



SeDeUse: A Model for Service-Oriented Computing in Dynamic Environments

Hervé Paulino and Carlos Tavares

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Mobilware 2009
Berlin, April 29th



Motivation

- Currently the possibility of using a service deployed anywhere in the world is a reality
- Use services to abstract resources in general, such as a Printer, not only businesses
- Existing coordination models, such Web service orchestration and choreography, do not handle dynamic environments with adequacy
 - Target business-specific rather than general interfaces
 - Couple resource usage and awareness



Motivation

- Several solutions have been proposed to overcome these limitations
 - Upgrade service description with semantic information by using ontologies
 - OWL-S, RDF, ...
 - Improve the composition mechanisms, BPEL in particular, by resorting to:
 - *Aspects: AOPBPEL*
 - Reflection: JOpera
 - Proxies: WS-Binder, MASC



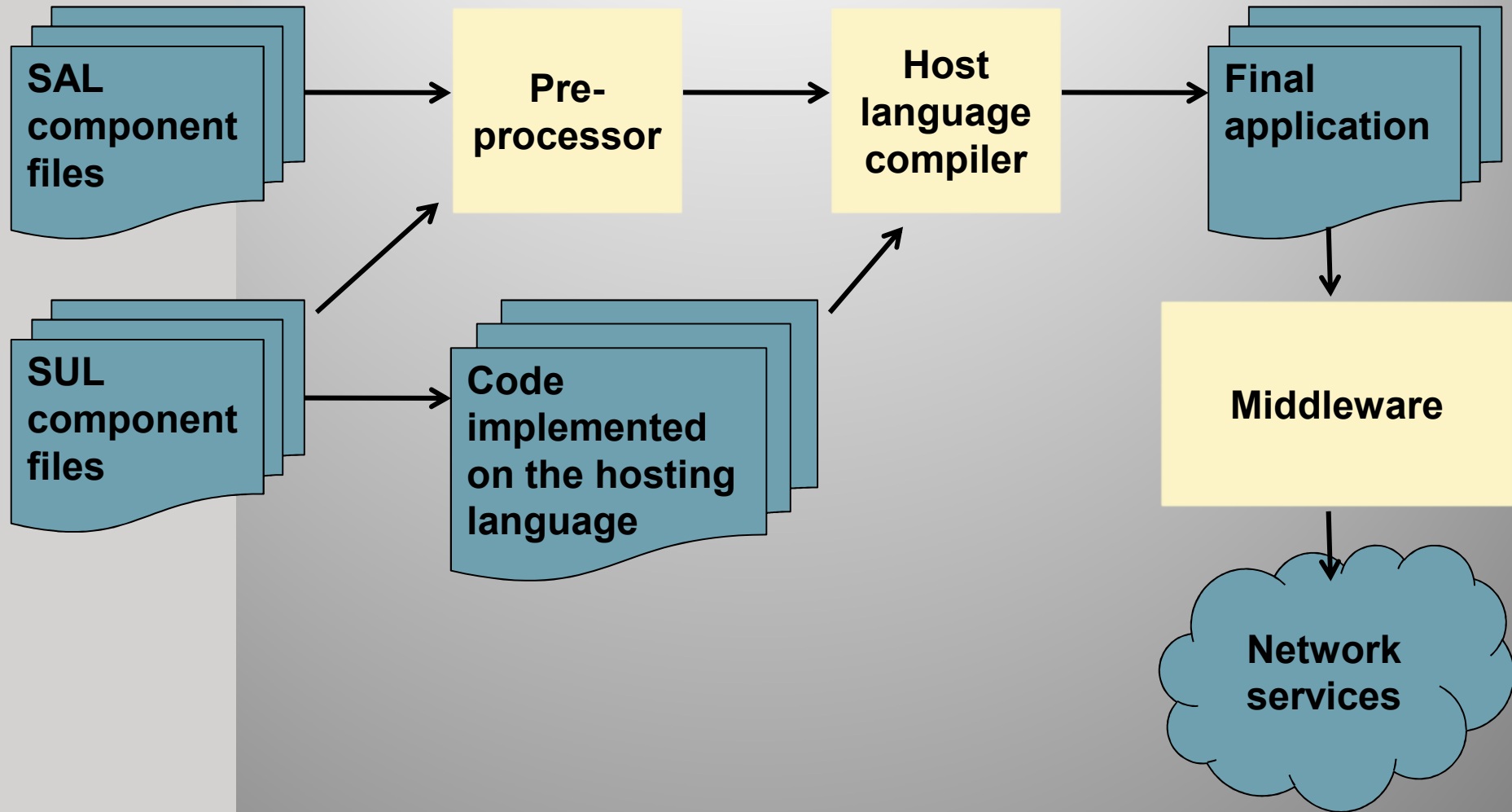
The SeDeUse Model

- Hide the idiosyncrasies of using SOC in dynamic environments
- Define a model orthogonal to common programming languages
 - Decouple service usage (functionality) from awareness (non-functionality)
 - Service Awareness Layer (SAL)
 - Service Usage Layer (SUL)
 - Code mobility transparent for SUL
 - Sustain its execution with a middleware that lives between the application and the standard service technologies



The SeDeUse Model

Compilation and execution





The SeDeUse Model

Identifiers and values

s, r

- *Service identifier*

o

- *Service operation identifier*

a

- *Variable identifier*

t

- *Type identifiers of the hosting language*

x

- *Exception identifiers of the hosting language*

v

- *Values of the hosting language*

A sequence of elements of a given syntactic category C is denoted by C^+



The SeDeUse Model

SAL - Syntax

Component

$D ::= D D$ *Sequence of declarations*
 | $s \{ A^+ \}$ *Service kind declaration*
 | $s \{ A^+ \} \text{alias } r$ *Service kind declaration with alias*

Attributes

$A ::= [\text{pref}] a = v$ *Attribute constraint*
 | $[\text{pref}] a \text{ in } \{ v^+ \}$ *Attribute soft constraint*



The SeDeUse Model

SAL - Examples

```
Printer {  
    colors = "b&w",  
    paper = "letter"  
}
```

```
Printer {  
    colors = "color",  
    pref paper = "a4"  
}
```



The SeDeUse Model

SAL - Examples

```
Printer {  
  colors = "color",  
  paper = "a4"  
} ColorPrinter
```

```
ColorPrinter {  
  type in {  
    "laser",  
    "inkjet"  
  }  
} PublicPrinter
```



The SeDeUse Model

SUL - Syntax

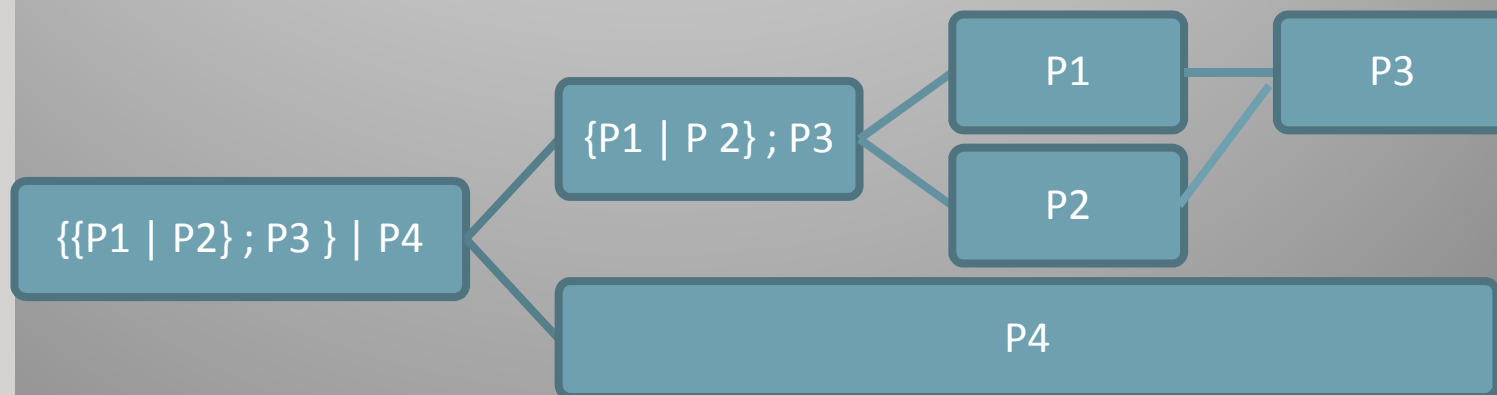
$P ::= \text{use } S^+ \text{ in } C(t_1 a_1 \dots t_n a_n) P X^+$	Service use abstraction
$P \mid P$	Parallel composition
$P ; P$	Sequential composition
$\{ P \}$	Grouping
$[a =] E$	Assignment
retry in e	Restart a transaction
i^+	Hosting language process
$E ::= \text{new } C(e^+)$	An instance of an use abstraction
$s.o(E^+) \mid s[e].o(E^+)$	Method invocation
e	Hosting language expression
$S ::= [\text{volatile}] E s$	
$[\text{volatile}] \text{ all } s$	Service allocation
$X ::= \text{catch } (x a) \{ P \}$	Exception handling



The SeDeUse Model

SUL – Defining computations

- Local computation is performed by sequences of instructions of the hosting language
- They can parallelly (|) and sequentially (;) composed
- Example: $\{ \{ P1 \mid P2 \} ; P3 \} \mid P4$





The SeDeUse Model

SUL – Using services

- The **use** construct allows to abstract computation in both local parameters and services

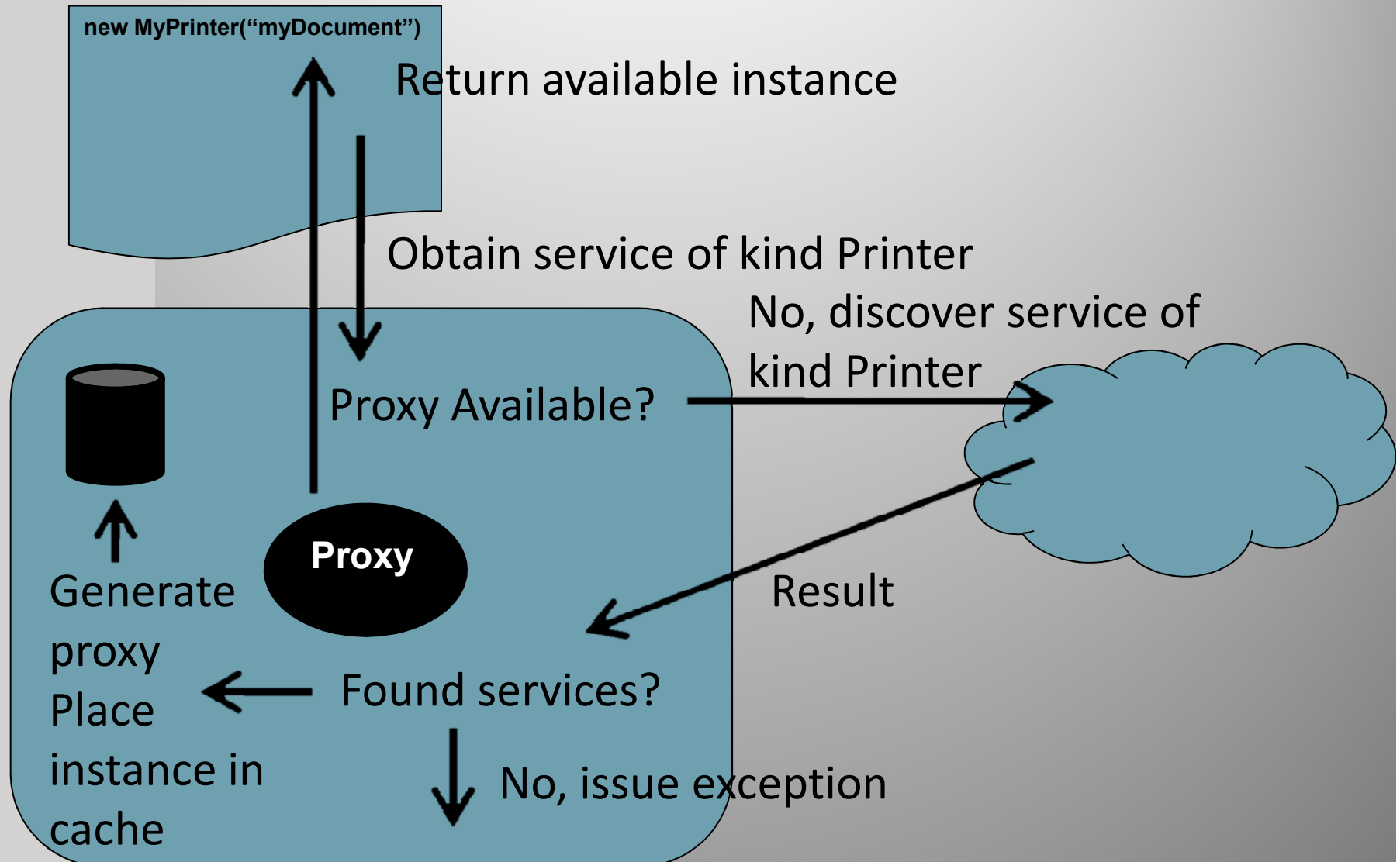
```
use Printer in MyPrinter (String doc) {  
    Printer.print(doc)  
}  
new MyPrinter("myDocument")
```

- The binding for *Printer* is obtained on-the-fly by the middleware layer



The SeDeUse Model

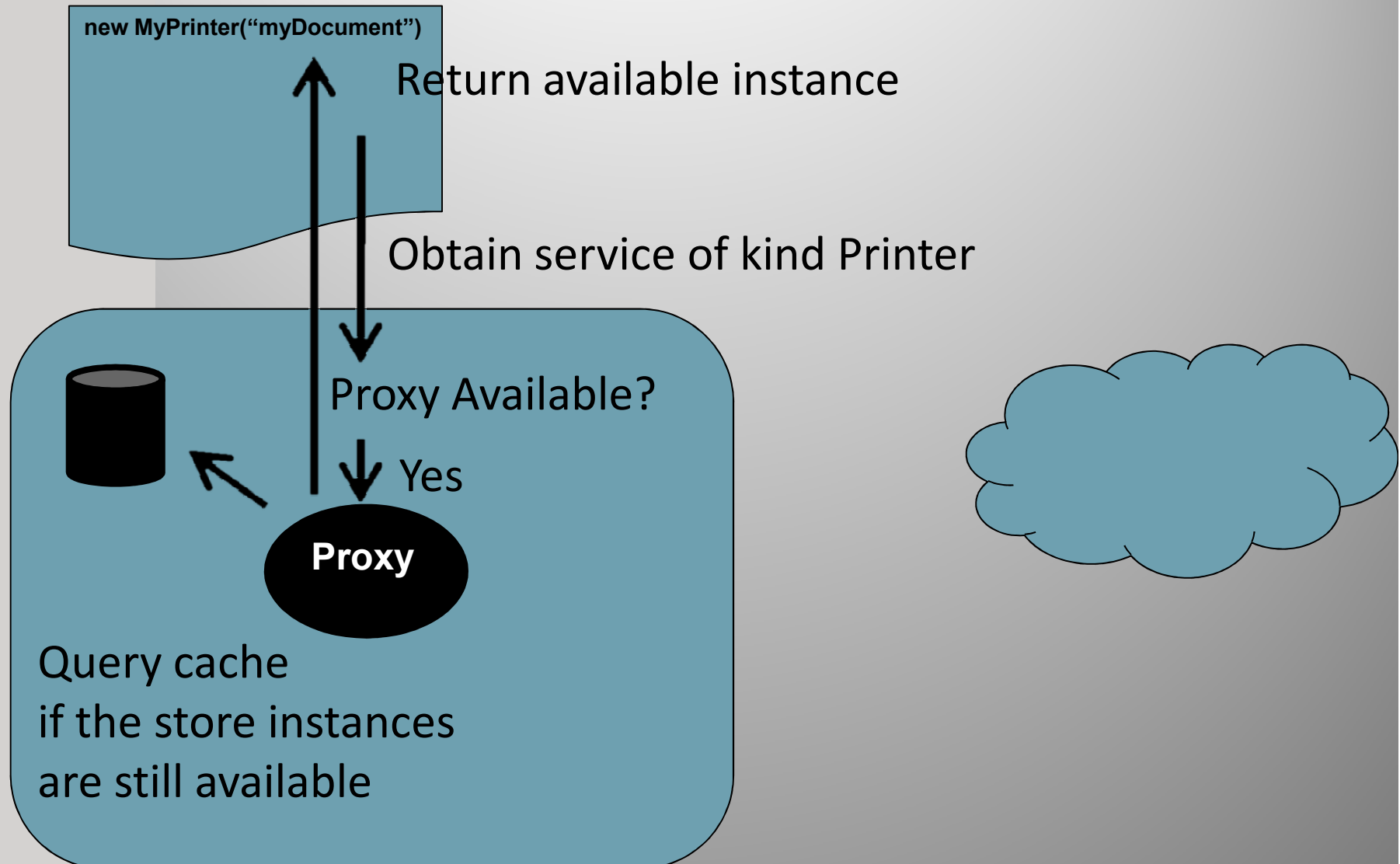
Obtaining an instance of a service kind





The SeDeUse Model

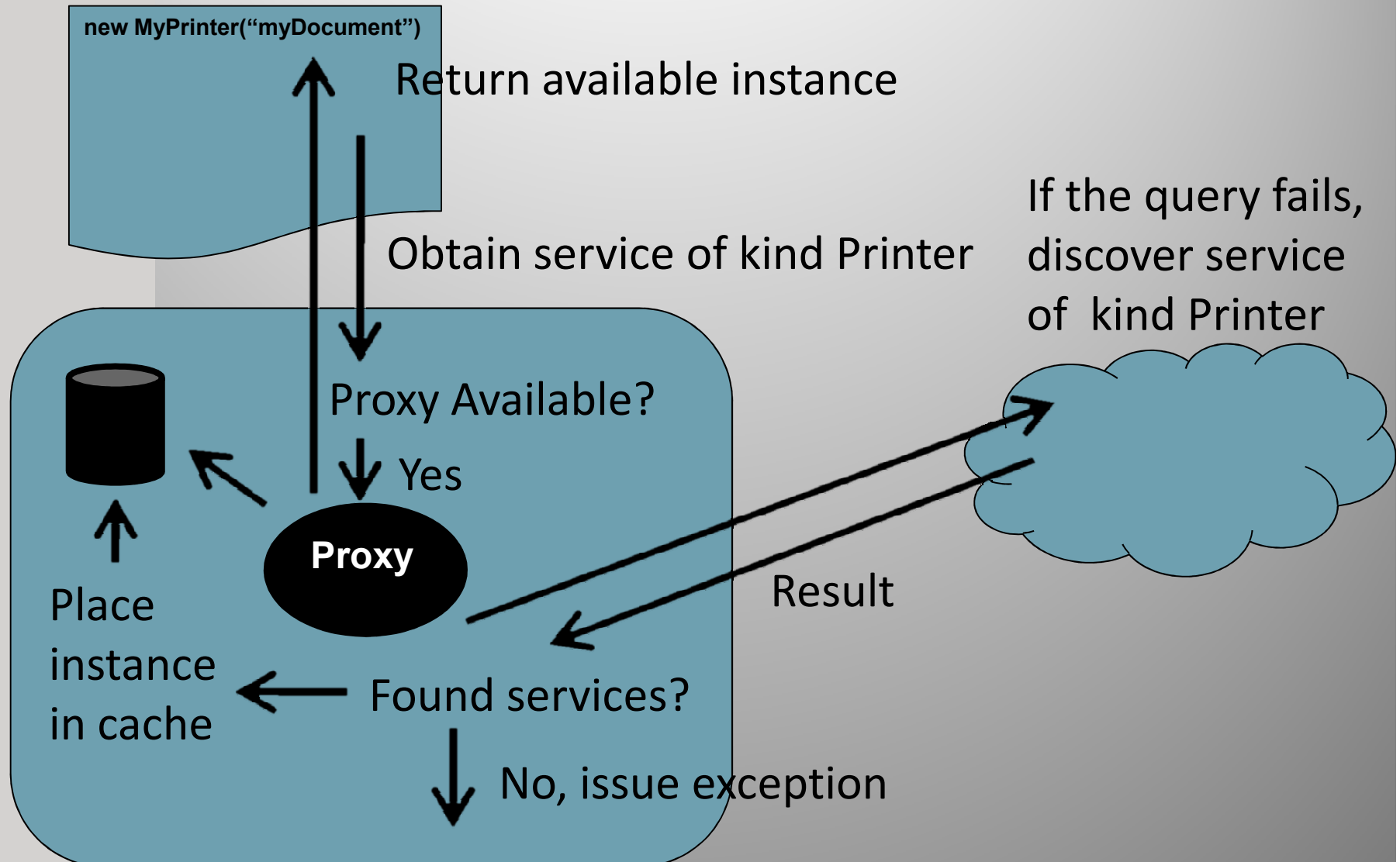
Obtaining an instance of a service kind





The SeDeUse Model

Obtaining an instance of a service kind





The SeDeUse Model

SUL – Using multiple service instances

- Use distinct instances of a service kind
 - Load-balance requests
 - Synchronize data between instances
- The value supplied defines an upper bound to avoid the raising of exceptions

```
use 2 SearchEngine in
    Search(String query) {
    SearchEngine[0].search(query) |
    SearchEngine[1].search(query)
    }
```



The SeDeUse Model

SUL – Using multiple service instances

- Indexes can be omitted
- $s.o(e^{\sim})$ is syntactic sugar for $s[i++].o(e^{\sim})$
 - i is initialized with 0
- Instances are ranged with a round-robin policy

```
use 2 SearchEngine in
    Search(String query) {
    SearchEngine.search(query) |
    SearchEngine.search(query)
    }
```



The SeDeUse Model

SUL – Failure recovery

- Exception handling mechanism is similar to Java
- However, it protects the use of the service rather than a sequence of instructions

```
use Printer in
    VirtualPrinter(VirtualPC vpc) {
        vpc.setPrinter(Printer)
    }
catch (ServiceException e) {
    vpc.unsetPrinter( )
}
new VirtualPrinter(vpc);
```



The SeDeUse Model

SUL – Failure recovery

- The code limited by **use** can be seen as a transaction
- If an exception is raised the transaction can be restarted by **retry**
 - This restarts the discovery procedure and eliminates the current instance from the cache

```
use Printer in
    VirtualPrinter(VirtualPC vpc) {
        vpc.setPrinter(Printer)
    }
catch (ServiceException e) { retry in 0 }
```



The SeDeUse Model

SUL – Stateless use

- Independent service invocations such as:

```
use Service in A() { Service.op1() }  
new A(); new A()
```

- Can be written as follows:

```
use volatile Service in A() {  
    Service.op1() ;  
    Service.op2()  
}  
new A()
```

- If the binding to an instance of Service is lost a new will be obtained without issuing an exception



The SeDeUse Model

Handling software mobility

- No explicit references to mobility
 - The program is not ordered to visit a given host
- A special attribute (@) in the SAL allows the programmer to state if a resource should be:
 - local local to the device
 - remote remote to the device
 - coupled local to the computation
 - closest as close to the computation as possible
 - performance the one that provides better performance



The SeDeUse Model

Handling software mobility - example

```
Printer { @ = "closest" }
```

```
Display { type = "TFT" }
```

```
CPU {  
    processor in { "Intel", "AMD" },  
    OS = "Linux",  
    @ = "coupled"  
}
```



Conclusions

- Model with novel and simple abstractions orthogonal to the common programming languages
- Resource usage completely separated from resource awareness
- Push technology-dependent details to the middleware
- Software mobility expressed in a non-functional manner
 - The same code can be used with and without mobility



Future Work

- Ongoing work
 - Application of the model to Java (SeDJ)
 - Implementation of the middleware in Java using Web service technologies
- Future work
 - Intelligent service discovery, based on service interface compliance
 - Porting to mobile phones



Thank you