Introduction to adatptive computing systems

Sara Bouchenak
Associate Professor, University of Grenoble – LIG, France

Sara.Bouchenak@imag.fr
http://membres-liglab.imag.fr/bouchenak/teaching/

Why adaptive computing systems?

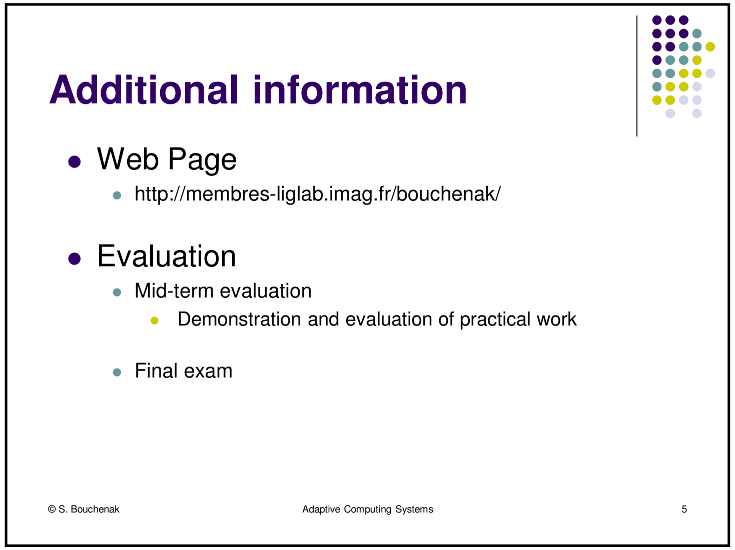
- Applications need to evolve
  - Scalability
  - Quality-of-service
- Applications hosted in changing environment
  - Mobility
    - Logical mobility (mobile code and data)
    - Physical mobility (mobile users and devices)
  - Dynamic connection and disconnection
  - Variable communication quality

Objectives

- Advanced aspects of adaptive systems
- Real applications
- Prepare to
  - Implement adaptive applications in an industrial context
  - Conduct research in the area of middleware and distributed systems

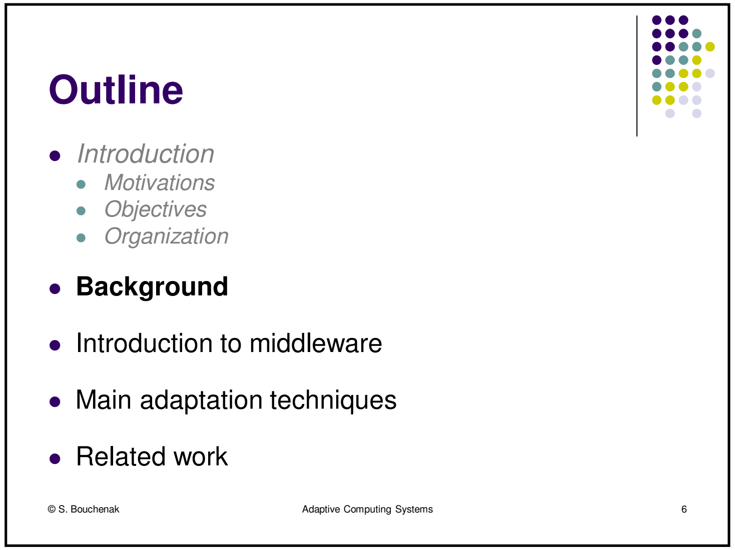
Agenda

Lecture, Monday, 14:00 – 17:00	Lab, Monday, 14:00 – 17:00
Introduction to adaptive computing systems	
	Java Management eXensions – JMX
AOP-based adaptive systems	
	Introduction to AspectJ
Interruption week	
Non-functional aspects of computing systems (logging, security, dependability, etc.)	
	Logging with AspectJ
Autonomic computing (case studies)	
	Security with AspectJ
Self-adaptive systems (case studies)	
	Dependability with AspectJ
-	
Interruption week	
Summary and future directions	
	Evaluation



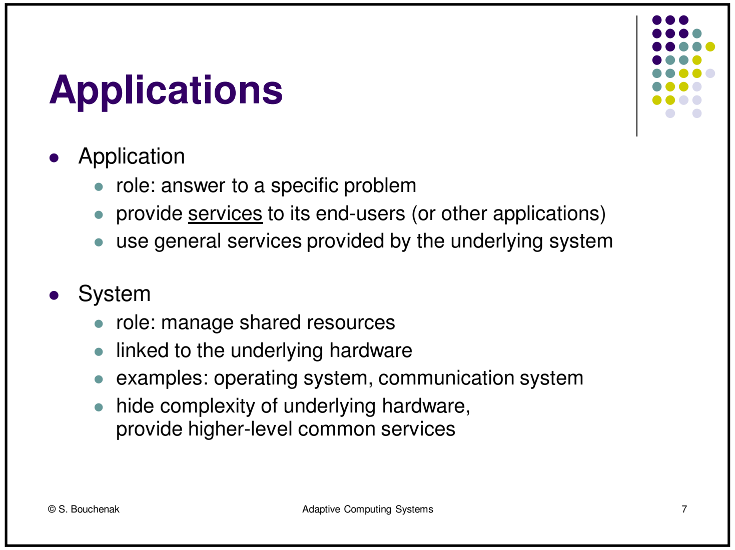
Additional information

- Web Page
  - http://membres-liglab.imag.fr/bouchenak/
- Evaluation
  - Mid-term evaluation
    - Demonstration and evaluation of practical work
  - Final exam



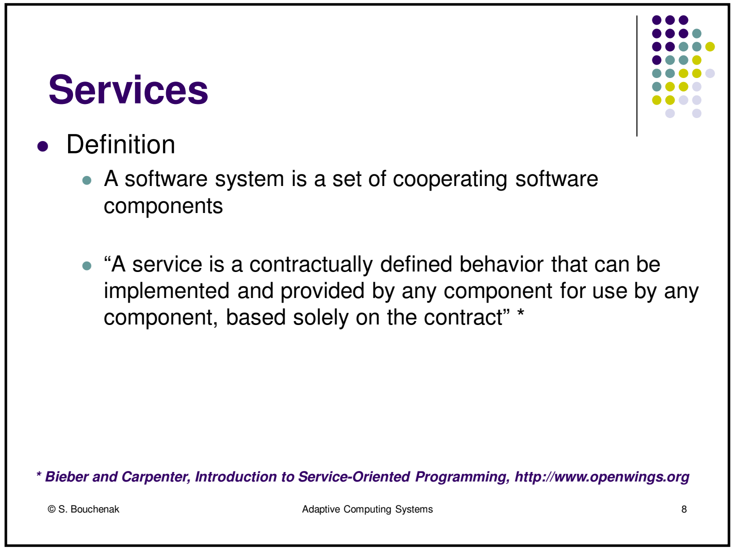
Outline

- *Introduction*
  - *Motivations*
  - *Objectives*
  - *Organization*
- **Background**
- Introduction to middleware
- Main adaptation techniques
- Related work



Applications

- Application
  - role: answer to a specific problem
  - provide services to its end-users (or other applications)
  - use general services provided by the underlying system
- System
  - role: manage shared resources
  - linked to the underlying hardware
  - examples: operating system, communication system
  - hide complexity of underlying hardware,
    provide higher-level common services



Services

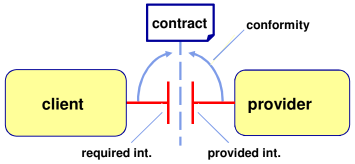
- Definition
  - A software system is a set of cooperating software components
  - “A service is a contractually defined behavior that can be implemented and provided by any component for use by any component, based solely on the contract” *

*** Bieber and Carpenter, Introduction to Service-Oriented Programming, http://www.openwings.org***

Services and interfaces

- Implementation
  - A service is accessible via one or multiple interfaces
  - An interface describes the interaction between serice povider and service client
    - Operational point of view:
      define operations and data structures for service implementation
    - Contractual point of view:
      define contract between service provider and service customer

Interface definition



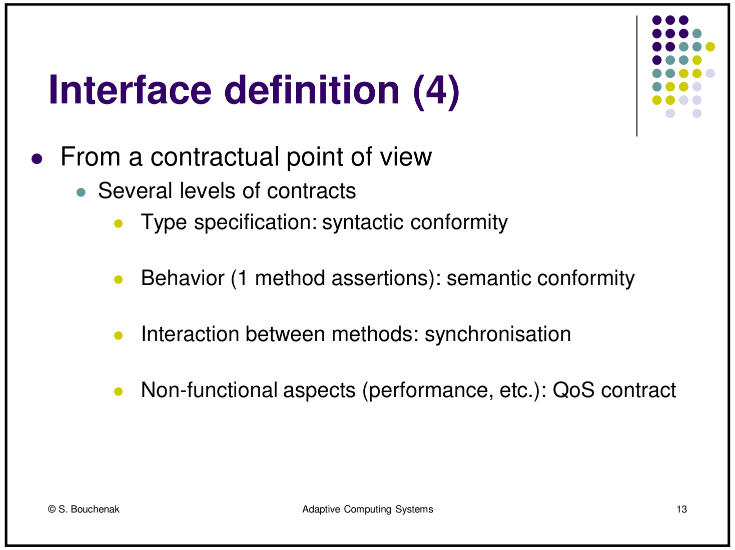
- A service involves <u>two</u> interfaces
  - Required interface (from client side)
  - Provided interface (from provider side)

Interface definition (2)

- Contract specifies compatibility (i.e. conformity) between interfaces
  - Client and provider see each other as a "black-box" (encapsulation)
  - Consequence: client and provider can be replaced, as long as the contract is met
- Contract may specify aspects non-included in the interface
  - Non-functional properties, i.e. Quality-of-service (QoS) properties

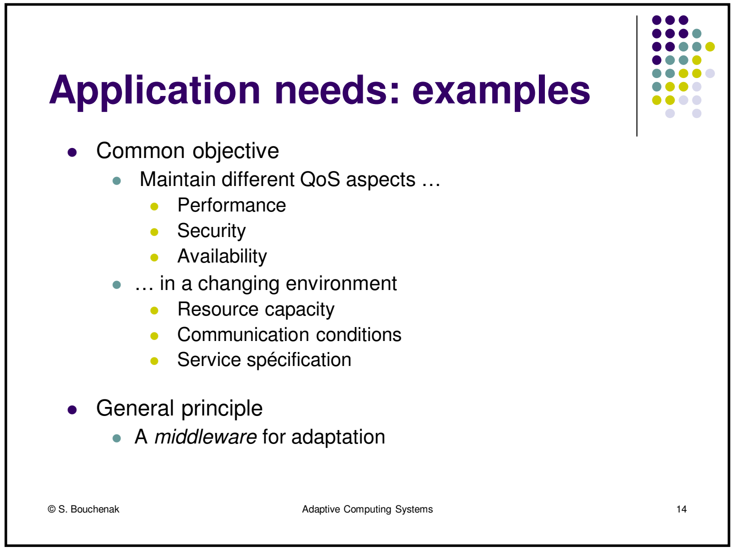
Interface definition (3)

- From an operational point of view
  - Interface Definition Language (IDL)
    - No standard
    - Based on an existing language
      - CORBA IDL in C++
      - Java et C# define their own IDL



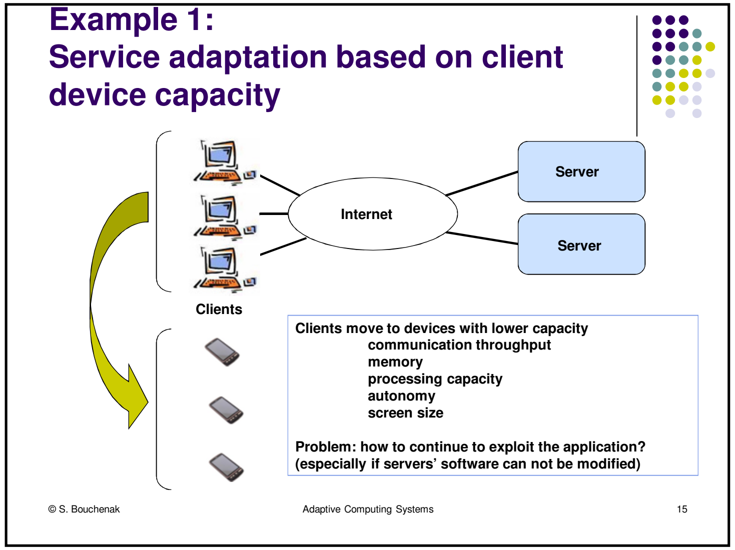
Interface definition (4)

- From a contractual point of view
  - Several levels of contracts
    - Type specification: syntactic conformity
    - Behavior (1 method assertions): semantic conformity
    - Interaction between methods: synchronisation
    - Non-functional aspects (performance, etc.): QoS contract



Application needs: examples

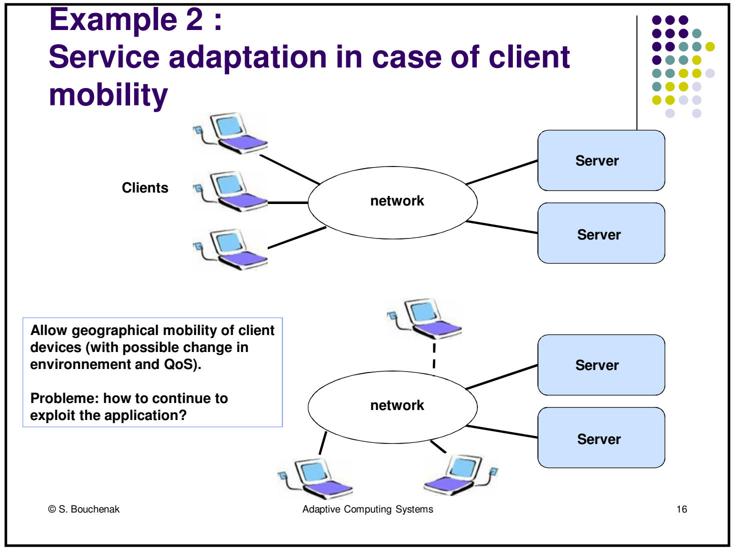
- Common objective
  - Maintain different QoS aspects …
    - Performance
    - Security
    - Availability
  - … in a changing environment
    - Resource capacity
    - Communication conditions
    - Service spécification
- General principle
  - A *middleware* for adaptation



Example 1: Service adaptation based on client device capacity

Clients move to devices with lower capacity
communication throughput
memory
processing capacity
autonomy
screen size

Problem: how to continue to exploit the application?
(especially if servers' software can not be modified)

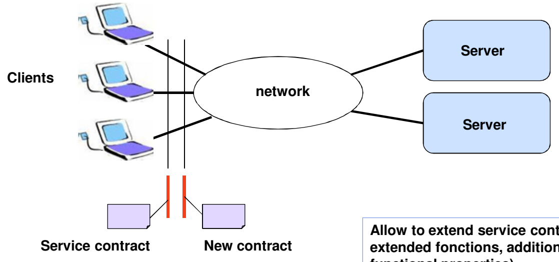


Example 2 : Service adaptation in case of client mobility

Allow geographical mobility of client devices (with possible change in environnement and QoS).

Probleme: how to continue to exploit the application?

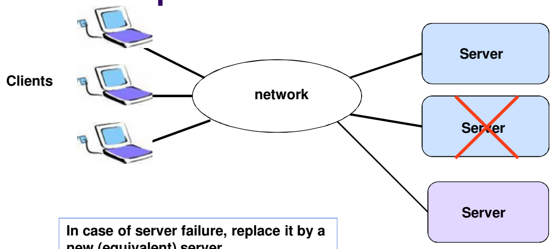
Example 3 : Service extension and evolution



Allow to extend service contract (e.g. extended fonctions, additional non-functional properties)

Problem: how to allow this evolution (without service interruption)?

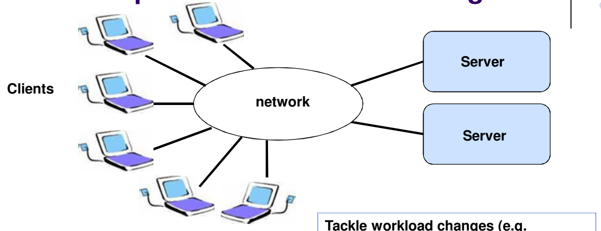
Example 4 : Service adaptation for fault-tolerance



In case of server failure, replace it by a new (equivalent) server

Problem: how to tolerate failures (failure detection, server replacement), without service interruption?

Example 5 : Service adaptation for workload changes



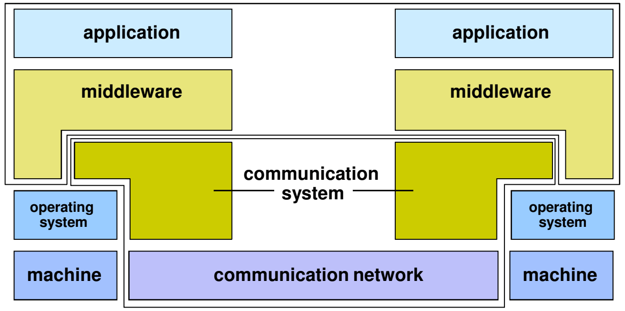
Tackle workload changes (e.g. #concurrent clients)

Problem: how to maintain an acceptable level of QoS (e.g. service request response time) ?

Outline

- *Introduction*
- *Background*
  - *Services and interfaces*
  - *Application needs*
- **Introduction to middleware**
- Main adaptation techniques
- Related work

Middleware



Middleware functionalities

- Middleware has four main functions
  - High-level interface or API (*Application Programming Interface*) to applications
  - Mask heterogeneity of underlying hardware and software systems
  - Transparency of distribution
  - General/reusable services for distributed applications

Middleware and distributed programming

- Middleware aims at making distributed progamming easier
  - Software development, evolution, reusability
  - Portability of applications between platforms
  - Interoperability between heterogeneous applications

Middleware examples

- CORBA
- Sun JVM
- Microsoft .NET
- Sun J2EE / EJB
- …

Why adaptable middleware?

- Adaptation of middleware and applications
  - Dynamic discovery of services
  - Dynamic reconfiguration
  - Adaptive behavior

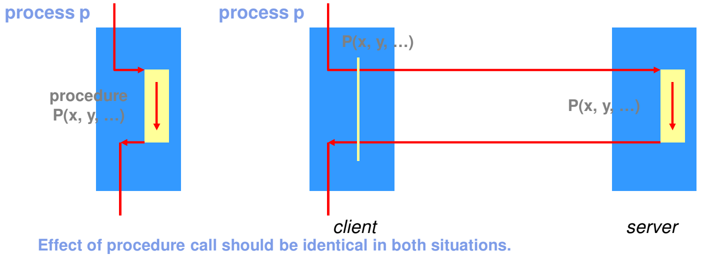
Types of middleware

- Classification criteria
  - Nature of communicating entities
    - Objects
    - Components
    - Others
  - Access mode to services
    - Synchronous (client-server)
    - Asynchronous (event-based)
    - Hybrid
  - Other criteria
    - Static vs. mobile entities
    - Guaranteed vs. non-guaranteed QoS

No rigorous classification, different implementations

A simple middleware example: RPC

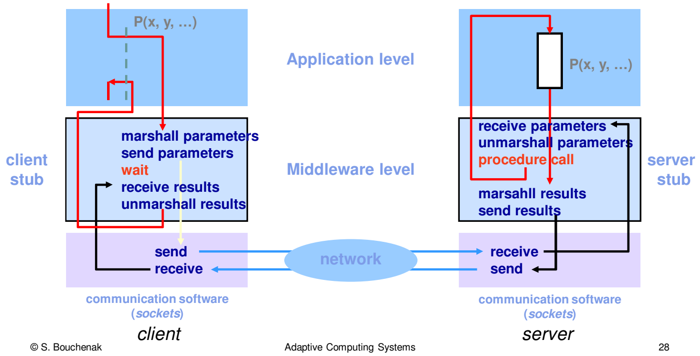
- Remote procedure call (RPC), a tool to build client-server distributed applications

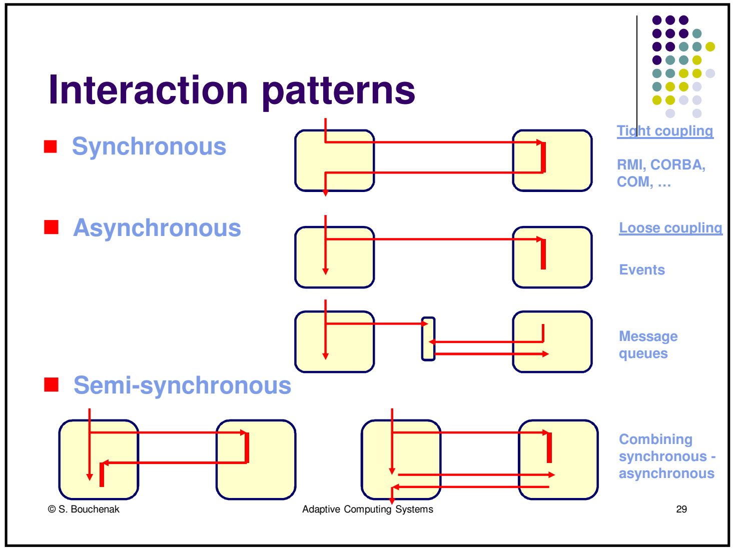


Effect of procedure call should be identical in both situations.
Impossible in case of failures.

A simple middleware example: RPC (2)

- Implementation of remote procedure call

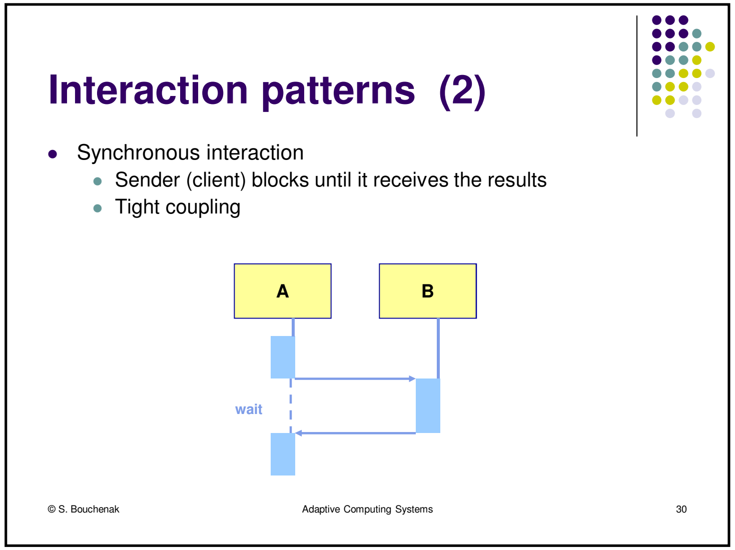




Interaction patterns

- Synchronous
- Asynchronous
- Semi-synchronous

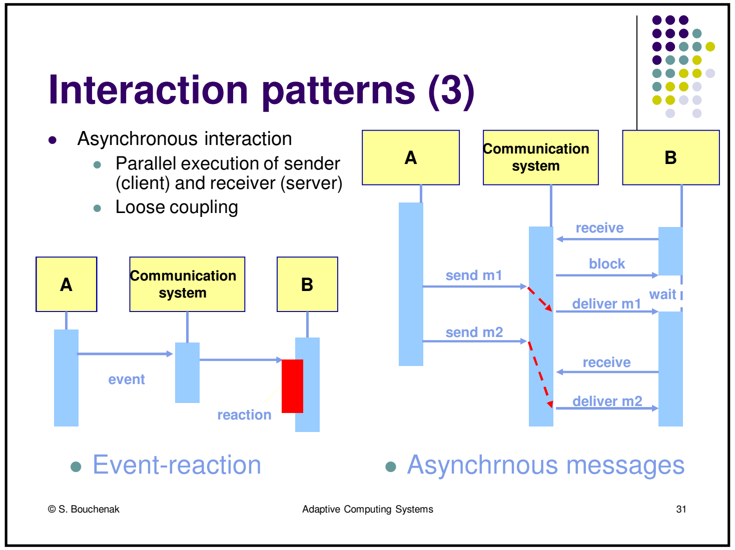
© S. Bouchenak Adaptive Computing Systems 29



Interaction patterns (2)

- Synchronous interaction
  - Sender (client) blocks until it receives the results
  - Tight coupling

© S. Bouchenak Adaptive Computing Systems 30

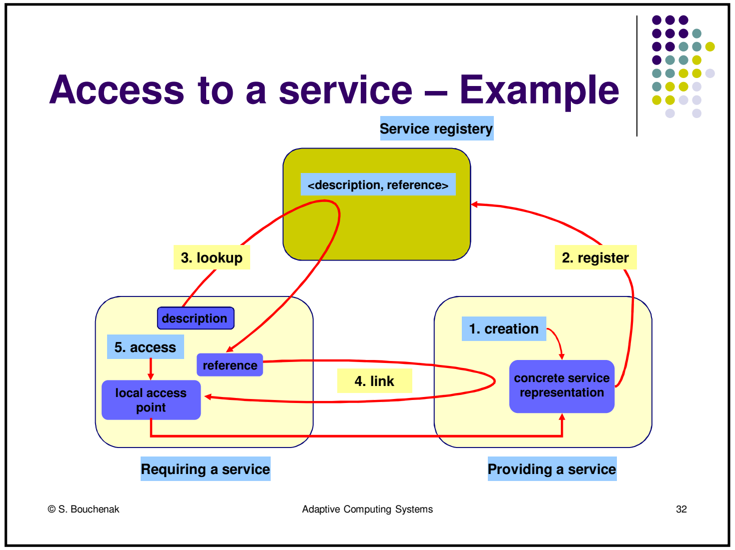


Interaction patterns (3)

- Asynchronous interaction
  - Parallel execution of sender (client) and receiver (server)
  - Loose coupling

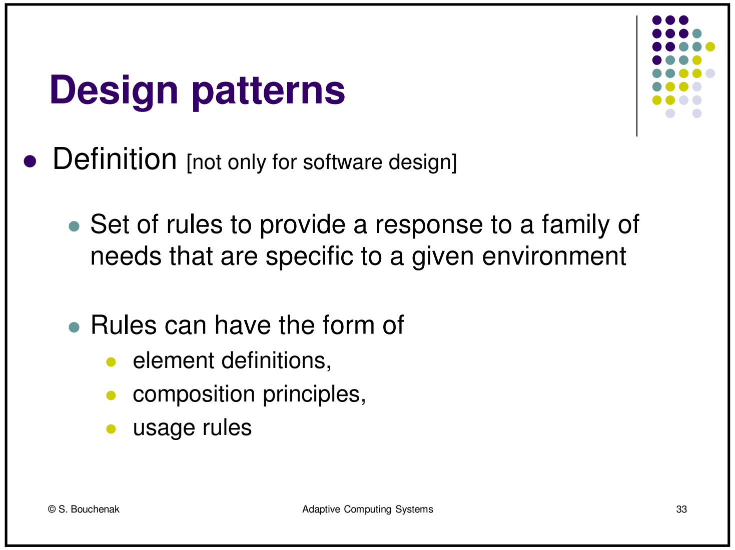
- Event-reaction
- Asynchrnous messages

© S. Bouchenak Adaptive Computing Systems 31



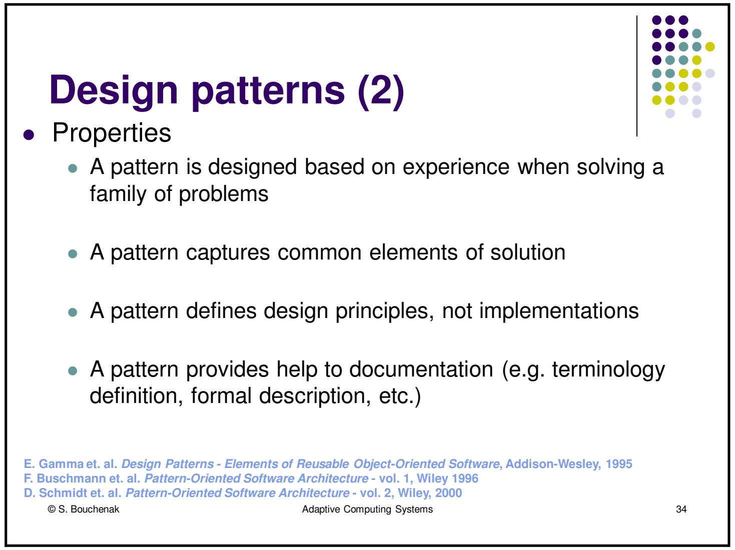
Access to a service – Example

© S. Bouchenak Adaptive Computing Systems 32



Design patterns

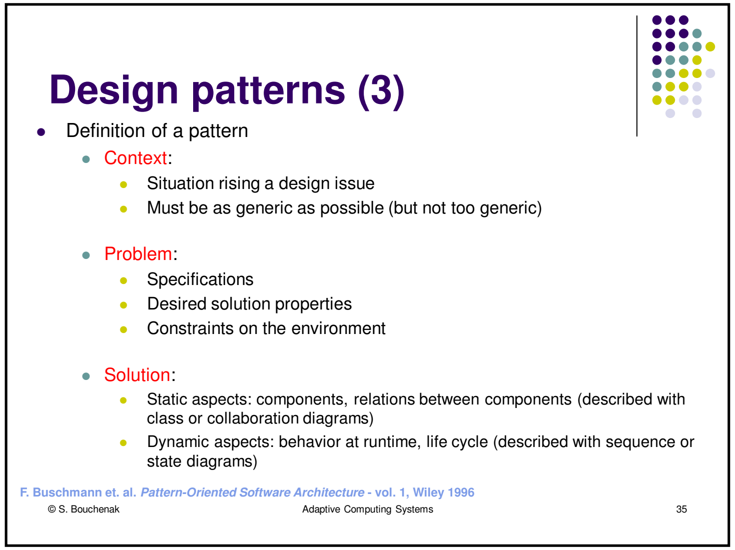
- Definition [not only for software design]
  - Set of rules to provide a response to a family of needs that are specific to a given environment
  - Rules can have the form of
    - element definitions,
    - composition principles,
    - usage rules



Design patterns (2)

- Properties
  - A pattern is designed based on experience when solving a family of problems
  - A pattern captures common elements of solution
  - A pattern defines design principles, not implementations
  - A pattern provides help to documentation (e.g. terminology definition, formal description, etc.)

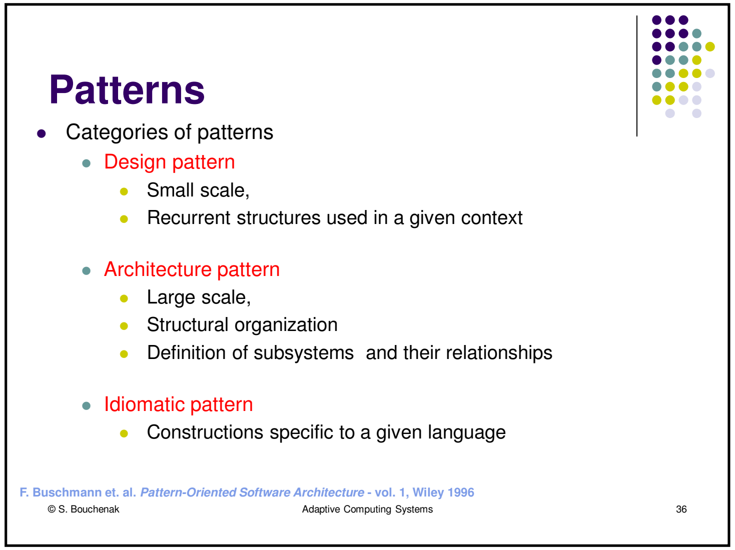
E. Gamma et. al. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
F. Buschmann et. al. *Pattern-Oriented Software Architecture* - vol. 1, Wiley 1996
D. Schmidt et. al. *Pattern-Oriented Software Architecture* - vol. 2, Wiley, 2000



Design patterns (3)

- Definition of a pattern
  - Context:
    - Situation rising a design issue
    - Must be as generic as possible (but not too generic)
  - Problem:
    - Specifications
    - Desired solution properties
    - Constraints on the environment
  - Solution:
    - Static aspects: components, relations between components (described with class or collaboration diagrams)
    - Dynamic aspects: behavior at runtime, life cycle (described with sequence or state diagrams)

F. Buschmann et. al. *Pattern-Oriented Software Architecture* - vol. 1, Wiley 1996



Patterns

- Categories of patterns
  - Design pattern
    - Small scale,
    - Recurrent structures used in a given context
  - Architecture pattern
    - Large scale,
    - Structural organization
    - Definition of subsystems and their relationships
  - Idiomatic pattern
    - Constructions specific to a given language

F. Buschmann et. al. *Pattern-Oriented Software Architecture* - vol. 1, Wiley 1996

Examples of patterns

- *Proxy*
  - Design pattern: representative for remote access
- *Factory*
  - Design pattern: object creation
- *Wrapper* [*Adapter*]
  - Design pattern: interface transformation
- *Interceptor*
  - Architecture pattern: service adaptation

These patterns are largely used in middleware implementations

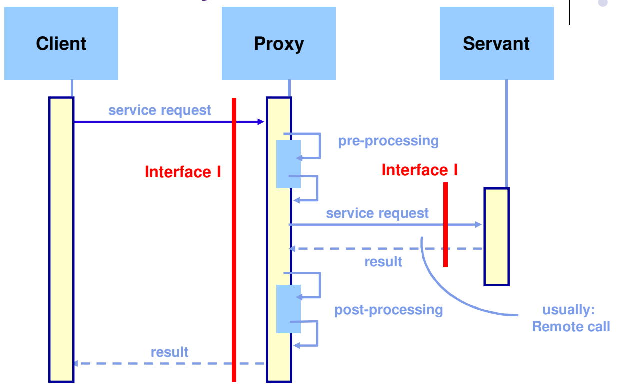
Proxy (Representative)

- Context
  - Applications as sets of distributed objects;
  - Client accesses services provided by a possibly remote object (servant)
- Problem
  - Define service access mechanisms that prevent
    - hand-coding cserver location in client code
    - having a detailed knowledge of communication protocols
  - Desired properties
    - efficient and dependable acces
    - simple programming model for client (ideally, no difference between local and remote service access)
  - Constraints
    - Distributed environment (no shared memory)

Proxy (Representative) (2)

- Solutions
  - Servant representative used locally at client-side (hide servant, and communication system to client)
  - Servant representative exposes same interface as servant
  - Define a uniform servant structure to ease its automatic generation

Use of *Proxy*



Client

Proxy

Servant

service request

Interface I

pre-processing

Interface I

service request

result

post-processing

usually: Remote call

result

Examples of patterns

- *Proxy*
  - Design pattern: representative for remote access
- ***Factory***
  - **Design pattern: object creation**
- *Wrapper* [*Adapter*]
  - Design pattern: interface transformation
- *Interceptor*
  - Architecture pattern: service adaptation

These patterns are largely used in middleware implementations

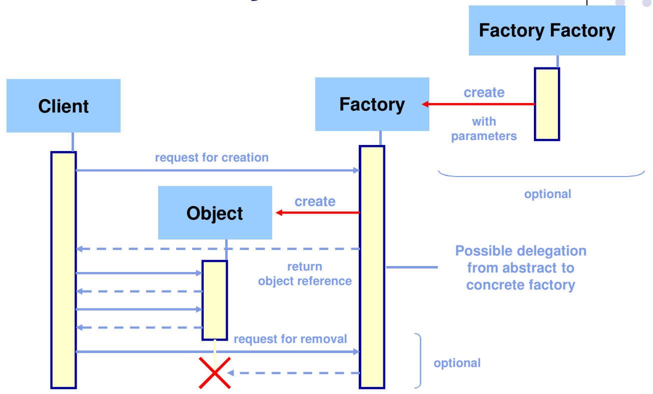
Factory

- Context
  - Application = set of objects in a distributed environment
- Problem
  - Dynamic creation of multiple instances of a class of objects
  - Desired properties
    - Instances may be parameterized
    - Easy evolution (no hand-coded decision)
  - Constraints
    - Distributed environment (no shared memory)

Factory (2)

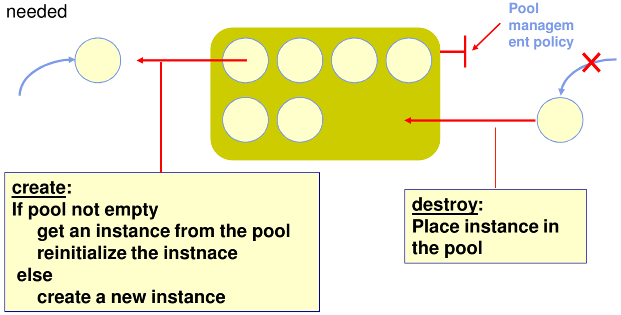
- Solutions
  - *Abstract Factory*
    - Define an interface and a generic organization for object creation
    - Effective object creation is delegated to a concrete factory that implements creation methods

Use of *Factory*



Use of a Pool in a *Factory*

- Problem: online resource (e.g. objet) creation is expensive
- Objective: reduce costs underlying resource creation
- Technique: create a set of resources in advance and reuse them whenever needed



<u>create</u>:
```
If pool not empty
    get an instance from the pool
    reinitialize the instnace
else
    create a new instance
```

<u>destroy</u>: Place instance in the pool

Examples of use of *Pool*

- Memory management
  - *Pool* of memory regions (of possibly different sizes)
  - Prevent the overhead of garbage-collection
- Activity management
  - *Pool* of *threads*
  - Prevent overhead of online thread creation
- Communication management
  - *Pool* of connections
  - Prevent cost of online communication channel creation

Examples of patterns

- *Proxy*
  - Design pattern: representative for remote access
- *Factory*
  - Design pattern: object creation
- ***Wrapper* [*Adapter*]**
  - **Design pattern: interface transformation**
- *Interceptor*
  - Architecture pattern: service adaptation

These patterns are largely used in middleware implementations

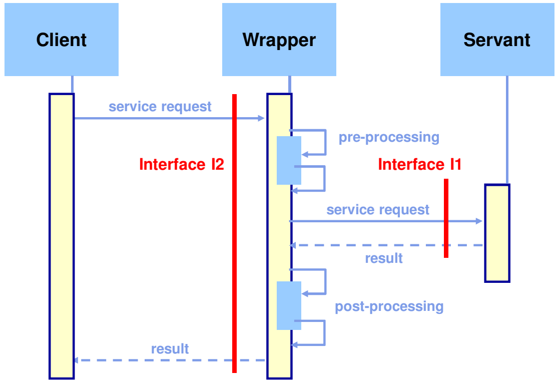
Wrapper (or *Adapter*)

- Context
  - Clients require services
  - Servants provide services
  - Services defined through interfaces
- Problem
  - Reuse an existing servant, while modifying its interface/functions to satisfy client needs (or a subset of clients)
  - Desired properties: efficiency, reusable and adaptable to different needs

Wrapper (or *Adapter*) (2)

- Solutions
  - *Wrapper* isolates servant by intercepting calls to servant interface
  - Each call to servant interface is preceded by by a prologue and followed by an epilogue in the *Wrapper*
  - Parameters of servant interface calls and results of calls can be modified

Use of *Wrapper*



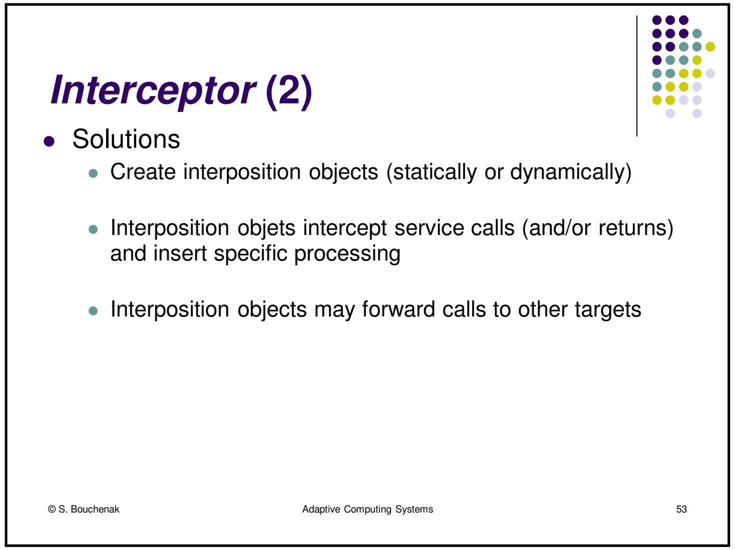
Examples of patterns

- *Proxy*
  - Design pattern: representative for remote access
- *Factory*
  - Design pattern: object creation
- *Wrapper* [*Adapter*]
  - Design pattern: interface transformation
- ***Interceptor***
  - **Architecture pattern: service adaptation**

These patterns are largely used in middleware implementations

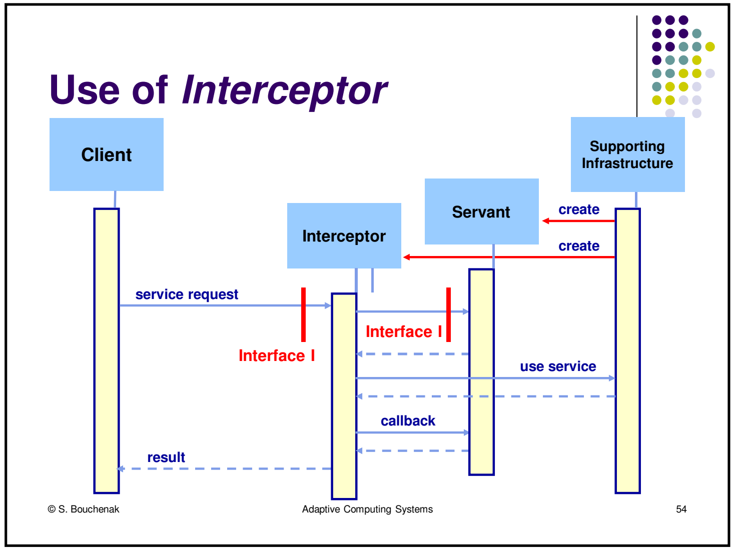
Interceptor

- Context
  - Provide services
    - Client-server, peer-to-peer, hierarchical
    - Uni- or bi-directional, synchronous or asynchronous
- Problem
  - Transform a service (add new functions)
    - Add a new processing level (cf. *Wrapper*)
    - Modify the target of the call
  - Constraints
    - Client and server programs must not be modified
    - Services may be dynamically added or removed

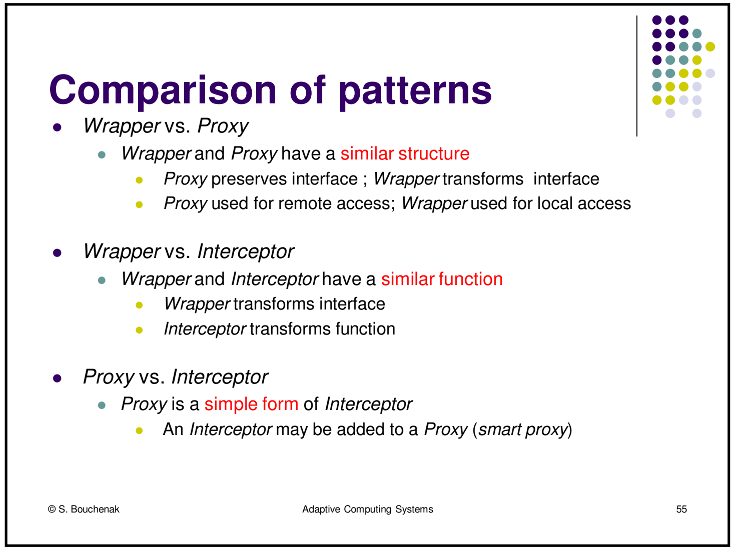


Interceptor (2)

- Solutions
  - Create interposition objects (statically or dynamically)
  - Interposition objets intercept service calls (and/or returns) and insert specific processing
  - Interposition objects may forward calls to other targets

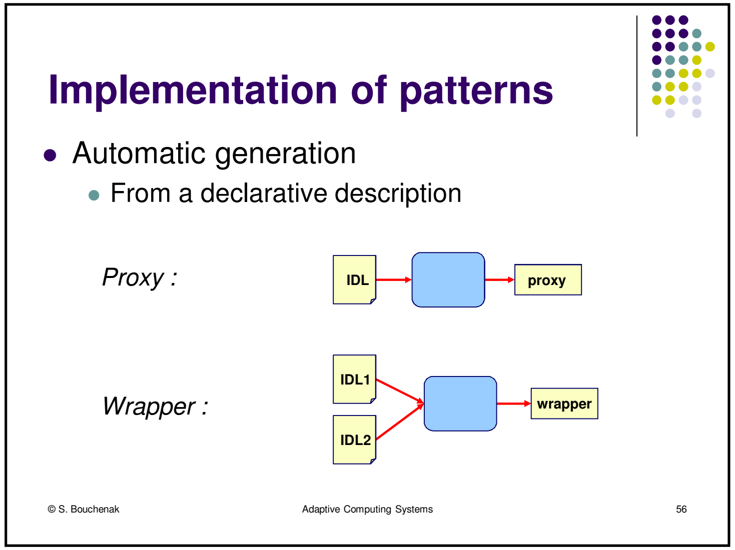


Use of *Interceptor*



Comparison of patterns

- *Wrapper* vs. *Proxy*
  - *Wrapper* and *Proxy* have a similar structure
    - *Proxy* preserves interface ; *Wrapper* transforms interface
    - *Proxy* used for remote access; *Wrapper* used for local access
- *Wrapper* vs. *Interceptor*
  - *Wrapper* and *Interceptor* have a similar function
    - *Wrapper* transforms interface
    - *Interceptor* transforms function
- *Proxy* vs. *Interceptor*
  - *Proxy* is a simple form of *Interceptor*
    - An *Interceptor* may be added to a *Proxy* (*smart proxy*)



Implementation of patterns

- Automatic generation
  - From a declarative description

Proxy :

Wrapper :

Implementation of patterns (2)

- Optimizations
  - Eliminate indirections (performance overhead)
    - Shorten indirection chains
    - Code injection (insertion of generated code in application code)
    - Low-level code generation (e.g. Java bytecode)
    - Reversible techniques (for adaptation)

Software frameworks

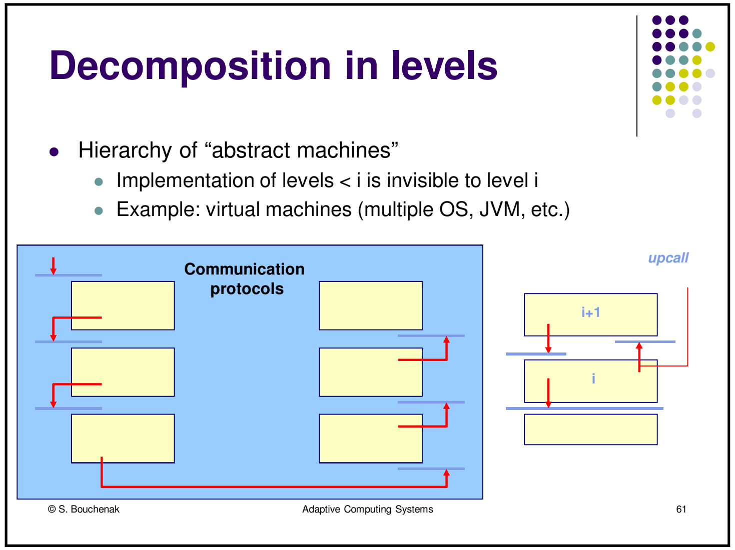
- Definition
  - A framework is a programme "squeleton" that can be used (adapted) for a famility of applications
  - A framework implements a model (not always explicit)
  - In object-oriented languages, a framework consists in
    - A set of (abstract) classes that must be adapted (via inheritance) to different contexts
    - A set of usage rules for these classes

Software frameworks (2)

- Patterns and frameworks
  - Two techniques for reuse
  - Pattrens reuse design principles
  - Frameworks reuse code implementation
  - A framework usually implement one or more patterns

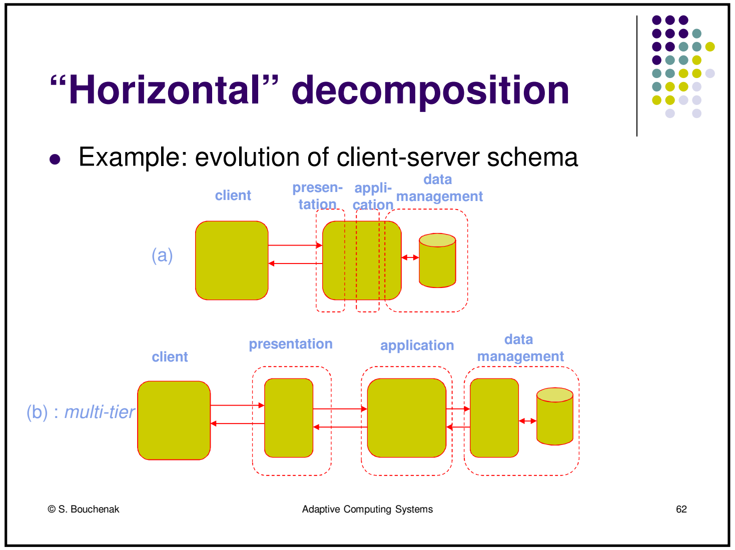
Decomposition schemes

- Objectives
  - Ease software development
    - Structure reflects design approach
    - Interfaces and inter-dependencies are exhibited
  - Ease software evolution
    - Encapsulation
- Example
  - Multi-level structures
    - “verticale” or “horizontal” decomposition



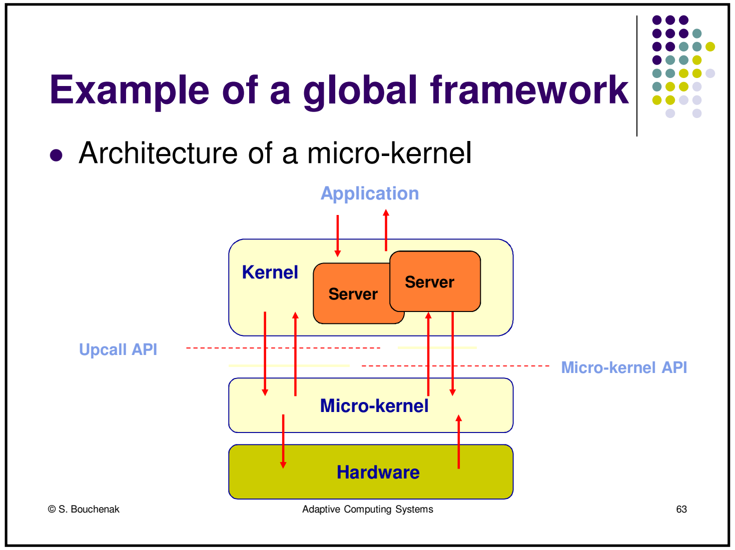
Decomposition in levels

- Hierarchy of “abstract machines”
  - Implementation of levels < i is invisible to level i
  - Example: virtual machines (multiple OS, JVM, etc.)



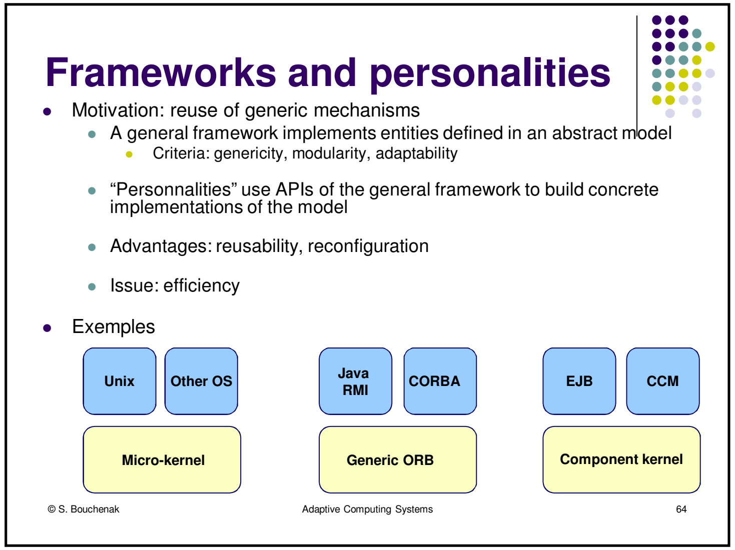
“Horizontal” decomposition

- Example: evolution of client-server schema



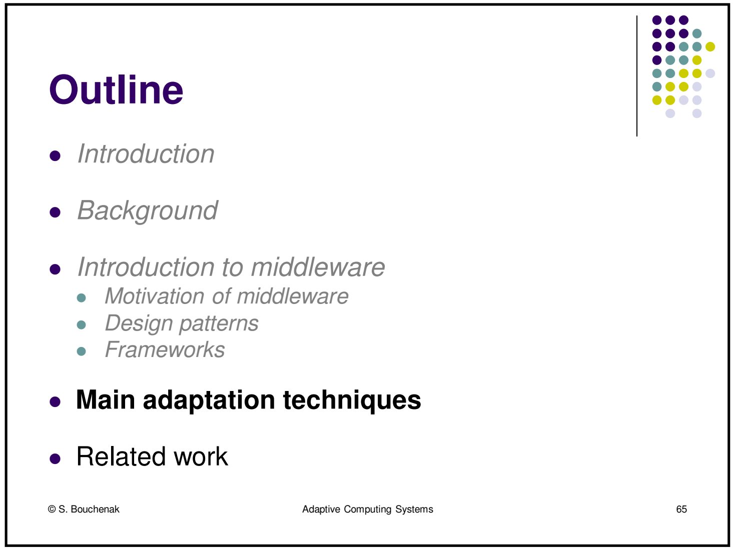
Example of a global framework

- Architecture of a micro-kernel



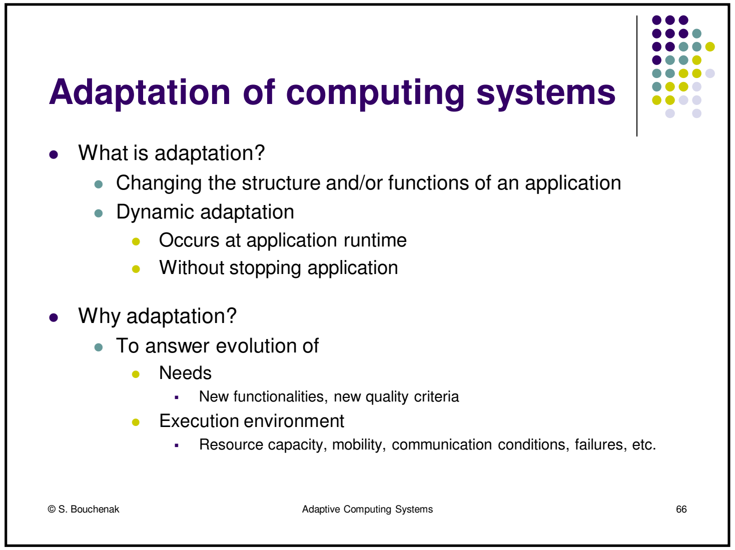
Frameworks and personalities

- Motivation: reuse of generic mechanisms
  - A general framework implements entities defined in an abstract model
    - Criteria: genericity, modularity, adaptability
  - “Personnalities” use APIs of the general framework to build concrete implementations of the model
  - Advantages: reusability, reconfiguration
  - Issue: efficiency
- Exemples



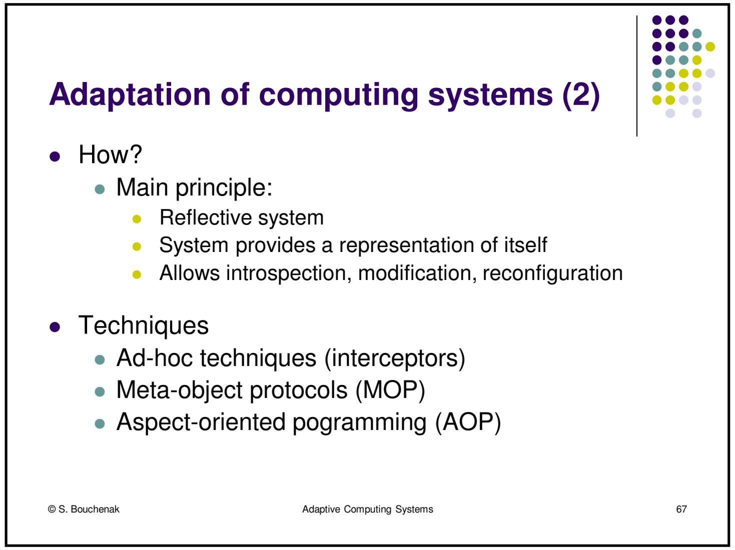
Outline

- *Introduction*
- *Background*
- *Introduction to middleware*
  - *Motivation of middleware*
  - *Design patterns*
  - *Frameworks*
- **Main adaptation techniques**
- Related work



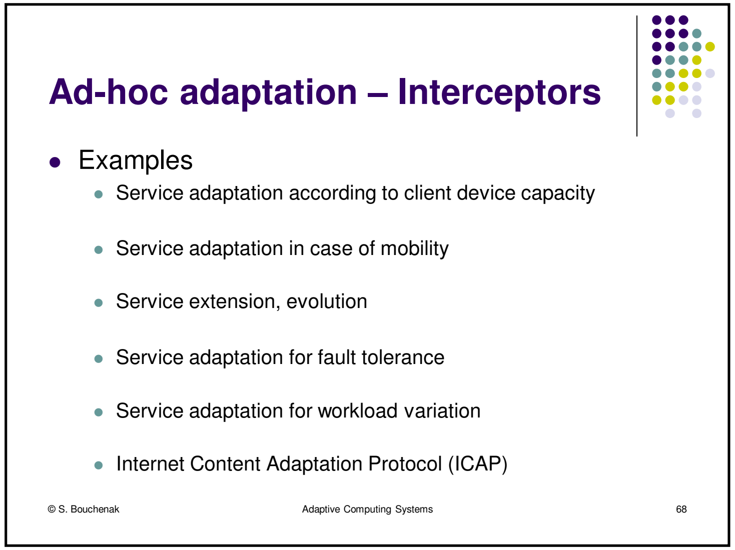
Adaptation of computing systems

- What is adaptation?
  - Changing the structure and/or functions of an application
  - Dynamic adaptation
    - Occurs at application runtime
    - Without stopping application
- Why adaptation?
  - To answer evolution of
    - Needs
      - New functionalities, new quality criteria
    - Execution environment
      - Resource capacity, mobility, communication conditions, failures, etc.



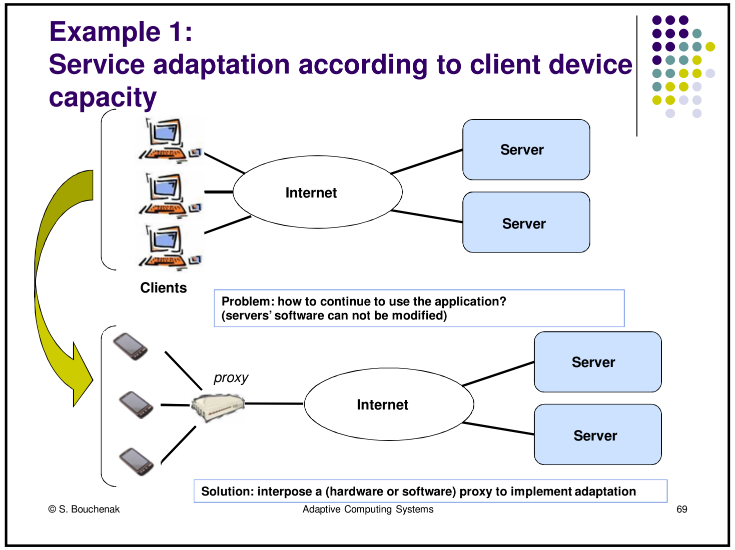
Adaptation of computing systems (2)

- How?
  - Main principle:
    - Reflective system
    - System provides a representation of itself
    - Allows introspection, modification, reconfiguration
- Techniques
  - Ad-hoc techniques (interceptors)
  - Meta-object protocols (MOP)
  - Aspect-oriented pogramming (AOP)



Ad-hoc adaptation – Interceptors

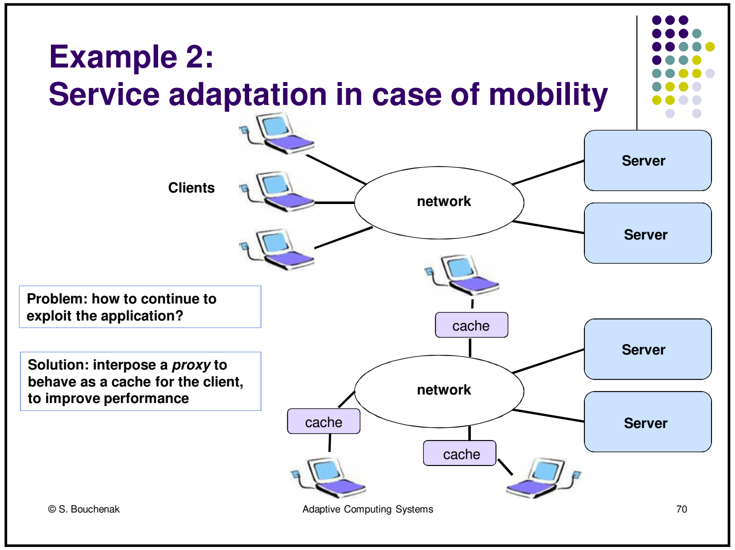
- Examples
  - Service adaptation according to client device capacity
  - Service adaptation in case of mobility
  - Service extension, evolution
  - Service adaptation for fault tolerance
  - Service adaptation for workload variation
  - Internet Content Adaptation Protocol (ICAP)



Example 1: Service adaptation according to client device capacity

Problem: how to continue to use the application?
(servers' software can not be modified)

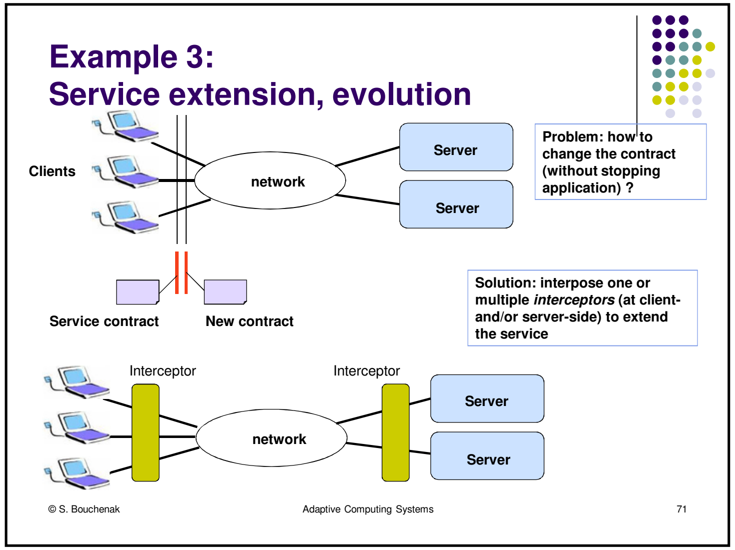
Solution: interpose a (hardware or software) proxy to implement adaptation



Example 2: Service adaptation in case of mobility

Problem: how to continue to exploit the application?

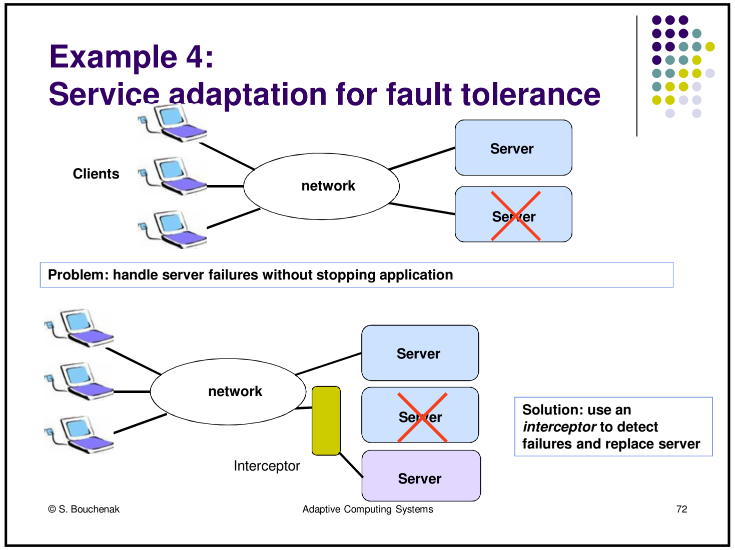
Solution: interpose a *proxy* to behave as a cache for the client, to improve performance



Example 3: Service extension, evolution

Problem: how to change the contract (without stopping application) ?

Solution: interpose one or multiple *interceptors* (at client- and/or server-side) to extend the service

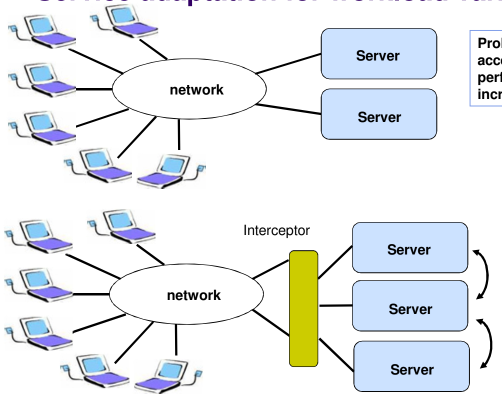


Example 4: Service adaptation for fault tolerance

Problem: handle server failures without stopping application

Solution: use an *interceptor* to detect failures and replace server

Example 5:
Service adaptation for workload variation

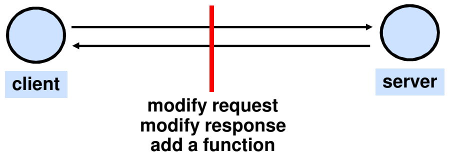


Problem: how to maintain an acceptable level of performance when workload increases?

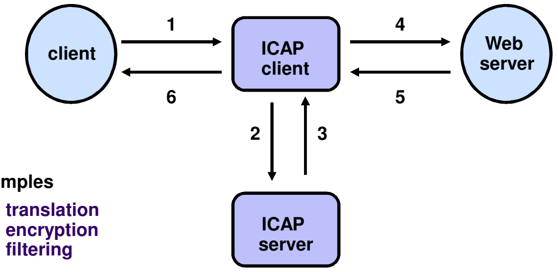
Solution: dynamic server provisioning and load balancing via an *interceptor*

Example 6:
ICAP (Internet Content Adaptation Protocol) protocol

- Definition
  - A lightweight HTTP-like protocol used to extend transparent proxy servers
- Motivations
  - Implement functions (virus scanning, content filtering, etc.)
  - Off-loading value-added services from Web servers to ICAP servers
- How it works
  - interposition in an HTTP client-server system



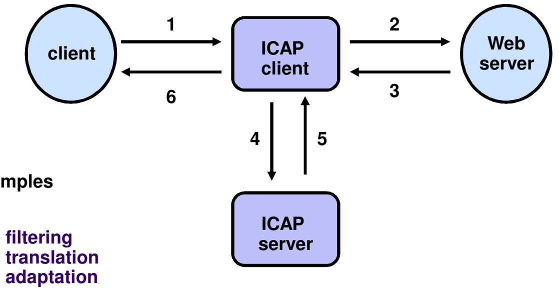
ICAP protocol: modify a request



Examples

translation
encryption
filtering

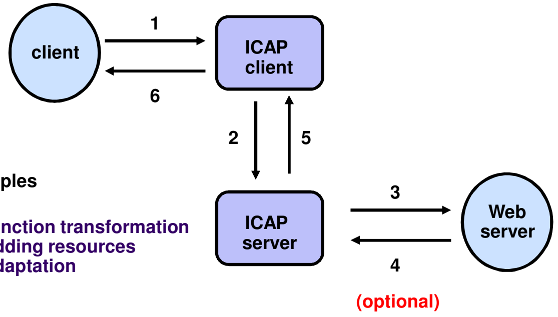
ICAP protocol: modify a response



Examples

filtering
translation
adaptation
Advertisemnt insertion

ICAP protocol: interpose a function



Examples

function transformation
adding resources
adaptation

Adaptation of computing systems

- How?
  - Main principle:
    - Reflective system
    - System provides a representation of itself
    - Allows introspection, modification, reconfiguration
- Techniques
  - *Ad-hoc techniques (interceptors)*
  - **Meta-object protocols (MOP)**
  - Aspect-oriented pogramming (AOP)

Meta-object protocol (MOP)

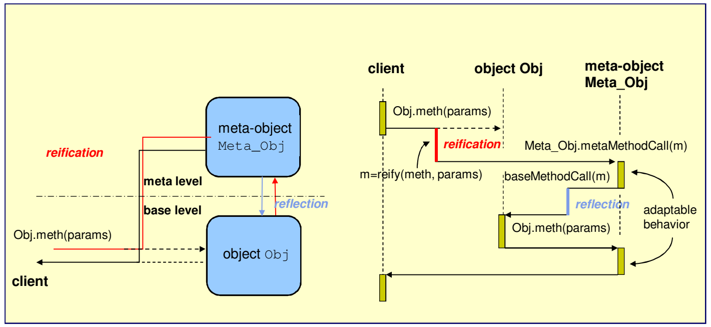
- An adaptable service is organized in two levels
  - Base level
    - Implement functions defined by specifications
  - Meta-level
    - Use a representation of the base level to observe or modify its behavior
    - This meta-level representation is causally connected to the base level

Meta-object protocol (2)

- Relations between levels
  - Creation of the representation of an entity: reification
  - Action of the meta-level on the base level: reflection
- This organization may be repeated recursively
  - “Reflective tour” : meta-meta-level, etc.
  - In practice, 2 or 3 levels

Meta-object protocol: example

- Reification of a method call:



Adaptation of computing systems

- How?
  - Main principle:
    - Reflective system
    - System provides a representation of itself
    - Allows introspection, modification, reconfiguration
- Techniques
  - *Ad-hoc techniques (interceptors)*
  - *Meta-object protocols (MOP)*
  - **Aspect-oriented pogramming (AOP)**

Aspect-oriented programming (AOP)

- Main principle
  - Separate concerns
  - Idenify a basic behavior and additional “aspects” as independent as possible
  - Separately describe the basic behavior and aspects
  - Integrate all elements in a unique program
- Methodology
  - Individual description of each aspect
  - Integration (“weaving”) of aspects, static or dynamic weaving

Aspect-oriented programming (2)

- Definitions
  - *Join point*
    - point where to insert aspect code
  - *Pointcut*
    - Set of join points logically correlated
  - *Advice*
    - definition of relations basetween inserted code and base code (e.g.before, after, etc.)

Aspect-oriented programming: example

- Implementing a *Wrapper* in AspectJ

```
public aspect MethodWrapping  {

/* point cut definition */
    pointcut Wrappable(): call(public * MyClass.*(..));

 /* advice definition */
    around(): Wrappable() {
             <prelude>      /* a sequence of code to be inserted before the call */
             proceed();     /* performs the call to the original method  */
             <postlude>   /* a sequence of code to be inserted after the call */
    }
}
```

Result: encapsulate a call to a public method of class MyClass with <prelude> and <postlude>

Possible usage: logging, assertion test, etc.

Outline

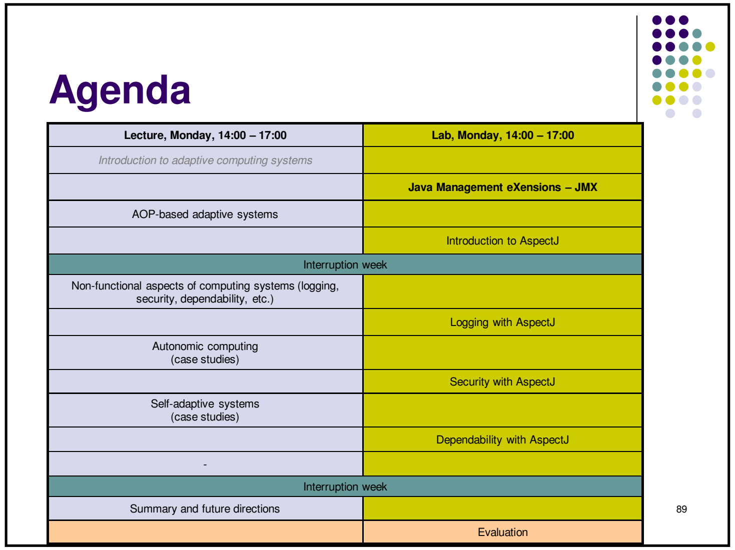
- *Introduction*
- *Background*
- *Introduction to middleware*
- *Main adaptation techniques*
  - *Motivations*
  - *Ad-hoc adaptation techniques*
  - *Meta-object protocols (MOP)*
  - *Aspect-oriented programming (AOP)*
- **Related work**

Related work

- SARDES research group (INRIA – LIG laboratory)
  - ~20 people
  - http://sardes.inrialpes.fr/
- Research topics :
  - middleware, distributed systems, cloud computing, autonomic computing
- SARDES =
  - *Systems Architecture for Reflective Distributed EnvironmentS*
  - *Self-Administrable and Reconfigurable Distributed EnvironmentS*

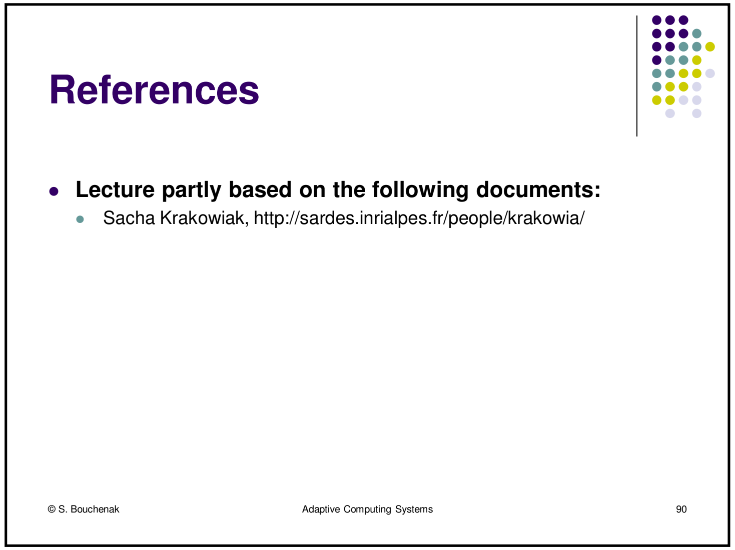
Related work (2)

- Collaborations
  - OW2 consortium
    - Open source middleware solutions
    - http://www.ow2.org/
  - Industrial partners
    - Bull
    - Microsoft
    - Orange Labs
    - ST Microelectronics
    - Start-ups: We Are Cloud, Scalagent, …
  - International collaborations
    - European projects
    - …



Agenda

Lecture, Monday, 14:00 – 17:00	Lab, Monday, 14:00 – 17:00
Introduction to adaptive computing systems	
	Java Management eXtensions – JMX
AOP-based adaptive systems	
	Introduction to AspectJ
Interruption week	
Non-functional aspects of computing systems (logging, security, dependability, etc.)	
	Logging with AspectJ
Autonomic computing (case studies)	
	Security with AspectJ
Self-adaptive systems (case studies)	
	Dependability with AspectJ
-	
Interruption week	
Summary and future directions	
	Evaluation



References

- **Lecture partly based on the following documents:**
  - Sacha Krakowiak, http://sardes.inrialpes.fr/people/krakowia/