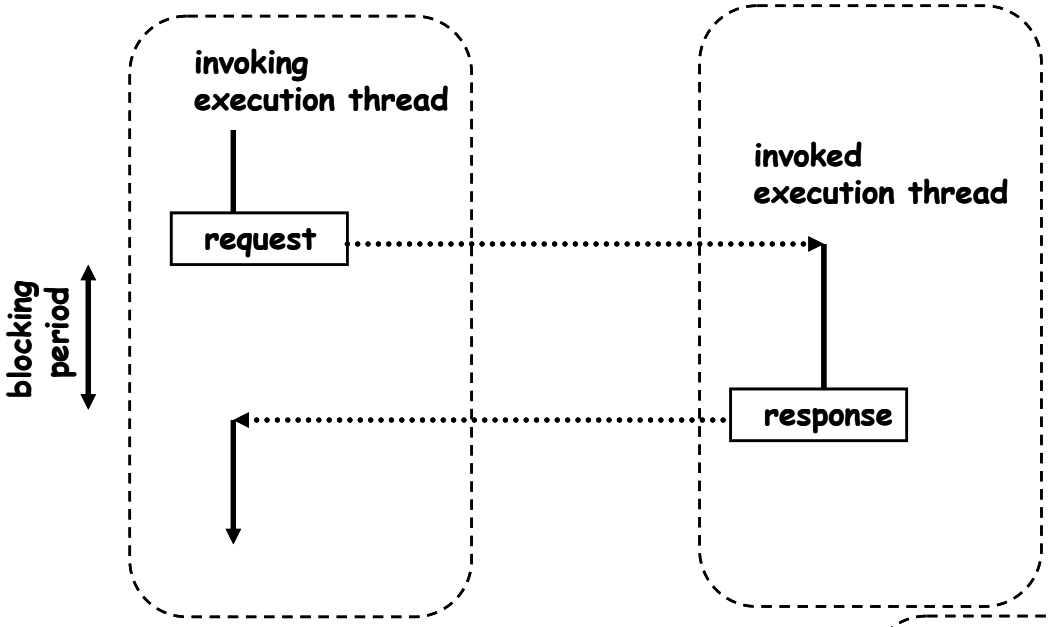
Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

Sapienza
Università di Roma

Progetto di Applicazioni Software

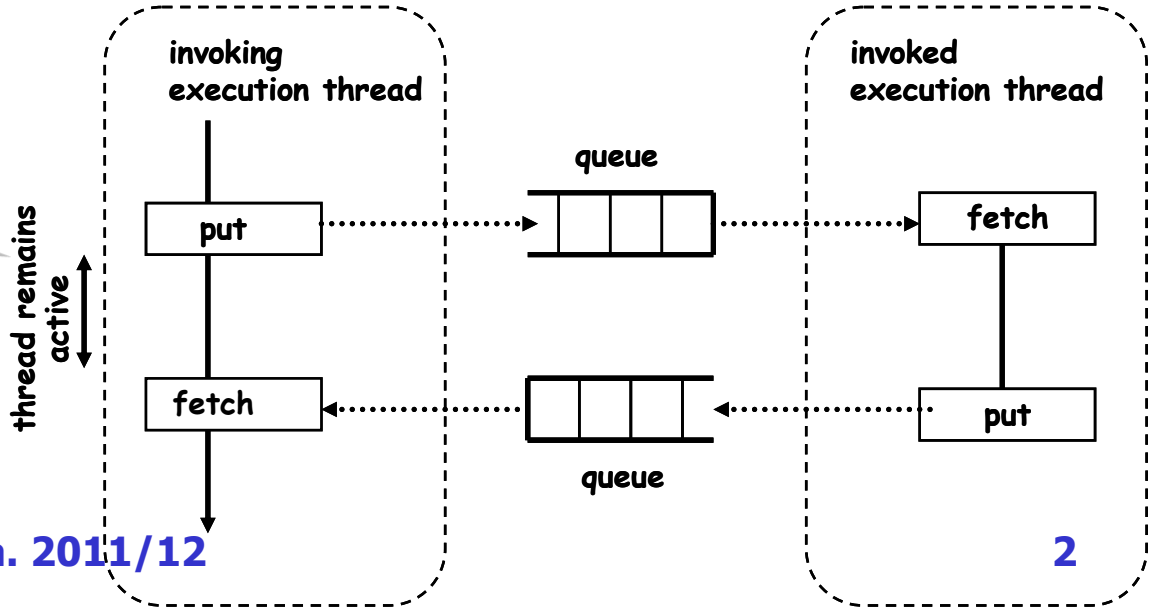
Tecnologie di Base per la Programmazione in Ambienti Distribuiti Remote Procedure Call (RPC) e Message Oriented Middleware (MOM) RPC e MOM in Java: Java RMI e JMS

Paradigmi di Comunicazione



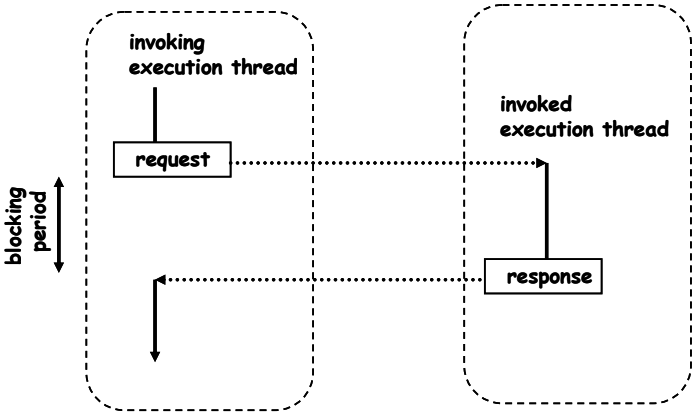
Interazione bloccante (impropriamente detta “sincrona”)

Interazione non-bloccante (impropriamente detta “asincrona”)



Blocking or Synchronous Interaction

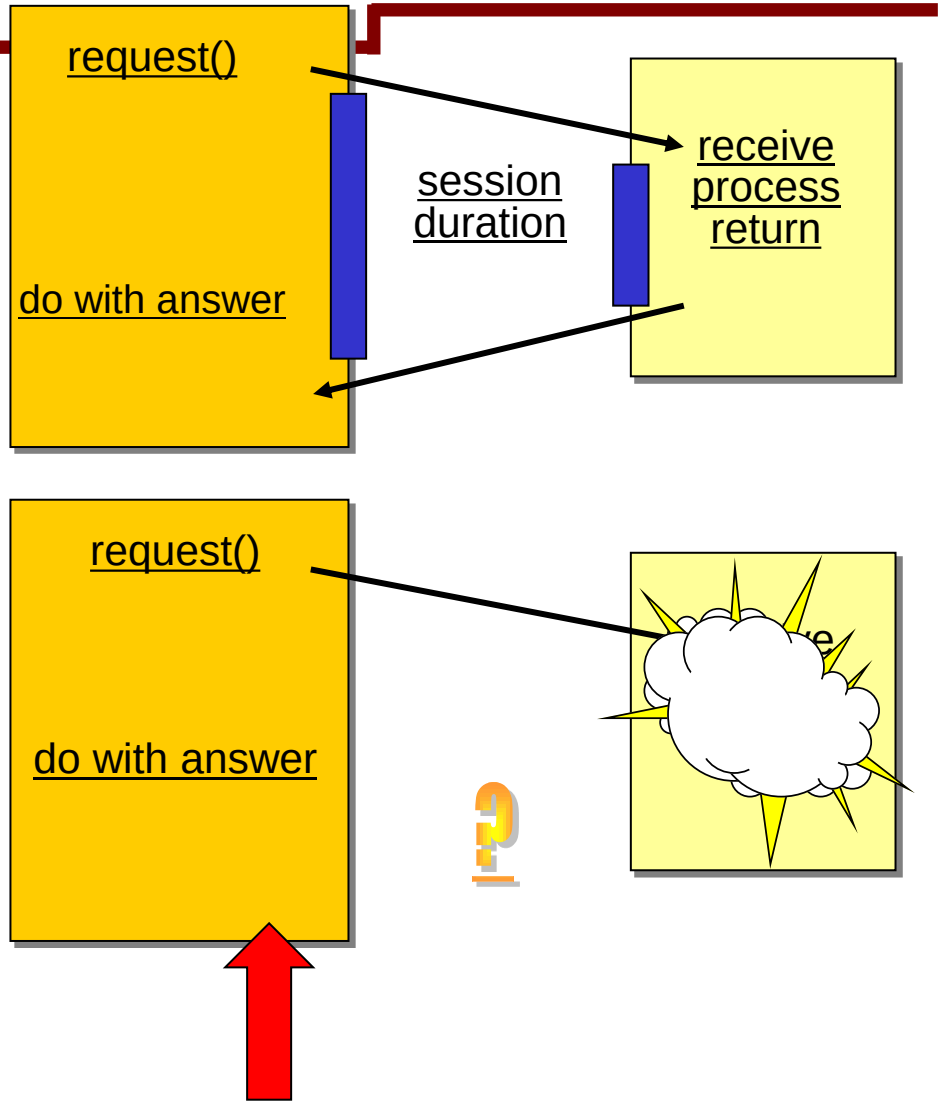
- Traditionally, distributed applications use blocking calls (the client sends a request to a service and waits for a response of the service to come back before continuing doing its work)
- Synchronous interaction requires both parties to be “on-line”: the caller makes a request, the receiver gets the request, processes the request, sends a response, the caller receives the response
- The caller must wait until the response comes back. The receiver does not need to exist at the time of the call (e.g., some technologies create an instance of the service/server /object when called if it does not exist already) but the interaction requires both client and server to be “alive” at the same time



- Because it synchronizes client and server, this mode of operation has several disadvantages:
  - connection overhead
  - higher probability of failures
  - difficult to identify and react to failures
  - it is a one-to-one system; it is not really practical for nested calls and complex interactions (the problems becomes even more acute)

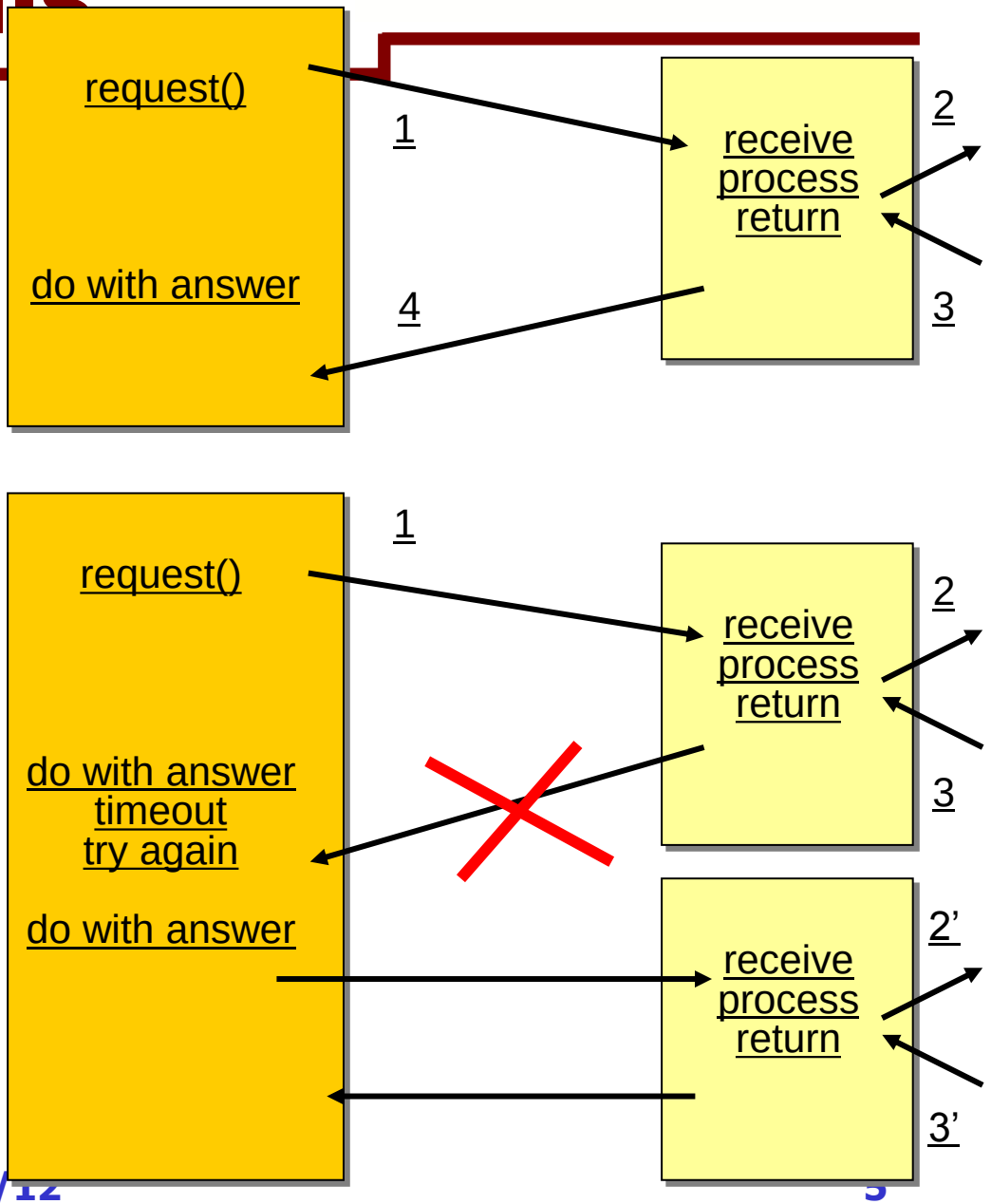
Overhead of Synchronism

- Synchronous invocations require to maintain a session between the caller and the receiver
- Maintaining sessions is expensive and consumes CPU resources. There is also a limit on how many sessions can be active at the same time (thus limiting the number of concurrent clients connected to a server)
- For this reason, client/server systems often resort to connection pooling to optimize resource utilization
  - have a pool of open connections
  - associate a thread with each connection
  - allocate connections as needed
- Synchronous interaction requires a context for each call and a context management system for all incoming calls. The context needs to be passed around with each call as it identifies the session, the client, and the nature of the interaction.



Failures in Synchronous Calls

- If the client or the server fail, the context is lost and resynchronization might be difficult.
  - If the failure occurred before 1, nothing has happened
  - If the failure occurs after 1 but before 2 (receiver crashes), then the request is lost
  - If the failure happens after 2 but before 3, side effects may cause inconsistencies
  - If the failure occurs after 3 but before 4, the response is lost but the action has been performed (do it again?)
- Who is responsible for finding out what happened?
- Finding out when the failure took place may not be easy. Worse still, if there is a chain of invocations (e.g., a client calls a server that calls another server) the failure can occur anywhere along the chain.



Two Solutions

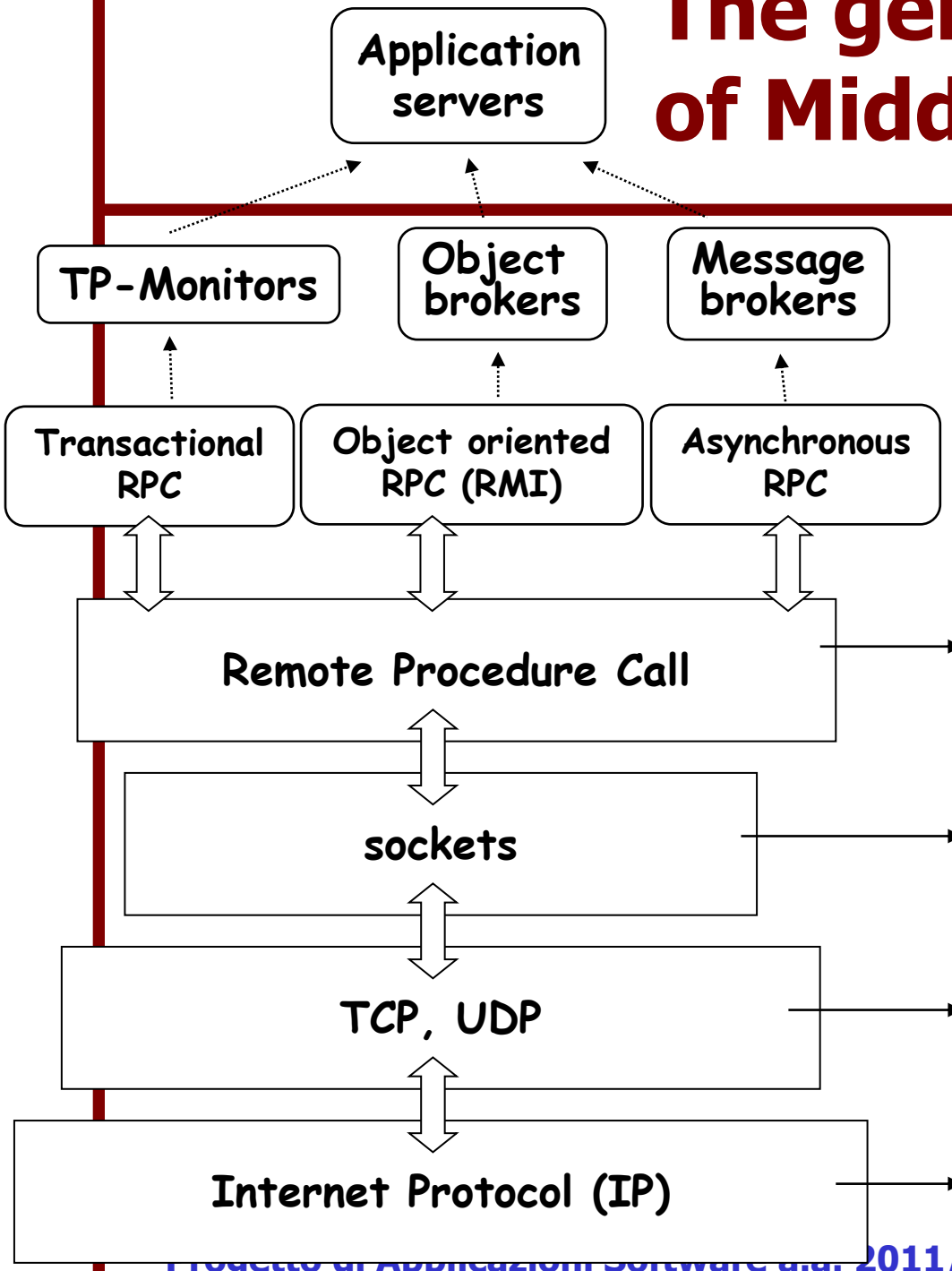
ENHANCED SUPPORT

- Client/server systems and middleware platforms provide a number of mechanisms to deal with the problems created by synchronous interaction:
  - Transactional interaction: to enforce exactly once execution semantics and enable more complex interactions with some execution guarantees
  - Service replication and load balancing: to prevent the service from becoming unavailable when there is a failure (however, the recovery at the client side is still a problem of the client)

ASYNCHRONOUS INTERACTION

- Using asynchronous interaction, the caller sends a message that gets stored somewhere until the receiver reads it and sends a response. The response is sent in a similar manner
- Asynchronous interaction can take place in two forms:
  - non-blocking invocation (a service invocation but the call returns immediately without waiting for a response, similar to batch jobs)
  - persistent queues (the call and the response are actually persistently stored until they are accessed by the client and the server)

The genealogy of Middleware



Remote Procedure Call:
hides communication details behind a procedure call and helps bridge heterogeneous platforms

sockets:
operating system level interface to the underlying communication protocols

TCP, UDP:
User Datagram Protocol (UDP) transports data packets without guarantees
Transmission Control Protocol (TCP) verifies correct delivery of data streams

Internet Protocol (IP):
moves a packet of data from one node to another

Understanding Middleware

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

PROGRAMMING ABSTRACTION

- Intended to hide low level details of hardware, networks, and distribution
- Trend is towards increasingly more powerful primitives that, without changing the basic concept of RPC, have additional properties or allow more flexibility in the use of the concept
- Evolution and appearance to the programmer is dictated by the trends in programming languages (RPC and C, CORBA and C++, RMI and Java, Web services and SOAP-XML)

INFRASTRUCTURE

- Intended to provide a comprehensive platform for developing and running complex distributed systems
- Trend is towards service oriented architectures at a global scale and standardization of interfaces
- Another important trend is towards single vendor software stacks to minimize complexity and streamline interaction
- Evolution is towards integration of platforms and flexibility in the configuration (plus autonomic behavior)

Middleware as a Programming Abstraction

- Programming languages and almost any form of software system evolve always towards higher levels of abstraction
  - hiding hardware and platform details
  - more powerful primitives and interfaces
  - leaving difficult task to intermediaries (compilers, optimizers, automatic load balancing, automatic data partitioning and allocation, etc.)
  - reducing the number of programming errors
  - reducing the development and maintenance cost of the applications developed by facilitating their portability
- Middleware is primarily a set of programming abstractions developed to facilitate the development of complex distributed systems
  - to understand a middleware platform one needs to understand its programming model
  - from the programming model the limitations, general performance, and applicability of a given type of middleware can be determined in a first approximation
  - the underlying programming model also determines how the platform will evolve and fare when new technologies evolve

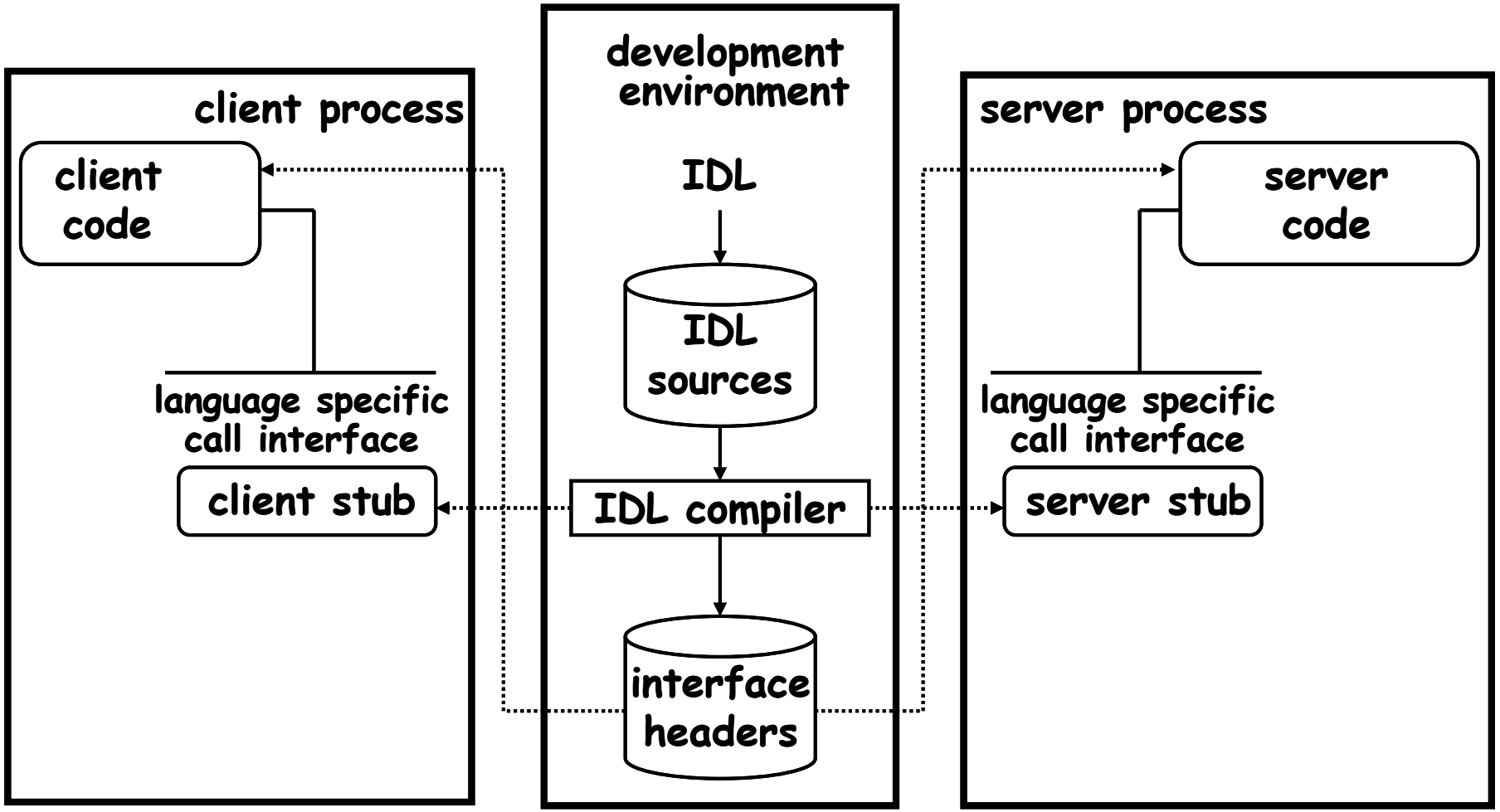
Middleware as Infrastructure

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
Sapienza Università di Roma

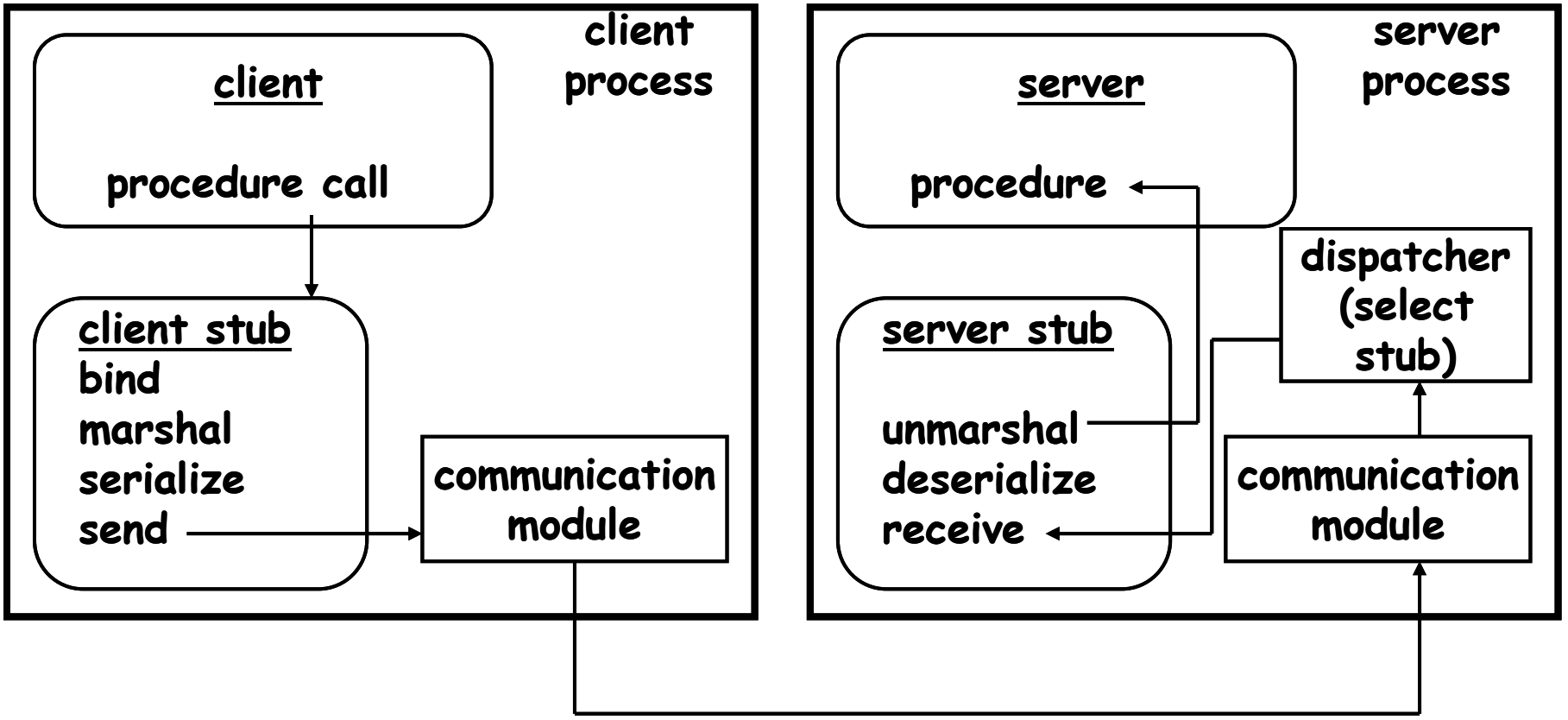
- As the programming abstractions reach higher and higher levels, the underlying infrastructure implementing the abstractions must grow accordingly
  - Additional functionality is almost always implemented through additional software layers
  - The additional software layers increase the size and complexity of the infrastructure necessary to use the new abstractions

- The infrastructure is also intended to support additional functionality that makes development, maintenance, and monitoring easier and less costly
  - RPC => transactional RPC => logging, recovery, advanced transaction models, language primitives for transactional demarcation, transactional file system, etc.
  - The infrastructure is also there to take care of all the non-functional properties typically ignored by data models, programming models, and programming languages: performance, availability, recovery, instrumentation, maintenance, resource management, etc.

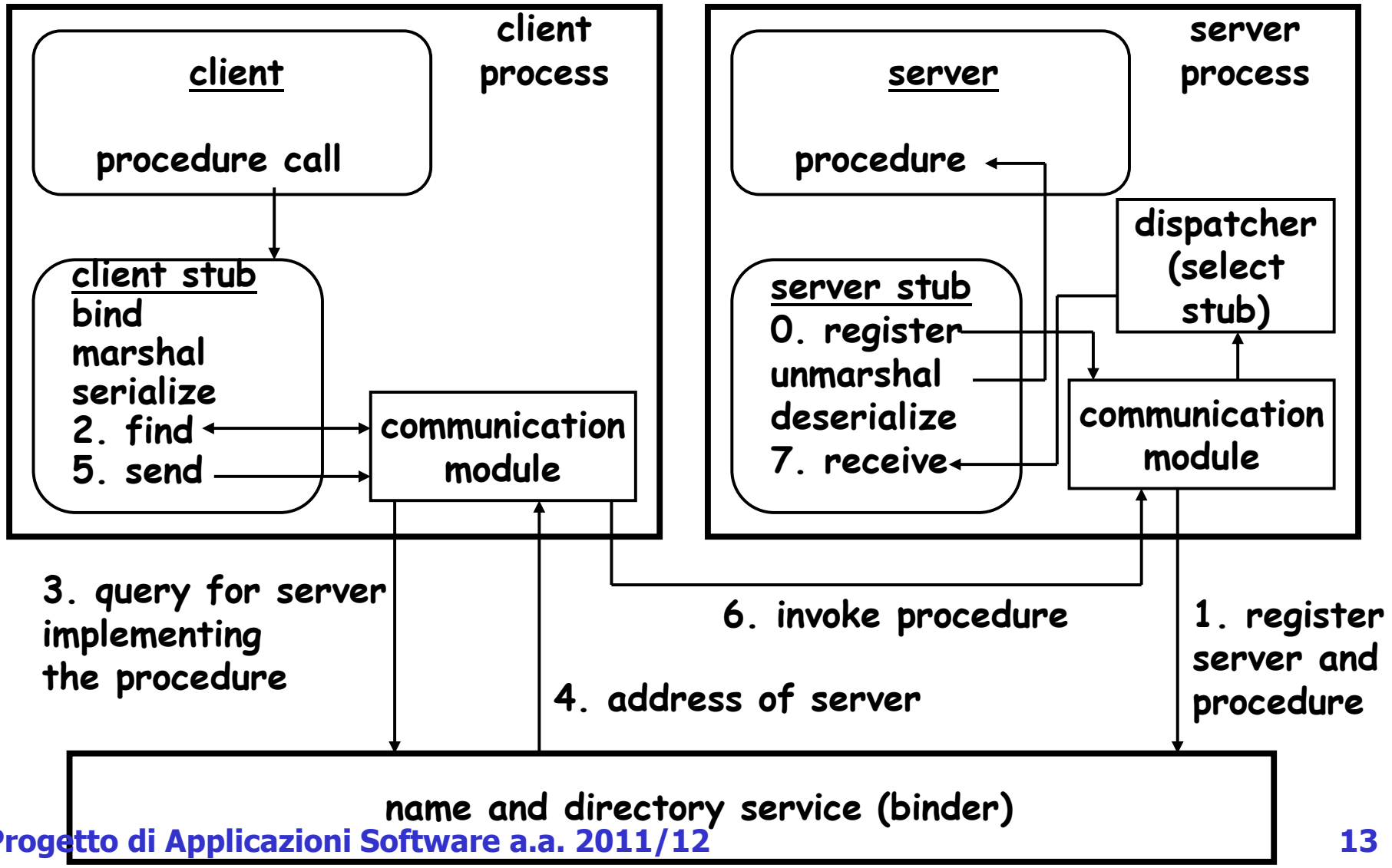
Basic Middleware: RPC (1)



Basic Middleware: RPC (2)



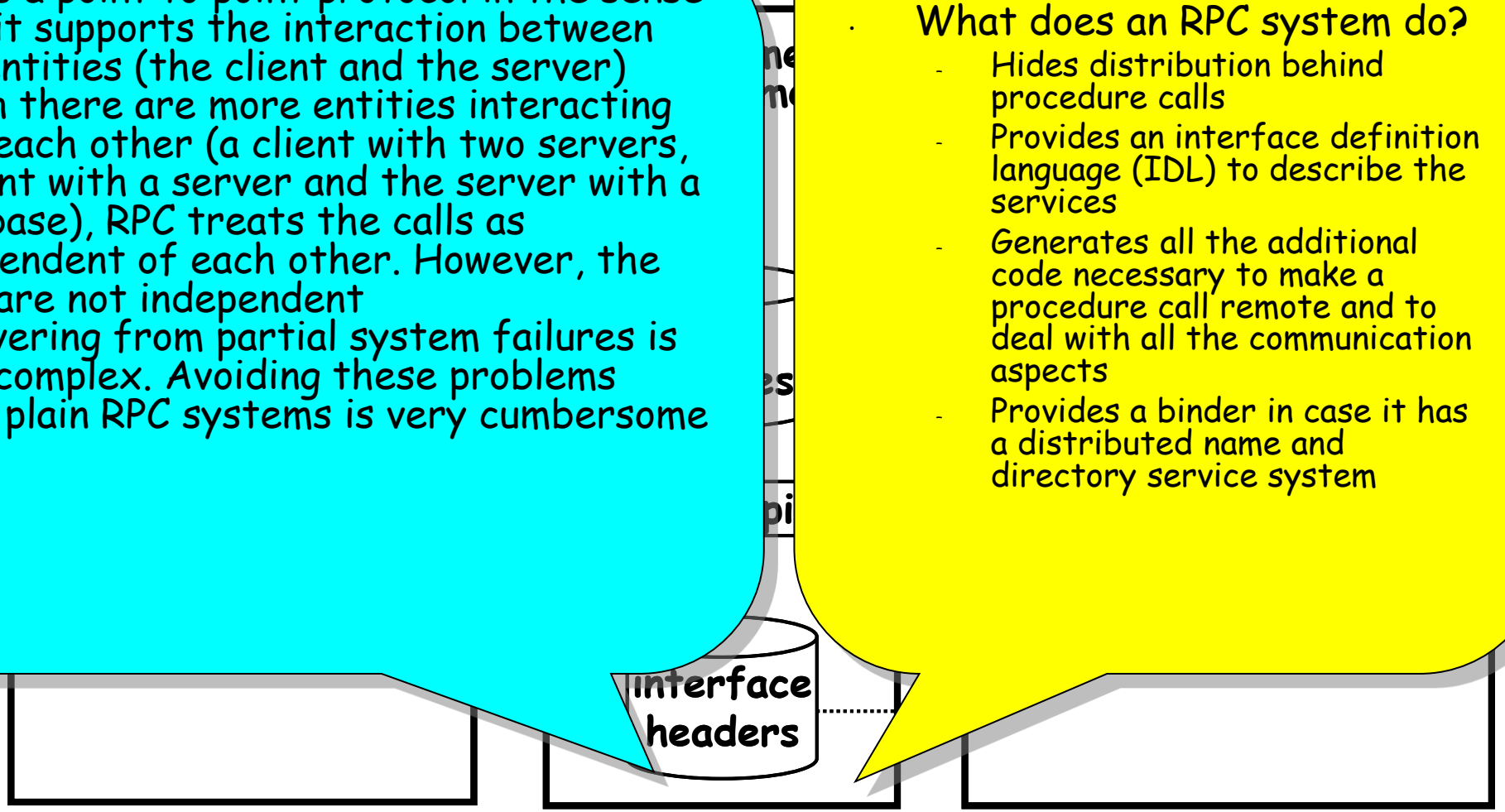
Basic Middleware (3)



Comments

RPC is a point to point protocol in the sense that it supports the interaction between two entities (the client and the server)
When there are more entities interacting with each other (a client with two servers, a client with a server and the server with a database), RPC treats the calls as independent of each other. However, the calls are not independent
Recovering from partial system failures is very complex. Avoiding these problems using plain RPC systems is very cumbersome

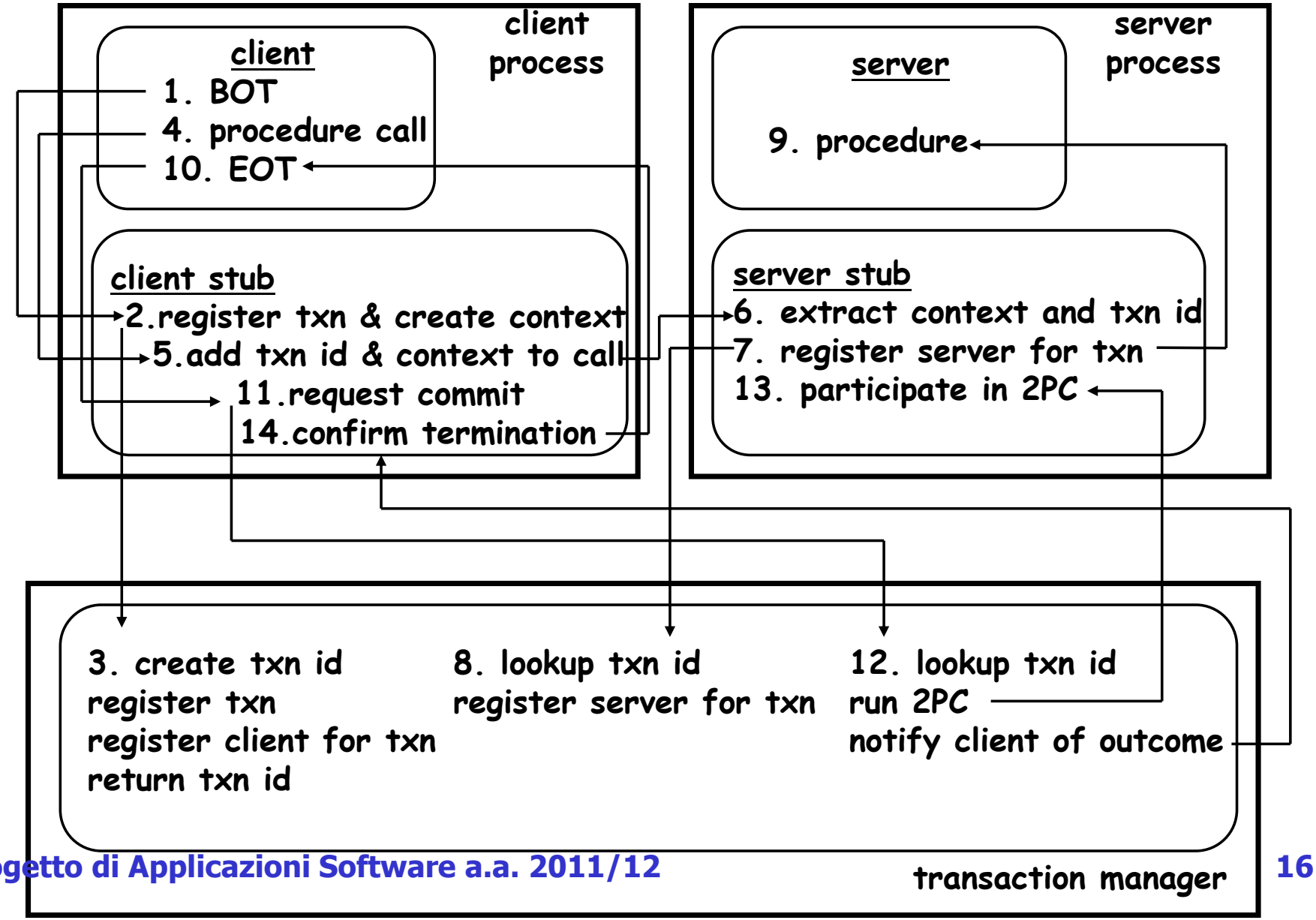
- One cannot expect the programmer to implement a complete infrastructure for every distributed application. Instead, one can use an RPC system (our first example of low level middleware)
- What does an RPC system do?
  - Hides distribution behind procedure calls
  - Provides an interface definition language (IDL) to describe the services
  - Generates all the additional code necessary to make a procedure call remote and to deal with all the communication aspects
  - Provides a binder in case it has a distributed name and directory service system



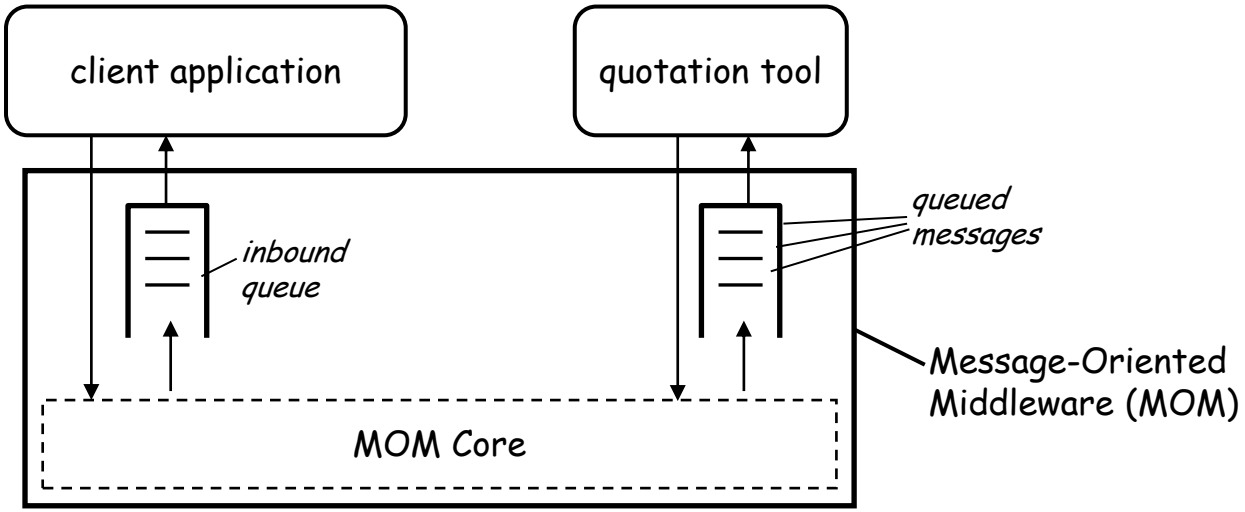
Transactional RPC

- The solution to this limitation is to make RPC calls transactional, that is, instead of providing plain RPC, the system should provide TRPC
- What is TRPC?
  - same concept as RPC plus …
  - additional language constructs and run time support (additional services) to bundle several RPC calls into an atomic unit
  - usually, it also includes an interface to databases for making end-to-end transactions using the XA standard (implementing 2 Phase Commit)
  - and anything else the vendor may find useful (transactional callbacks, high level locking, etc.)

Transactional RPC

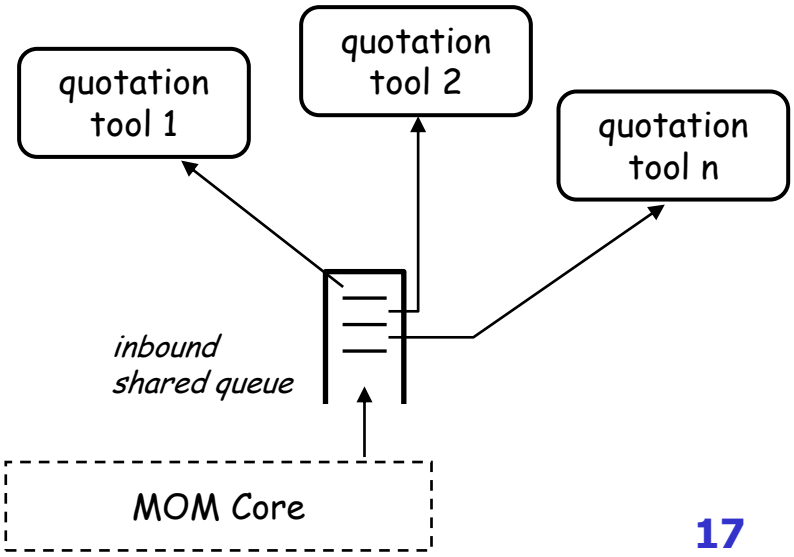


Message Oriented Middleware



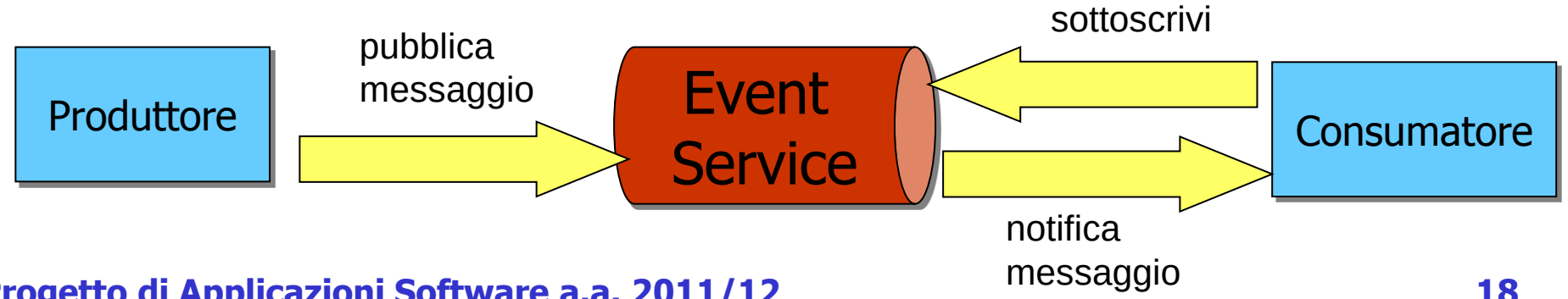
The queuing model

(Primitive) load balancing with shared queues



Publish/Subscribe

- Realizza un'interazione molti-a-molti e anonima
- I partecipanti si suddividono in due gruppi
  - publisher: produttori, inviano messaggi
  - subscriber: consumatori, esprimono interesse per determinate categorie di messaggi
- La comunicazione avviene attraverso un'entità centrale (event service) che riceve i messaggi dai publisher e li incrocia con gli interessi dei subscriber



Publish/Subscribe

- ***Topic-based***: le informazioni vengono suddivise in argomenti (*topic* o *subject*)
  - sottoscrizioni e pubblicazioni per argomento
  - canale logico ideale che connette un publisher a tutti i subscriber
- ***Content-based***: possibile utilizzare dei filtri per una selezione più accurata delle informazioni da ricevere (contenuto dei messaggi)
- La notifica può avvenire secondo due modalità
  - **push**: i sottoscrittori vengono invocati in callback, utilizzando un riferimento comunicato al momento della sottoscrizione
  - **pull**: i sottoscrittori eseguono un polling sull'event service quando hanno bisogno di messaggi

Esempio

```
Message QuoteREquest {
        QuoteRefereneceNumber: Integer
        Customer: String
        Item: String
        Quantity: String
        RequestedDeliveryDate: Timestamp
        DeliveryAddress: String
}
```

Con P&S topic-based un sottoscrittore può registrare il suo interesse a tutti i messaggi di tipo `QuoteRequest` (selezione della "classe")

Con P&S content-based un sottoscrittore può registrare il suo interesse a tutti i messaggi di tipo `QuoteRequest` tali che `RequestedDeliveryDate before June 30th, 2004 AND Quanity > 1000`

Si possono scrivere espressioni che permettono di filtrare tutti i messaggi di un tipo rispetto ai valori degli attributi (selezione dell' "istanza"). Diventa importante il potere espressivo del linguaggio di sottoscrizione (una sorta di query language)

"*E risuona il mio barbarico yawp sopra i tetti del mondo*„
("I sound my barbaric yawp over the roofs of the world„)

[W. Whitman, *Canto di me stesso*, in *Foglie d'Erba*]

RPC IN JAVA

RPC / 1

- Forniscono l'astrazione per cui le chiamate a procedure (cioè funzioni) sono mascherate dietro a chiamate locali
  - I *client* che invocano le procedure "hanno l'impressione" che le chiamate siano eseguite localmente mentre, invece, il corpo delle chiamate è eseguito su un calcolatore remoto
  - Nelle **Remote Procedure Call** (**RPC**), il *server* è il processo che gestisce effettivamente le chiamate sul calcolatore remoto.
- Il servizio remoto ha un'interfaccia ben definita attraverso un *Interface Description Language* (**IDL**).
  - Fornisce una rappresentazione astratta della procedura in termini di parametri di input/output
  - Essendo usato per descrizioni *language-neutral*, si rendono indipendenti le infrastrutture software ed i linguaggi di programmazione adottati su client e server
    - e.g., client Java, server C++

RPC / 2

- Questa astrazione è fornita, lato client, dallo ***stub*** mentre, lato server, si parla di ***skeleton***
  - Lo *stub* e lo *skeleton* vengono automaticamente generati da un software specifico, residente sulle macchine client e server o ad esse destinato, prendendo come input l'IDL.
    - Lo *stub* converte le chiamate locali del client in messaggi inviati su Socket verso lo skeleton
    - Lo *skeleton* riceve ogni chiamata tramite Socket e le converte effettivamente in chiamate locali, cattura l'eventuale output che viene convertito in un sequenza di byte inviata sul Socket verso il client richiedente.

RMI

- Java riprende i concetti di:
  - chiamate a procedure residenti su macchine remote;
  - descrizione di operazioni e segnature condivise fra client e server;
  - stub e skeleton;
- interpretandole secondo il proprio paradigma:
  - metodi remoti;
  - interfacce remote;
  - oggetti remoti.
- **Remote Method Invocation** (**RMI**)
- Attenzione: RMI *non è* language-neutral!
  - RMI è studiato per applicazioni distribuite su **JVM** residenti in macchine diverse.

Servizi di RMI / 1

RMI fornisce alcuni servizi base per la gestione delle applicazioni distribuite:

Naming/Registry service

- Quando un processo server-side esporta un servizio basato su RMI deve registrare uno o più oggetti RMI su un **Registry pubblico**
- Ciascun oggetto è registrato con un **nome logico**, usato dal client per ritrovarlo
  - Il client può invocare il metodo
    public static Remote Naming.lookup(String name)
    della classe
    java.rmi.Naming
    per ottenere uno *handle* ad un oggetto remoto, che implementa
    java.rmi.Remote
- Il riferimento viene convertito dal processo client in un riferimento *stub* che è restituito al chiamante
  - Una volta ricevuto il riferimento il client può iniziare la conversazione con il server

Servizi di RMI / 2

Il **servizio di attivazione** di un oggetto remoto

- Un **server object** è registrato ed in esecuzione all'interno di una JVM attiva ed è sotto il controllo dello RMI Registry
- Se lo RMI Registry viene fermato, gli oggetti remoti vengono disattivati e i riferimenti diventano inutilizzabili
- Il Registry gestisce anche le richieste dei client di riferimenti remoti, restituendo opportuni handle

Servizi di RMI / 3

Garbage Collector distribuito

- Per ogni oggetto remoto RMI, il *reference layer* mantiene la lista di riferimenti remoti registrati dai client per ogni oggetto "attivato"
  - Questo possono essere il risultato di una lookup oppure ottenuti come valori di ritorno di un altro metodo su altri oggetti remoti
- Quando la VM di client si accorge che un oggetto remoto non è più referenziato localmente, invia una notifica al server RMI in modo tale che il server possa eliminare la lista dei riferimenti
- Quando un oggetto non è più referenziato da alcun client né da oggetti locali, viene eliminato attraverso un'operazione di garbage collection
- Inoltre, il Garbage Collector prevede l'uso di un timeout per l'invocazione di un riferimento remoto
  - Quando il timeout scade il riferimento viene eliminato dalla lista e viene data notifica ai client che avevano quel riferimento

Uso di RMI

Definire l'interfaccia remota che definisce i "metodi di business" invocabili remotamente.

- L'interfaccia **java.rmi.Remote** serve ad identificare le interfacce i cui metodi possono essere invocati da VM non locali. Ogni oggetto RMI deve direttamente o indirettamente implementare tale interfaccia
- Solo quei metodi specificati in una interfaccia che estende **Remote** possono essere invocati remotamente.
- Tutti i metodi dell'interfaccia remota devono dichiarare il lancio di **java.rmi.RemoteException**

Compilare l'interfaccia remota

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Echo extends Remote {
        public String repeat(String phrase) throws RemoteException;
}
```

Uso di RMI (cont.)

Creare lo *stub* (per rendere utilizzabile remotamente l'oggetto RMI)

- Nota storica:
  - Tutte le versioni di Java, dalla 1.5 in poi, non richiedono più la realizzazione di file bytecode specifici per gli stub
    - All'uopo, si possono usare direttamente i file .class delle interfacce dichiarate sul server
    - In precedenza, occorreva richiamare un comando shell (RMI Compiler, rmic)
- Si consiglia di creare un **JAR** contenente tutte le interfacce remote del servizio e renderlo disponibile su indirizzi pubblici così da renderlo replicabile sulle macchine dei client

```
user:~/…/remoteInterface$
jar cvf echo.jar -C bin/ . ↵
added manifest
adding: compute.jar(in = 345) (out= 189)(deflated 45%)
adding: it/(in = 0) (out= 0)(stored 0%)
adding: it/uniroma1/(in = 0) (out= 0)(stored 0%)
adding: it/uniroma1/dis/(in = 0) (out= 0)(stored 0%)
adding: it/uniroma1/dis/pas/(in = 0) (out= 0)(stored 0%)
adding: it/uniroma1/dis/pas/rmi/(in = 0) (out= 0)(stored 0%)
adding: it/uniroma1/dis/pas/rmi/server/(in = 0) (out= 0)(stored 0%)
adding: it/uniroma1/dis/pas/rmi/server/Echo.class(in = 255) (out= 188)(deflated 26%)
user:~/…/remoteInterface$_
```

Uso di RMI (cont.)

Definita l'interfaccia remota, il passo successivo è implementarla e compilarla (dopo aver copiato nel percorso relativo il JAR contenente la definizione dell'interfaccia remota).

```
import java.rmi.RemoteException;

public class EchoEngine implements Echo {
        @Override
        public String repeat(String phrase) throws RemoteException {
        return
                "\n\n\t\t\"I sound my barbaric "
                + phrase.trim().toLowerCase()
                + " over the roofs of the world\"\n\n";
         }
}
```

Uso di RMI (cont.)

In ogni momento è possibile avviare il servizio *registry* che associa (*bind*) ad un nome logico un oggetto remoto.

Da shell Unix/Linux con JDK installata, occorre eseguire:
rmiregistry [<*NumeroPorta*>]

- *NumeroPorta* , parametro opzionale, identifica la porta in cui il registry accetta i bind e le lookup (le ricerche dai client).
  - Default: **1099**

Su macchine Windows:
start rmiregistry [<NumeroPorta>]

Il registry durante le lookup e le invocazioni remote deve avere accesso allo stub

- Si deve dunque
  1) scrivere un programma che effettui il bind, registrando l'oggetto RMI,
  2) compilarlo
  3) eseguirlo, una volta avviato il registry

Uso di RMI (cont.)


```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Logger;

public class EchoServerApplication {
public static final String BINDING_NAME = "Echo";

public static void main(String[] args) {
        EchoEngine engine = new EchoEngine();
        try {
                Echo skeleton =
                        (Echo)
                        UnicastRemoteObject.exportObject(engine, 0);
                Registry registry = LocateRegistry.getRegistry();
                registry.rebind(BINDING_NAME, skeleton);
        } catch (RemoteException e) {
                e.printStackTrace();
                System.exit(1);
        }
        Logger.getLogger(BINDING_NAME).info(
                        EchoEngine.class.getName() +
                        " successfully bound to " +
                        BINDING_NAME);
}
}
```

RMI Name Registry

- Le operazioni di ricerca e registrazione accettano come parametro un URL il cui formato è:

  rmi://host:port/name

  - **rmi://** **è il nome del protocollo ed è opzionale**
  - **host:** **è il nome del server RMI (default *localhost*)**
  - **port:** **è la porta su cui sta in ascolto il registry (default 1099)**
  - **name:** **il nome logico (obbligatorio)**
- Ad esempio un URL del tipo Echo è equivalente a rmi://localhost:1099/Echo

Il client

```
import it.uniroma1.dis.asos.rmi.server.Echo;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class VoiceForTheEcho {

public static void main(String[] args) {
        if (args.length < 1) {
                System.err.println("I needed something to hear...");
                System.exit(1);
        }

        String name = "Echo";
        try {
                Registry registry = LocateRegistry.getRegistry();
                Echo echoEngine = (Echo) registry.lookup(name);
                System.out.println(echoEngine.repeat(args[0]));
        } catch (Exception e) {
                System.err.println(
                                "Sadly, your voice did not flow beyond the walls");
                e.printStackTrace();
        }
}
}
```

Serializzazione

- Il meccanismo base utilizzato da RMI per la trasmissione dei dati fra client e server è quello della **serializzazione** (**serialization**)
- L'obiettivo principale della serializzazione è permettere la codifica di oggetti e strutture in sequenze di byte
  - Tali sequenze poi vengono trasferite all'opposto end-point dove vengono "deserializzate" per ricostruire una copia dell'oggetto remoto
  - È necessario che entrambi gli end-point abbiano la stessa definizione della classe
    - Entrambi, cioè, abbiano il `.class` relativo
- Gli oggetti di una certa classe Java sono considerati *serializzabili* se la classe implementa l'interfaccia **java.io.Serializable** ed eventuali oggetti incapsulati sono a loro volta *serializzabili*
  - L'interfaccia Serializable è puramente dichiarativa, ovverosia non definisce alcun metodo

Ulteriori informazioni

- Un'utile guida introduttiva
  `http://download.oracle.com/javase/tutorial/rmi/overview.html`
- Documentazione:
  `http://download.oracle.com/javase/6/docs/api/java/rmi/package-summary.html`

Dipartimento di Informatica
e Sistemistica Antonio Ruberti

Sapienza
Università di Roma

MOM in Java

Java Message Service - JMS

"*Magnifico ambasciatore. «Tarde non furon mai grazie divine». Dico questo perché mi pareva di aver non perduto, ma smarrito la grazia Vostra, essendo stato Voi molto tempo senza scrivermi; ed ero nel dubbio chiedendomi donde potesse nascerne la causa.*„

[N. Machiavelli, *Lettera a Francesco Vettori in Roma*]

Java Message Service

- Java message service **(JMS)** è l'insieme di API che consentono lo scambio di messaggi tra applicazioni Java distribuite sulla rete
- La prima specifica JMS è stata rilasciata nel 1998, mentre l'ultima versione delle API è la 1.1, pubblicata nel 2002

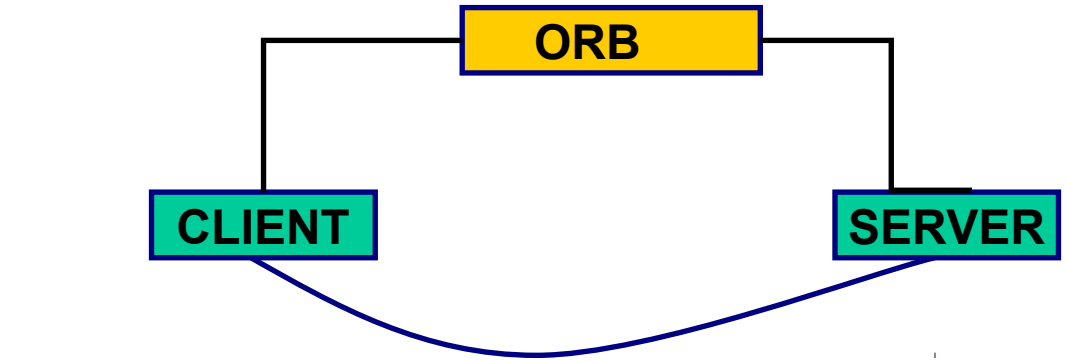
Messaggio

- In ambito JMS, un messaggio è un raggruppamento di dati che viene inviato da un sistema ad un altro
  - I dati sono solitamente utilizzati per notificare eventi ed informazioni
  - Sono pensati per essere utilizzati da software

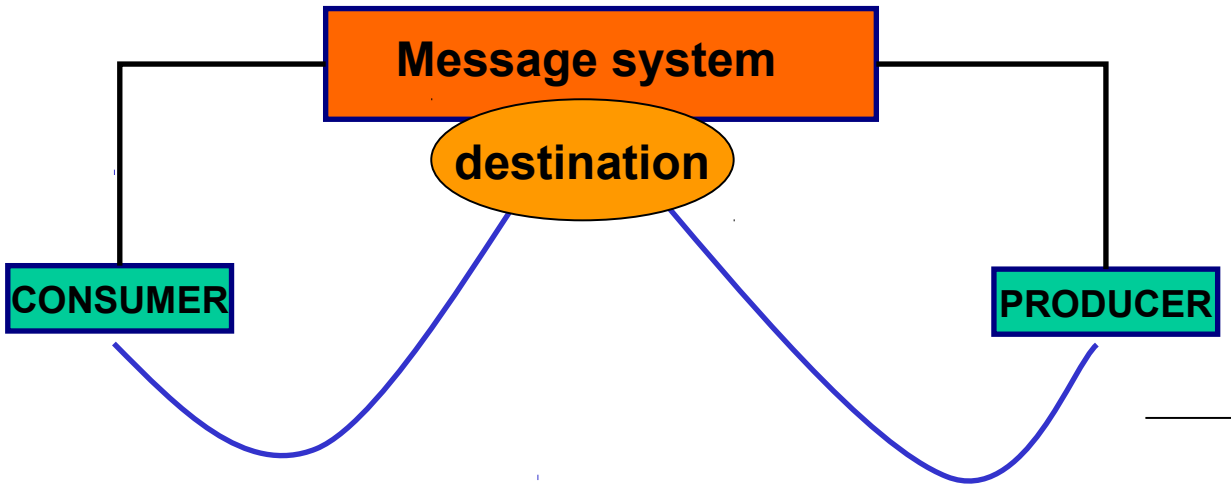
Messaging System

- Con il termine ***messaging*** ci si riferisce ad un meccanismo che consente la comunicazione **asincrona** tra client remoti:
  - Un ***producer*** invia un messaggio ad uno o più *consumer*
  - Il ***consumer*** riceve il messaggio ed esegue le operazioni correlate, **in un secondo momento**
- *Producer* e *consumer* sono definiti entrambi ***JMS client***.
  - Occorre non confondersi con il paradigma client/server
- I *JMS client* sfruttano l'infrastruttura fornita dal ***JMS Provider***

Messaging System (2)



Modello ORB (RMI)



MODELLO MESSAGE ORIENTED

JMS non include…

- Un sistema di Load balancing/Fault Tolerance: la API JMS non specifica un modo in cui diversi client possano cooperare al fine di implementare un unico servizio critico
- Notifica degli Errori
- JMS non prevede API per il controllo della privacy e dell'integrità del messaggio

Architettura JMS

- **JMS Client**: programma che manda o riceve i messaggi JMS
- **Messaggio**: ogni applicazione definisce un insieme di messaggi da utilizzare nello scambio di informazioni tra due o più client
- **JMS provider**: sistema di messaggistica che implementa JMS e realizza delle funzionalità aggiuntive per l'amministrazione e il controllo della comunicazione attraverso messaggi
- **Administered objects**: sono oggetti JMS preconfiguarti, creati da un amministratore ad uso dei client. Incapsulano la logica specifica del JMS provider nascondendola ai client, garantendo maggiore portabilità al sistema complessivo.

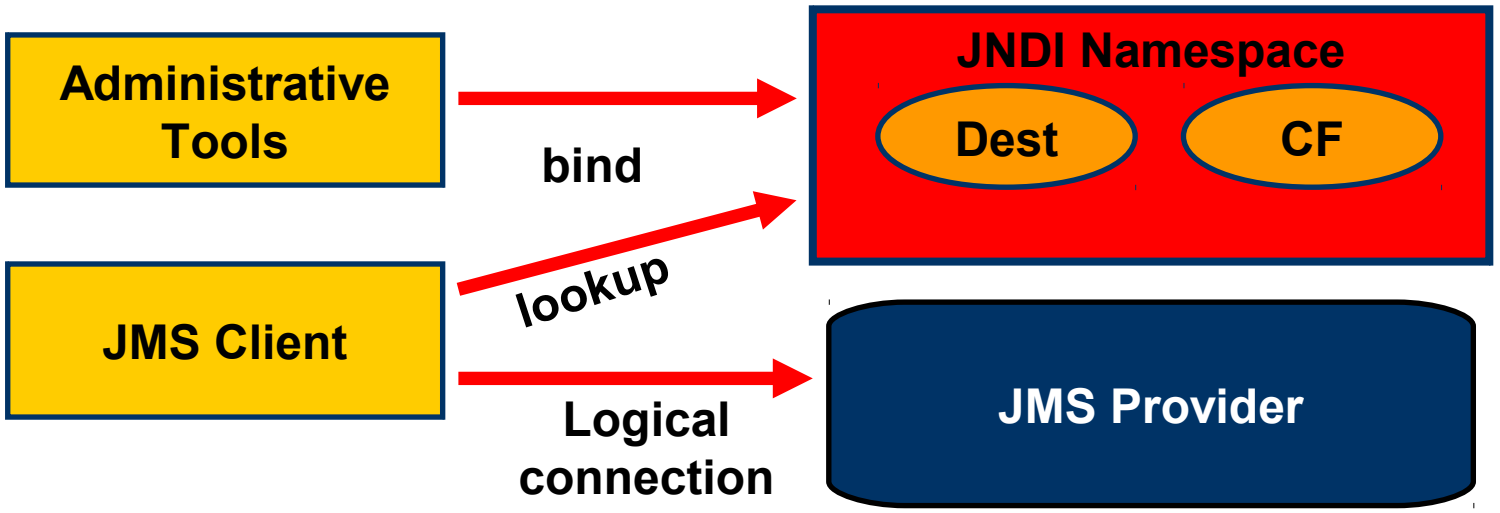
Administered Objects

- **Connection Factory:** oggetto utilizzato da un client per realizzare la connessione con il provider
- **Destination (queue/topic):** oggetto utilizzato da un client per specificare la destinazione dove il messaggio è spedito/ricevuto.

Administered Objects (2)

- Le Connection Factory e le Destination sono legati a *nomi simbolici*
- I servizi **JNDI** (***Java Naming and Directory Service***) sono tipicamente resi disponibili dall'Application Server per ottenere gli oggetti remoti registrati.
  - I nomi degli oggetti sono completamente indipendenti dal luogo fisico in cui gli stessi sono resi disponibili.
  - Il "Naming" degli oggetti in JMS è gestito dall'amministratore di sistema e *non* dai client (come in RMI)
- Una applicazione che usa JMS adopera JNDI per ottenere i riferimenti a tali oggetti.
  - Per esempio, un sender o un receiver.

Bind e lookup



- Attraverso gli *Administrative Tools*, gli amministratori responsabili del dispiegamento dei componenti effettuano il deploy ed associano (***bind***) un nome JNDI agli oggetti *Destination* e *Connection Factory*
- I client JMS ricercano gli Administered Objects (***lookup***) ed instaurano una connessione logica ad un JMS Provider

Modelli di messaggistica (message domain)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- **Point-to-point (PTP):**
  - Il *producer* si definisce ***sender***
  - Il *consumer* si definisce ***receiver***
  - Più JMS Client possono condividere la stessa *Destination* (**queue**)
  - Un messaggio può essere consumato da **un solo receiver**
  - Consente una comunicazione **uno-a-uno**
- **Publish/Subscribe (Pub/Sub):**
  - Il *producer* si definisce **publisher**
  - I *consumer* si definiscono **subscriber**
  - Più JMS Client possono condividere la stessa *Destination* (**topic**)
  - Il messaggio pubblicato viene ricevuto da **tutti i subscriber** che si siano dichiarati **interessati**
  - Consente una comunicazione **uno-a-molti**

Point-to-point domain

- Ogni JMS Client spedisce e riceve messaggi mediante canali virtuali conosciuti come **queue** (coda)
- È un modello di tipo "*pull-based*"
  - spetta ai receiver prelevare i messaggi dalle code

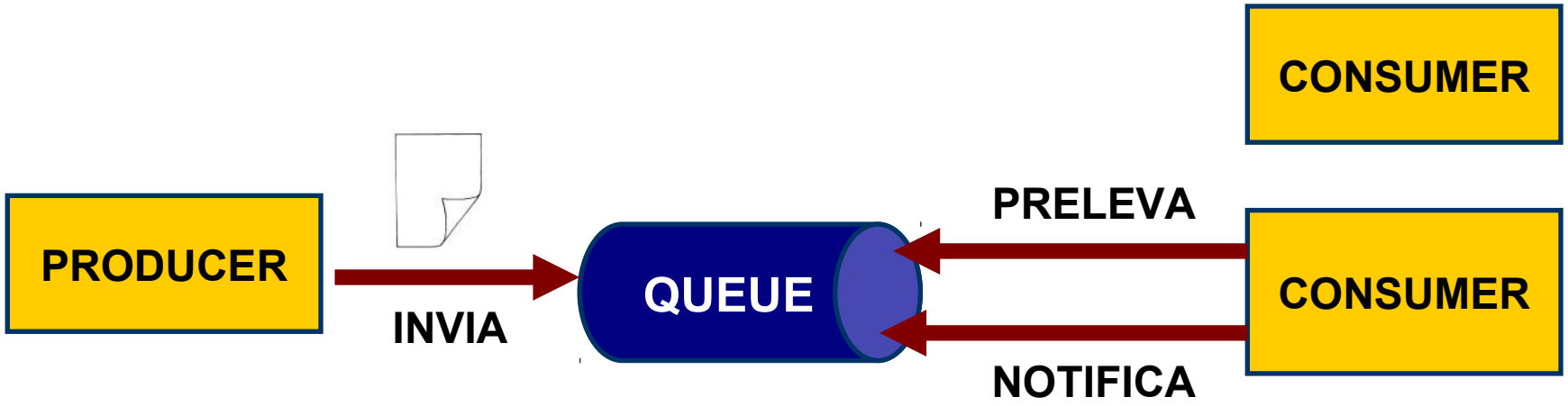
Point-to-point domain (2)

Dipartimento di Informatica
e Sistemistica Antonio Ruberti
Sapienza
Università di Roma

- Più producer e più consumer possono condividere la stessa coda, ma…
- Ogni messaggio può essere letto da un solo consumer
- Sender e receiver non hanno alcuna dipendenza temporale rispetto ai messaggi
- Il receiver notifica l'avvenuta ricezione e processamento del messaggio (**acknowledgement**)
- Il PTP si rivela utile quando è necessario garantire che un messaggio arrivi ad un solo destinatario che notifichi la corretta ricezione

Point-to-point domain (3)

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA



Publish/Subscribe domain

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Ogni producer spedisce messaggi a molti consumer mediante canali virtuali conosciuti come **topic** (argomento)
- È un modello di tipo "*push-based*"
  - i messaggi vengono automaticamente inviati in **broadcast** ai consumer
- Per ricevere i messaggi, i consumer devono *sottoscriversi* ad un topic
- Qualsiasi messaggio spedito al topic viene consegnato **a tutti** i consumer sottoscritti, ciascuno dei quali riceverà una copia identica di ciascun messaggio inviato al topic

Publish/Subscribe domain (2)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Il publisher ed i subscriber hanno una dipendenza temporale:
  - Un consumer che si sottoscrive ad un topic può consumare solamente messaggi pubblicati *dopo* la sua sottoscrizione.
  - Il subscriber può continuare a consumare messaggi solo nel periodo in cui rimane attivo.
    - Durante il periodo di inattività i messaggi che dovrebbe ricevere andrebbero persi.
- Per ovviare al problema il client JMS può dichiararsi **Durable Subscriber**
  - Questo gli consente di disconnettersi e riconnettersi in un secondo momento, ricevendo tutti i messaggi pubblicati durante il periodo di disconnessione

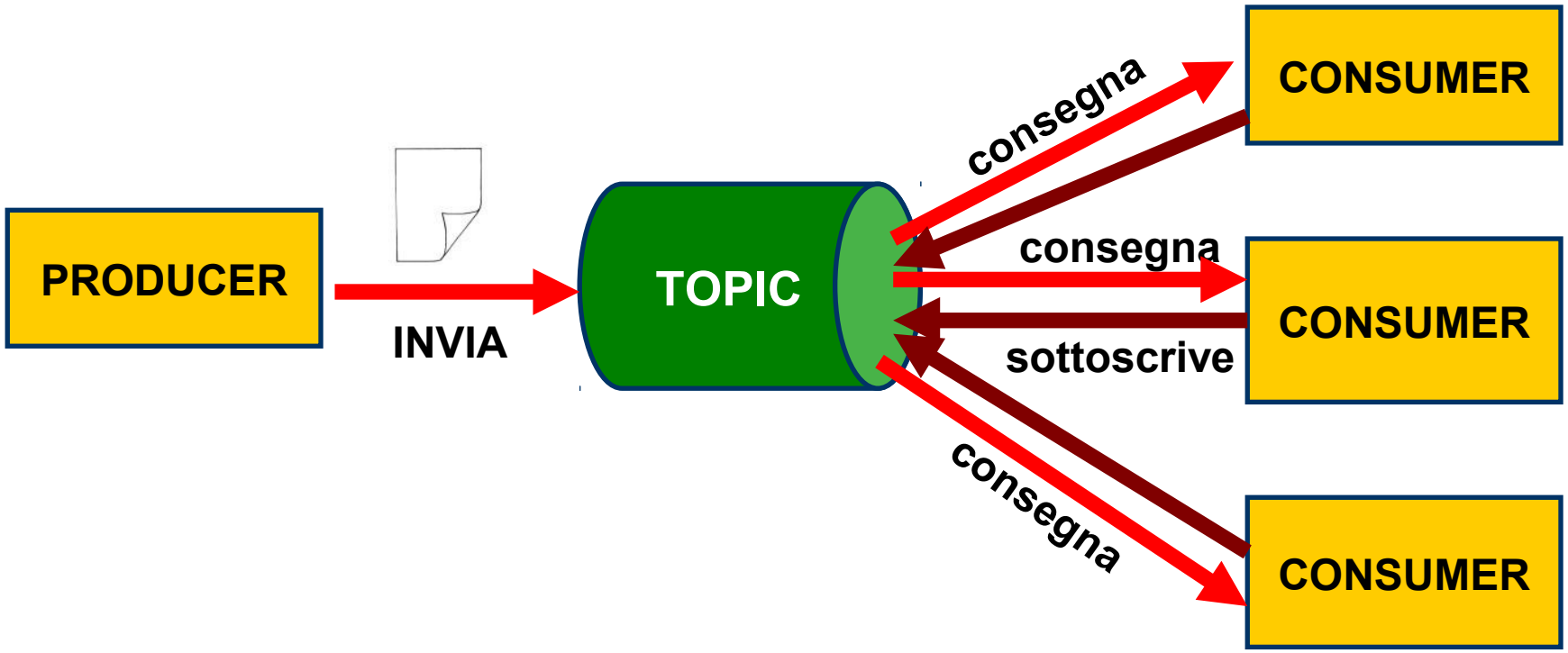
Publish/Subscribe domain (3)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Il modello **pub/sub** prevede due tipi di **sottoscrizione**:
  - **Topic-based**:
    - le informazioni vengono suddivise in argomenti (*topic* o *subject*)
    - le sottoscrizioni e le pubblicazioni vengono effettuate scegliendo come discriminante un argomento (letteralmente, *topic*)
  - **Content-based**:
    - utilizza dei **filtri** (*message selector*) per una selezione più accurata delle informazioni da ricevere sul topic

Publish/Subscribe domain (4)

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA



Modalità di ricezione

- Un messaggio JMS può essere consumato secondo due modalità:
  - **Modalità sincrona:** il subscriber (o il receiver) prelevano direttamente il messaggio dalla coda, tramite l'invocazione del metodo **`receive()`.**
    - Il client rimane bloccato finché non arriva il messaggio o fino allo scadere di un timeout
  - **Modalità asincrona:** il client registra un *message listener* su una destination.
    - Ogni qual volta un messaggio è pronto per essere consegnato ad un client, il JMS Provider lo consegna ed il listener associato viene invocato.

Connection Factory

- È l'interfaccia utilizzata dal JMS Client per creare connessioni con il JMS Provider onde accedere al servizio JMS.
  - javax.jms.ConnectionFactory
- ConnectionFactory è estesa delle interfacce:
  - **QueueConnectionFactory**
  - **TopicConnectionFactory**

Connection Factory (2)

- Nel caso di PTP Domain si utilizza l'interfaccia javax.jms.QueueConnectionFactory

```
Context ctx = new InitialContext();
QueueConnectionFactory queueConnectionFactory =
(QueueConnectionFactory)ctx.lookup("ConnectionFactory");
```

- Context è l'interfaccia base per la specifica di un naming context di JNDI.
- Si crea un oggetto di tipo context
- Si recupera il riferimento logico all'oggetto con il metodo lookup

Connection Factory (3)

- Nel caso di dominio publish/subscribe si utilizza l'interfaccia TopicConnectionFactory

```
Context ctx = new InitialContext();
TopicConnectionFactory topicConnectionFactory =
(TopicConnectionFactory)ctx.lookup("ConnectionFactory");
```

Connection

- Dopo aver effettuato il *lookup* per recuperare lo handle ad un'implementazione di Connection Factory, è possibile creare un oggetto di tipo **javax.jms.Connection**
- PTP Domain:
  - Si ottiene lo handle all'interfaccia **QueueConnection** invocando il metodo **createQueueConnection()** sull'oggetto **QueueConnectionFactory**:
- Pub/Sub Domain:
  - Si ottiene lo handle all'interfaccia **TopicConnection** invocando il metodo **createTopicConnection()** sull'oggetto **TopicConnectionFactory.**

```
QueueConnection queueConnection =
queueConnectionFactory.createQueueConnection();
```

```
TopicConnection topicConnection =
topicConnectionFactory.createTopicConnection();
```

Session

- Un oggetto che implementi l'interfaccia javax.jms.Session può essere creato a partire da istanze Connection
- Permette l'istanziazione di
  - producer,
  - consumer,
  - messaggi.
- L'interfaccia è estesa, rispettivamente per la gestione di topic e queue, da
  - javax.jms.TopicSession
  - javax.jms.QueueSession

Destination

- È l'interfaccia verso gli *Administered Object* che astraggono *queue* e *topic*
  - **javax.jms.Destination**
- Nel PTP Domain, la destination è specializzata dall'interfaccia **javax.jms.Queue**
- Nel Pub/Sub Domain, la Destination è specializzata dall'interfaccia **javax.jms.Topic**
- Anche in questo caso il client identifica la destinazione mediante l'utilizzo delle API JNDI

Lookup delle Destination

Dipartimento di Informatica
e Sistemistica Antonio Ruberti
Sapienza
Università di Roma

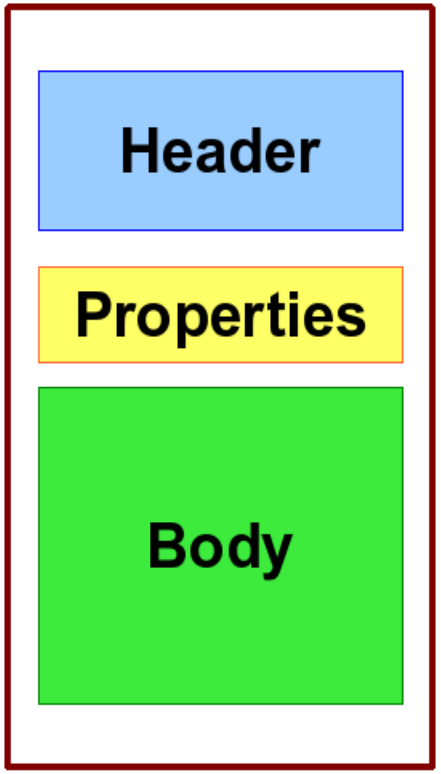
- Un consumer deve ottenere il riferimento alla destination invocando il metodo lookup
- L'input è il nome della destination precedentemente dispiegata
  - Nell'assegnazione dello handle, occorre non dimenticare il cast al tipo opportuno (javax.jms.Queue o javax.jms.Topic)

```
Queue queue = (Queue)
ctx.lookup("/queue/myQueue");

Topic topic = (Topic)
ctx.lookup("/topic/myTopic");
```

Message

- Cositutito da:
- **Header (obbligatorio):** contiene informazioni sul messaggio
  - e.g., il **JMSCorrelationID**, adoperato per connettere due messaggi in relazione l'uno con l'altro
- **Property (opzionale):** contiene alcune proprietà opzionali del messaggio solitamente utilizzate per gestire la compatibilità tra sistemi di messaggistica differenti.
  - I suoi campi sono esaminati dai un consumer mediante i **message selector**
- **Body (opzionale):** contiene la parte informativa trasferita all'interno del messaggio



Message Body

- Esistono vari tipi di messaggio ma qui ci limiteremo ai messaggi di testo
  - Un oggetto **TextMessage** viene creato per mezzo dei metodi **createTextMessage()** forniti dall'interfaccia **Session**
- Nel caso **PTP**, per creare un messaggio di tipo **TextMessage** utilizziamo il seguente codice:

```
TextMessage txtMsg = queueSession.createTextMessage();
txtMsg.setText("Ciao!");
producer.send(txtMsg);
[…]
```

Si invoca il metodo createTextMessage() su un oggetto `queueSession` specifico per il dominio PTP

Message Property

- I campi property possono essere impostati e letti mediante i metodi:
  - set<*Type*>Property(String name, <*Type*> value)
  - <*Type*> get<*Type*>Property(String name)

    dove <*Type*> è il tipo di property
    - Boolean, Byte, Integer, String, Object...
- Il seguente esempio imposta il valore di due property

```
TextMessage txtMsg = queueSession.createTextMessage();
txtMsg.setText("Ciao!");
txtMsg.setStringProperty("Sender", "Massimiliano");
txtMsg.setIntegerProperty("Priority", 2);
```

Message Producer

- Il **`message producer`** è l'oggetto che ha il compito di inviare messaggi ad una destination
- Implementa l'interfaccia
  javax.jms.MessageProducer
  ed è generato da un oggetto
  javax.jms.Session
  attraverso il metodo
  MessageProducer **createProducer**(
  Destination destination
)
- Come d'uopo, si hanno le specializzazioni per queue e topic
  - **TopicPublisher createPublisher**(
    Topic topic
)
    in
    javax.jms.**TopicSession**
  - **QueueSender createSender**(
    Queue queue
)
    in
    javax.jms.**QueueSession**

Message Consumer

- Il **`message consumer`** è l'oggetto che ha la possibilità di trarre messaggi da una destination
- Implementa l'interfaccia
  javax.jms.MessageConsumer
  ed è generato da un oggetto
  javax.jms.Session
  attraverso il metodo
  MessageConsumer **createConsumer**(
  Destination destination[, String messageSelector]
)
- Come d'uopo, si hanno le specializzazioni per queue e topic
  - **TopicSubscriber createSubscriber**(
    Topic topic[, String messageSelector, boolean noLocal]
)
    in
    javax.jms.**TopicSession**
  - **QueueReceiver createReceiver**(
    Queue queue[, String messageSelector]
)
    in
    javax.jms.**QueueSession**

Modalità Asincrona

- È la modalità standard di trasmissione di messaggi in JMS
- Il producer non è tenuto ad attendere che il messaggio sia ricevuto per continuare il suo funzionamento
- Per supportare il consumo asincrono dei messaggi, deve essere utilizzato un oggetto *listener* che implementi l'interfaccia

  javax.jms.MessageListener
- Tramite overriding del metodo

  void onMessage(Message message)

  si possono definire le operazioni da effettuare all'arrivo di un messaggio.

Message Listener

- Passi:
  - Generare l'oggetto listener

```
TopicListener topicListener = new TextListener();
QueueListener queueListener = new TextListener();
```

  - Passare l'oggetto listener creato come input al metodo
    void setMessageListener(
    MessageListener listener
)
    degli oggetti **MessageConsumer**

Message Listener

- Il listener, dopo essere stato connesso **si mette in ascolto** di un messaggio, avviando la connessione

```
queueConnection.start();
topicConnection.start();
```

- All'arrivo di un messaggio viene invocato il metodo **onMessage()**

```
public class TextListener implements MessageListener
{
        public void onMessage(Message message)
        {
                // Codice per consumare il messaggio
        }
}
```

Modalità Sincrona

- I prodotti di messaging sono intrinsecamente asincroni ma un message consumer può ricevere i messaggi anche in modo sincrono
- Il consumer **richiede esplicitamente** alla destinazione di prelevare il messaggio (*fetch*) invocando il metodo

  Message receive([long timeout])

  offerto dall'interfaccia

  javax.jms.MessageConsumer

  – È sospensivo, ovverosia rimane bloccato fino alla ricezione del messaggio, a meno che non si espliciti un **timeout**, scaduto il quale il metodo termina

Modalità Sincrona (2)

```
while(<condition>) {
  Message m = queueReceiver.receive();
  if( (m != null) && (m instanceof TextMessage) ) {
    message = (TextMessage) m;
    System.out.println("Rx: " + message.getText());
  }
}
```

Indipendentemente dalla modalità di ricezione, la comunicazione viene conclusa invocando un'operazione di close() sulla connessione

```
queueConnection.close();
topicConnection.close();
```

Message Selector

- I message selector sono parametri utilizzati per filtrare messaggi in arrivo, permettendo ad un consumer di specificare quali siano quelli di suo interesse, sulla base delle informazioni contenute all'interno del messaggio
  - La selezione avviene ***solo*** a livello di ***header*** **e** ***property***
    non di ***body***
- Assegnano il lavoro di filtraggio al JMS Provider anziché all'applicazione
  - Il filtraggio avviene a livello di server e permette di inoltrare ai client lungo la rete i messaggi strettamente necessari o utili, risparmiando così la banda del canale
- Un message selector è un oggetto di tipo String che contiene un'espressione
  - la sintassi appartiene ad un sottoinsieme dello standard SQL92
- Il metodo **createConsumer()** ammette opzionalmente di specificare un message selector

  MessageConsumer createConsumer(
  Destination destination[, String **messageSelector**]
)

Esempio

```
ctx = new InitialContext(properties);
cf = (TopicConnectionFactory)ctx.lookup("ConnectionFactory");
destination = (Topic)ctx.lookup("topic/allarme");
connection = cf.createTopicConnection();
session = connection.createTopicSession(
        false,  Session.AUTO_ACKNOWLEDGE
);

selector = "sender = 'Massimo' AND priority > 0";
subscriber =
session.createSubscriber(destination,selector,false);

subscriber.setMessageListener (new TextListener());
connection.start();
```