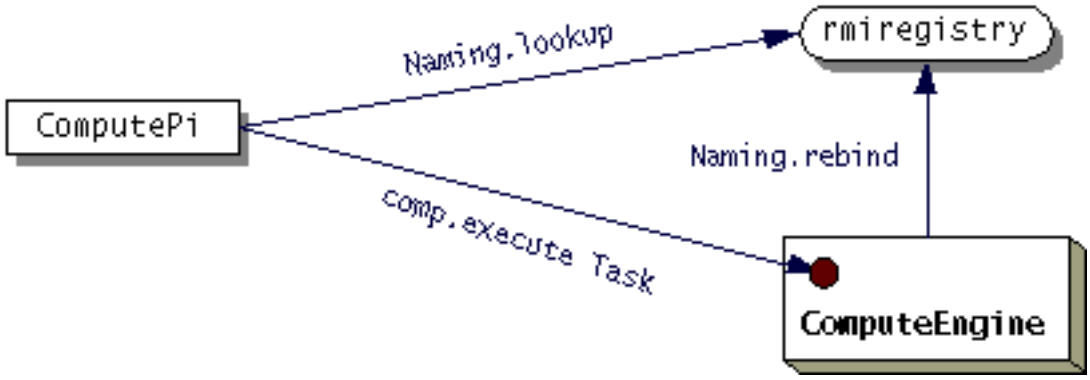


FIGURE 21 A client : compute Pi

Then locally, the client creates an object of the type representing the task to be executed, it is given to the server, then it gets the result

```
package client;
import java.rmi.*; import java.math.*; import compute.*;

public class ComputePi {
        public static void main(String args[]) {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            try {
                String name = "//" + args[0] + "/Compute";
                Compute comp = (Compute) Naming.lookup(name);
                Pi task = new Pi(Integer.parseInt(args[1]));
                BigDecimal pi = (BigDecimal) (comp.executeTask(task));
                System.out.println(pi);
            } catch (Exception e) {
                System.err.println("ComputePi exception: " +
                                   e.getMessage());
                e.printStackTrace();
            }
        }
    }
```

The implementation of classe Pi is rather classical, it must verify two things :

- implement Task
- be Serializable (the attributes)

The method execute, that represent the work to be achieved, must be implemented.

```
package client;

    import compute.*;
    import java.math.*;

    public class Pi implements Task {

      /** constants used in pi computation */
      private static final BigDecimal ZERO =
        BigDecimal.valueOf(0);
      private static final BigDecimal  ONE =
        BigDecimal.valueOf(1);
      private static final BigDecimal FOUR =
        BigDecimal.valueOf(4);

      /** rounding mode to use during pi computation */
      private static final int roundingMode =
        BigDecimal.ROUND_HALF_EVEN;

      /** digits of precision after the decimal point */
      private int digits;

      /**
       * Construct a task to calculate pi to the specified
       * precision.
       */
      public Pi(int digits) {
        this.digits = digits;
      }
```

```
 /**
  * Calculate pi.
  */
                    /* Code that was developped by the CLIENT ,
                     * but that will be executed by the SERVER */
public Object execute() {
   return computePi(digits);
}

 /**
  * Compute the value of pi to the specified number of
  * digits after the decimal point.  The value is
  * computed using Machin's formula:
  *
  *         pi/4 = 4*arctan(1/5) - arctan(1/239)
  *
  * and a power series expansion of arctan(x) to
  * sufficient precision.
  */
 public static BigDecimal computePi(int digits) {
     int scale = digits + 5;
     BigDecimal arctan1_5 = arctan(5, scale);
     BigDecimal arctan1_239 = arctan(239, scale);
     BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                          arctan1_239).multiply(FOUR);
     return pi.setScale(digits,
                     BigDecimal.ROUND_HALF_UP);
 }
 /**
  * Compute the value, in radians, of the arctangent of
  * the inverse of the supplied integer to the speficied
  * number of digits after the decimal point.  The value
  * is computed using the power series expansion for the
  * arc tangent:
  *
  * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
  *     (x^9)/9 ...
  */
 public static BigDecimal arctan(int inverseX,
                          int scale)
 {
     BigDecimal result, numer, term;
     BigDecimal invX = BigDecimal.valueOf(inverseX);
```

```
            BigDecimal invX2 =
                BigDecimal.valueOf(inverseX * inverseX);

            numer = ONE.divide(invX, scale, roundingMode);

            result = numer;
            int i = 1;
            do {
                numer =
                    numer.divide(invX2, scale, roundingMode);
                int denom = 2 * i + 1;
                term =
                    numer.divide(BigDecimal.valueOf(denom),
                                 scale, roundingMode);
                if ((i % 2) != 0) {
                    result = result.subtract(term);
                } else {
                    result = result.add(term);
                }
                i++;
            } while (term.compareTo(ZERO) != 0);
            return result;
        }
    }
```

Source code at :

http://java.sun.com/docs/books/tutorial/rmi/index.html

5.5 Callback, synchro, multithread, and RMI

5.5.1 Model

What is going on if N clients are, at the same time, calling a single RMI server ?

That is not specified in the RMI semantics.

At least one thread, bus implicit.

One cannot know if it will get several threads, or if the calls will be sequentialized one after antother.

If a single thread, there have to be a delay queue to postpon the pending calls.
In any case, one does not know the order of execution.

- FIFO is the simplest, can be implemented with locks
- however, most implementations use a pool de threads.

It is feasible, and “correct”, to implement RMI with a thread for each call.

Current implémentation of RMI jdk 1.4.1 (Oct. 2002):

RMI opens a TCP/IP socket per client.

One server thread is dedicated to reading the input stream of the socket

It seems to be that thread that achieves the call onto the server method.

Conclusion: it is more or less one thread per client.

5.5.2 Asynchronous calls

How to achieve Asynchronous calls with RMI ?

You have to explicitely deal with it with a thread

In the simplest case:

a method returning void

not much to do faire.

The thread is simply used not to block the caller.

If we want an asynchronous call for a method not returning void:

One needs to reserve a special place (future) to put the resultat to come

One also needs a method to test if the result is back (or notify the thread that might be waiting for the result)

One also needs a synchronization method to hold the caller if we want to (not with buzy wait).

---> Example

a Call Back is also needed from the server to the client.

5.5.3 Call back

How to achieve a Call back with RMI ?

Very useful, for example to inform a client of a value change, or condition change within a server.

Avoid "Pooling": repeated calls to monitor a potential change.

No direct means, a remote object is needed on the client side.

Two solutions:

- the client object is itself a remote object

- another object of the client VM is a remote objet, and there is a mutual reference between itself and the client

  ----> Picture

Be careful to at least 2 things:

--> • concurrent access

--> • potential deadlocks when calling back

http://java.sun.com/docs/books/tutorial/rmi/index.html

CHAPITRE 6 Active Objects

Active Objects are studied within the framework of the ProActive library: a LGPL framework and interactive environment for parallel and distributed programming.

It is a project of the ObjectWeb consortium.

(http://www.objectweb.org)

ProActive is a Java library for Parallel, Distributed, and Concurrent computing, also featuring Mobility and Security in a uniform framework.

Overview:

- ProActive: Basic features
- MOP
- Meta-Objects for Distribution
- IC2D: graphical visualization and control

(See other material)

CHAPITRE 7 Mobility and Formal Models

Communicating Mobile Agents

Localization issues and Performance modeling

Formal Calculus for Objects and Asynchronous Objects:

- Sigma calculus
- ASP: Asynchronous Sequential Processes

(See other material)

CHAPITRE 8 Hands-on Practical

Session

By yourself or Option 10

- The ProActive environment

  http://www.inria.fr/oasis/ProActive/
- Interactive monitoring and visualization: IC2D
- Programming a Communicating Mobile Agent