Middleware

Distributed Systems
Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018

1 Overview

2 Communication

3 Naming

4 Object-Oriented Middleware

5 CORBA

Acknowledgement

- part of these slides derive from a presentation by Giovanni Rimassa, which we warmly thank
- other slides contain material from [Tanenbaum and van Steen, 2007]
- slides were made kindly available by the authord
- every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of the professor of this course

Next in Line. . .

1. Overview
2. Communication
3. Naming
4. Object-Oriented Middleware
5. CORBA

What is Middleware?

Traditional definition

- what is middleware?
  - the word suggests something belonging to the middle
  - but middle between what?
- the traditional middleware definition
  - the middleware lies in the middle between the Operating System and the applications
- the traditional definition stresses *vertical* layers
  - applications on top of middleware on top of the OS
  - middleware-to-application interfaces (top interfaces)
  - middleware-to-OS interfaces (bottom interfaces)

Why Middleware?

Behind middleware

- problems of today
  - software development is *hard*
  - experienced designers are *rare* (and *costly*)
  - applications become more and more complex
- what can middleware help with?
  - middleware is *developed once* for *many applications*
  - higher-quality *designers* can be afforded
  - middleware can provide *services* to applications
  - middleware *abstracts* away from the *specific OS*

Middleware and Models I

Interoperability

- a key feature of middleware is interoperability
  - applications using the same middleware can interoperate
  - this is true of any common platform (e.g. OS file system)
- however, many incompatible middleware systems exist
  - applications on middleware *A* can work together
  - applications on middleware *B* can work together, too
  - but, *A*-applications and *B*-applications most often cannot
- the *Enterprise Application Integration* (EAI) task
  - emphasis on *horizontal* communication
  - *application-to-application* and *middleware-to-middleware*

Middleware and Models II

Conceptual integrity

- software development does not happen *in vacuum*
  - almost any software project must cope with past systems
  - there is never time nor resources to start *from scratch*
  - legacy systems were built with their own approaches
- system integration is the only way out
  - take what is already there and add features to it
  - try to add without modifying existing subsystem
- first casualty: conceptual integrity
  - the property of a system of being understandable and explainable through a coherent, limited set of concepts

Middleware and Models III

Models from middleware to applications

- real systems are heterogeneous
  - piecemeal growth is a *very troublesome* path for software evolution
  - still, it is very popular – being asymptotically the most cost effective when development time goes to zero
- middleware technology is an *integration* technology
  - adopting a given middleware should ease *both* new application development *and* legacy integration
  - to achieve integration while limiting conceptual drift, middleware tries to cast a model on heterogeneous applications.

Middleware and Models IV

Integration middleware

- before: you have a total mess
  - a lot of systems, using different technologies
  - ad-hoc interactions, irregular structure
  - each piece must be described in its own reference frame
- then: the *integration middleware* (IM) comes
  - a new, shiny model is supported by the IM
  - existing systems are re-cast under the Model
  - new model-compliant software is developed
- after: you have the same total mess
  - but, no, now they are CORBA objects, or JADE agents

Middleware Technologies

Abstract vs. concrete middleware

- abstract middleware: a common *model*
- concrete middleware: a common *infrastructure*
- example: *distributed objects*
  - abstractly, any middleware modelling distributed systems as a collection of network reachable objects has the same model: OMG CORBA, Java RMI, MS DCOM, OSGI Architecture. . .
    - actually, even at the abstract level there are differences. . .
  - concrete implementations, instead, aim at actual interoperability, so they must handle much finer details
    - until CORBA 2.0, two CORBA implementations from different vendors were not interoperable
    - OSGI easily provides you with specifications—technology not so easy to find

Middleware Standards

The role of standards

- dealing with infrastructure, a key-issue is the so-called *network effect*
    - the value of a technology grows with the number of its adopters
- standardisation efforts become critical to build momentum around an infrastructure technology
    - large standard consortia are built, which gather several industries together

OMG CORBA `http://www.omg.org/spec/#MW`
FIPA `http://www.fipa.org/specifications/`
OSGi `http://www.osgi.org/developer/specifications/`
W3C `http://www.w3.org/standards/`

-
    - big industry players try to push their technology as *de facto* standards, or set up more open processes for them

Middleware Discussion Template

How to (re)present a middleware

- presentation and analysis of the model underlying the middleware
  - what do they want your software to look like?
- presentation and analysis of the infrastructure created by widespread use of the middleware
  - if they conquer the world, what kind of world will it be?
- discussion of *implementation issues* at the platform and application level
  - what kind of code should one write to use this platform?
  - what kind of code should one write to build his/her own platform?

Next in Line...

1. Overview
2. Communication
3. Naming
4. Object-Oriented Middleware
5. CORBA

Communication in a Distributed Setting

- communication does *not* belong to distributed systems only
  - communication mechanisms like procedure call and message-passing just require a plurality of interacting entities, not necessarily distributed ones
- however, communication in distributed systems presents more difficult challenges, like unreliability of communication and large scale
- of course, communication in distributed systems first of all deals with distribution / location transparency

! communication in distributed systems is mostly a middleware issue

Focus on. . .

1 Overview
2 Communication
- Layers & Protocols
- Types of Communication
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication

3 Naming
- Names, Identifiers, Addresses
- Flat & Structured Naming
- Attribute-based Naming

4 Object-Oriented Middleware
5 CORBA

Layered Communication I

Communication involves many problems at many different levels

- from the physical network level up to the application level
- communication can be organised on *layers*
- a reference model is useful to understand protocols, behaviours, and interactions

Layered Communication II

OSI model [Day, 1995]

- standardised by the International Standards Organization (ISO)
  - ISO/IEC 7498-1:1994
- designed to allow *open systems* to communicate
- rules for communication govern the format, content, and meaning of messages sent and received
- rules are formalised in *protocols*
- the collection of protocols for a particular system is its *protocol stack*, or *protocol suite*

Types of Protocols

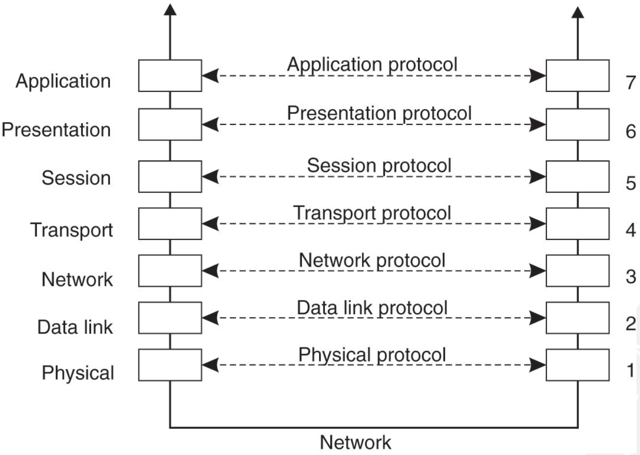
Connection-oriented protocols

- first of all, a connection is established between the sender and the receiver
- possibly, an agreement over the protocol to be used is reached
- then, communication occurs through the connection
- finally, the connection is terminated

Connectionless protocols

- no setup is required
- the sender just send a message when it is ready

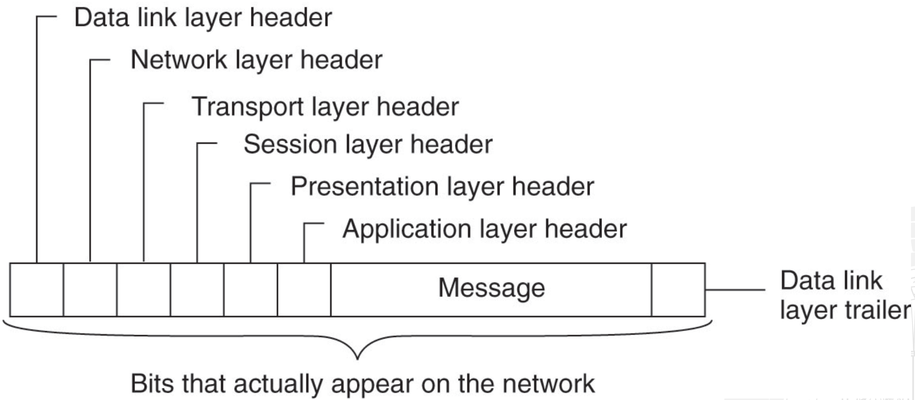
The OSI Reference Model



Layers, interfaces, and protocols in the OSI Model

[Tanenbaum and van Steen, 2007]

A Message in the OSI Reference Model



A typical message as it appears on the network

[Tanenbaum and van Steen, 2007]

OSI Model $\neq$ OSI Protocols

OSI protocols

- never successful
- TCP/IP is not an OSI protocol, and still dominates its layers

OSI model

- perfect to understand and describe communication systems through layers
- however, some problems exist when middleware comes to play

Middleware Protocols I

The problem

- middleware mostly lives at the application level
- protocols for middleware services are different from high-level application protocols

← middleware protocols are application-independent, application protocols are obviously application-dependent

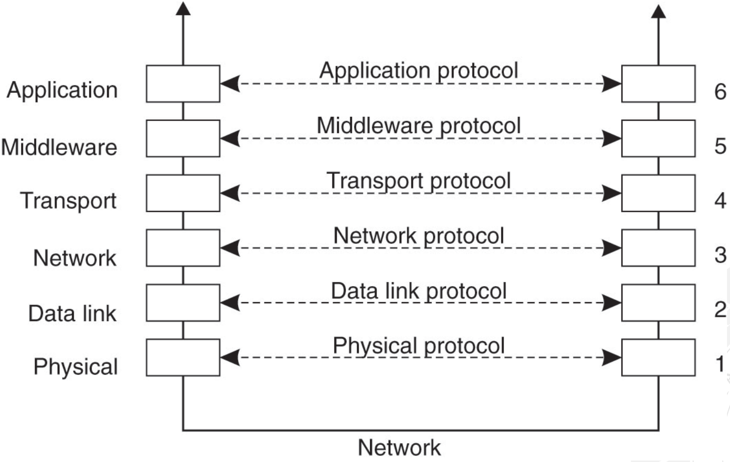
- how can we distinguish between the two sorts of protocols at the same layer?

Middleware Protocols II

Extending the reference model for middleware

- session and presentation layers are replaced by a middleware layer, which includes all application-independent protocols
- potentially, also the transport layer could be offered in the middleware one

Middleware as an Additional Service in C/S Computing



Adapted reference model for network communication

[Tanenbaum and van Steen, 2007]

Focus on. . .

1 Overview
2 Communication
- Layers & Protocols
- Types of Communication
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication

3 Naming
- Names, Identifiers, Addresses
- Flat & Structured Naming
- Attribute-based Naming

4 Object-Oriented Middleware
5 CORBA

Types of Communication I

Persistent vs. transient communication

persistent communication — a message sent is stored by the communication middleware until it is delivered to the receiver

→ no need for time coupling between the sender and the receiver

transient communication — a message sent is stored by the communication middleware only as long as both the receiver and the sender are executing

→ time coupling between the sender and the receiver

Types of Communication II

Asynchronous vs. synchronous communication

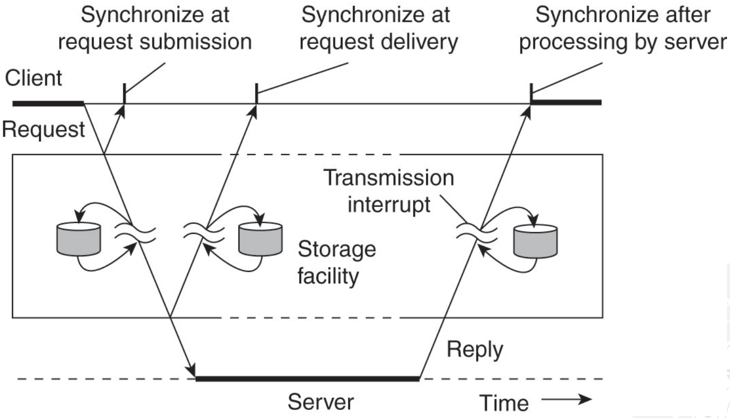
asynchronous communication — the sender keeps on executing after sending a message

→ the message should be stored by the middleware

synchronous communication — the sender blocks execution after sending a message and waits for response—until the middleware acknowledges trasmission, or, until the receiver acknowledges the reception, or, until the receiver has completed processing the request

→ some form of coupling in control between the sender and the receiver

Communications with a Middleware Layer



Viewing middleware as an intermediate (distributed) service in application-level communication

[Tanenbaum and van Steen, 2007]

Actual Communication in Distributed Systems I

Persistency & synchronisation in communication

- in the practice of distributed systems, many combinations of persistency and synchronisation are typically adopted
- persistency and synchronisation should then be taken as two dimensions along which communication and protocols could be analysed and classified

Actual Communication in Distributed Systems II

Discrete vs. streaming communication

- communication is not always *discrete*, that is, it does not always happen through complete units of information – e.g., messages
- discrete communication is then quite common, but not the only way available – and does not respond to all the needs
- sometimes, communication needs to be continuous—through sequences of messages constituting a possibly unlimited amount of information
- streaming communication — the sender delivers a (either limited or unlimited) sequence of messages representing the *stream* of information to be sent to the receiver

→ communication may be *continuous*

Focus on. . .

1 Overview
2 Communication
- Layers & Protocols
- Types of Communication
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication

3 Naming
- Names, Identifiers, Addresses
- Flat & Structured Naming
- Attribute-based Naming

4 Object-Oriented Middleware
5 CORBA

Remote Procedure Call (RPC)

Basic idea

- programs can call procedures on other machines
- when a process A calls a procedure on a machine B, A is suspended, and execution of procedure takes place on B
- once the procedure execution has been completed, its completion is sent back to A, which resumes execution

Information in RPC

- information is not sent directly from sender to receiver
- parameters are just packed and transmitted along with the request
- procedure results are sent back with the completion
- *no message passing*

Issues of RPC

Main problems

- the *address space* of the caller and the callee are separate and different

→ need for a common reference space

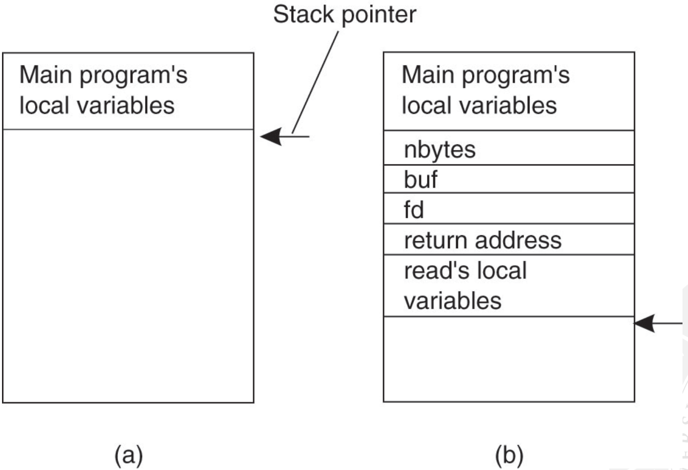
- *parameters* and *results* have to be passed and handled correctly

→ need for a common data format

- either / both machines could unexpectedly *crash*

→ need for suitable fault-tolerance policies

Conventional Procedure Call



Parameter passing in a local procedure call

[Tanenbaum and van Steen, 2007]

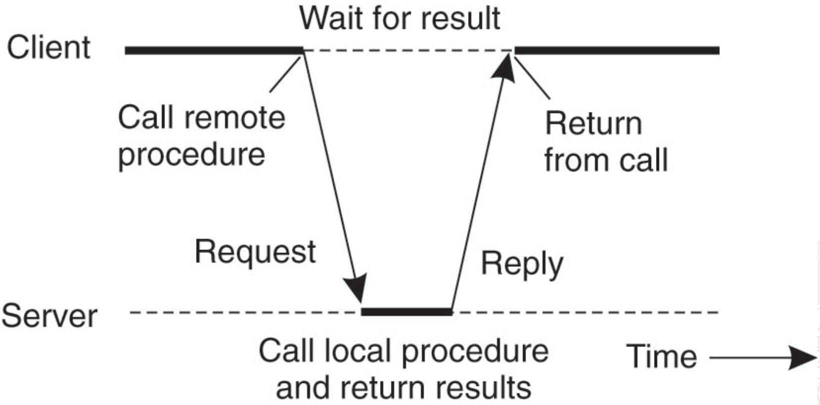
Client & Server Stubs I

Main goal: transparency

- RPC should be like local procedure call from the viewpoint of both the caller and the callee

→ procedure calls are sent to the *client stub* and transmitted to the *server stub* through the network to the called procedure

Client & Server Stubs II



Principle of RPC between a client and server program

[Tanenbaum and van Steen, 2007]

Steps for a RPC

- the client procedure calls the client stub in the normal way
- the client stub builds a message and calls the local operating system
- the client's OS sends the message to the remote OS
- the remote OS gives the message to the server stub
- the server stub unpacks the parameters and calls the server
- the server does the work and returns the result to the stub
- the server stub packs it in a message and calls its local OS
- the server's OS sends the message to the client's OS
- the client's OS gives the message to the client stub
- the stub unpacks the result and returns to the client

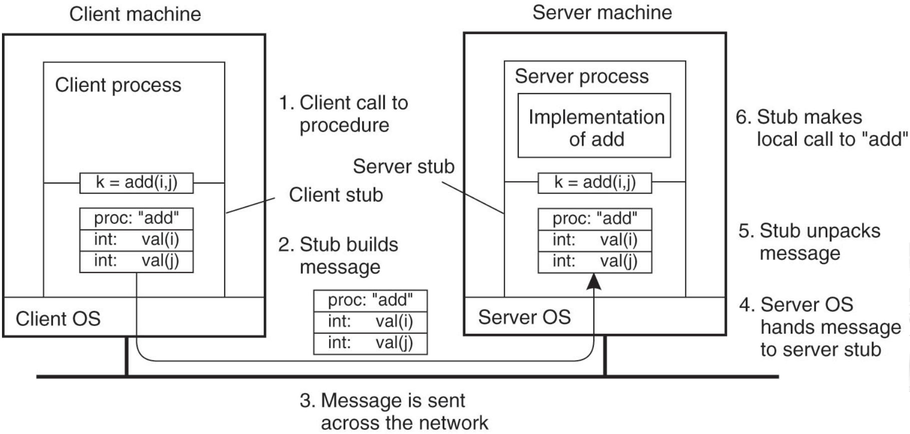
Parameter Passing I

Passing value parameters

- parameters are *marshalled* to pass across the network

→ procedure calls are sent to the *client stub* and transmitted to the *server stub* through the network to the called procedure

Parameter Passing II



Steps of a remote computation through a RPC

[Tanenbaum and van Steen, 2007]

Issues in Parameter Passing I

Passing value parameters

- problems of representation and meaning
  - e.g., little endian vs. big endian
- in order to ensure transparency, stubs should be in charge of the mapping & translation
- a possible approach: interfaces described through an IDL (Interface Definition Language), and consequent handling compiled into the stubs
  - e.g., CORBA IDL

Issues in Parameter Passing II

Passing reference parameters

- main problem: reference space is local
- first solution: forbidding reference parameters
- second solution: copying parameters (suitably updating the reference), then copying them back (according to the original reference)

→ call-by-reference becomes copy&restore

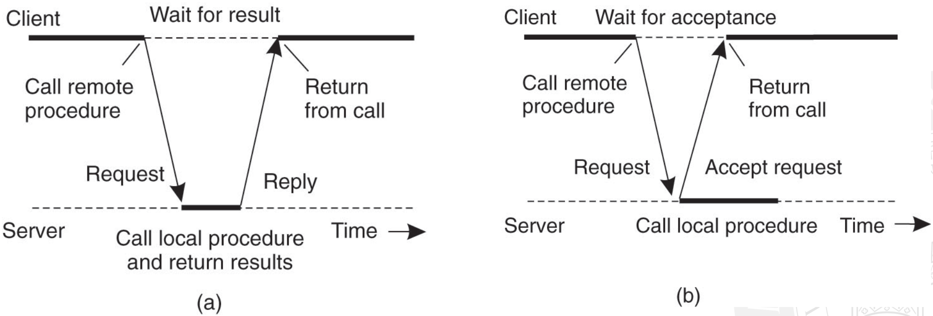
- third solution: creating a global/accessible reference to the caller space from the callee

Asynchronous RPC

Synchronicity might be a problem in distributed systems

- synchronicity is often unnecessary, and may create problems

→ *asynchronous RPC* is an available alternative in many situations



Traditional RPC

Asynchronous RPC

[Tanenbaum and van Steen, 2007]

Deferred Synchronous RPC I

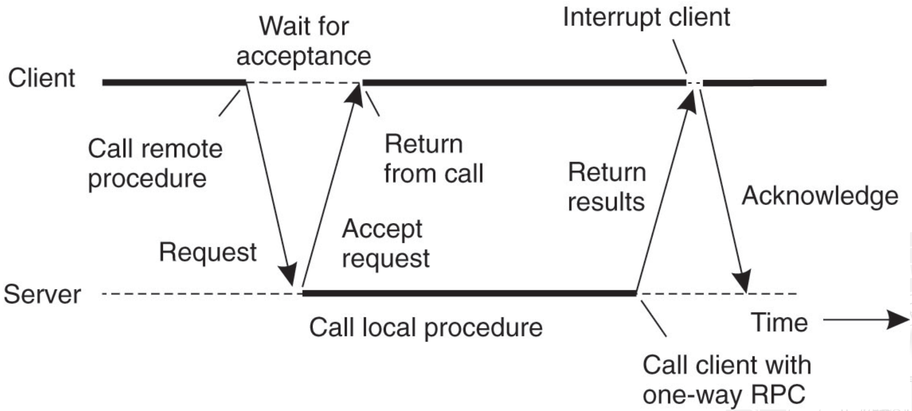
Combining asynchronous RPCs

- sometimes some synchronicity is required, but too much is too much

→ *deferred synchronous RPC* combines two asynchronous RPC to provide an *ad hoc* form of synchronicity

- the first asynchronous call selects the procedure to be executed and provides for the parameters
- the second asynchronous call goes for the results
- in between, the caller may keep on computing

Deferred Synchronous RPC II



Deferred synchronous RPC

[Tanenbaum and van Steen, 2007]

Limits of RPC

Coupling in time

- co-existence in time is a requirement for any RPC mechanism
- sometimes, a too-hard requirement for effective communication in distributed systems
- an alternative is required that does not require the receiver to be executing when the message is sent

The alternative: messaging

- please notice: message-oriented communication is not synonym of uncoupling
- however, we can take this road toward uncoupled communication

Focus on. . .

1 Overview
2 Communication
- Layers & Protocols
- Types of Communication
- Remote Procedure Call
- **Message-oriented Communication**
- Stream-oriented Communication

3 Naming
- Names, Identifiers, Addresses
- Flat & Structured Naming
- Attribute-based Naming

4 Object-Oriented Middleware
5 CORBA

Message-oriented Transient Communication

Basic idea

- messages are sent through a channel abstraction
- the channel connects two running processes
- time coupling between sender and receiver
- transmission time is measured in terms of milliseconds, typically

Examples

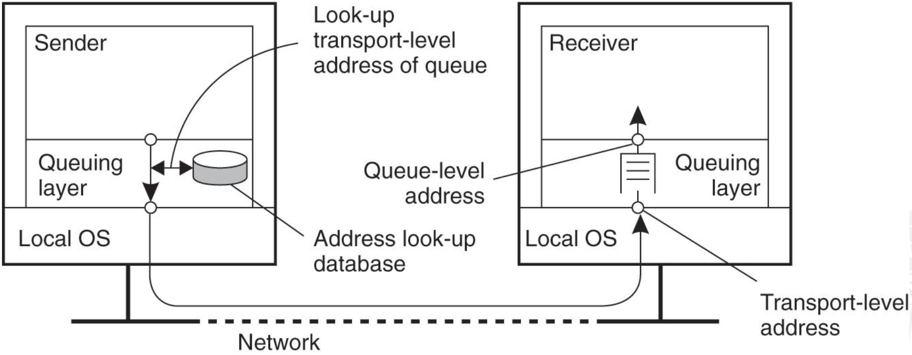
- Berkeley Sockets [Vessey and Skinner, 1990] — typical in TCP/IP-based networks
- MPI (Message-Passing Interface) [Gropp, 2011] — typical in high-speed interconnection networks among parallel processes

Message-Oriented Persistent Communication I

Message-queuing systems / Message-Oriented Middleware (MOM)

- basic idea: MOM provides message storage service
- a message is put in a queue by the sender, and delivered to a destination queue
- the target(s) can retrieve their messages from the queue
- time uncoupling between sender and receiver
- example: IBM's WebSphere Message-Queuing System

Message-Oriented Persistent Communication II



General architecture of a message-queuing system

[Tanenbaum and van Steen, 2007]

Focus on. . .

1. Overview
2. Communication
   - Layers & Protocols
   - Types of Communication
   - Remote Procedure Call
   - Message-oriented Communication
   - Stream-oriented Communication
3. Naming
   - Names, Identifiers, Addresses
   - Flat & Structured Naming
   - Attribute-based Naming
4. Object-Oriented Middleware
5. CORBA

Streams

Sequences of data

- a stream is transmitted by sending sequences of related messages
- single vs. complex streams: a single sequence vs. several related simple streams
- data streams: typically, streams are used to represent and transmit huge amounts of data
- examples: JPEG images, MPEG movies

Streams & Time I

Continuous vs. discrete media

- in the case of *continuous (representation) media*, time is relevant to understand the data – e.g., audio streams
- in the case of *discrete (representation) media*, time is not relevant to understand the data – e.g., still images

Streams & Time II

Transmission of time-dependent information

asynchronous transmission mode — data items of a stream are transmitted in sequence without further constraints—e.g., a file representing a still image

synchronous transmission mode — data items of a stream are transmitted in sequence with a maximum end-to-end delay—e.g., data generation by a pro-active sensor

isochronous transmission mode — data items of a stream are transmitted in sequence with both a maximum and a minimum end-to-end delay—e.g., audio & video

Streams & Quality of Service I

Quality of service

- timing and other non-functional properties are typically expressed as *Quality of Service* (QoS) requirements
- in the case of streams, QoS typically concerns *timing*, *volume*, and *reliability*
- in the case of middleware, the issue is how can a given middleware ensure QoS to distributed applications

Streams & Quality of Service II

A practical problem

- whatever the theory, many distributed systems providing streaming services rely on top of the IP stack
- IP specification allow for a protocol implementation dropping packets when needed
- QoS should be enforced at the higher levels

Next in Line. . .

1. Overview
2. Communication
3. Naming
4. Object-Oriented Middleware
5. CORBA

What is Naming? I

The issue of naming

- naming is mapping names onto computational entities
  - e.g., *resources* in REST
- finding the entity a name refers to is said *resolving* a name—name resolution

The issue of naming in distributed systems

- naming is an issue in computational systems in general
- features of distributed system makes naming even more difficult
  - openness
  - location
  - mobility

! naming in distributed systems is mostly a middleware issue

What is Naming? II

Naming systems

- the naming system is the portion of the system devoted to name resolution
- ! the naming system is an essential part of any middleware
  - e.g., see AMS and DF in JADE
- issues of naming systems
  - distribution
  - scalability
  - efficiency

Focus on. . .

1 Overview
2 Communication
- Layers & Protocols
- Types of Communication
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication

3 Naming
- **Names, Identifiers, Addresses**
- Flat & Structured Naming
- Attribute-based Naming

4 Object-Oriented Middleware
5 CORBA

Names

Defining a (distributed) naming system amounts at...

- defining a set of the *admissible names*
- defining the set of the *named entities*
- defining the association between names and entities

What is a name?

- a name is something that refers to an entity
- ... a string, a sequence of symbols, ...

! defining the set of the *admissible names* for its components determines how we can speak about the system

Entities

Entities are to be used

- an entity is something one can operate on
- by accessing to it
- through an access point

Access point

- a special sort of entity in distributed systems
- used to access an entity
- like, e.g., the cell phone to access yourselves

Addresses

Accessing an entity through an access point. . .

- requires an *address*
- like, e.g., your cell phone number
- for the sake of brevity, whenever there is no danger of confusion, the address of an access point to an entity can be called the address of the entity

What about using addresses as names?

- addresses are names of some sort
- however, they are quite unfriendly for humans
- . . . and, *location independence* might be desirable

Identifiers

An identifier is another type of name

1. an identifier *refers to at most one* entity
2. each entity is *referred to by at most one* identifier
3. an identifier *always refers to the same entity*—it is never reused

Addresses vs. identifiers

- identifiers are sorts of names
- however, with different purposes
  - e.g., while my user name `andrea.omicini` is not to be reused for another person of the Alma Mater (*identifier*), my cell number could instead be reused by someone else (*address*)

Human-friendly Names

Identifiers and addresses are often in machine-readable form

- humans cannot handle them easily
- *observability* is spoiled
- possibly creating problems in the use, monitoring, and control of distributed systems
- *human-friendly names*

Resolving Names to Addresses

Main issue in naming

- how do we *associate* names and identifiers to addresses?
- in large, distributed, mobile, open systems, in particular?

Examples

- the simplest case: *name-to-address binding*, with a table of $\langle name, address \rangle$ pairs

← problem: a centralised table does not work in large networks

- the DNS case: hierarchical composition

→ `www.apice.unibo.it` hierarchically resolved through a recursive lookup

Focus on. . .

1. Overview
2. Communication
   - Layers & Protocols
   - Types of Communication
   - Remote Procedure Call
   - Message-oriented Communication
   - Stream-oriented Communication
3. Naming
   - Names, Identifiers, Addresses
   - **Flat & Structured Naming**
   - Attribute-based Naming
4. Object-Oriented Middleware
5. CORBA

Flat Naming

Basic idea

- a name is just a flat sequence of chars / symbols
- works in LANs

Examples

broadcasting messages containing the identifier of the target entity is sent to everyone, only the machine containing the entity responds
  - e.g., ARP (Address Resolution Protocol)
  - problem: inefficient when the network grows

multicasting only a restricted group of hosts receives the request
  - e.g., data-link level in Ethernet networks

Structured Naming

Basic idea

- flat names are good for machines, not for humans
- structured names are composed by simple human-readable names – thus matching the natural limitations of human cognition

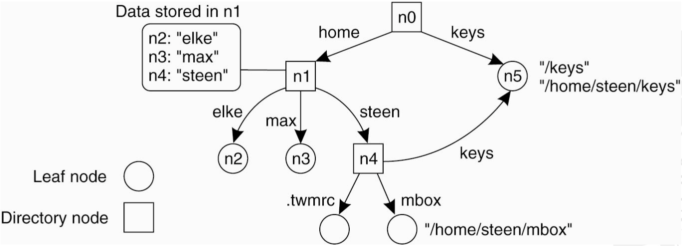
Example

- Internet name space

Name Spaces

Basic idea

- names are organised hierarchically, according to a *labelled, directed* graph—a naming graph
- *leaf nodes* represent named entities
- *directory nodes* have a number of outgoing edges, each labelled with an identifier



Symbolic link in a naming graph [Tanenbaum and van Steen, 2007]

The Internet Domain Name Space (DNS)

The DNS Name Space

- hierarchically organised as a rooted tree
- each node (except root) has exactly one incoming edge, labelled with the name of the node
- a subtree is a *domain*
- a path name to its root node is a *path name*
- a node contains a collection of *resource records*

Resource Records

Type of record	Associated entity	Description
SOA	Zone	Holds information on the represented zone
A	Host	Contains an IP address of the host this node represents
MX	Domain	Refers to a mail server to handle mail addressed to this node
SRV	Domain	Refers to a server handling a specific service
NS	Zone	Refers to a name server that implements the represented zone
CNAME	Node	Symbolic link with the primary name of the represented node
PTR	Host	Contains the canonical name of a host
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any entity-specific information considered useful

Most relevant types of resource records in a DNS node

[Tanenbaum and van Steen, 2007]

Focus on. . .

1 Overview
2 Communication
- Layers & Protocols
- Types of Communication
- Remote Procedure Call
- Message-oriented Communication
- Stream-oriented Communication

3 Naming
- Names, Identifiers, Addresses
- Flat & Structured Naming
- Attribute-based Naming

4 Object-Oriented Middleware
5 CORBA

Limits of Flat & Structured Naming

Beyond location independence

- flat naming allow for unique and location-independent way to refer to distributed entities
- structured naming also provides for human-friendliness
- however, distributed systems are more and more information-based – information could also be the basis for looking for an entity
- exploiting information associated to entities to locate them

Attribute-based Naming I

Description as pairs

- many way to describe an entity could be used
- most popular: a collection of $\langle attribute, value \rangle$ pairs associated to an entity to describe it
- *attribute-based naming*

A.k.a. *directory services*

- attribute-based naming systems are also known as directory services
- the essential point: choosing the right set of attributes to describe resources
- yet, things are more complex than that: from X.500 to LDAP

Attribute-based Naming II

X.500 standard

- directory services are mostly ruled by the X.500 standards
  `http://www.x500standard.com`
- ruling access protocols like DAP (Directory Access Protocol)
- including

DIT
: (Directory Information Tree) a hierarchical organisation of distributed entries distributed over servers

DSA
: (Directory System Agents) the servers hosting the DIT

entry
: each entry consists of a set of attributes, each one with possibly multiple values

DN
: each entry has a unique Distinguished Name, formed by combining its Relative Distinguished Name (RDN), some entry attributes, and the RDNs of each entry up to the DIT root

Hierarchical Implementations I

Combining structured & attribute-based naming

- distributed directory services
  - Lightweight Directory Access Protocol (LDAP)
  - allowing for DAP upon TCP/IP

Hierarchical Implementations II

Hierarchy through LDAP attribute-based names

- an LDAP directory service contains a number of *directory entries* – a collection of $\langle attribute, value \rangle$ pairs, similar to DNS resource records
- the directory entries in an LDAP directory service constitute the *directory information base* (DIB)—there, each record is uniquely named
- naming attributes are called Relative Distinguished Names (RDN)—they are combined to form a globally-unique name, which is a structured name
- as a result, the Directory Information Tree (DIT) is a collection of directory entries forming the naming graph of an LDAP directory

Hierarchical Implementations III

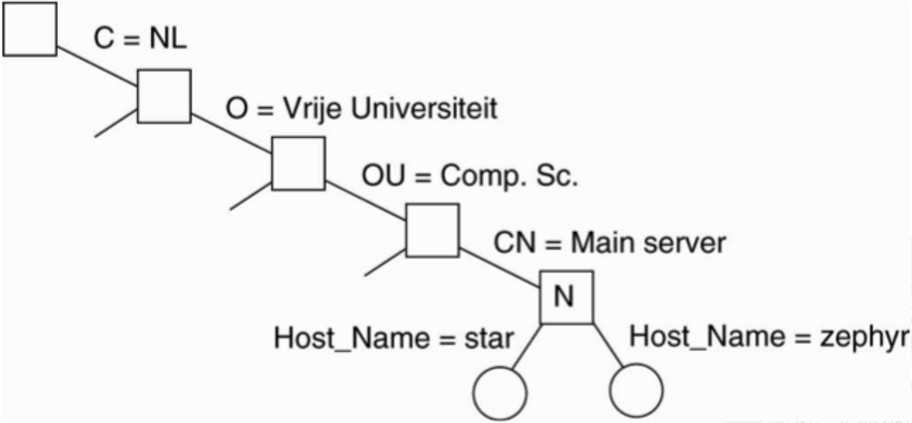
Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	137.37.20.10

Two LDAP directory entries with hierarchical naming. . .

[Tanenbaum and van Steen, 2007]

Hierarchical Implementations IV



... along with the corresponding (partial) DIT

[Tanenbaum and van Steen, 2007]

Next in Line. . .

1 Overview

2 Communication

3 Naming

4 Object-Oriented Middleware

5 CORBA

Distributed Objects

From OO to distributed OO

- distributed systems need quality software, and they are a difficult system domain
- OOP is a current software best practice
- questions are
  - can we apply OOP to distributed systems programming?
  - what changes and what stays the same?
- distributed objects apply the OO paradigm to distributed systems
  - examples: CORBA, DCOM, Java RMI, JINI, EJB, OSGi

Core of OOP I

What is the fundamental concept of OOP?

? from the very name of object-oriented programming, could it be

the **object**

?

- definitely not—and *you should know this*!

! the fundamental concept of object-oriented programming is

the **class**

!

Core of OOP II

Class: a definition

- a class is an abstract data type, with an associated module that implements it
- writing this as a conceptual equation *à la* Wirth,

$$\text{type} + \text{module} = \text{class}$$

Modules *vs.* Types

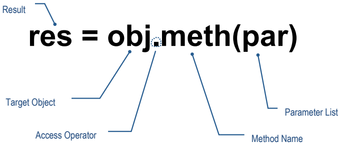
Modules & types

- modules and types look very different
  - modules give structure to the implementation
  - types specifies how each part can be used
- but they share the interface concept
  - in modules, the interface selects the *public* part
  - in types, the interface describes the allowed *operations* as well as their *properties*
- as a result, the interface is at the very core of the notion of class

OOP Mechanism

Method call

The fundamental OOP computation mechanism



OOP Extensibility

Subclassing

subclassing is the main OOP extension mechanism, and it is affected by the dual nature of classes

- type + module = class
- subtyping + Inheritance = subclassing

subtyping — a partial order on types

- a valid operation on a type is also valid on a subtype

LSP Liskov substitution principle [Liskov, 1987]: if S is a subtype of T, then replacing objects of type T with objects of type S does not alter the properties of a program

inheritance — a partial order on modules

- a module grants special access to its sub-modules
- open/closed principle: an OO language must allow the creation of modules *closed* for use but *open* for extension

Distributing the Objects

How to?

Q how can we extend OOP to a distributed system, preserving all its desirable properties?

A just pretend the system is not distributed, and then do business as usual!

- this is called *transparency*
  - as crazy as it may seem, it works
  - well, up to a point at least, but generally enough for a lot of applications
- problems arise from failure management
  - in reliable and fast networks, things run smooth...
  - whenever a failure comes from what we abstracted away – e.g., a network failure –, we are just plain dead

Core of Distributed OOP

What is the fundamental concept of Distributed OOP?

- could it be

  the object

  or, again,

  the class

  ?
- clearly not
- the fundamental concept of distributed OOP is

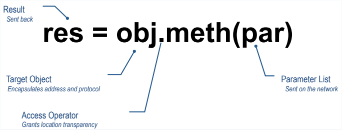
  the remote interface

  !

Distributed OOP Mechanism

Remote Method Call

The fundamental Distributed OOP computation mechanism



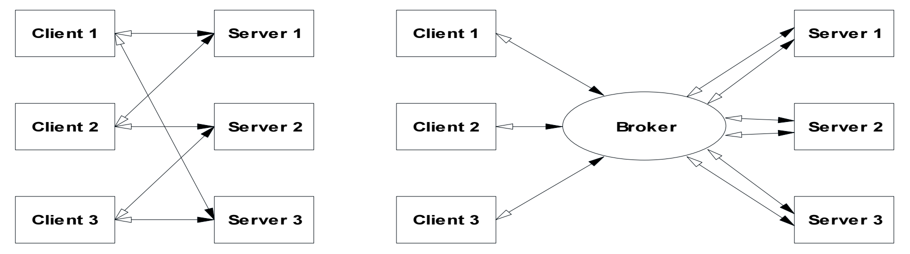
Distributed OOP: Communication Model

The communication model of distributed objects...

- is implicit
  - transmission is implicit, everything happens through stubs
  - the stub turns an ordinary call into an Inter-Process Communication (IPC) mechanism
  - as a result, both local and remote calls are handled homogeneously—*location transparency*
- is object-oriented
  - only *objects* exist, invoking operations on each other
  - interaction is *client/server* with respect to the individual call—micro C/S, not necessarily macro C/S
  - each call is attached to a specific *target object*: the result can depend on the target object state
  - callers refer to objects through an *object reference*

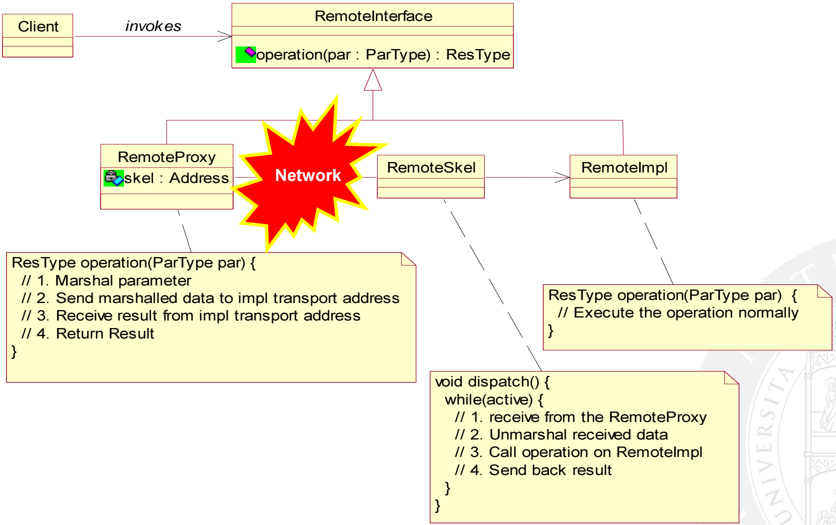
Broker Architecture

Broker architectural pattern [Buschmann et al., 1996]



- stock market metaphor
- publish/subscribe scheme
- extensibility, portability, interoperability
- a broker reduces communication channels from $N_c x N_s$ to $N_c + N_s$

Proxy and Impl, Stub and Skeleton



Next in Line...

1 Overview

2 Communication

3 Naming

4 Object-Oriented Middleware

5 CORBA

What is CORBA I

A standard

- acronym for *Common ORB Architecture*
- ORB is an acronym again, standing for *Object Request Broker*
- CORBA is a standard, not a product
- a standard proposed by OMG

What is CORBA II

Object Management Group (OMG)

- a consortium of more than 800 companies, founded in 1989
- including all major tech companies

  http://www.omg.org
- CORBA is a standard, not a product
- the same institution that took up the Unified Modeling Language (UML) specification from its original creator, Rational Software Corporation

Behind CORBA I

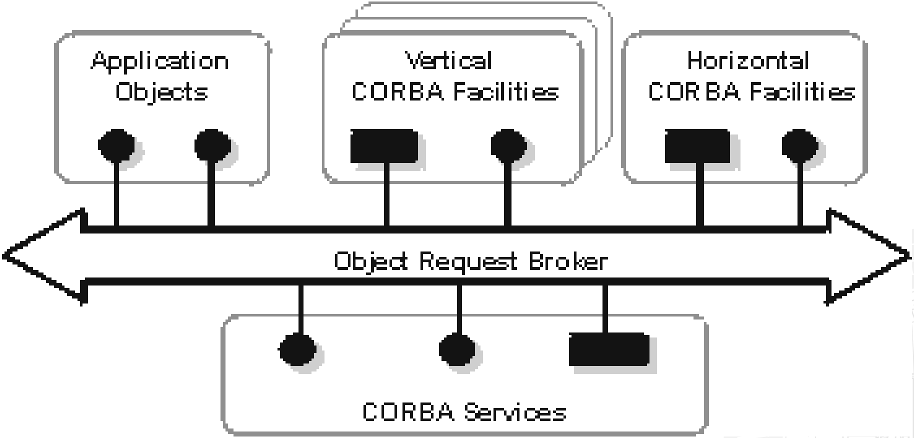
Object Management Architecture (OMA)

- represents the OMG vision for distributed computing
- the architecture standardises component interfaces to create a plug-and-play component software environment based on OO technology

! nowadays, the OMA vision has been superseded by the Model Driven Architecture (MDA), almost a meta-standard in itself

http://www.omg.org/mda/

Behind CORBA II



http://www.omg.org/oma/

Behind CORBA III

ORB the Object Request Broker is OMA backbone

- **IOOP** the IIOP protocol is the standard application transport that grants interoperability

Services The Common Object Services serve as CORBA system libraries, bundled with the ORB infrastructure

- Naming and Trader Service
- Event Service
- Transaction Service
- ...

Facilities The Common Facilities are frameworks to develop distributed applications in various domains

- *Horizontal Common Facilities* handle issues common to most application domains—e.g., GUI, Persistent Storage, Compound Documents
- *Vertical Common Facilities* deal with traits specific of a particular domain—e.g., Financial, Telco, Health Care

RMI in OMA I

Communication in OMA

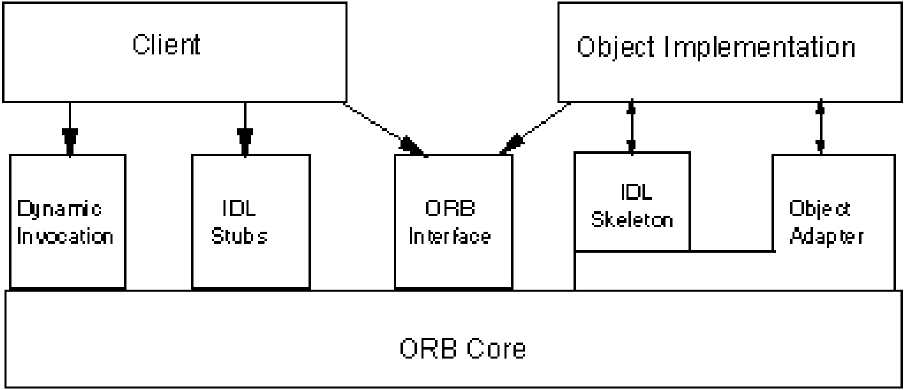
- part of the OMA deals with communication mechanisms
- it allows remote method invocation regardless of
  - location and network protocols
  - programming language
  - operating system
- the transport layer is hidden from applications using *stub* code

RMI in OMA II

Participants in RMI

- a Request is the *closure* of an invocation, complete with target object, actual parameters, etc.
- the Client is the object making the request
- the Object Implementation is the *logical object* serving the request
- the Servant is the *physical component* that incarnates the Object Implementation
- the ORB connects Client and Servant

ORB Core Interfaces I



ORB Core Interfaces II

Interfaces

- client-side interfaces
  - Client Stub
  - Dynamic Invocation Interface (DII)
- server-side interfaces
  - Static Skeleton
  - Dynamic Skeleton Interface (DSI)
  - Object Adapter (OA)

ORB Core Interfaces III

Client (IDL) Stub

- specific of each remote interface and operation, with static typing and dynamic binding
- automatically generated by compilation tools
- conversion of request parameter in network format (marshalling)
- synchronous, blocking invocation

ORB Core Interfaces IV

Dynamic Invocation Interface (DII)

- generic, with *dynamic typing* and *dynamic binding*
- directly provided by the Object Request Broker
- both *synchronous* and *deferred synchronous* invocations are possible
- provides a reflective interface
    - request
    - parameter
    - ...

ORB Core Interfaces V

Static Skeleton (IDL)

- corresponds to the Client Stub on Object Implementation side
- automatically generated by compilation tools
- builds parameters from network format (unmarshalling), calls the operation body, and sends back the result

Dynamic Skeleton Interface (DSI)

- conceptually analogous to Dynamic Invocation Interface
- allows the ORB to forward requests it does not manage to Object Implementations
- can be used to make bridges between different ORBs

ORB Core Interfaces VI

Object Adapter (OA)

- connects the Servant – the component containing an Object Implementation – to the ORB
- since in CORBA the Object Implementation is reactive, the OA has the task of activating and deactivating it
- there can be many Object Adapters
  - the CORBA 2.0 standard specifies the Basic Object Adapter (BOA)
  - the CORBA 2.3 standard specifies the Portable Object Adapter (POA)

ORB Core Interfaces VII

ORB Interface

- common interface for maintenance operations
- initialization functions
- bi-directional translation between Object Reference and strings
- operations of this interface are represented as belonging to pseudo-objects

CORBA Interoperability I

Evolution of the standard

- CORBA is heterogeneous for operating system, network transport, and programming language
- with the 1.2 version of the standard, interoperation was limited to ORBs from the same vendor.
    - in CORBA 1.2 two objects managed by ORBs from different vendors could not interact

  → very limited notion of interoperability
- CORBA 2.x grants interoperability among ORBs from different vendors

CORBA Interoperability II

Recipe for interoperability

- *communication protocols* shared among ORBs
- *data representation* common among ORBs
- *object reference format* common among ORBs

Common communication protocols

- the standard defines the *General Inter-ORB Protocol* (GIOP), requiring a reliable and connection-oriented transport protocol
- upon TCP/IP CORBA the standard defines Internet Inter-ORB Protocol (IIOP)
- *object reference format* common among ORBs

CORBA Interoperability III

Common data representation

- Common Data Representation (CDR) format is specified as a part of GIOP
- CDR acts at the presentation layer in the ISO/OSI stack

Common object reference format

- Interoperable Object Reference (IOR) format
    - contains all information to contact a remote object (or more)

OMA Common Object Services I

Design guidelines for CORBAservices

- essential and flexible services
- widespread use of multiple inheritance (mix-in)
- service discovery is orthogonal to service use
- both local and remote implementations are allowed

! CORBAservices are ordinary Object Implementations

Naming Service

- it handles name $\leftrightarrow$ object reference associations
- White Pages service for name resolution
- it allows tree-like naming structures (naming contexts)
- fundamental as a bootstrap mechanism

OMA Common Object Services II

Object Trader Service

- Yellow Page service for CORBA objects
- it enables highly dynamic collaborations among objects

Life Cycle Service

- object *creation* has different needs with respect to object *use*

→ the Factory concept is introduced

- Factory Finders are defined, to have location transparency even at creation time
- this service does not standardise Factories (which are class-specific), but *copy*, *move*, and *remove* operations.

OMA Common Object Services III

Event Service

- (most) objects are reactive
- the Event Service enables notification delivery, decoupling the producer and the consumer with an event channel
- it supports both the push model (observer) and the pull model for event distribution
- suitable administrative interfaces allow event supplier and event consumer of push or pull kind to be connected

Notification Service

- it improves over the Event Service, with more expressiveness and flexibility

OMA Common Object Services IV

Transaction Service

- transactions are a cornerstone of business application
- a two-phase commit protocol grants ACID properties
- it supports flat and nested transactions

Concurrency Control Service

- it manages lock objects, singly or as part of groups
- integration with the Transaction Service
  - transactional lock objects

OMG IDL Language I

Motivation

- CORBA is *neutral* with respect to programming languages
- different parts of an application can be written in *different languages*
- a language to specify interactions across language boundaries is required

→ Interface Definition Language (IDL)

OMG IDL Language II

Overall features

- syntax and lexicon similar to C/C++/Java
- it expresses the *declarative part* of a language only
- services are exported through *interfaces*
- it provides support for *OOP concepts* such as inheritance or polymorphism

Programming with CORBA I

Overall picture

- the Broker architecture makes it possible to build distributed applications, heterogeneous with respect to
  - operating system
  - network protocol
- the OMG IDL language allows to build distributed applications, heterogeneous with respect to
  - programming language
- in the end, the distributed system should be implemented in some real programming languages

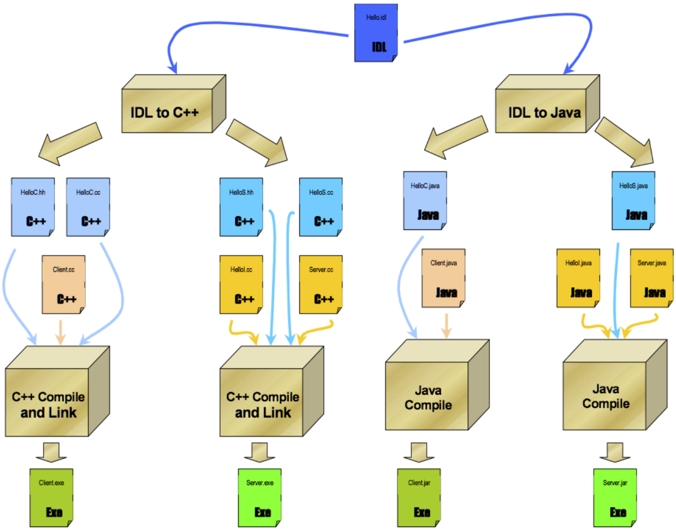
  → the IDL specification have to be cast into those languages

Programming with CORBA II

From IDL to real languages

- CORBA programming environments feature a tool called IDL compiler
  - it accepts OMG IDL as input, and generates code in a concrete implementation language
- with respect to a given IDL interface, a component may be a client and/or a server
  - the *client* requests the service, the *server* exports it
  - the IDL compiler generates code for both

Programming with CORBA III



Programming with CORBA IV

Language mappings

- for each supported programming language, the CORBA standard specifies a language mapping, specifying
  - how every OMG IDL construct is to be translated
  - programming techniques that are to be used
- the number of supported languages is large, and includes
  - C++
  - Java
  - SmallTalk
  - Perl
  - Ada
  - Ruby
  - Python

Objects and Metadata I

Meta-level

- seeking *flexibility* typically means looking for the ability to change dynamically with awareness
- this requires a new level allowing for
  - explicit description of system features
  - ability to enforce system change at run-time
- since this further level uses the first level as the object of its activity, it is called meta-level

Objects and Metadata II

Metadata

- since data belonging to the meta-level are *data about other data*, they are metadata—e.g., the schema of a DB

! systems have a (usually small) number of meta-levels—e.g. objects, classes and metaclasses in Smalltalk, or, the four-layer meta-model of UML

- OO software system were soon given metadata
  - Smalltalk has metaclasses
  - CLOS (Common Lisp Object System) introduced the concept of Meta-Object Protocol
  - Java has a Reflection API since version 1.1

! *reflection* is an architectural pattern [Buschmann et al., 1996]

Objects and Metadata III

Reflection & reification

- metadata are essential in open systems, to address heterogeneity, since they allow talking about system & component features
- *reification* is a pre-condition for reflection, making the representation of system properties explicitly available
- *reflection* is a basic mechanism for systems for self-observation—awareness
    - *reflective computation* works over reified system properties
    - *reflective update* dynamically affects system properties

! in a distributed system, metadata have to be *persistent*, *consistent*, and *available*

Objects and Metadata IV

Metadata in CORBA

Accordingly, metadata are used in several parts in the OMA architecture

- the Dynamic Invocation Interface allows to act on the remote operation invocation mechanism itself
- the Interface Repository allows runtime discovery of new IDL interfaces and their structure
- the Trader Service gathers services exported by objects into a yellow-page structure

The Dynamic Invocation Interface I

Goals of the DII

- the DII provides a complete and flexible interface to the remote invocation mechanism, around which CORBA is built
- the central abstraction supporting the DII is the Request pseudo-object, which reifies an instance of a remote call (Command design pattern, [Buschmann et al., 1996]

The Dynamic Invocation Interface II

IDL interfaces for the DII

- first, a request attached to a CORBA object needs be created
- the `create_request()` operation, belonging to the Object pseudo-interface (minimum of the inheritance graph), is to be used
- when a request is created, it is associated to its original Object Reference for its whole lifetime
- IDL is exploited to create a request
- after creation, a request object can be used via IDL

The Dynamic Invocation Interface III

```
// IDL create_request
module CORBA { // PIDL
  pseudo interface Object {
  typedef unsigned long ORBStatus;
  ORBStatus create_request(in Context ctx,
    in Identifier operation, // Operation name
    in NVList arg_list, // Operation arguments
    inout NamedValue result, // Operation result
    out Request request, // Newly created request
    in Flags req_flags; // Request flags);
  }; // End of Object pseudo interface
}; // End of CORBA module
```

The Dynamic Invocation Interface IV

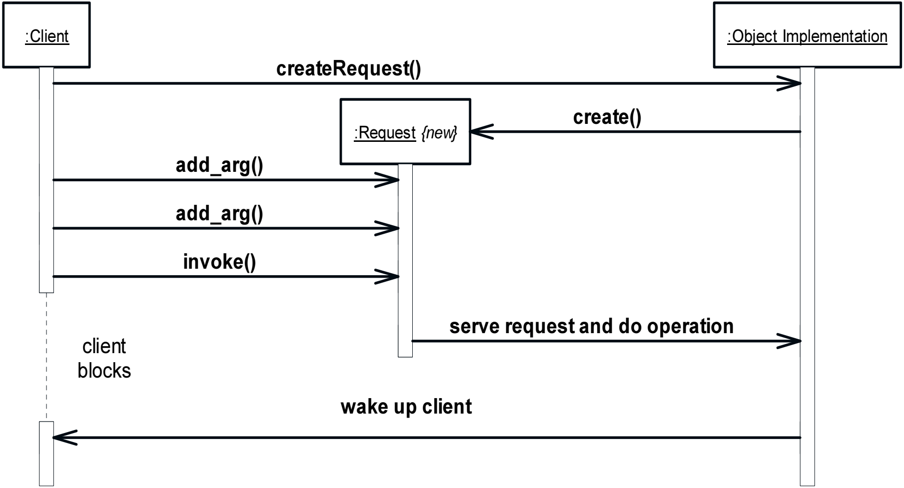
```
// IDL use object
module CORBA {
  typedef unsigned long Status;
  pseudo interface Request {
    Status add_arg(in Identifier name,
      in TypeCode arg_type,
      in any value, in long len,
      in Flags arg_flags);
    Status invoke(in Flags invoke_flags);
    Status delete(); // Destroy request object
    Status send(in Flags invoke_flags);
    Status get_response(in Flags response_flags);
  }; // End of Request interface
}; // End of CORBA module
```

The Dynamic Invocation Interface V

Communication via DII

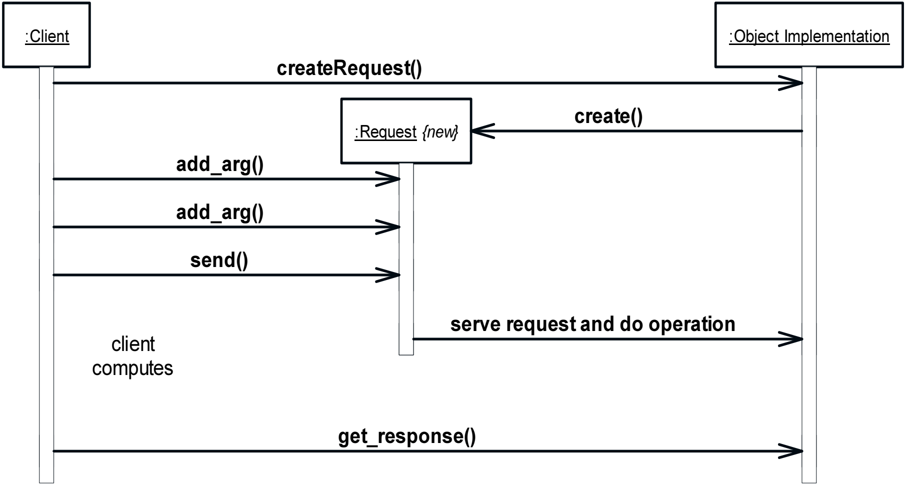
- through request objects the DII allows selecting the rendezvous policy
  - *synchronous call* with `invoke()`
  - *deferred synchronous call* with `send()`
- with deferred synchronous invocations, a group of requests can be sent all at once
- thhe new Asynchronous Method Invocation (AMI) specification of CORBA 2.4 also introduces *asynchronous calls*

The Dynamic Invocation Interface VI



Synchronous Call with the DII

The Dynamic Invocation Interface VII



Deferred synchronous Call with the DII

The Interface Repository

Goals & features

- the Interface Repository keeps the descriptions of all the IDL interfaces available in a CORBA domain
- using the Interface Repository, programs can discover the structure of types they do not have the stubs for
- a complete OO representation of the IDL language is stored within the Interface Repository
- with Repository IDs, more interface repositories can be federated

Dynamic Collaboration I

- CORBA objects are more adaptable than ordinary programming language objects such as Java or C++ objects
- two CORBA objects A and B, initially knowing nothing about each other, can set up a collaboration
  - object A uses `get_interface()` to get an `InterfaceDef` describing B
  - by browsing the Interface Repository, A discovers the syntax of the operations supported by B
  - using the DII, A creates a request and sends it to B
- with CORBA, the syntax of the operations can be discovered at runtime

Dynamic Collaboration II

The issue of semantics

- the specification of the semantics of operations is missing in CORBA
  - OMG IDL cannot specify preconditions, postconditions, and invariants
  - the domain of discourse cannot be semantically represented in CORBA

! more complex systems (like multi-agent systems) require languages to describe the domain of the discourse (*ontologies*)

Summing Up I

Middleware...

- mediates between different OS and distributed applications
- aims at interoperability
- provides integration technologies
- targets conceptual integrity
- represented as abstract vs. concrete middleware

Summing Up II

Communication

- Remote Procedure Call
- message-oriented models
- streaming
- other forms like multicasting and epidemic protocols are important, but are not a subject for this course

Summing Up III

Naming

- naming is a general issue, particularly relevant in the distributed setting
- naming system is typically provided by middleware
- different approaches to naming are possible: flat, structured, attribute-based
- typically, naming systems take a hybrid stance to the naming problem
- DNS and LDAP are paradigmatic examples of naming systems

Summing Up IV

Object-oriented middleware

- it provides a coherent framework for distributed OOP, both conceptually and technologically
- it extends OOP to distributed systems
- it hides the complexity of programming DS
- it is supported by open standards—such as OMG CORBA and OSGi
- it promotes integration across OSs, networks and languages
- it counts on a lot of free implementations available

Summing Up V

CORBA

- the historical reference for OO middleware
- OMA: ORB, Services, Facilities
- core interfaces: IDL stub & skeleton, DII & DSI, OA
- interoperability: IDL & Interface Repository
- programming with CORBA
- metadata
- dynamic object collaboration

References I

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996).
Pattern-Oriented Software Architecture: A System of Patterns, volume 1.
John Wiley & Sons, New York, NY.

Day, J. (1995).
The (un)revised OSI reference model.
ACM SIGCOMM Computer Communication Review, 25(5):39–55.

Gropp, W. (2011).
MPI (Message Passing Interface).
In Padua, D., editor, *Encyclopedia of Parallel Computing*, pages 1184–1190. Springer US, Boston, MA, USA.

Liskov, B. (1987).
Data abstraction and hierarchy.
In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87) – Addendum*, pages 17–34, New York, New York, USA. ACM Press.

Tanenbaum, A. S. and van Steen, M. (2007).
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.

References II

Vessey, I. and Skinner, G. (1990).
Implementing Berkeley Sockets in System V release 4.
In *Proceedings of the Winter 1990 USENIX Conference*, pages 177–193, Washington, DC, USA.

Middleware

Distributed Systems

Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018