Distributed Systems

(3rd Edition)

Chapter 02: Architectures

Version: March 20, 2022

Architectural styles

Basic idea

A style is formulated in terms of

- (replaceable) components with well-defined interfaces
- the way that components are connected to each other
- the data exchanged between components
- how these components and connectors are jointly configured into a system.

Architectural styles

Component

A component is a modular unit with well-defined required and provided interfaces that is replaceable within its environment.

Connector

A mechanism that mediates communication, coordination, or cooperation among components. Example: facilities for (remote) procedure call, messaging, or streaming.

Architectural styles

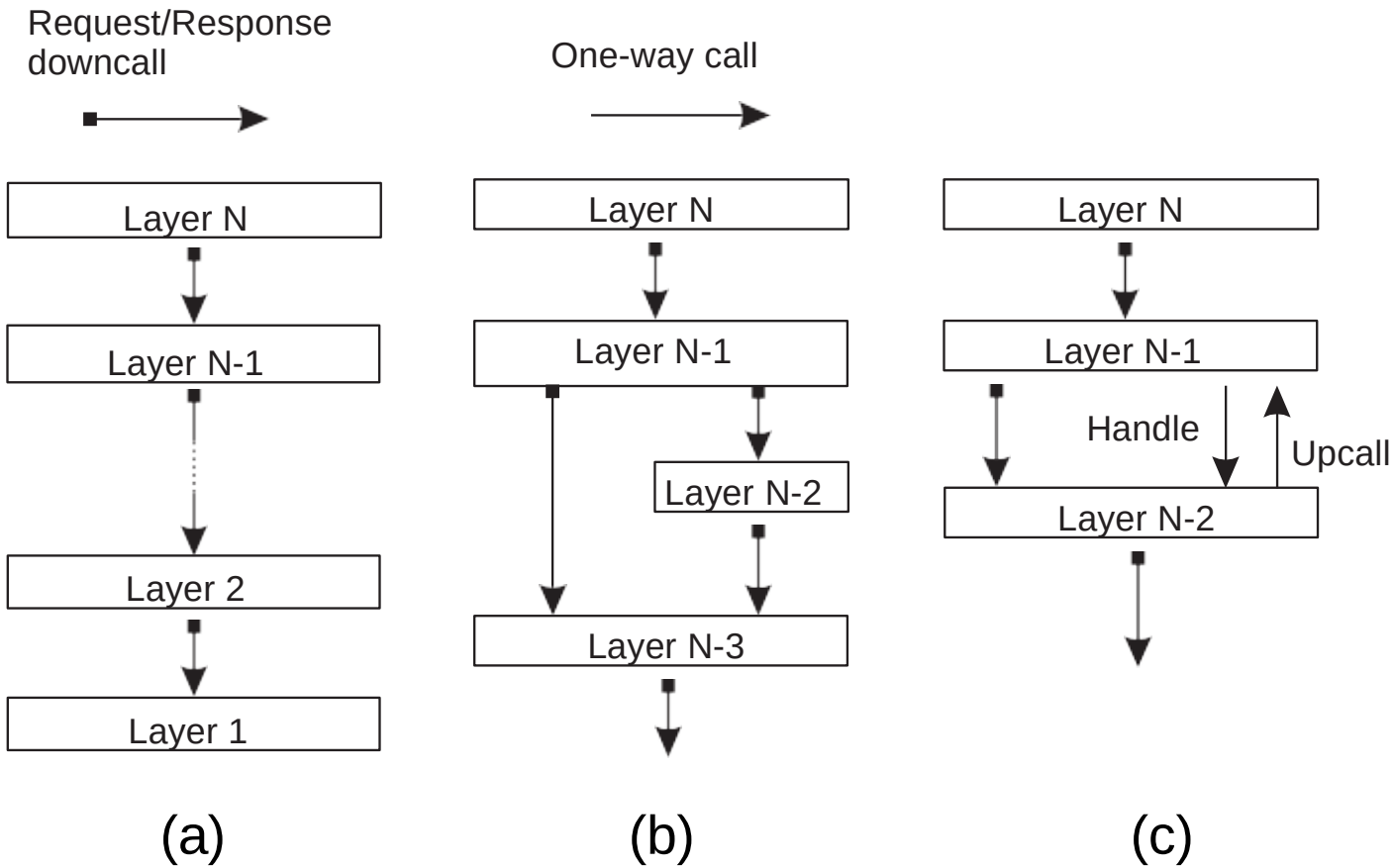
Classification of architectural styles

Several styles have by now been identified

- Layered architectures
- Object-based architectures
- Resource-centered architectures
- Event-based architectures

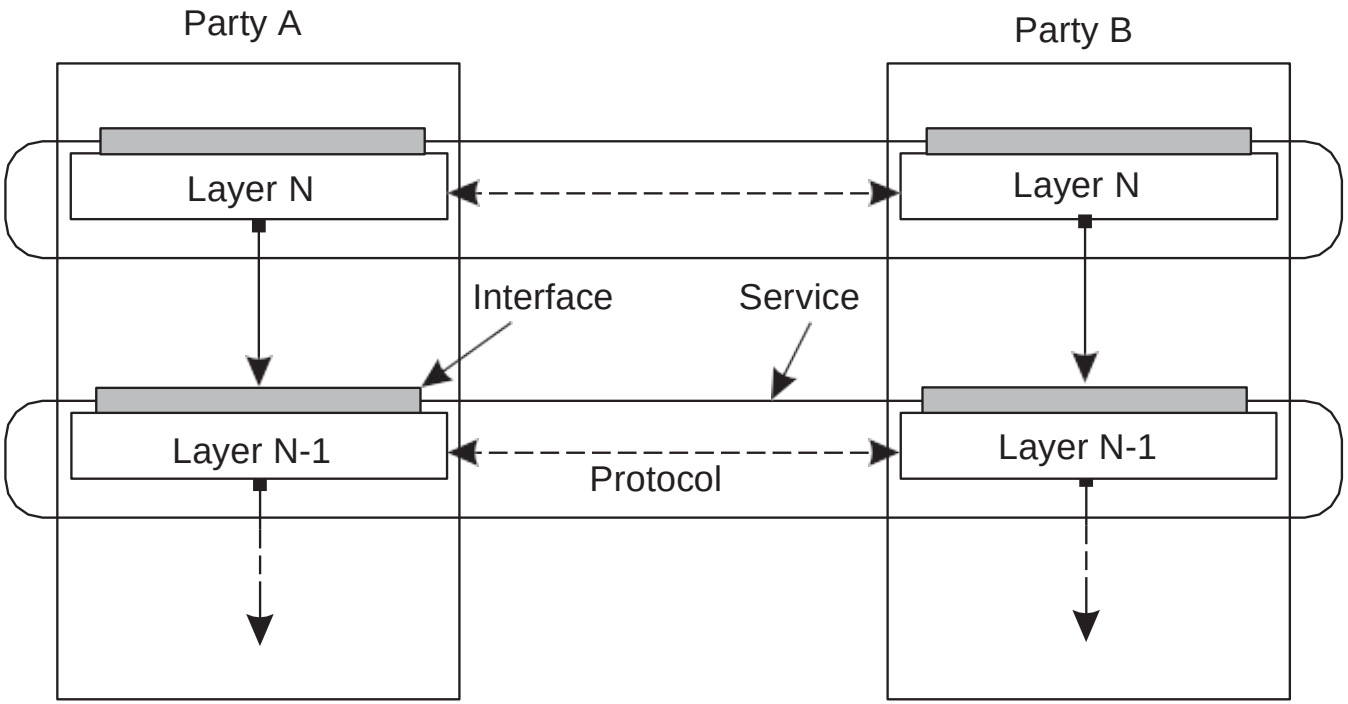
Layered architecture

Different layered organizations



Example: communication protocols

Protocol, service, interface



Two-party communication

Server

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept()  # returns new socket and addr. client
4 while True:                # forever
5   data = conn.recv(1024)   # receive data from client
6    if not data: break      # stop if client stopped
7    conn.send(str(data)+"*") # return sent data plus an "*"
8 conn.close()               # close the connection
```

Client

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # connect to server (block until accepted)
4 s.send('Hello, world')  # send some data
5 data = s.recv(1024)     # receive the response
6 print data               # print the result
7 s.close()               # close the connection
```

Application Layering

Traditional three-layered view

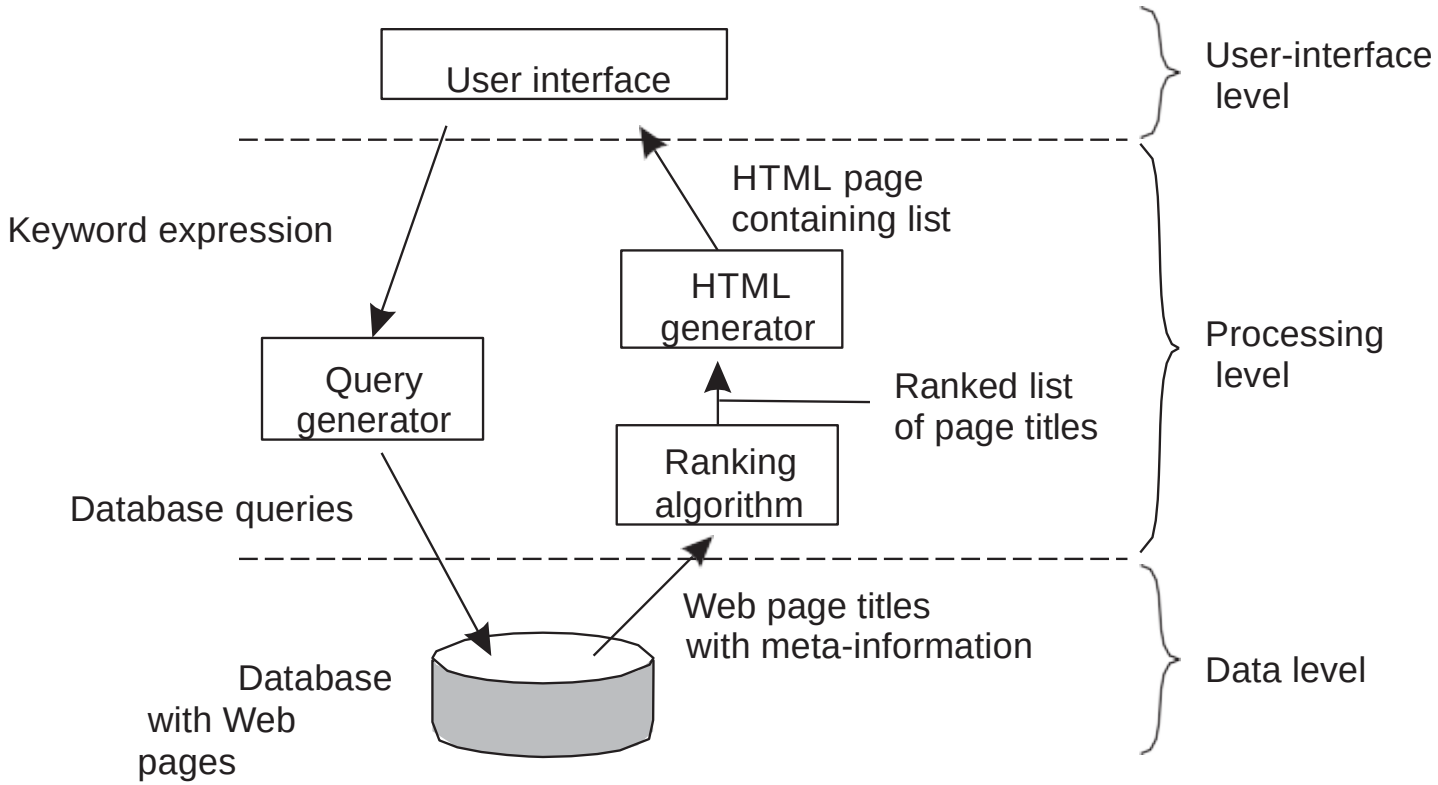
- Application-interface layer contains units for interfacing to users or external applications
- Processing layer contains the functions of an application, i.e., without specific data
- Data layer contains the data that a client wants to manipulate through the application components

Observation

This layering is found in many distributed information systems, using traditional database technology and accompanying applications.

Application Layering

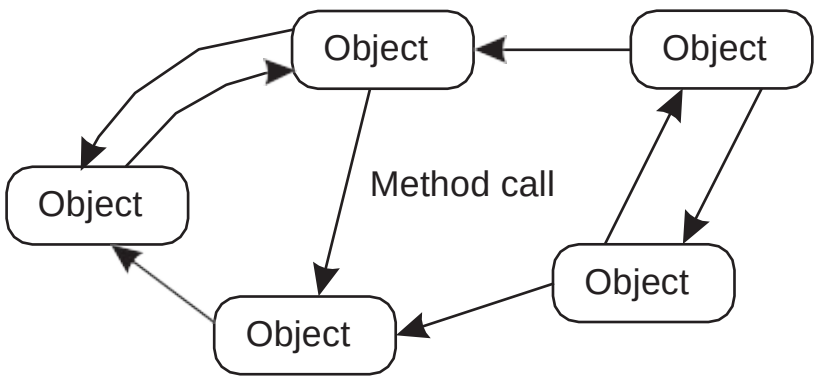
Example: a simple search engine

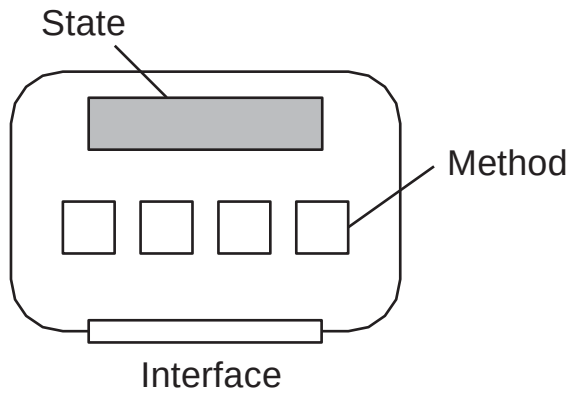


Object-based style

Essence

Components are objects, connected to each other through procedure calls. Objects may be placed on different machines; calls can thus execute across a network.

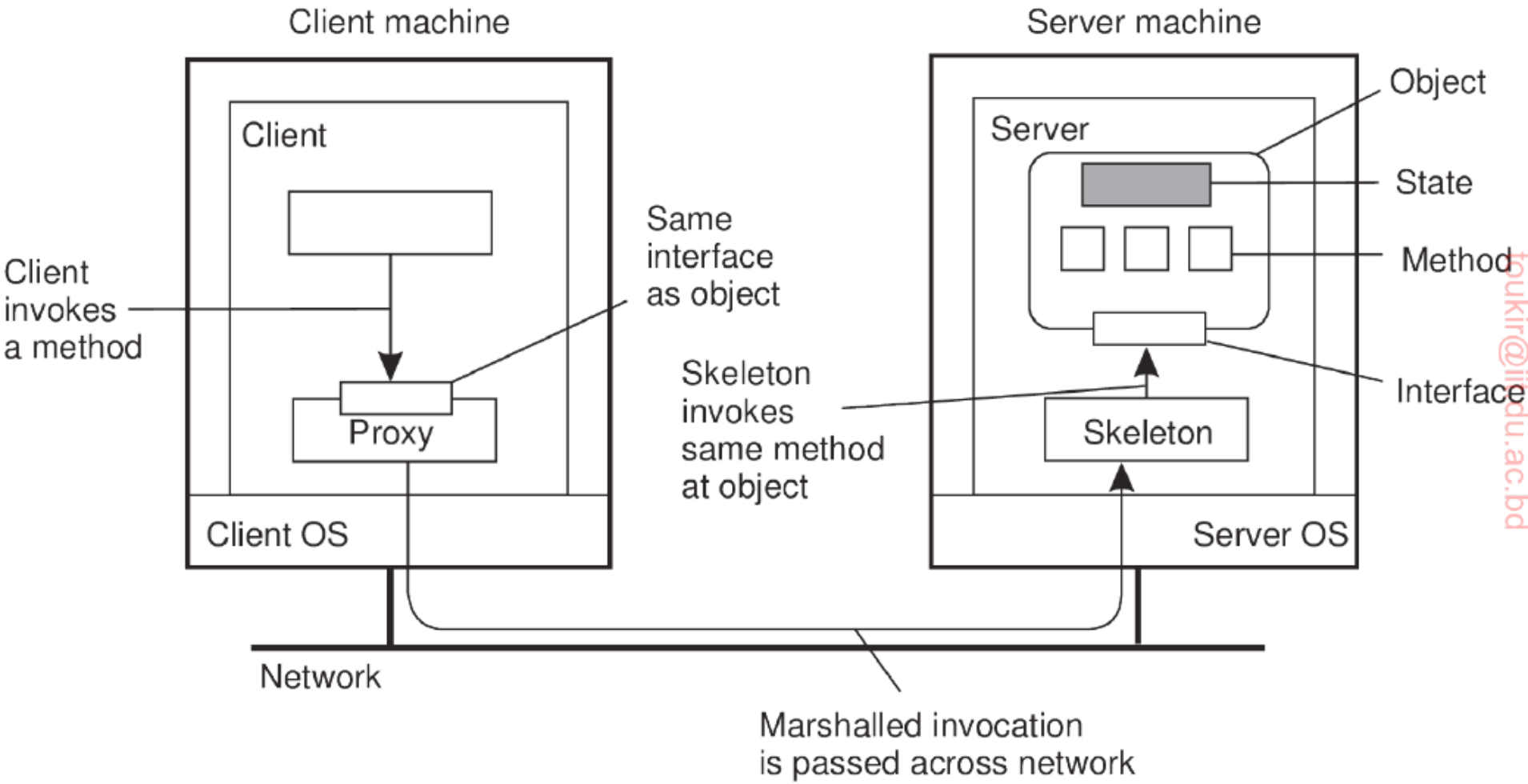




Encapsulation

Objects are said to encapsulate data and offer methods on that data without revealing the internal implementation.

Object-based style



Object-based style

Marshall and Unmarshall

To "marshal" an object means to record its state and codebase(s) in such a way that when the marshalled object is "unmarshalled," a copy of the original object is obtained, possibly by automatically loading the class definitions of the object.

For example, in Java a marshaller serializes an object to XML, and an unmarshaller deserializes XML stream to an object.

RESTful architectures

Essence

View a distributed system as a collection of resources, individually managed by components. Resources may be added, removed, retrieved, and modified by (remote) applications.

1. Resources are identified through a single naming scheme
2. All services offer the same interface
3. Messages sent to or from a service are fully self-described
4. After executing an operation at a service, that component forgets everything about the caller

Basic operations

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state

Example: Amazon's Simple Storage Service

Essence

Objects (i.e., files) are placed into buckets (i.e., directories). Buckets cannot be placed into buckets. Operations on ObjectName in bucket BucketName require the following identifier:

http://BucketName.s3.amazonaws.com/ObjectName

Typical operations

All operations are carried out by sending HTTP requests:

- Create a bucket/object: PUT, along with the URI
- Listing objects: GET on a bucket name
- Reading an object: GET on a full URI

On interfaces

Issue

Many people like RESTful approaches because the interface to a service is so simple. The catch is that much needs to be done in the parameter space.

Amazon S3 SOAP interface

Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy

On interfaces

Simplifications

Assume an interface `bucket` offering an operation `create`, requiring an input string such as `mybucket`, for creating a bucket “mybucket.”

SOAP

```
import bucket
bucket.create("mybucket")
```

RESTful

```
PUT "
http://mybucket.s3.amazonsws.com/
"
```

Conclusions

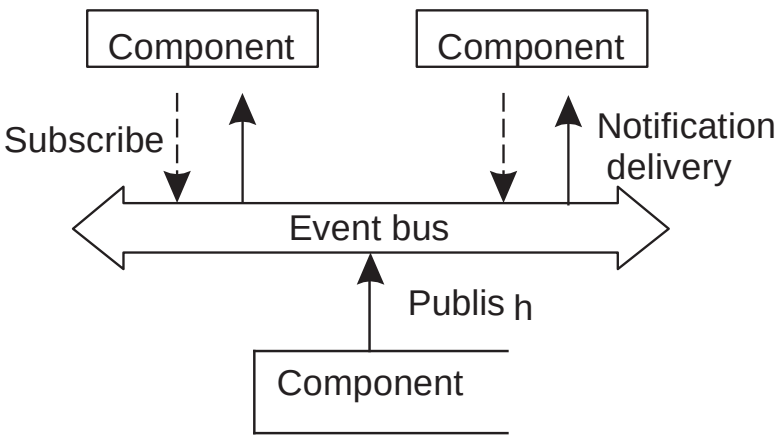
Are there any to draw?

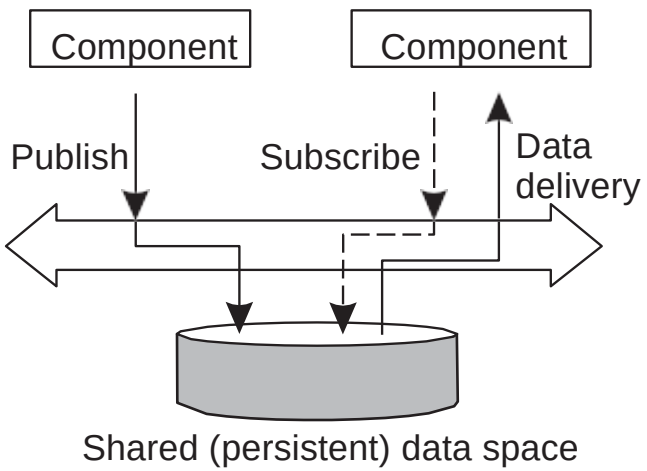
Coordination

Temporal and referential coupling

	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

Event-based and Shared data space





Example: Linda tuple space

Three simple operations

- `in(t)`: remove a tuple matching template `t`
- `rd(t)`: obtain copy of a tuple matching template `t`
- `out(t)`: add tuple `t` to the tuple space

More details

- Calling `out(t)` twice in a row, leads to storing two copies of tuple
- `t` ⇒ a tuple space is modeled as a multiset.

  Both `in` and `rd` are blocking operations: the caller will be blocked until a matching tuple is found, or has become available.

Example: Linda tuple space

Bob

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("bob","distsys","I am studying chap 2"))
4 blog._out(("bob","distsys","The linda example's pretty simple"))
5 blog._out(("bob","gtcn","Cool book!"))
```

Alice

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("alice","gtcn","This graph theory stuff is not easy"))
4 blog._out(("alice","distsys","I like systems more than graphs"))
```

Chuck

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 t1 = blog._rd(("bob","distsys",str))
4 t2 = blog._rd(("alice","gtcn",str))
5 t3 = blog._rd(("bob","gtcn",str))
```

Using legacy to build middleware

Problem

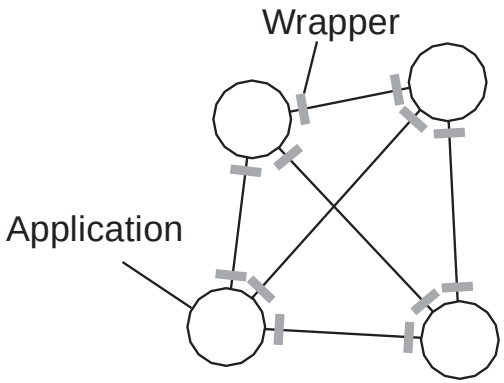
The interfaces offered by a legacy component are most likely not suitable for all applications.

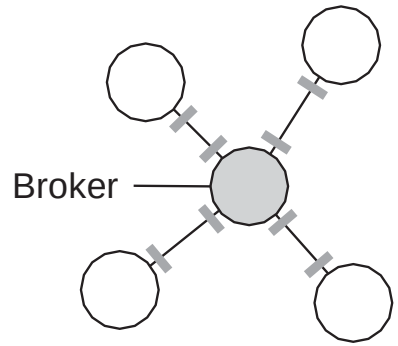
Solution

A wrapper or adapter offers an interface acceptable to a client application. Its functions are transformed into those available at the component.

Organizing wrappers

Two solutions: 1-on-1 or through a broker





Complexity with N applications

- 1-on-1: requires $N \times (N - 1) = O(N^2)$ wrappers
- broker: requires $2N = O(N)$ wrappers