Middleware for synchronous requests

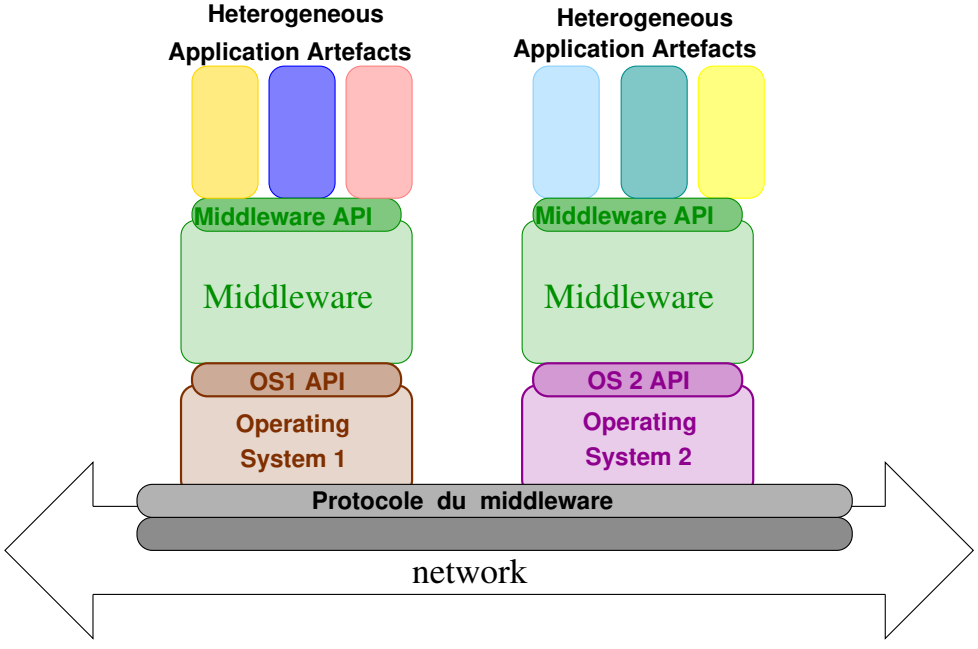
Revision : 528

Sophie Chabridon and Chantal Taconet

September 2021

1 Introduction

1.1 Middleware for distribution



- **Middleware** is a software layer which provides :
  - Programming interfaces (common API)
  - Protocol for interoperability
    - With data exchange format

. . . to support distribution and heterogeneity.

1.2 Goal : interoperability

- Existing *"legacy code"*,
- Numerous languages,
- Several operating systems,
- Various hardware (e.g., little endian, big endian),
- Several network protocols

$\Rightarrow$ need for interoperability!

1.3 Distribution models

- Point to point message
- Point to multipoint message
- Event/action
- Publish/subscribe
- **Client/server**
- Mobile code
- Virtual shared memory

1.4 Client-server models

- **Procedural**
  - Remote Procedure Call - RPC
- **Object-oriented**
  - Remote Method Invocation (Java RMI, Common Object Request Broker Architecture CORBA)
- **Data-oriented**
  - SQL requests
  - REST (Representational State Transfer)- create, read, update, delete over HTTP
- Traditionnal **Web** (HTTP requests)
- **Web Services** (SOAP over HTTP)

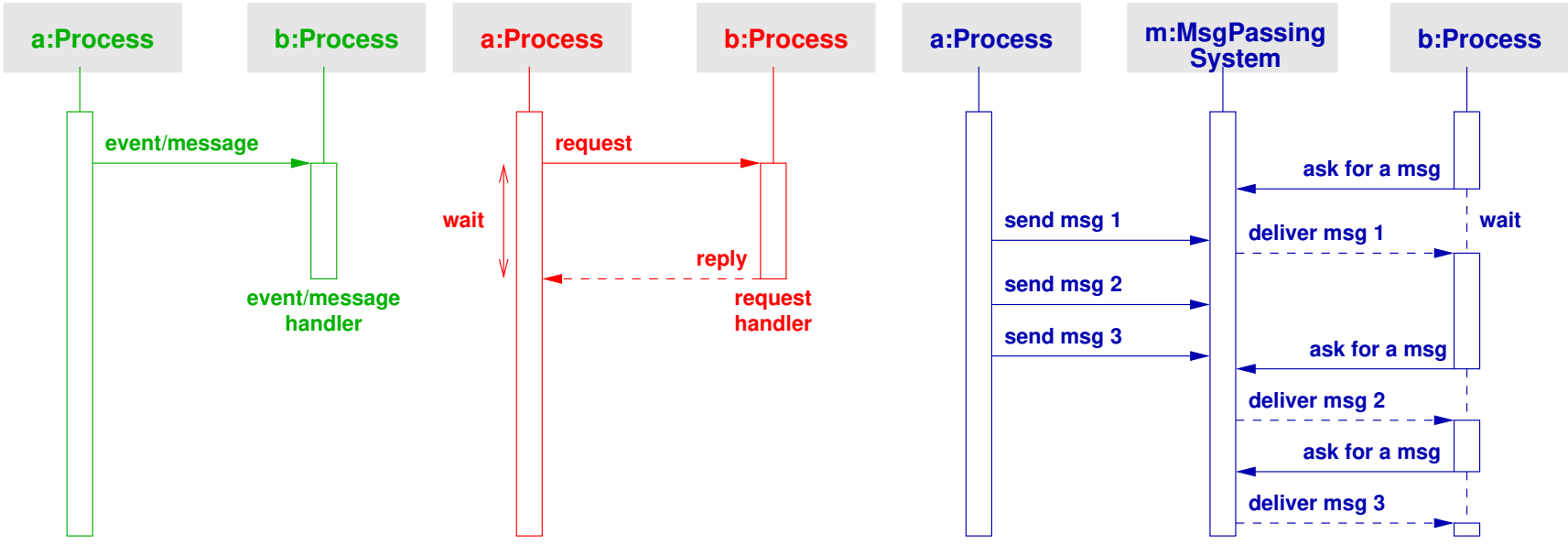
1.5 Middleware for distributed objects history

- Comes from two technologies :
  - **Objects (inheritance, encapsulation and polymorphism)**
  - RPC or Remote Procedure Call (distribution, heterogeneity, data marshalling and unmarshalling)

1.6 Synchronous vs asynchronous mode

- Two entities (*e.g.*, processus) communicate
  - In **synchronous** mode: the two entities (client and server) are active at the same time, after a request, client is waiting for server response.
  - In *asynchronous* mode: entities send messages, they don't wait for responses, they don't know when the message will be delivered

1.7 Asynchronous call, synchronous call, buffered message

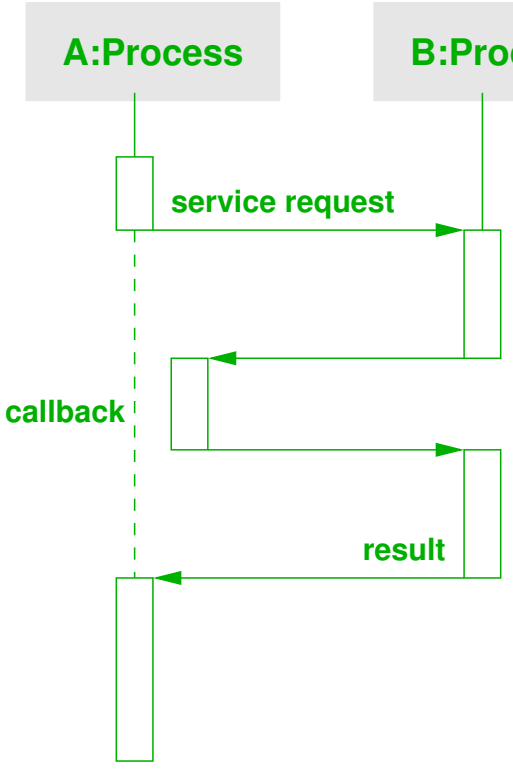


Asynchronous event (push)

Synchronous call

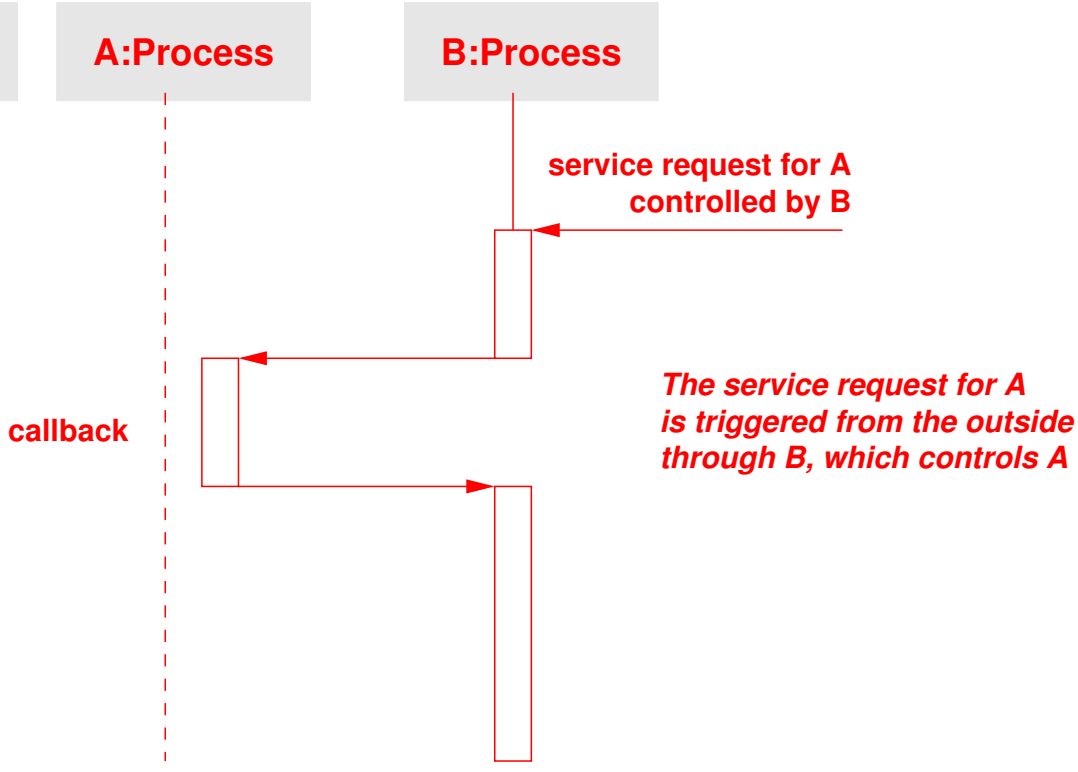
Buffered messages (pull)

1.8 Call-back and Inversion of control



Synchronous call with callback

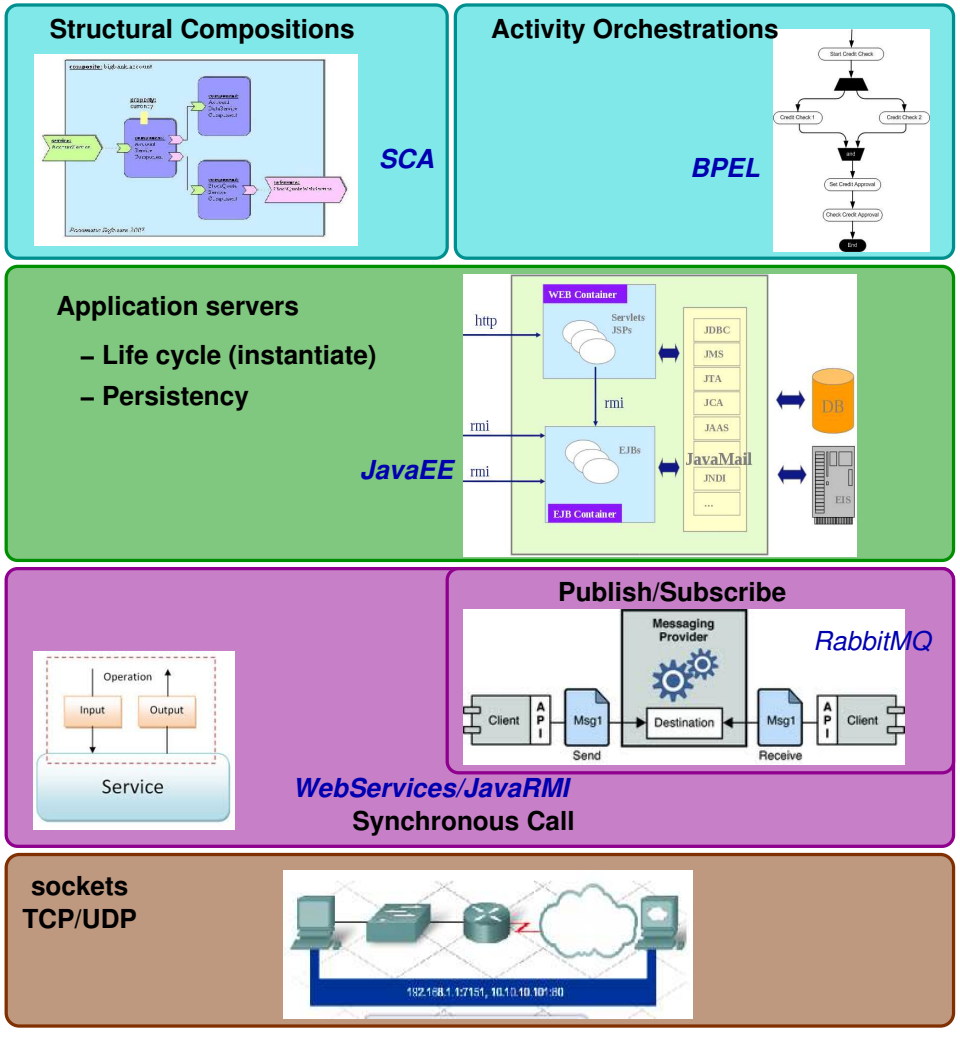
A callback is first registered
and later called asynchronously.



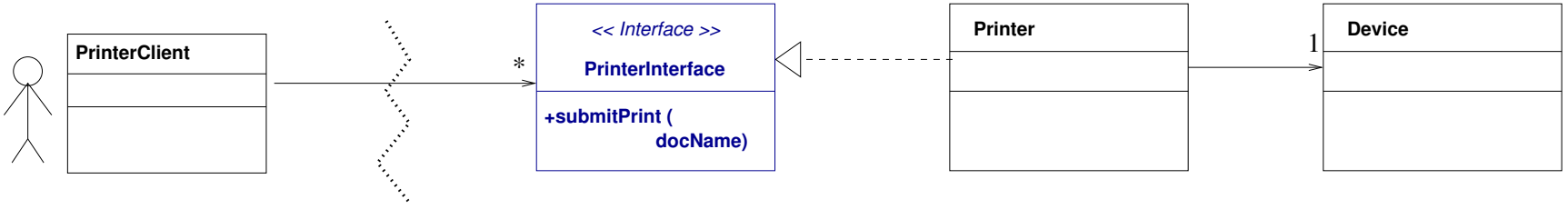
Inversion of control

The control flow is no more under the responsability of
the application but controlled by the framework.

2 Synchronous middleware and the big picture

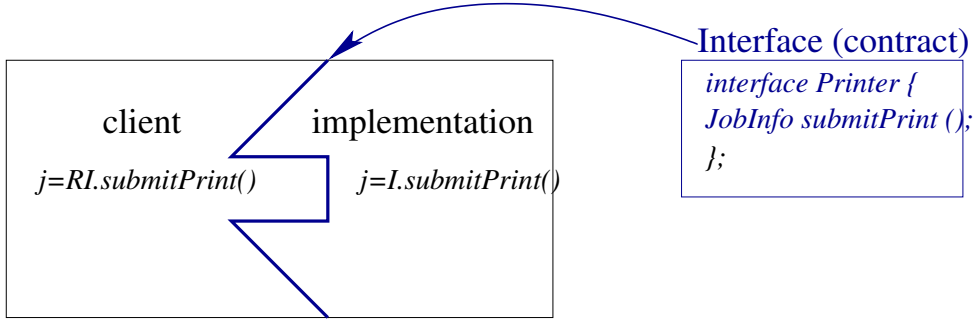


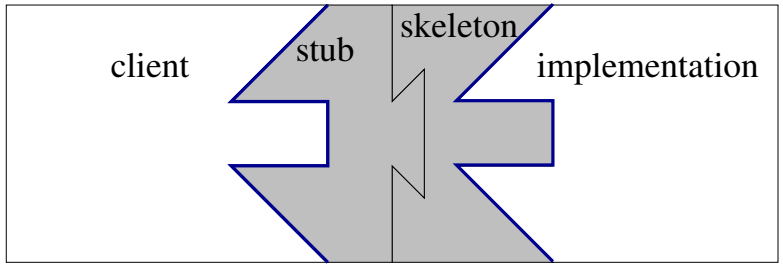
2.1 Introduction of the distributed example



- Which distribution ?
- Which abstractions (service, object) ?
- Which middleware ?

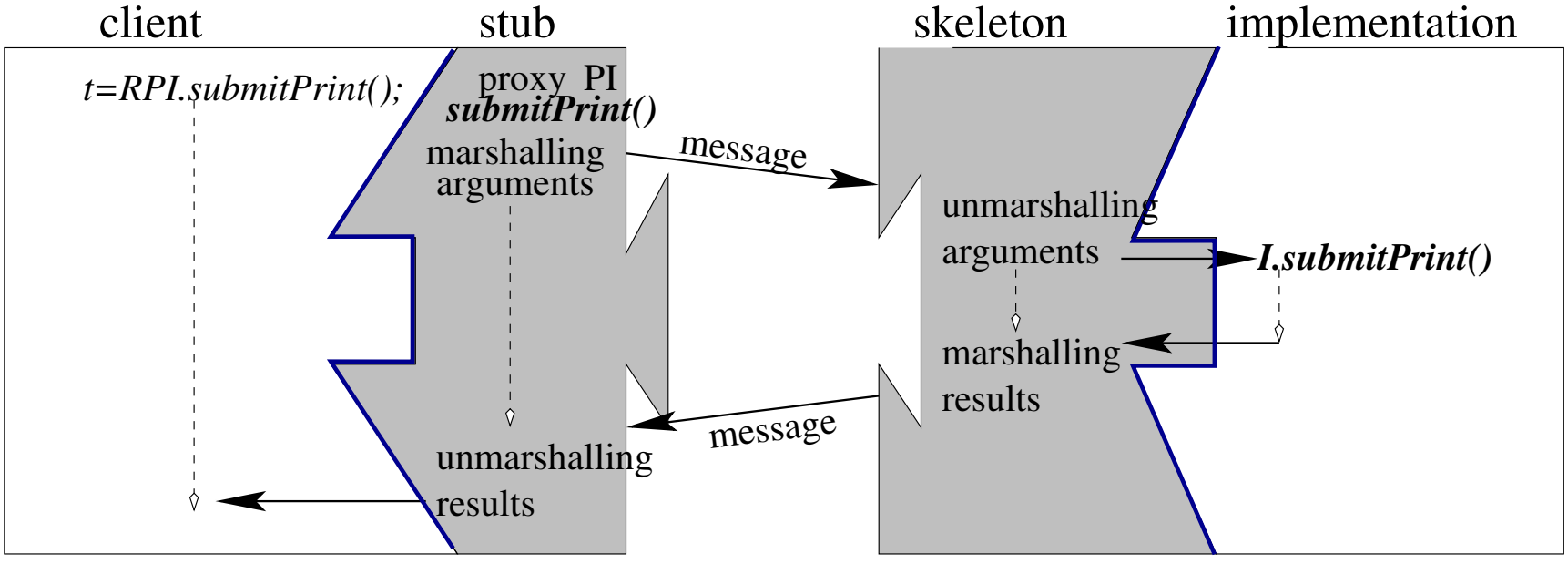
2.2 Principle of distributed objects





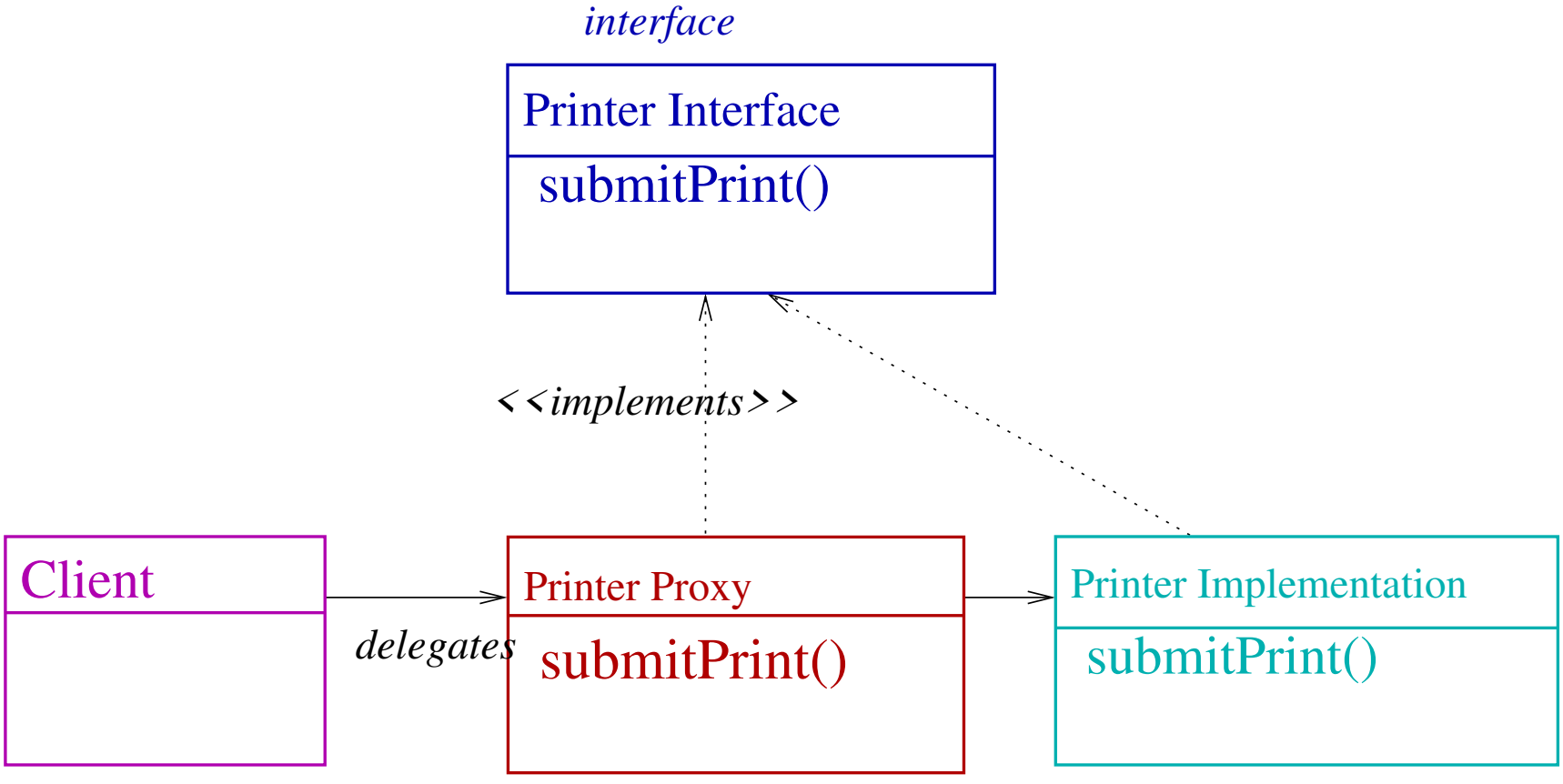
Automatic generation
stub and skeleton

2.3 The stub and the skeleton



2.4 Proxy Object and inheritance tree

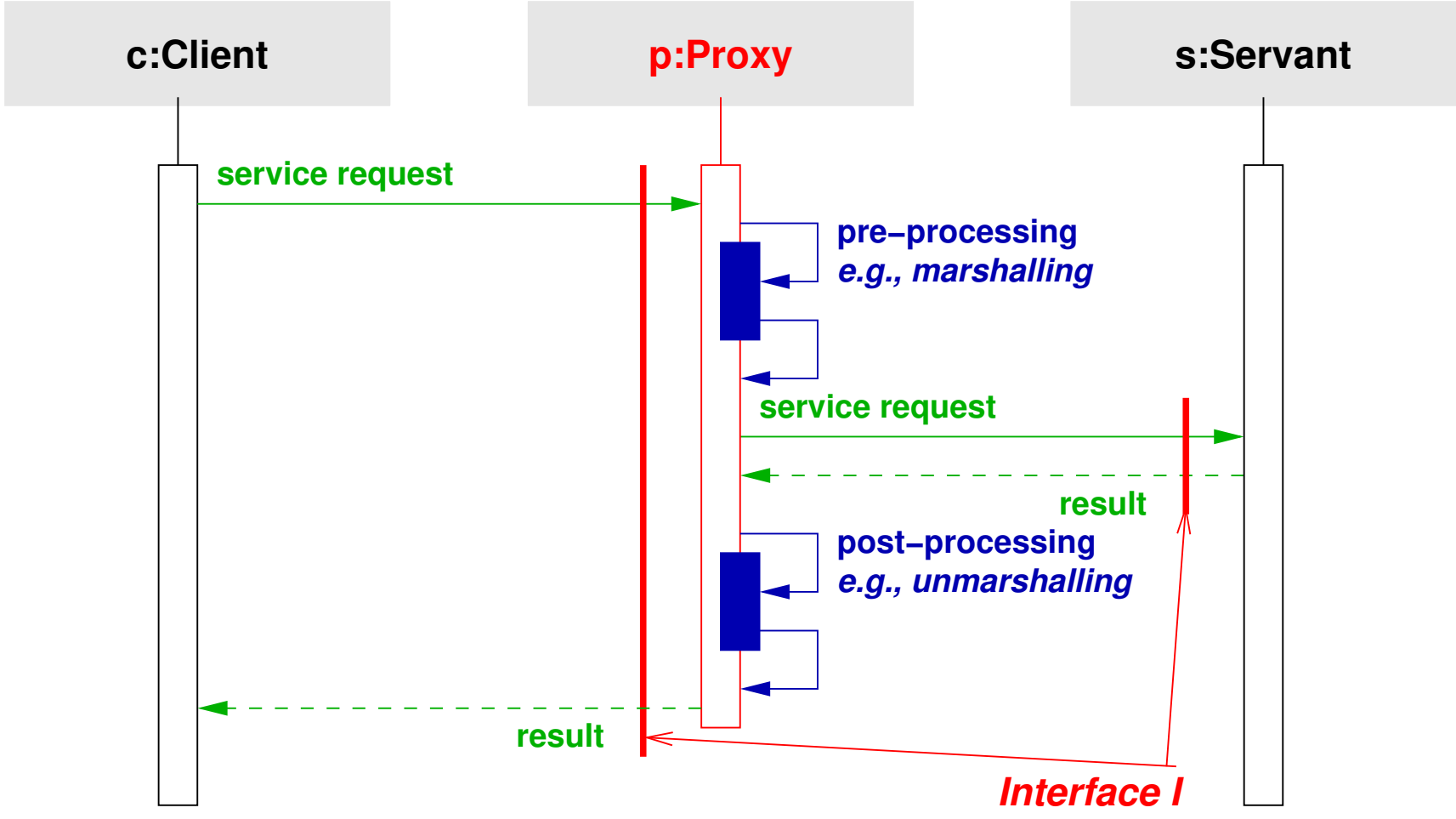
- Proxy: Representative for remote access



2.5 Proxy design pattern

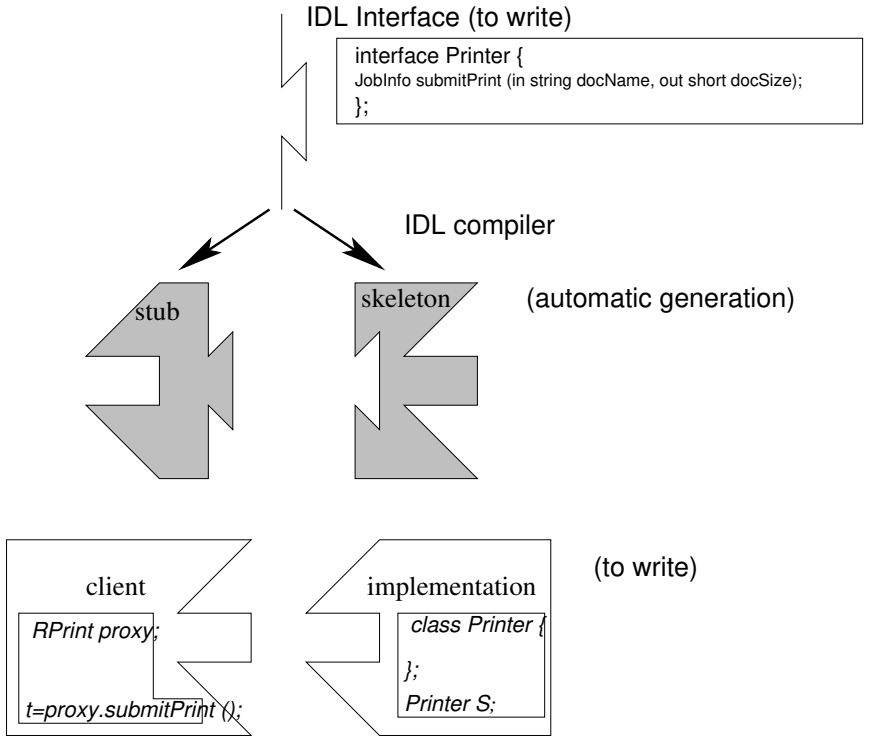
- Context: A client needs access to a remote service provided by some entity (called the "servant")
- Problem
  - Define an access mechanism that does not involve
    - Hard-coding the location of the servant into the client code
    - Deep knowledge of the communication protocols by the client
  - Desirable properties
    - Access should be efficient at run-time and secure
    - Programming should be simple: No difference between local and remote access
  - Constraints: Distributed environment (no single address space)
- Solutions
  - Use a local representative of the server on the client side that isolates the client from the communication system and the servant
  - Keep the same interface for the representative as for the servant
  - Define a uniform proxy structure to facilitate automatic generation

2.5.1 Sequence diagram of Proxy

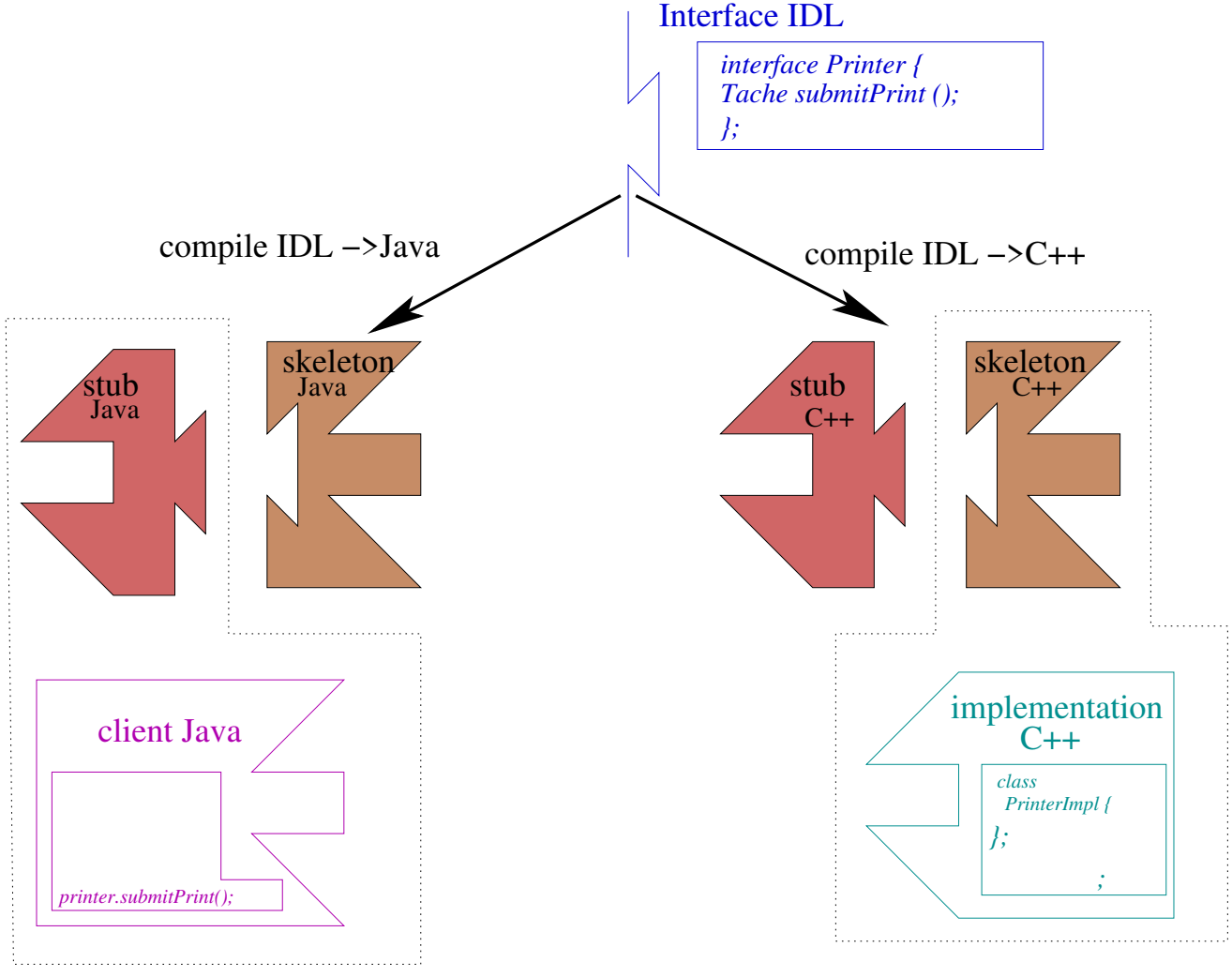


2.6 Distribution Implementation Process

1. Description of the interface in **IDL**
2. IDL compiler creates the stub and the skeleton
3. Write both **client and server implementations**



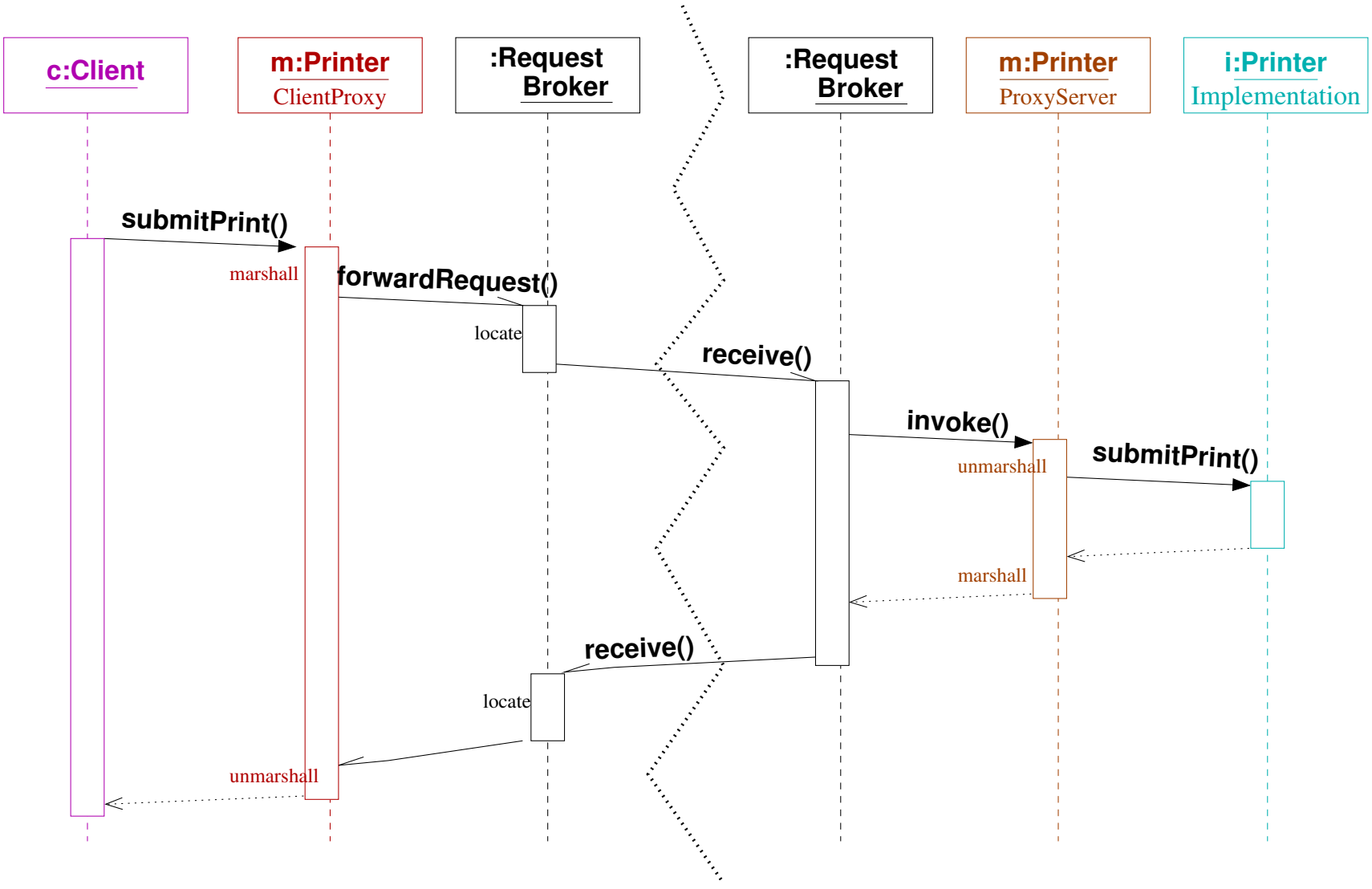
2.7 Multi-languages (or multi-ORBs, or multi-OSs)



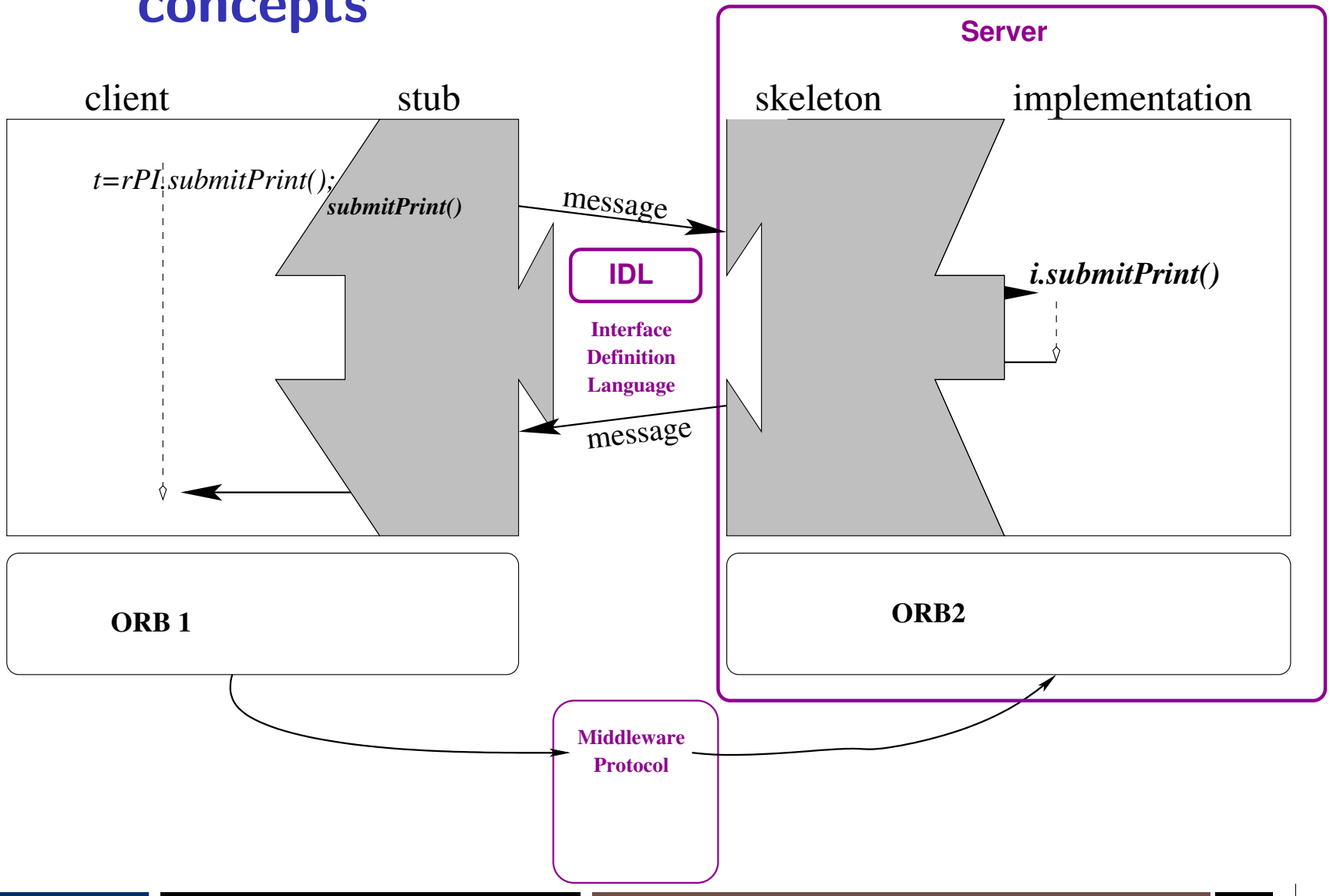
2.8 Distribution implications

- Objects/service implementation are in different spaces (not the same process, not the same computer . . .):
  - Assign a **unique identifier** to each object/service in different spaces
  - Localize objects/service implementations
  - Transports requests and replys
  - Use of a neutral network format for the data

2.9 Invocation sequence diagram



2.10 Middleware for synchronous requests : main concepts



2.11 Inherent complexity of distribution

- No global state
- Poor debugging tools
- Partial failures, network partition
- Requests in parallel (concurrency management)
- Trusting the caller (authentication)

3 Conclusions

3.1 Main distributed object middleware

CORBA (OMG) 1991

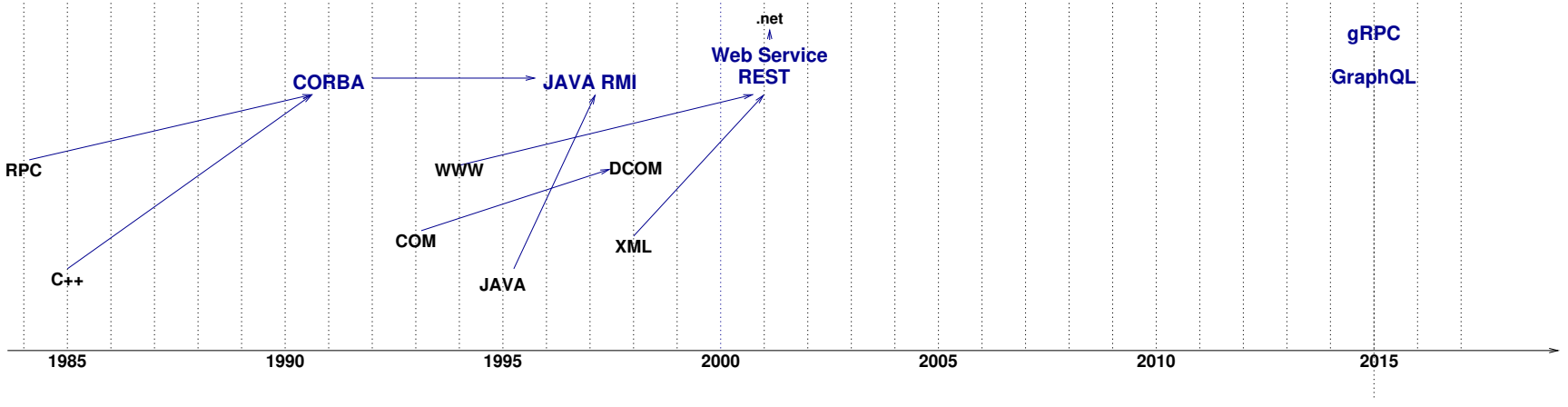
Java RMI (Sun) 1997

SOAP WebService (w3C) 2001

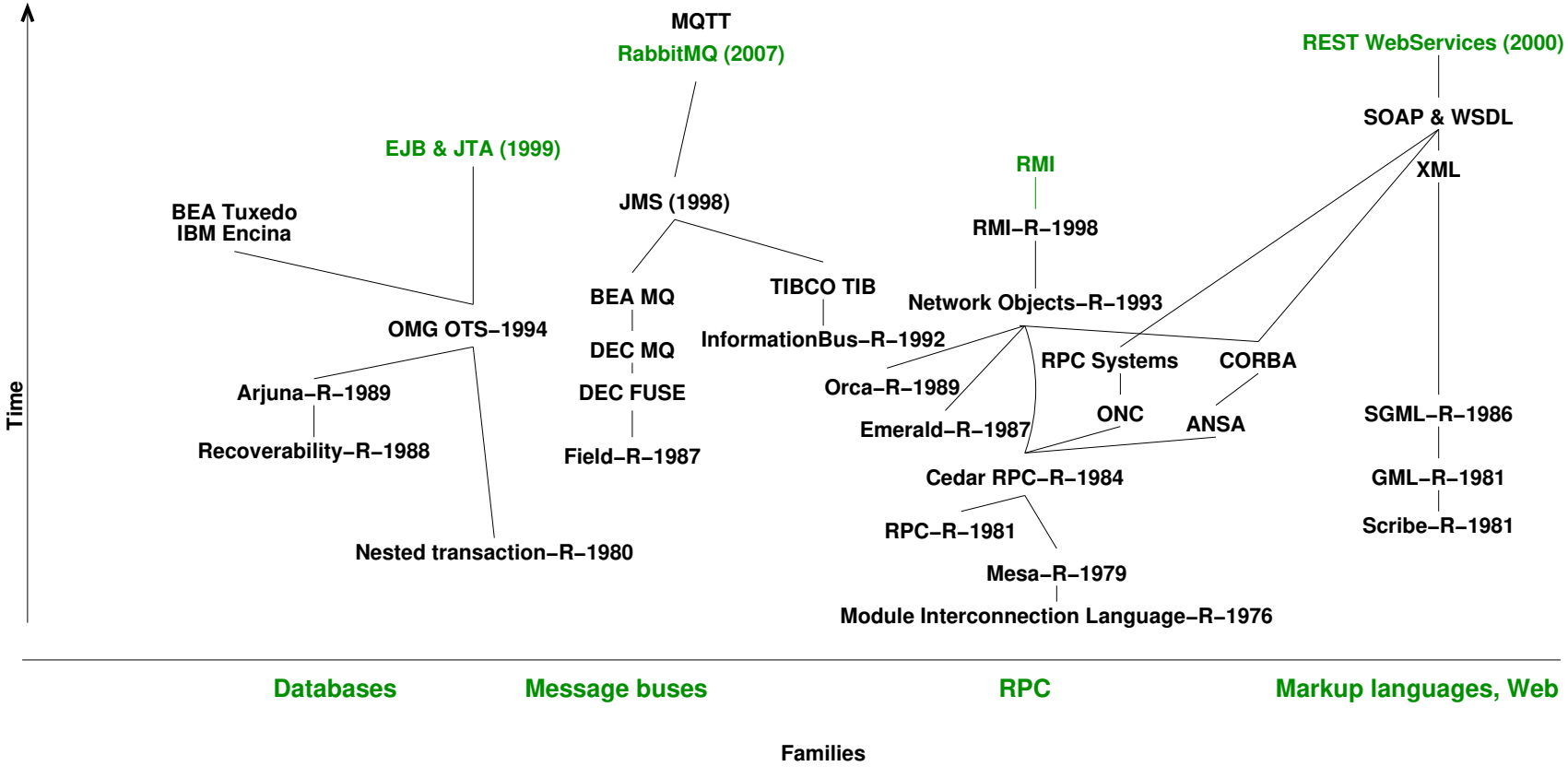
REST WebService (w3C) 2001

GoogleRPC (Google) 2015

GraphQL (Facebook) 2015



3.2 Middleware history



3.3 Comparison of historical synchronous middleware

	CORBA	RMI	SOAP	REST	
Origin	OMG	SUN	W3C	W3C	
Prog. language	multi	Java	multi	multi	
IDL	IDL CORBA	interface Java	WSDL	URIs	
data presentation	CDR/binary	serialisation/binary	SOAP Envelope/XML	JSON/XML/text	p
protocole	IIOP	IIOP	SOAP Protocol	HTTP1	
connexions	connected	connected	short connexions	short connexions	
object references	IOR (location independant)	symbolic names/IP+port	URL	URI	
naming service	NS, trading	RMI registry, JNDI	UDDI, WSIL	/	
main advantages	services/efficiency	easy to use in java	SOA	simple	efficient
main difficulties	complex to learn	Java/Java	complex	low level/need a third party	

3.4 Conclusions

- Granularity of distribution variable (object, service),
- Complexity of distribution
- Synchronous request middleware is the necessary foundation to build higher level middleware
  - Message Oriented Middleware (MOM) (asynchronous middleware)
  - Application servers
  - Component middleware
  - Compositions and orchestrations

References

Krakowiak, S. (2009).

Middleware Architecture with Patterns and Frameworks.

.