Communication Support for Distributed Systems

Alessio Bechini Dept. of Information Engineering, Univ. of Pisa

a.bechini@ing.unipi.it

Outline

Communication Frameworks to Support Distributed Applications

- Direct vs Indirect
- Request/reply
- RPC
- RMI
- Garbage Collection, DGC
- Web Services
- Towards JEE Architecture

Direct vs Indirect Communication

Abstracting Communication

Support to inter-process communication has to be based on ***more abstract*** ways to shape communication actions.

- **Direct communication** - each of the involved parties directly refers to the counterpart
- **Indirect communication** - fully asynchronous relationship, by leveraging an *intermediate means* that "holds" messages (it will be discussed later)

Direct Communication

We address increasingly abstract ways to provide direct communication:

- **Request/Reply Protocols** - designed to support message exchange in client-server systems
- **Remote Procedure Call (RPC)** - provides remote execution of code, on the basis of explicit requests
- **Remote Method Invocation (RMI)** - As RPC, in OO perspective

Le RMI sono una tecnologia specifica di Java che consente a un oggetto Java in un processo di invocare metodi su un oggetto Java in un processo remoto. Le RMI estendono il concetto di RPC per il mondo Java, consentendo agli oggetti distribuiti di comunicare tra loro.

Request/Reply Protocols

Generalities

Two roles: client (asks), server (replies)

The protocols are **typically synchronous**: the client sends a REQ, and stops waiting for a REPLY (the reply plays also the ACK role).

What about the System Model?

Usually, UDP-like, + no restriction on datagram length.

In practice, other transport supports may be used.

General Primitives (I)

A general form of the protocol can be shaped using three primitives:

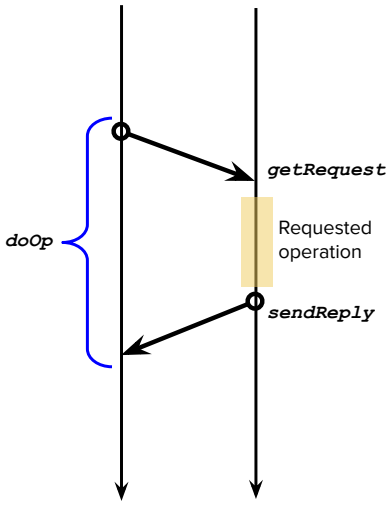
1. `public byte[] doOp(RemoteRef s, int opId, byte[] args)` used on client
   Sends a request message to the remote server identified by `s`,
   to request the specific operation `opId`
   with arguments *marshalled* in `args`, and returns the reply.
   It is implemented by:
   a. send of a message, and
   b. blocking receive to get the reply message.

Marshalling := convertire i parametri della funzione in una sequenza di byte che possono essere trasmessi attraverso una rete.

La funzione sarà composta da due parti principali: nella prima parte viene inviato il messaggio al server, mentre nella seconda parte vi sarà una blocking receive per attendere il risultato.

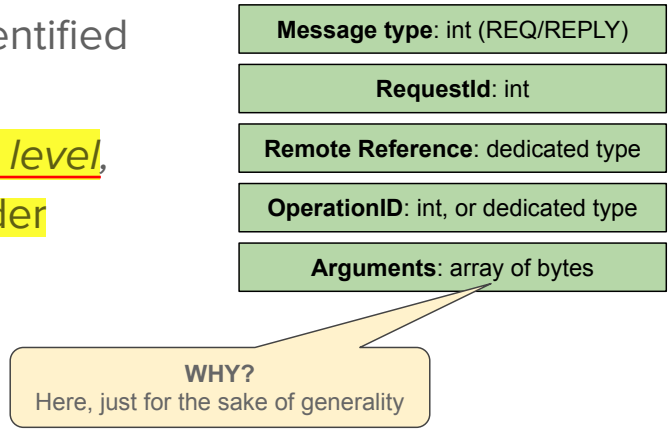
General Primitives (II)

2. `public byte[] getRequest()` used on server
   Blocking operation; gets the request message
3. `public void sendReply (RemoteRef c, byte[] reply)`
   Sends the reply message to the client at `c`,
   possibly with the result
   of the requested operation



Structure of a Message

Any message is uniquely identified by a `requestId`, *assigned at the middleware level*, plus the address of the sender (IPaddr+port)



What's "Marshalling"?

The exchanged data in the context of a REQ-REPLY communication could be highly structured (e.g., objects, along with their state).

The two parties must agree on *how to encode the exchanged data*, so to correctly reconstruct all data items.

TAKE CARE!
Possible different languages for the parties!

The transformation of the memory representation of an object to a data format suitable for transmission is said **marshalling** (and un-marshalling the opposite operation).

pensa, ad esempio, alle funzioni hton() e ntoh() viste a Reti Informatiche...

Failure Model

Regarding the transmission medium, along with UDP:

- possible message loss
- possible message reordering

Regarding processes: usually, fail-stop (no Byzantine…)

To check whether the server is down,
a **timeout** is employed in the implementation of `doOp`

How to Overcome Adversities

avversità

Timeout, to check *possible server failures* (as said)
Request retransmission, to compensate
for *possible loss of req messages*
Duplicate filtering: 1+ copies of the same message
may arrive, so duplicates must be discarded

- **Re-execution vs caching** on the server side;
caching is possible only for **idempotent** requests

We have a requestID…

Similar idea to memo-ization in a sequential setting

BTW: Idempotency What?

- We refer to **idempotency** of a function f **w.r.t. sequential composition**, i.e. two subsequent calls of f: $f(\cdot);f(\cdot)$
- Formally, idempotency of f means that,

  if f is called subsequently twice with the same arguments, the second call will have no side effects and will return the same output as the first call.

Questo non significa necessariamente che la funzione è pura, perché una funzione può avere effetti collaterali e ancora essere idempotente.

- Of course, this is a significant property for functions *with* side effects:

  pure functions are idempotent by nature!

Una funzione si dice "pura" se soddisfa due condizioni principali:
- Referential Transparency: Per ogni insieme di argomenti, restituisce sempre lo stesso risultato. Ciò significa che l'output della funzione dipende solo dai suoi input e non da uno stato esterno.
- Assenza di Effetti Collaterali: Non modifica lo stato esterno del programma. Cioé NON modifica variabili globali, variaibli esterne o parametri passati per riferimento.

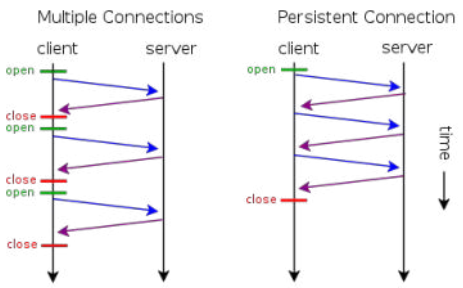
Safe := it doesn't change the internal state of the system
Idempotent := doesn't change the response, so if I call a function of this type multiple times I will always get the same result

safe + idempotent = pure

Example of Req-Reply protocol: HTTP

Mapping of communications onto underlying NW protocols:
Use of a single TCP connection per request/response, or (v. 1.1) *persistent connections*, aka **HTTP keep-alive**, possibly closed by a peer upon some kind of timeout.

"NW protocols" si riferisce ai "Network Protocols" (protocolli di rete). I protocolli di rete sono insiemi di regole e convenzioni che definiscono come i dati devono essere trasmessi, ricevuti e gestiti su una rete di computer.



Addressing: using URI/URL, that account also for the requested operation, and parameters as well

```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

Tutti gli URL sono URI, ma non tutti gli URI sono URL. Gli URL sono una sottoclasse di URI che forniscono anche il mezzo per recuperare la risorsa indicata. In altre parole, un URL non solo identifica una risorsa, ma ne fornisce anche la posizione.

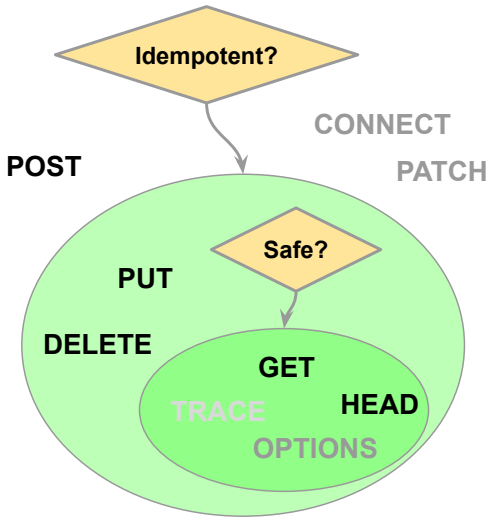
Usando gli URL posso specificare in una singola riga, l'indirizzo, il servizio ed i parametri ("tre piccioni con una fava")

HTTP “Methods”

Methods: types of messages, intended for semantically different classes of operations. GET, POST, etc.

Idempotence: the side-effects of N > 0 identical requests are the same as for a single request ➔ caching of results is *possible*!

Nullipotence (or “safety”): msgs determine retrieval, no state change on the server

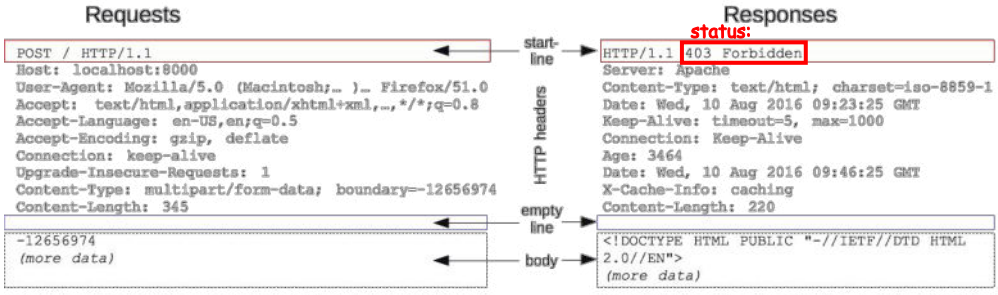


- **GET**: retrieves a representation of the state of the resource (safe and idempotent).
- **POST**: requests that the representation enclosed be processed (neither safe nor idempotent). It usually results in a new resource being created or the target resource being updated. May be used for configuration purposes.
- **DELETE**: requests the resource to be deleted (not safe but idempotent)
- **PUT**: requests the update of the status of an existing resource with the enclosed representation (not safe but idempotent). Is generally associated with an actuator (get associated with sensor).

Safe := it doesn’t change the internal state of the system
Idempotent := doesn’t change the response, so if I call a function of this type multiple times I will always get the same result

Structures of HTTP Messages

The structure is different for REQ and RESPONSE messages (the latter have STATUS, to account for possible errors!



From developer.mozilla.org

The same fields have possible different meanings (or can be used differently), depending on the type/method

Abbiamo detto che il "keep-alive" è una modalità che ci permette di migliorare le performance, vediamo ora un altro trick sempre per migliorare le prestazioni: HTTP pipelining.

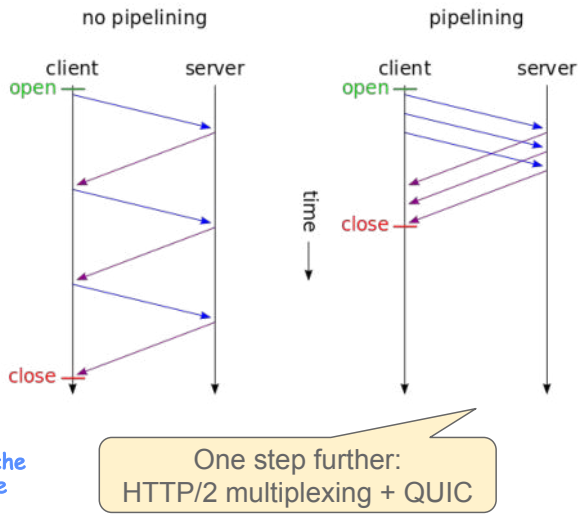
Addressing Performance in HTTP

A method to improve the performance of HTTP communication, beyond keepalive:

HTTP pipelining, i.e. sending requests one after the other without waiting for each reply before issuing the next request.

Of course, this cannot be done in case of non-idempotent requests.

questo perché in caso di richieste non idempotenti devo prima assicurarmi che la modifica sia stata effettuata prima di procedere con successive richieste (altrimenti potrei ottenere un risultato non aggiornato)



One step further:
HTTP/2 multiplexing + QUIC

Marshalling? No, Content Specification!

Marshalling is not used in HTTP; instead, a different trick is employed because everything is specified as a sequence of bytes, and it is explicitly stated what type of content is present in the message. This is done by specifying the so-called MIME type.

Approach to the identification of type of data contents:

data is prefixed by **MIME specification**, La specifica MIME (Multipurpose Internet Mail Extensions) definisce gli standard per la rappresentazione e lo scambio di informazioni multimediali su Internet.

so client and server may agree on how to use it.

MIME is a two-part identifier for file formats and format contents transmitted on the Internet: type/subtype

Examples:

`text/plain` `text/html` `text/css` `image/jpeg` `image/gif` `application/javascript` `application/json` `application/zip` `application/pdf` `application/sql` `application/vnd.ms-excel` (.xls) `application/vnd.ms-powerpoint` (.ppt)

RPC - Remote Procedure Call

Il Remote Procedure Call (RPC) è un protocollo di comunicazione che consente a un programma di richiamare (invocare) una procedura (funzione o metodo) in un altro indirizzo di spazio degli indirizzi (solitamente su una macchina remota) come se fosse una chiamata locale

Introducing RPC

It's a form of inter-process communication between a *client* and a *server*.

The requested operation is the execution of a procedure on the server, getting back results; *in general, client/server languages may be different.*

Crucial aspects:

- **Interface-based programming** → abstraction, "portability"
- Diverse **call semantics** can be chosen/implemented
- **Transparency**, wrt *access* (local vs remote), *location*, etc. (concurrency, replication, failure, mobility, performance, scaling...)
  se una funzione viene chiamata in locale oppure se questa viene eseguita in remoto, dal punto di vista del programma è esattamente uguale

Why do we want to specify interfaces using yet another language? This additional language will be used by all the other languages to obtain the way to deal with the messages. So, the interface specification gives me the possibility to obtain interoperability across different languages. In fact, they are used by those languages to map the parameters onto a specific way in each specific language to use such parameters.

Interfaces in RPC

RPC interfaces are specified using an **IDL** (Interface Definition Language)

Interface specifications → **cross-language interoperability**, because they are used to map parameters onto each specific way to use them.

An interface definition is usually compiled by using specific tools, so to obtain the modules (stub/skeleton) to be deployed on the different nodes to support the communication.

stub: client side; skeleton: server side

Among "IDLs," *with different facets* : Sun XDR for RPC, CORBA IDL for RMI, WSDL for Web Services... and **Protocol Buffers** as well (defined at Google).

Just data serialization

Nel caso delle RPC e dei servizi web, un'interfaccia descrive un insieme di procedure o metodi che possono essere chiamati da remoto. In altre parole, un'interfaccia in questo contesto stabilisce quali funzioni sono disponibili per l'invocazione da parte di un client su un server, e come questi metodi devono essere chiamati, quali parametri accettano e quali tipi di dati restituiscono.
L'uso di IDL garantisce che queste interfacce possano essere interpretate correttamente in diversi ambienti e linguaggi di programmazione.

Parameter Passing

The ordinary "local" way to deal with parameter passing cannot be plainly extended to the distributed setting:
peers don't share an address space,
so the *usual* **pass-by-reference** convention **does not make sense**:
ordinary addresses cannot be passed as params!

In a distributed system, parameter passing between different components (or "peers") has to be handled differently than in a local system. This is because, unlike local systems where components share the same memory address space, distributed systems do not share memory.

Instead, as in pl/SQL, usually a parameter can be specified as

- IN (input only)
- OUT (output only)
- INOUT (input in Request, output in Response)

Pass-by-Reference := quando un parametro viene passato per riferimento, la funzione riceve un riferimento (indirizzo di memoria) del dato originale. Ciò significa che qualsiasi modifica fatta al parametro all'interno della funzione influenzerà direttamente il dato originale.

RPC Semantics wrt Faults

Exactly-once Semantics := Indica che un'operazione sarà eseguita esattamente una volta, garantendo che non vi saranno duplicati o omissioni.
Nei sistemi distribuiti, garantire l'exactly-once semantics è molto più complesso. È difficile garantire che una richiesta venga ricevuta ed elaborata esattamente una volta, specialmente in presenza di ritrasmissioni dovute a timeout o fallimenti temporanei.

Local calls have the *exactly-once* semantics;
not possible in a distributed setting.

What about remote calls, in presence of faulty behaviours?

depending on how these different tricks are used at the middleware level we will have a different semantics for RPC

Call semantics	*REQ retransmission*	*Duplicate filtering*	*Re-exec vs Re-transmit*	*Examples*
Maybe	NO	Not applicable	Not applicable	-
At-least-once	YES	NO	Re-execution	SUN (ONC) RPC
At-most-once	YES	YES	Re-transmission	CORBA, RMI

La trasparenza nel contesto delle RPC è un concetto chiave che mira a rendere l'interazione tra client e server il più semplice e naturale possibile, lo si fa nascondendo i dettagli della comunicazione remota, permettendo agli sviluppatori di invocare funzioni su un server remoto come se fossero chiamate locali.

How to Deal with Transparency in RPC

Two possible points of view:

- Local calls *syntactically identical* to remote ones
- Local calls ***not*** *syntactically identical* to remote ones

The difference is very relevant from a non-functional point of view, because having the result from a local call will take less time than having a reply from a remote call.

Remote calls yield (producono) much higher delays (some orders of magnitude), and this difference should become evident at the program level.

Current solutions:

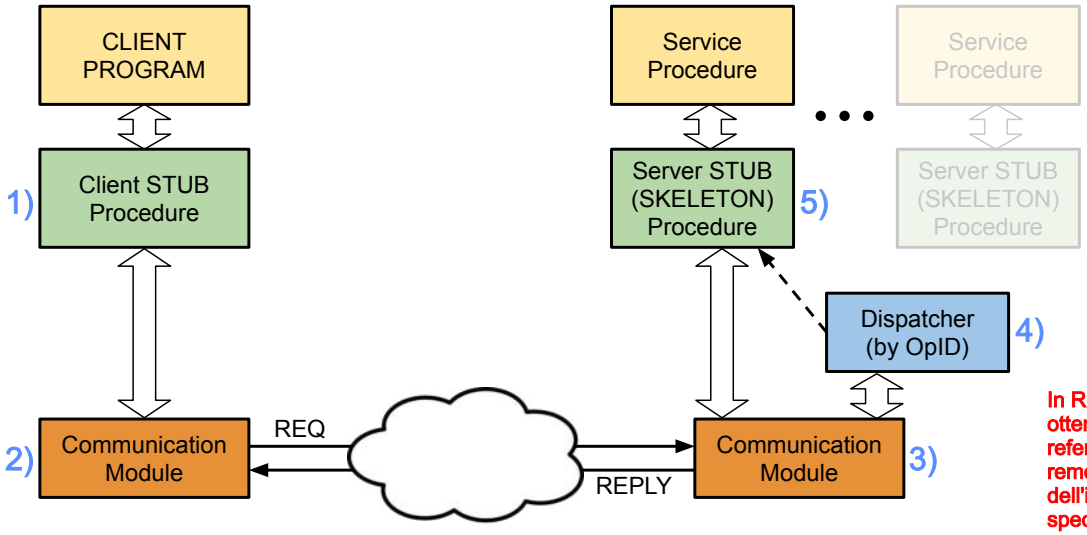
the difference between local and remote procedures is remarked *by the relative interfaces*, but not different syntax is used for calls.

Esempio: pensa a due interfacce Java, una progettata per le chiamate remote e una per le chiamate locali. Queste due interfacce conterranno funzioni con lo stesso nome e gli stessi parametri.

Le procedure locali e remote sono definite tramite interfacce diverse, permettendo al programmatore di essere consapevole della natura della chiamata (locale o remota) tramite l'interfaccia utilizzata. Nonostante le interfacce siano diverse, la sintassi utilizzata per chiamare una procedura remota è la stessa di quella per chiamare una procedura locale.

• stub := è una porzione di codice che risiede sul *lato client* e agisce come un proxy per la chiamata al metodo remoto. Quando un client desidera invocare una funzione remota, chiama il metodo nello stub locale, che si occupa di preparare e inviare la richiesta al server remoto.
• skeleton := è la controparte dello stub sul *lato server*. Riceve le richieste di invocazione dal client, le deserializza (unmarshalling), chiama il metodo appropriato sul server, e quindi invia la risposta al client.

RPC Implementation (chiesta molto spesso all'esame)



In RPC, il processo per ottenere una remote reference della procedura remota varia a seconda dell'implementazione specifica del sistema RPC.

1) Client stub procedure: This part of the system enables the local client to handle the remote call as if it were local. It makes the client program use the communication module to support transparency.
The stub is automatically created by specific tools by inspecting the specification of the communication interface you wrote using the specific IDL language.

2) To communicate with counterparts, it is important to have a specific module that handles communication using the services provided by the network.

3) The communication module on the server side, upon receiving the request, will pass the operation ID portion of the message to another block, the dispatcher.

4) The dispatcher maintains the mapping between the operation IDs and the actual procedures to be called. It indicates which specific server skeleton module to use based on the operation ID.

5) The skeleton contains all the code for interacting with the service procedure. It will pass the request, which will be executed as if it were a local request.

Afterward, it is still the skeleton's responsibility to wait for the result and then send it back through the communication module.

RMI - Remote Method Invocation

Le RMI sono una tecnologia specifica di Java che consente a un oggetto Java in un processo di invocare metodi su un oggetto Java in un processo remoto. Le RMI estendono il concetto di RPC per il mondo Java, consentendo agli oggetti distribuiti di comunicare tra loro.

If you work with Java Enterprise applications, it's important to understand that the middleware you use relies on RMI to communicate with other Java nodes in your system.
Even if you don't directly use RMI in your code, it's crucial for the communication between your software modules.

In Java, you have to interact with objects, which can be, in the case of distributed systems, remote objects. When we pass an object to a Java function, we are effectively passing its reference, so the ordinary call-by-value semantics for parameter passing means that the value of the reference will be copied. But references to local objects are not a problem. However, if we are dealing with remote objects, this method is no longer suitable...

CALL-BY-VALUE in Java
· Passaggio del Riferimento: Quando passi un oggetto a una funzione, stai passando una copia del riferimento all'oggetto. Questo significa che la funzione lavorerà sulla stessa istanza dell'oggetto, ma attraverso un riferimento diverso (temporaneo, "che punta all'oggetto passato come parametro").
· Modifiche all'Oggetto: Qualsiasi modifica fatta all'oggetto all'interno della funzione sarà riflessa nell'oggetto originale al di fuori della funzione, poiché entrambi i riferimenti puntano allo stesso oggetto.

RMI: Towards OO, and More

Influenced by CORBA (interoperable OO RPC)

Similar to RPC wrt:

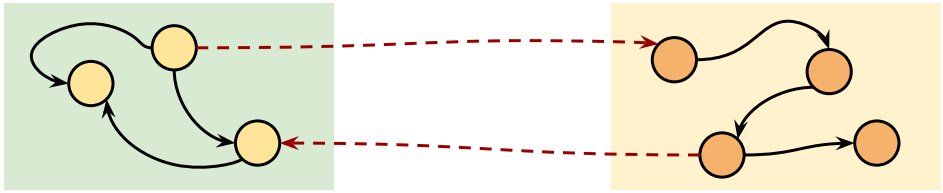
programming by interfaces, *call semantics*, *transparency issues*.

Additional features:

- **Object orientation** - support to *remote* objects
- **Use of remote references** - as parameters as well
- **Use of exceptions** - also for anomalous conditions related to communication (`java.rmi.RemoteException`)

OO: Going Distributed, and Issues

Elegant system model, both local and remote objects (...CORBA...).



oltre alla

Beyond communication, further issues arise, and in particular:

- **Class loading** - how to implement it in a distributed setting?
- **Garbage collection** - how to account for remote references hold by remote client objects?

Remote References

RMI extends the concept of (local) reference to a (local) object:
A **remote reference** is the means to access a remote object.
mezzo

It is an ID for a particular unique remote object in the whole system;
it is created by RRM (Rem. Ref. Module), a dedicated component.

Ideally, its general representation can be structured as follows:

32 bits	32 bits	32 bits	32 bits	
IP Address	Port	Creation time	Object nr.	*Interface of r. obj*

The identifier for a specific remote object must be unique across the entire system and is generated by a dedicated module known as the RRM (Remote Reference Module). This module is an integral part of the runtime middleware support for RMI (Remote Method Invocation) technology.

Remote Interfaces for Remote Objs

An object is a remote object if it implements a *remote interface*, qui con interf intendiamo una interf Java. Cioè una Classe deve avere "implements Remote"
which is ➔ an interface that declares a set of methods
that may be invoked from a remote ("client") JVM.

An interface is "remote" if it at least extends (possibly indirectly)
`java.rmi.Remote` , which is a *marker* interface (no method).
cioè l'interfaccia Remote non contiene metodi, ma extenderla serve per marcarla come accessibile da remoto

Each method declaration in a remote interface
(or its super-interfaces) must be a *remote method* declaration, i.e. ...
e cioè deve: (guarda pag successiva)

Un'interfaccia NON può usare *implements* per implementare un'altra interfaccia. Invece, un'interfaccia può solo estendere un'altra interfaccia usando la parola chiave *extends*.
Questo permette all'interfaccia derivata di ereditare i metodi della superinterfaccia.

Remote Method Declaration

... must:

```
public interface MyRemoteInterface extends Remote {
    void myRemoteMethod() throws RemoteException;
}
```

- include `java.rmi.RemoteException` (or one of its super-interfaces) in its `throws` clause, in addition to any application-specific exceptions (which do not have to extend `java.rmi.RemoteException`);
- In a remote method declaration, a remote object declared as a parameter or return value (even if embedded within a non-remote object in a parameter) must be declared *as the remote interface*, not the implementation class of that interface.

```
public interface RemoteService extends Remote {
    void remoteMethod(RemoteInterface remoteObj) throws RemoteException;
}
```


Because all the interactions with such objects will be handled only according to what you declared in the interface. Of course, inside your implementation, you can include additional methods, but those methods can only be used locally, not when you interact with this object in a remote context.

Summing Up: Parameter Passing in RMI

riassumendo

In RMI, for remote methods of remote objects, arguments are IN, the return value is the only OUT.

A differenza delle chiamate locali, dove i riferimenti agli oggetti possono essere direttamente condivisi in memoria, in un sistema distribuito non è possibile condividere riferimenti diretti agli oggetti.
Questo significa che i dati devono essere trasferiti come una copia (serializzazione) e non possono essere modificati direttamente dal server per riflettere le modifiche nel client.

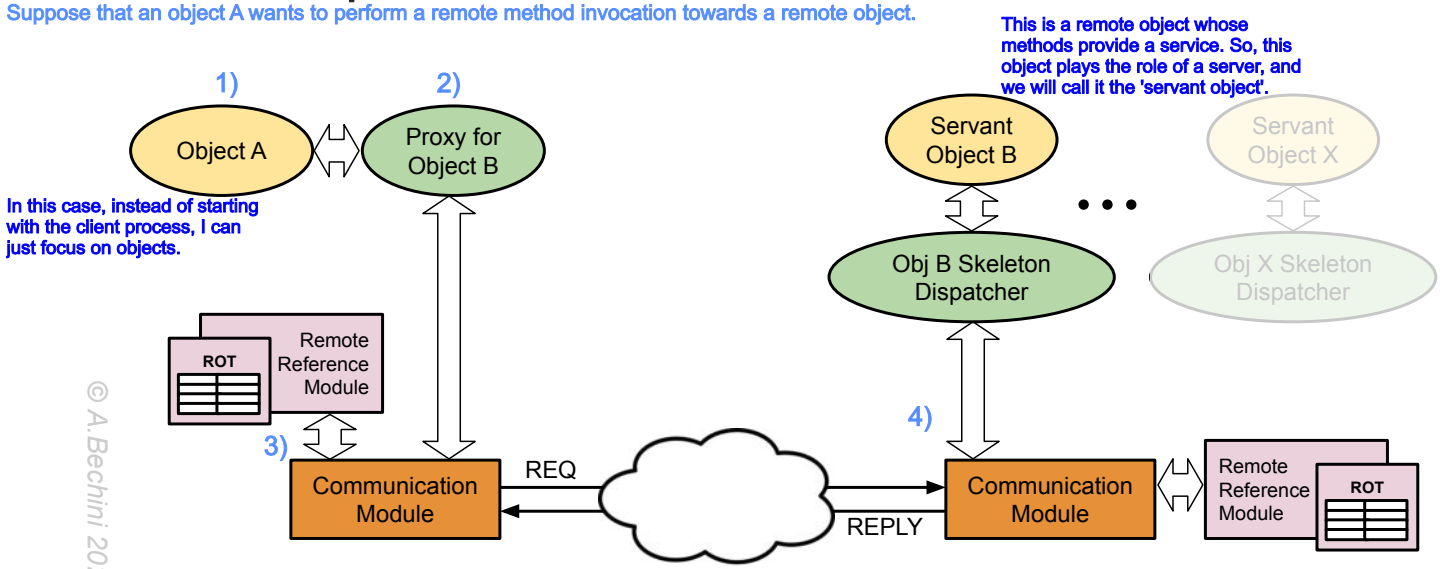
Parameters

- **Ordinary "by value"**; must undergo marshalling, thus are required to implement `Serializable`
- **Remote references**; they must be typed as `Remote` (interface)

"The following diagram is similar to what we have seen for RPC, with some differences: in this case, at the program level, we have local objects that may interact with other objects that are remote.
In practice, we need to provide some proxies to the remote objects, which are a kind of 'avatar' -> virtual objects that are not the original ones. These objects allow us to work with them as if we were working with local objects (when, in reality, we are working with remote objects).
It's up to the local proxy (for the remote obj) to put in place all the actions to communicate with the real object somewhere else. The proxy will interact with the communication module, which is part of the middleware, and it will take care of handling the communication with the counterpart on the server side."

RMI Implementation

Suppose that an object A wants to perform a remote method invocation towards a remote object.



Also in this case, communication modules are present to support the actual communication over the network according to the standard request-reply scheme.

1) We have an object that wants to perform a remote invocation on a remote object.

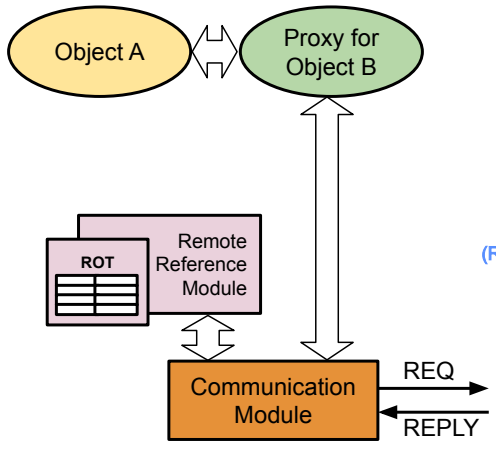
2) The Proxy is a special object, which acts as the local counterpart of the remote one. This object appears to be an ordinary object, but when you invoke a method on it, the computation is not performed locally. Instead, the Proxy activates communication over the system. This object is automatically created by the system based on the classes marked as "remote" by the programmer.

3) Everything is managed through remote references, which are handled by a specific module: the RRM (Remote Reference Module). Inside the RRM, there is a table called the ROT (Remote Object Table), which maintains the binding between a remote reference ID (used by the client) and the reference to a local object on the remote server.

4) On the server side, we have a component very similar to the Proxy on the client side, known as the Skeleton.
On a remote node, there might be many objects on the server acting as remote objects, resulting in many servants, and it's important to know which specific object to contact.
How is the exact object to which the request was sent identified? Once the communication module obtains the request, it precisely identifies the requested service through the RRM. It determines which local object should be used and delegates the Skeleton to perform the actual invocation on the local object. From this point on, everything proceeds in the usual way: an ordinary method invocation occurs over the server, producing a result.

Once the result is obtained, it is then passed to the Skeleton, which sends it back to the server-side communication module. This module then replies to the client-side communication module, and the result finally reaches the original client object.

On Client Side: Proxy (Stub)



Proxy: local counterpart (controparte) of the remote obj, in charge of marshalling/unmarshalling.

The proxy can be constructed AUTOMATICAMENTE (senza che il programmatore faccia nulla) on the basis of the relative remote interface (requires `rmic` for Java < 5)

(Rem. Ref. Module)
RRM: on both sides, creates remote references and, in table ROT, keeps the match between remote/local references.

Dispatcher:
● Il dispatcher è un componente che gestisce la ricezione delle richieste dai client e le indirizza allo skeleton appropriato.
● In un sistema con più oggetti remoti o classi, potresti avere diversi skeleton, ognuno responsabile per un tipo di oggetto remoto o per un insieme di operazioni. Il dispatcher si occupa di determinare a quale skeleton deve essere inviata ogni richiesta in base, ad esempio, all'identificatore dell'operazione (opId).
● Il dispatcher può essere considerato come un livello di smistamento o di routing che assicura che le richieste siano gestite dallo skeleton corretto.

On Server Side: Dispatcher and Skeleton

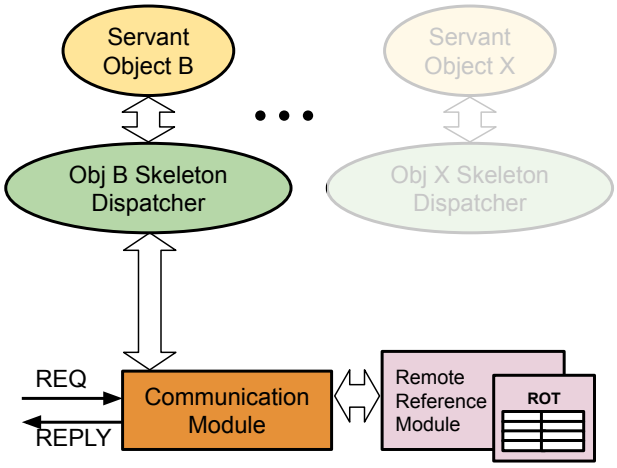
Il motivo per cui un dispatcher e uno skeleton sono creati per ogni classe e non per ogni istanza è che il codice generato per instradare e gestire le chiamate remote è specifico per la classe, non per le singole istanze. Ogni classe ha una definizione fissa dei metodi remoti che può esporre, e quindi il codice del dispatcher e dello skeleton può essere generato una volta per tutte le istanze di quella classe.

One dispatcher + skeleton *for each class* representing a remote object.

Upon REQ, the Communication Module asks RRM for the *local* reference of the involved dispatcher/skeleton.

The dispatcher, based on *opId* (operation ID), determines the method to invoke on the skeleton.

The skeleton performs the unmarshalling and invokes the correct method on the correct servant.



Skeleton:
● Uno skeleton è un ogg. lato server in un sistema distribuito che agisce come un gateway per le richieste in arrivo da client remoti.
● Quando un client invoca un metodo su un oggetto remoto, lo skeleton sul lato server riceve questa richiesta. Lo skeleton si occupa del processo di "unmarshalling", che consiste nel tradurre la richiesta ricevuta, spesso in forma di flusso di byte, in una chiamata di metodo comprensibile all'oggetto locale (detto "servant").
● Dopo l'unmarshalling, lo skeleton invoca il metodo corrispondente sull'oggetto servant. In sostanza, lo skeleton è la rappresentazione lato server dell'oggetto remoto lato client.

Building Up Servants (I)

Servants live in a server process, and are *instances* of a class that implements a remote interface:

- The class *usually* extends `java.rmi.server.UnicastRemoteObject`, inheriting its own remote behavior.
- The class can implement any number of remote interfaces.
- The class can extend another remote implementation class.
- The class can define also non-remote methods, but they can only be used (only) locally.

Servant:
● È l'oggetto lato server che esegue effettivamente le operazioni richieste. È qui che risiede la logica di business del servizio o dell'applicazione distribuita.
● Dopo aver processato la richiesta, il servant restituisce il risultato allo skeleton, che poi lo re-impacchetta e lo invia indietro al client tramite lo stub.

un "servant" è tipicamente un'istanza di una classe Java che implementa un'interfaccia remota

Building Up Servants (II)

A servant must be *exported* to be callable by remote clients.

"exported" (esportato) qui significa rendere un oggetto locale (servant) disponibile per l'accesso remoto

If its class extends `UnicastRemoteObject`, this is done implicitly;
otherwise, the operation must be done explicitly by invoking
`UnicastRemoteObject.exportObject(obj)`

Note: server's capabilities are provided by classes
`java.rmi.server.RemoteObject`, `java.rmi.server.RemoteServer`,
`java.rmi.server.UnicastRemoteObject`, and
`java.rmi.activation.Activatable`

So once you have a servant and you have exported it, a potential client must know about it.
A servant can only be accessed once its remote reference is made available to potential clients. So, how can a client find the remote reference for a servant?
We use a dedicated registry, typically called RMI Registry. This registry allows clients to look up remote references using names, functioning similarly to a traditional name service.

How to Find Remote Objects?

A servant can be accessed once its remote reference is available.

How to find it? We can exploit a **registry**, namely *rmiregistry*, to lookup remote refs from names (i.e. it acts as a *name service*).

Un *name service*, consente di utilizzare nomi facili da ricordare al posto di indirizzi numerici complessi o identificatori univoci. Esempio: il DNS.

Steps:

1) A servant binds to *rmiregistry*, making its rem. ref available on it
2) Any client can lookup the relative rem. ref in the rmiregistry
3) Using the rem. ref, the actual call can be executed

the client has to know where the RMI registry is

Misc on rmiregistry

Di solito, non esistono registry RMI pubblici aperti e accessibili a chiunque per registrare i propri servizi.
Per registrare un servizio su un RMI registry, dovrai creare e gestire il tuo proprio RMI registry sul server dove risiede il servizio.

- Management (e.g., creation) of rmiregistry: by static methods of class `java.rmi.registry.LocateRegistry`
- Usually, it accepts lookup requests on port 1099
- RMI URLs: `rmi://[host][:port][/[object]]`
- Accesses to rmiregistry (bind, rebind, lookup...) are provided by static methods of class `java.rmi.Naming`

RMI and Multithreading

Typical possible approaches:

- **Thread-per-request** - Each request is handled by a separate thread
- **Thread-per-connection** - One thread dedicated to a single connection
- **Thread-per-object** - Requests to a single servant objects are serialized over one dedicated thread

Other combinations of these policies can be devised.

ideate

RMI Multithreading: Specifications

"A method dispatched by the RMI runtime to a remote object implementation (a server) may or may not execute in a separate thread. Calls originating from different clients Virtual Machines will execute in different threads. From the same client machine it is not guaranteed that each method will run in a separate thread"

"So, whenever you write code for your servants, please make sure that your code is thread-safe. You have to imagine that the code will be executed

In practice:

the code related to request processing **must be thread-safe**

Questa è la ragione per cui, nelle lezioni precedenti, abbiamo parlato dei modi per fornire la sincronizzazione in Java

Il class loading in Java è un processo fondamentale che consiste nel caricare le classi bytecodes (i file .class) nella Java Virtual Machine (JVM) per la loro esecuzione.
Ogni classe viene caricata nella JVM solo quando è necessaria per la prima volta, seguendo il principio del "lazy loading".

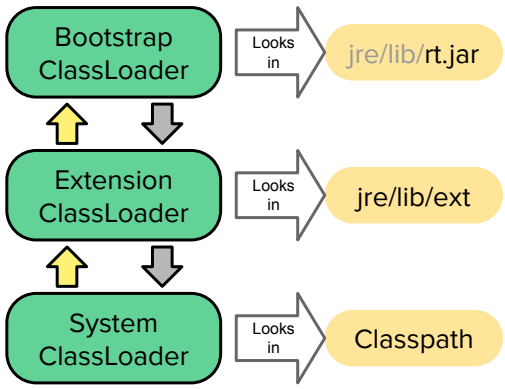
Class Loading

in ordinary Java local program

Ogni volta che una classe deve essere caricata viene prima chiamato il System ClassLoader, questo subito delega ai due ClassLoader superiori (senza nemmeno chiedersi se può caricarla lui), se questi non riescono a caricare la classe allora restituiranno false e sarà il SystemCL a doverla caricare. Funziona così perché se c'è una classe String, si deve caricare quella definita da Java e non quella dell'utente.

Class loading is performed by special components: *Class Loaders*.

They are organized in a **delegation hierarchy**; a classloader asks first its parent to load a class and, in case it fails, it tries to perform the task on its own.



On the right: Standard upper levels.

Properties of "loading": uniqueness, visibility.

Explicit loading: `cl.loadClass(...)`, `Class.forName(...)`

La class Class è usata da Java per mantenere i "metadati" delle varie classi. Ogni volta che una classe viene caricata, viene creata (automaticamente) una istanza di Class per essa. Il metodo forName(String className) permette di caricare dinamicamente una classe data il suo nome completo.

1) Bootstrap Class Loader: È il primo class loader nel processo di class loading. È responsabile per caricare le classi core della Java API presenti nel JDK. Queste classi sono tipicamente quelle disponibili nel rt.jar e altri archivi fondamentali. rt.jar is where very basic Java classes are placed. It contains all the ordinary standard Java library. Typically, it is located in the 'libs' subdirectory of the directory where the JRE is placed.
2) Extension Class Loader: Dopo il Bootstrap Class Loader, entra in gioco l'Extension Class Loader. Il suo compito è caricare le classi che sono estensioni della standard Java API. Queste classi sono solitamente presenti nella directory lib/ext o in qualsiasi altra directory specificata dalla proprietà di sistema java.ext.dirs.
3) System Class Loader (o Application Class Loader): Questo è il class loader che carica le classi del percorso di classi (classpath) dell'applicazione. Include le classi e i package che sono parte dell'applicazione stessa o librerie di terze parti. È il class loader che si utilizza più comunemente durante lo sviluppo di applicazioni Java.

"We have seen that in Java (locally), there are three class loaders. Is it possible to extend this mechanism so that classes could be loaded not just from files present on our system but also from files that may reside elsewhere (on other nodes)?
Certainly, and this is not something particularly new. This capability was achieved at the very beginning of Java history with applets (small Java programs launched from HTML and run in a web browser, locally). In this case, the JVM was present in the client's browser. You had to find classes, where some could be present on the client system, but others defined by the programmer were present on a server. So, in that case, there was a need to load the classes not from the local file system but from that of the server. To achieve this, a specific class loader was added, allowing classes to be loaded from remote nodes."

Distributed Dynamic Class Loading

Distributed Dynamic Class Loading permette il caricamento di classi da remoto, ovvero da server specifici, e questo processo può coinvolgere sia il lato client che il lato server.

Class loading can be performed *remotely*,
by downloading .class files from specific server nodes.
This can be accomplished by a *specific classloader*.

How to know about the (http/file) server to download a class from?
Because initially, we don't know where to find it. So, we need a way to specify where to find those class files so that clients can retrieve them.

1) A remote reference can be annotated with this information (URL)
   quando un oggetto o una classe viene trasmesso in un ambiente distribuito, può includere un URL che indica la sua posizione originale.
2) The object sent within a remote method call is annotated with the URL.

In alternativa, l'oggetto stesso che viene inviato come parte di una chiamata di metodo remoto può essere annotato con l'URL da cui il class loader può scaricare la classe se necessario.

Classloaders are informed on where to load classes
by the *codebase* property, i.e. for RMI: java.rmi.server.codebase

In un ambiente RMI, è spesso necessario caricare classi da una posizione remota. Qui entra in gioco la proprietà "codebase". La java.rmi.server.codebase è una proprietà Java utilizzata in RMI per specificare l'ubicazione (URL) delle classi che devono essere caricate dalla JVM remota.
La proprietà java.rmi.server.codebase può essere usata per indicare un URL da cui la JVM ricevente può caricare queste classi mancanti.

Distributed Dynamic Class Loading

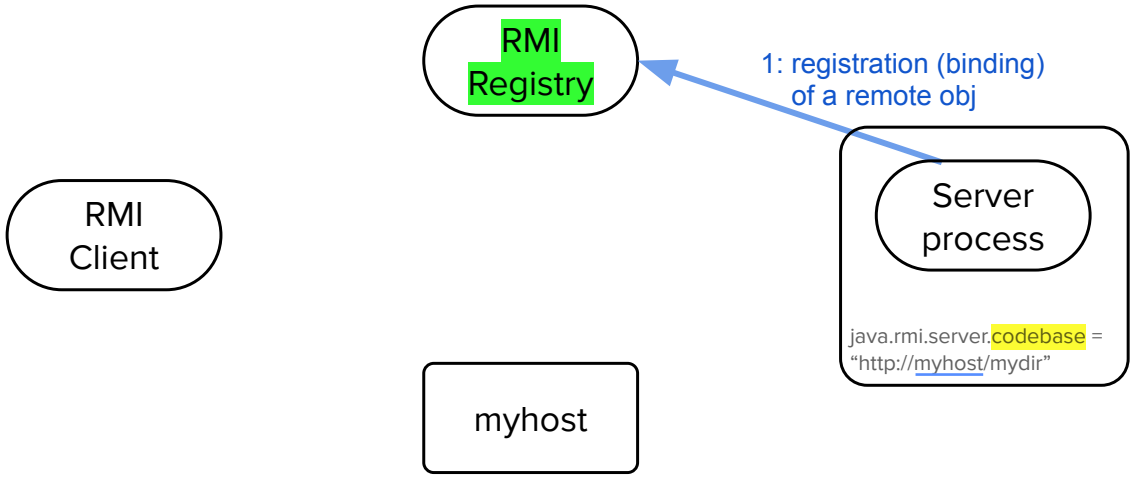
What do you know about Java properties?

© A.Bechini 2023

A. Bechini - UniPI

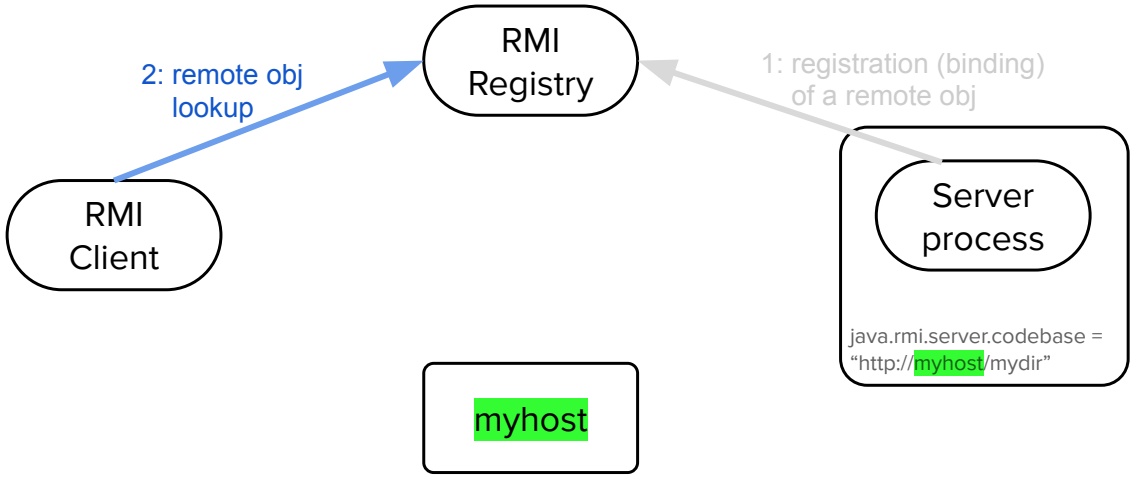
Le "Java Properties" sono un mezzo attraverso il quale Java gestisce le impostazioni di configurazione.
Questo meccanismo è particolarmente utile per gestire le impostazioni che possono cambiare senza la necessità di ricompilare il codice, come le credenziali di accesso a un database o qualsiasi altra configurazione che potrebbe variare in base all'ambiente in cui l'applicazione è eseguita.

Distributed Class Loading - Scenario

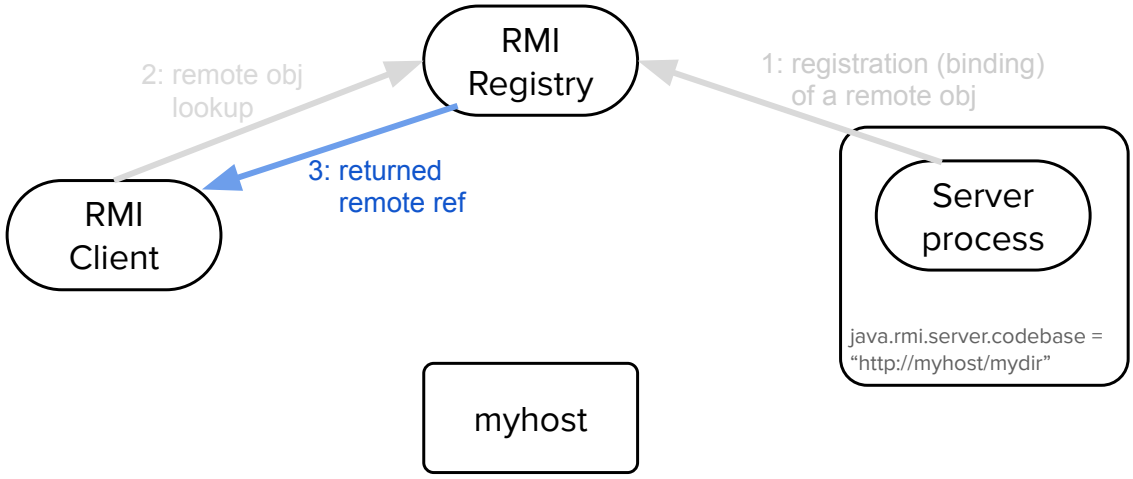


Distributed Class Loading - Scenario

now a client can look up the corresponding remote reference.
The entry in the registry will contain also the annotation about where to obtain the corrisponding .class file

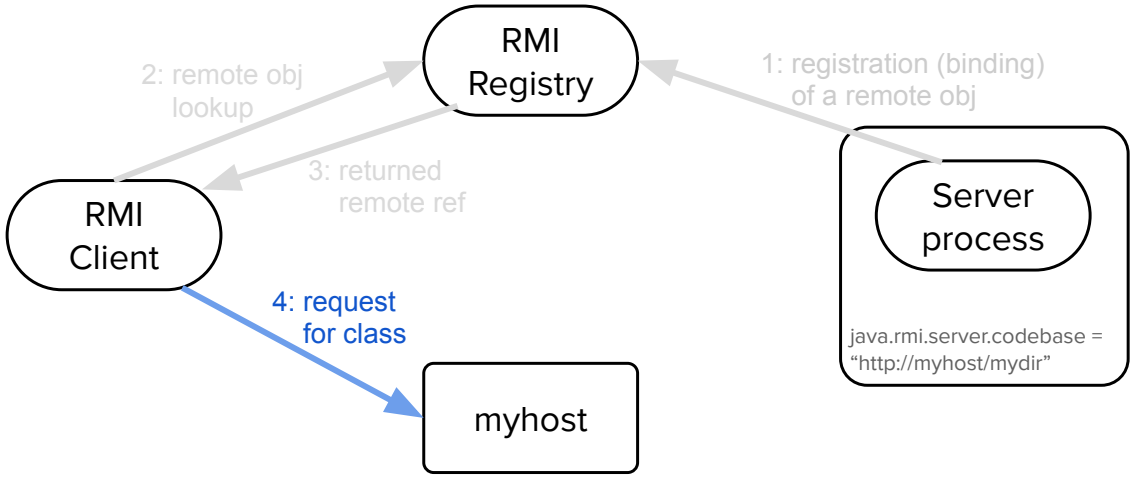


Distributed Class Loading - Scenario

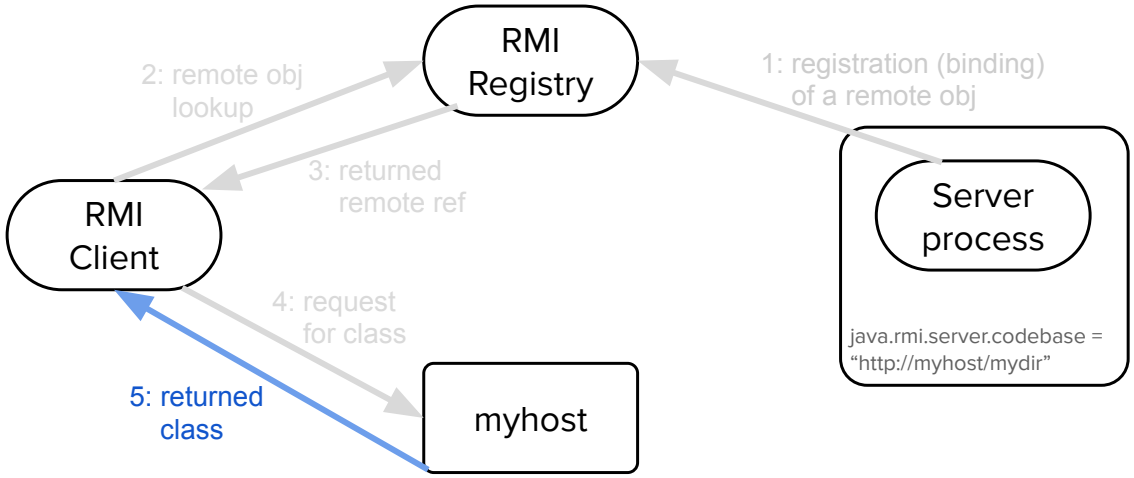


So, the remote reference is returned back. Now, the RMI client needs to retrieve the details for the class. The request for the class is issued toward 'myhost' because 'myhost' was specified as an annotation.

Distributed Class Loading - Scenario



Distributed Class Loading - Scenario



The class is returned back, and now the client has all the information needed to proceed with RMI invocation.

So, we've seen that by using an additional class loader and employing some tricks to add information about the location from which to retrieve .class files, we are able to extend the class loading process even to a distributed environment.

Garbage Collection and DGC

We also need to extend the garbage collection system to accommodate a distributed setting.

In general, for garbage collection, we can rely on two different classes of algorithms. One class is called Mark and Sweep, and the other is known as Reference Counting.
We will cover both of these in the following slides.

Garbage Collection: Mark and Sweep

Idea: Each *reachable* object is marked first,
and then all the unmarked objects are removed (swept away).
spazzati via

The marking algorithm starts from *a pool of* *GC root objects*.
In Java, the GC roots are:

1) local variables in the main method
2) the main thread
3) static variables of the main class

Esempi di tipi di GC roots:
- Oggetti referenziati dallo stack dei thread: Include tutte le variabili locali e i parametri attivi dei metodi chiamati.
- Oggetti referenziati dallo heap: Includono oggetti che sono direttamente referenziati da oggetti nello heap, come classi statiche e singole istanze.

il "Tri-Color Making" è un algoritmo della tipologia "Mark and Sweep"

BTW: Tri-Color Marking (I)

This algorithm does not require “freezing” the system.

Three sets of objects:

- **White**, with candidates to be swept;
  Inizialmente, tutti gli oggetti sono considerati bianchi, ovvero potenzialmente non raggiungibili e quindi eliminabili.
- **Grey**: objs reachable from roots, *not scanned yet for refs to White*
  sono gli oggetti raggiunti dalle radici (roots), cioè sono accessibili attraverso variabili e riferimenti attivi nel programma, ma non sono ancora stati completamente esaminati. Un oggetto grigio potrebbe avere riferimenti ad altri oggetti che devono ancora essere esplorati e potenzialmente trasformati da bianchi a grigi o neri.
- **Black**: not candidates for collection;
  reachable from roots, and *no refs to objs in White*;
  sono considerati sicuramente raggiungibili e non sono candidati per la raccolta. Sono oggetti raggiungibili dalle radici che sono stati già esaminati e non hanno riferimenti ad alcun oggetto bianco.

BTW: Tri-Color Marking (II)

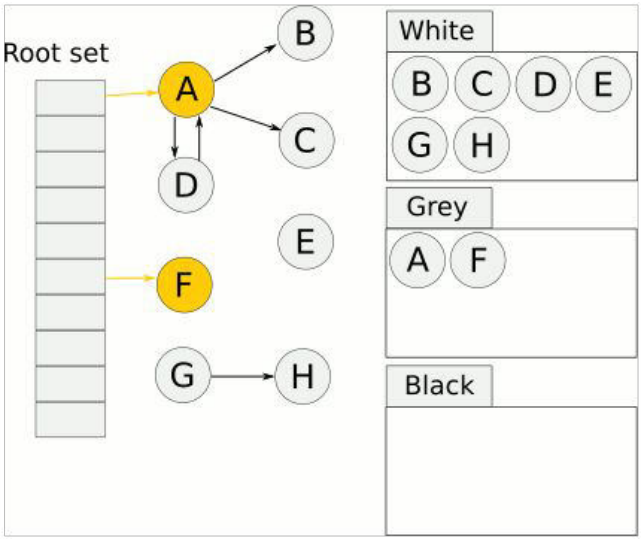
Structure of marking algorithm:
At init, Black is empty, Grey contains the objs directly reachable by roots, White has all the rest.

While Grey is not empty:

- Pick one obj X from Grey, and move it to Black;
- Move into Grey all the objs in White that are referenced by X.

Property: no obj in Black has a ref to objs in White
Thus, at the end Grey is empty ➔ all the objs in White can be swept.

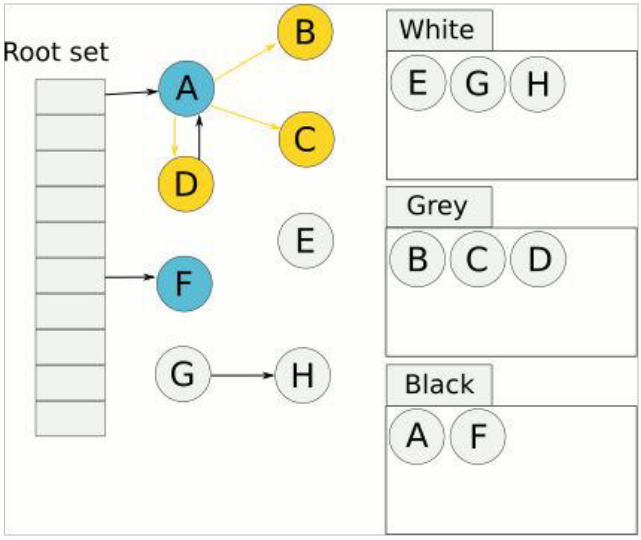
Tri-Color Marking GC - Init



a partire dal root set posso raggiungere gli oggetti A ed F

(from WikiCommons)

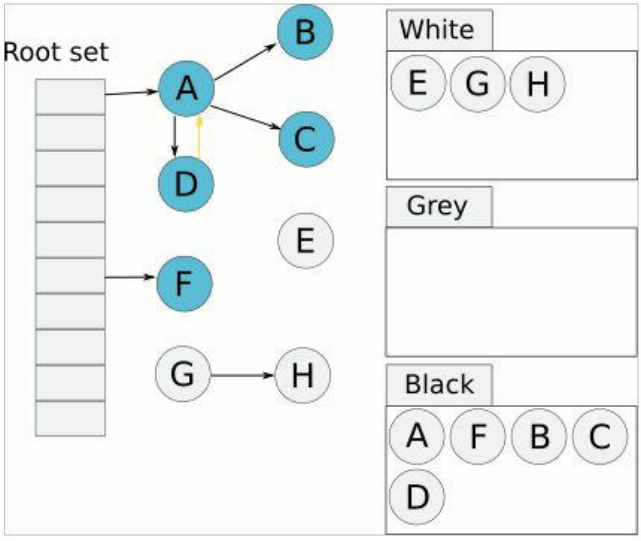
Tri-Color Marking GC - Iteration 1



porto A e F in black.
Sposto tutti quelli raggiungili tramite A e F in grey.

(from WikiCommons)

Tri-Color Marking GC - Iteration 2

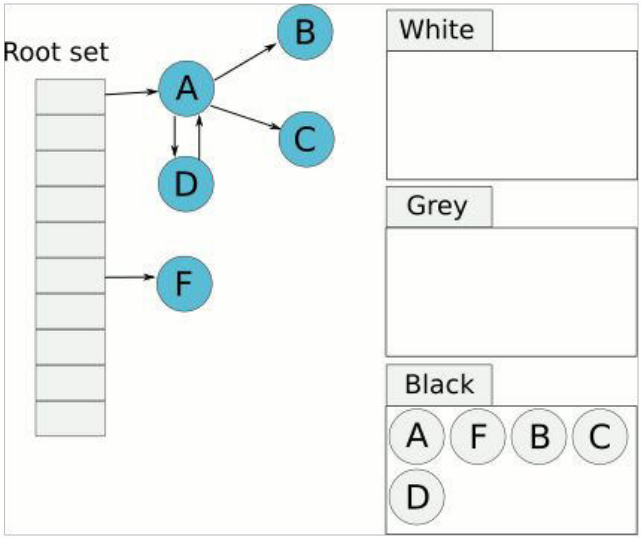


grey vuoto -> fine dell'algoritmo.
tutti quelli in white li posso cancellare

(from WikiCommons)

Tri-Color Marking GC - Sweeping!

pulizia



(from WikiCommons)

Garbage Collection: Reference Counting

Idea: Each object is associated with a *reference counter*;
Object refs added/removed ➔ the counter is updated

The GC is allowed to remove all objects whose counter = 0

Problem: possible cyclic references lead to memory leaks!

Thus, some periodic checks for this must be done.

(se ho due oggetti che si referenziano tra di loro, che però sono distaccati da tutto il resto, entrambi mantengono il contatore ad 1 e quindi non vengono deallocati, anche se non sono raggiunti da altre referenze)

Local GC in Java

Local GC is usually implemented using Mark&Sweep approaches.

gli sono però state fatte delle ottimizzazioni:

Different objects show different utilization patterns (typically, high “infant mortality” is observed), and this affects the GC performance.

Per ottimizzare il processo, gli oggetti vengono divisi in "popolazioni" (ad esempio, giovani e vecchi) in base al loro comportamento. Questo permette di focalizzare il GC principalmente sugli oggetti più giovani, che sono più frequentemente soggetti ad essere eliminati.

For this reason, objects are divided into “populations” (e.g. young and old) according to their behavior, so to optimize the overall process (only younger objs are set to white).

Distributed Garbage Collection (I)

Based on reference counting; Local GC & DGC must collaborate!
Clients should inform servers about what rem refs they are using.

On the server, the Remote Reference Module must keep, for each ROT entry, the list of clients that hold remote references to that object (the “servant”).

Such a list has to be updated each time a remote reference is created/duplicated/removed.

Distributed Garbage Collection (II)

The DGC functionality is described by the `java.rmi.dgc.DGC` interface, used for the server-side component of the overall system. It has two (remote) methods, used by the RMI runtime:

- `dirty(<list of rem refs>, ... , <Client JVM_ID>)`
- `clean(<list of rem refs>, ... , <Client JVM_ID>)`

perché ci serve l'ID della JVM e non ad esempio l'indirizzo di essa?
perché su una macchina potrei avere più JVM

Note: the underlying messages have the at-most-once semantics

Use of Dirty...

- `dirty(<list of rem refs>, ... , <Client JVM_ID>)`

It is used to inform the server that the client is making use of the reported references ➔ update of the relative lists of holders.

This invocation must be done by the RMI runtime just before constructing the proxies for the corresponding remote objects.

The dirty method is called by the client JVM to inform the server that it has a reference to remote objects, ensuring that they are not collected by garbage collection on the server side, as long as the client is still active.

Chi esegue le operazioni di marcatura delle risorse come 'dirty'? Il programmatore?
R: No, queste operazioni sono gestite dal supporto runtime del middleware. Il programmatore non se ne occupa direttamente; sono le API sottostanti a occuparsene.

… and Clean

- `clean(<list of rem refs>, ... , <Client JVM_ID>)`

It is used to inform the server that the client makes NO MORE use of the reported remote references

➔ removal of JVM_ID from the relative lists of “holders”.

As one list becomes empty and also no local reference is present for the object ➔ it can be removed from the heap.

The clean method is used to inform the server that the client no longer needs certain remote references. This call helps the server identify which remote objects are no longer in use and can be collected. Essentially, it "cleans up" the references that are no longer needed, enabling efficient memory management in a distributed environment.

DGC - Problems

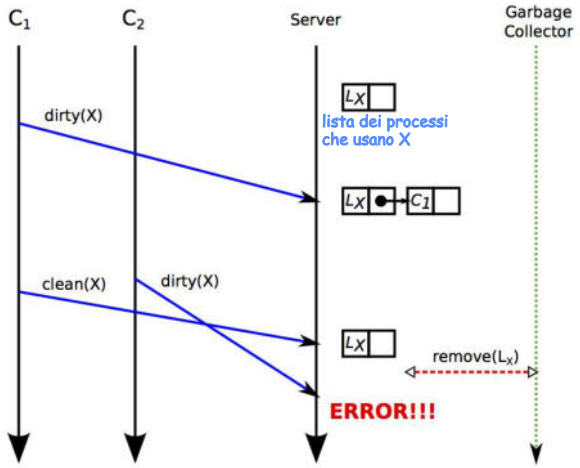
Two problems arise with the described DGC mechanism, asking for proper solutions.

1) Consequences of a data race between `clean(X, …)` and `dirty(X, …)` when the list of holders for X has one single element
2) Crashes of clients may prevent a list from ever become empty, leading to a memory leak.

in una data race tra due client che accedono lo stesso oggetto: se il primo invia la clean prima che il secondo abbia inviato la dirty, il server nel mentre potrebbe deallocare l'oggetto in quanto (per un momento) nessuno lo stava usando

se un client crasha non potrà mai dire al server che non usa più delle reference…

Clean/Dirty Race



The race of clean(X) from C_1 and dirty(X) from C_2 may lead to C_2 trying to use a remote object that is no more present.

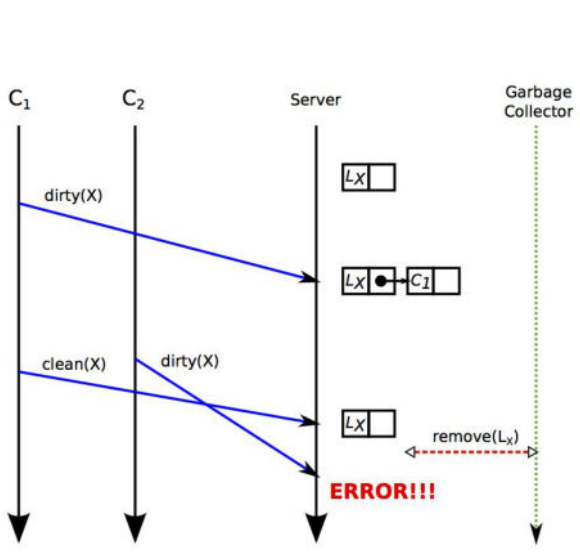
Solution: the server must be informed that X has been given to C, and a dummy placeholder for C is placed in the list of holders for X, waiting for the corresponding dirty(X) to come.

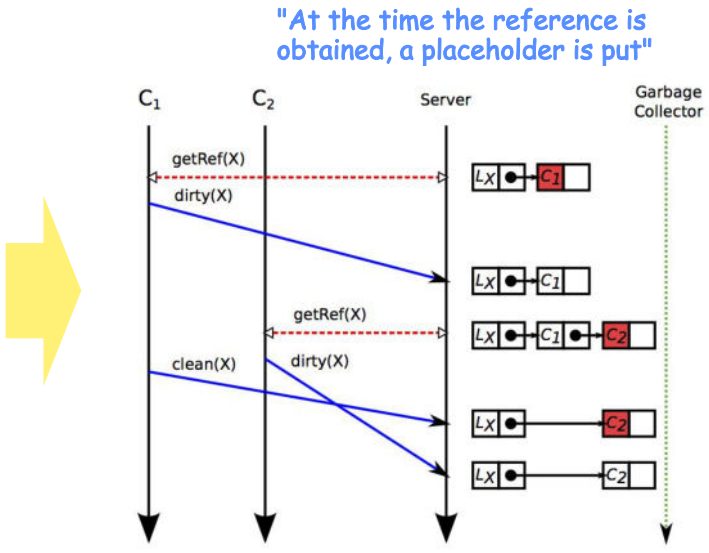
Idea: non appena un client richiede la reference di un oggetto remoto, il server lo marchia così anche se la dirty() ancora non è arrivata, so che non devo cancellarlo

Of course, if this dirty message was sent, it means that at some point in time before, C2 performed the lookup towards the RMI registry to obtain the remote reference.
If a client process has performed the lookup of a remote reference, it's because likely, it has the intention to use it. As soon as the lookup request is answered, a placeholder is set to indicate that shortly, the client will request that object.

Furthermore, we can set the placeholder precisely because if it is possible to respond to the lookup, the object is still available (at that moment), and thus we immediately set the placeholder.

Solution to Clean/Dirty Race





“Leasing,” to Deal with Crash of Clients

The problem can be solved introducing a **leasing** approach:
The semantics of `dirty(…)` has to be changed:
`dirty(...<Lease>)` corresponds to ask for a *lease time*, and a Lease time will be returned.

It’s up to the client runtime to renew the lease by successively invoking `dirty(…)` . Whenever a Lease time expires without timely renewals, the client is removed from the list of holders.

Web Services

I Web Services sono una componente fondamentale dell'architettura delle applicazioni web moderne. Sono sistemi software progettati per supportare l'interazione tra macchine su una rete. Attraverso i Web Services, le applicazioni possono comunicare tra loro e scambiare dati, indipendentemente dalle piattaforme e dai linguaggi di programmazione usati.

Web Services: Why on the Scene?

- So far: discussion on ways to support and abstract communication between client/server, or peers.
- Often, a server is used to expose a given functionality (service).

In molti scenari, un server non è semplicemente un repository di dati, ma un fornitore di funzionalità specifiche, o servizi.

- HTTP has become a pervasive communication means for *machine-to-machine communication*.

è diventato il mezzo di comunicazione dominante

Idea: let's use HTTP to provide clients with *a description of a service result*.

To be formalized: i) required interactions ii) format for the result

I Web Services estendono l'uso di HTTP oltre la semplice consegna di pagine web, sfruttandolo per esporre e descrivere servizi complessi in modo che possano essere facilmente consumati da client remoti. La formalizzazione delle interazioni e dei formati di risultato è cruciale per garantire che questi servizi siano interoperabili, affidabili e comprensibili.

NB: un Web Service non è necessariamente progettato per essere utilizzato solo nelle applicazioni web (anche se questo è il motivo principale del suo utilizzo), ma il nome deriva dal fatto che utilizziamo HTTP, che è il protocollo web archetipico per la comunicazione.

What are Web Services?

- Focus on *interoperability*.
- ⇨ WS are client/server apps that communicate over HTTP.
  As described by W3C, WS provide a standard means of interoperating between software applications running on a variety of platforms and frameworks.
- Programs providing *simple* services can interact to deliver more complex added-value services (e.g.: "mashups" in web apps).
- From ordinary APIs to server **Web APIs**.

Un mashup, nel contesto del web, è un'applicazione che unisce contenuti, dati, o funzionalità da diverse fonti per creare qualcosa di nuovo. I mashup sfruttano la facilità con cui si possono accedere a programmi e dati su Internet tramite il protocollo HTTP. Grazie a HTTP, è possibile richiedere e ricevere dati da diversi servizi web in modo standardizzato e integrarli in un'unica applicazione.

Two Ways to Develop Web Services

From the technological standpoint, two ways of implementing WSs:

- **"Big" Web Services**: emphasis on "service" access, by means of the application level protocol SOAP, XML encodings, and additional protocols. W3C standard for WSs.
- **RESTful Web Services**: emphasis on "resource" access, classic HTTP request methods are used, descriptions also encoded with JSON or XML.

Both require adequate support at server and client sides.

- SOAP e WSL offrono un approccio più formale, dove le funzionalità e le interfacce che le offrono vengono descritte con modalità standardizzate, quindi più rigorose e più affidabili
- REST offre un approccio "mordi e fuggi", meno vincolato. No formal interface definition → producer and consumer must have a mutual understanding. Va bene per applicazioni meno importanti

Big Web Services: Characteristics

- Use of XML messages according to SOAP (Simple Object Access Prot.), with XML encoding of data types
- Formal description of the provided service (i.e. "interface" with the relative operations) by WSDL - WS Description Language
- Addressing of non-functional aspects (transactions, security, etc.)
- Heavy-weight development, but complexity of development can be reduced with use of supporting IDEs.
- Java API: JAX-WS, making SOAP transparent to programmers.

- SOAP è un protocollo standard per lo scambio di messaggi (codificati in XML) tra web services.
- WSDL è utilizzato per descrivere formalmente i web services: definizione delle operazioni disponibili, i tipi di dati richiesti e restituiti, gli endpoint di rete, ecc.. Questo permette ai client di capire come interagire con il servizio senza conoscere i dettagli interni dell'implementazione.

Big Web Services: Architecture

UDDI (Universal Description, Discovery, and Integration) è uno standard per un repository di servizi web. Permette alle aziende di registrare e pubblicizzare i loro servizi web e ai client di cercarli e localizzarli. Funziona come una sorta di "rubrica telefonica" per servizi web.

- Development: either “WSDL first”, or “Implementation first”
- Broker Repository: use of the UDDI standard (Universal Description Discovery and Integration)
- Aimed at targeting SOA systems (Service-Oriented Architecture)
- Actually, not very popular nowadays

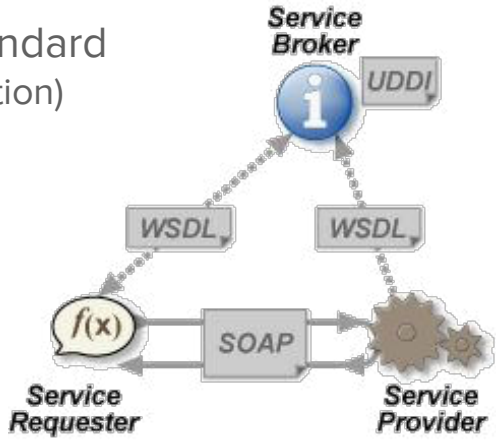


Image from Wikimedia Common

I Big Web Services sono spesso utilizzati nell'ambito delle architetture orientate ai servizi (SOA), dove l'obiettivo è creare applicazioni modulari composte da servizi indipendenti. SOA è un modello architetturale che permette l'integrazione di servizi distribuiti, scalabili e indipendenti, che possono essere riutilizzati in diverse applicazioni.

Nonostante i loro vantaggi in termini di standardizzazione e robustezza, i Big Web Services non sono attualmente molto popolari. Questo può essere dovuto alla loro complessità di implementazione e alla crescente preferenza per approcci più leggeri e flessibili come RESTful Web Services.

RESTful Web Services: Characteristics

- Lightweight infrastructure, easy IDE-based WS development.
- REST architectural style (**RE**presentational **S**tate **T**ransfer - no “official” standard)
- No XML messages, *no formal interface definition*: producer and consumer must have a mutual understanding of the context and content being passed along.
- Basic idea: a “resource” on the server, and HTTP requests towards it can either return a description of its state, or update it
- Restricted ways to respond to requests (REST architectural constraints)

A differenza dei SOAP Web Services, i RESTful Web Services generalmente non utilizzano messaggi XML strutturati o una definizione formale dell'interfaccia come WSDL. Produttori e consumatori di un servizio REST devono avere una comprensione reciproca del contesto e del contenuto degli scambi, che spesso avviene attraverso formati come JSON.

"The idea here is to think of services as resources. The HTTP request towards such a resource can either return a description of the state of the resource, which corresponds to the output of the service, or, if the service has some kind of state information, such messages can also update the state for the resource."

Architectural REST Constraints

- Client-server Architecture
- *Stateless*-ness, i.e. no client context stored on server side.
- Cacheability - clients and intermediaries can cache responses.
  Le risposte del server devono essere etichettate come cacheabili o non-cacheabili.
- Layered System - no way for a client to tell apart the server and an intermediary.
  stratificato distinguere
- Code on demand - possibility to transfer code (e.g. JavaScript...)
  REST permette l'opzionalità di scaricare codice eseguibile (come JavaScript) dal server, che può essere eseguito sul client.
- Uniform Interface: subject to specific constraints.

REST Uniform Interface

Resources are manipulated by a fixed set of operations, specified by the HTTP message “method”:

- PUT creates a new resource, and DELETE deletes one
- GET retrieves the current state of a resource *in some representation*. Resources are decoupled from their representation: their content can be accessed in many formats, such as HTML, XML, plain text, PDF, JPEG, JSON, ...
- POST transfers a new state onto a resource.
  existing

REST Uniform Interface

Resou operations,
speci

- P deletes one
- G
  in led
  from their representation: their content can be accessed in many formats, such as HTML, XML, plain text, PDF, JPEG, **JSON**, ...
- POST transfers a new state onto a resource.

JSON is one
of the most widely used formats
for interoperable data exchange,
and deserves a few words

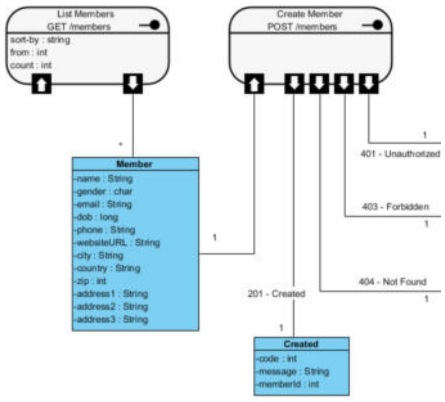
RESTful APIs/interfaces

A RESTful service should be described accurately by a “Web API”

The **design of a proper API** for a service exposed via a REST approach is **crucial** from a SW Engineering viewpoint.

Several proposed DLs (Description Languages):

RSDL (RESTful Service Description Language),
RAML (RESTful API Modeling Language),
OpenAPI - paired to graphical designers



La creazione di un'API RESTful non è solo una questione di implementazione tecnica, ma anche una sfida di ingegneria del software. Una buona API dovrebbe essere chiara, consistente, e facile da usare per gli sviluppatori. Questo significa definire chiaramente le risorse disponibili, i metodi HTTP supportati (come GET, POST, PUT, ecc.), i formati dei dati in input e output, e qualsiasi altro dettaglio necessario per utilizzare il servizio.

Java RESTful WSs with JAX-RS

(Java API for RESTful Web Services)

Reference implementation: Jersey, inside GlassFish

Idea: use (runtime) annotations to decorate a "resource class" (typically a POJO) to make it managed as a REST web resource.

A *JAX-RS helper servlet* is internally used to catch the relative reqs.

"A JAX-RS helper servlet is a special internal component, always present in the application. It is used to catch the relative request and delegate it to the appropriate method in the resource class."

Some typical annotations:

- `@Path` - a relative URI path indicating the "placing" of the class
- `@GET` - annotation for the class method to process GET requests
- `@Produces` - specifies the MIME type of the content produced

- Jersey: È l'implementazione di riferimento per JAX-RS. Fornisce una libreria che facilita lo sviluppo di Web Services RESTful in ambienti Java. (viene spesso utilizzata all'interno del server applicativo GlassFish)
- GlassFish: È un server applicativo che può funzionare come contenitore per applicazioni JAX-RS/Jersey.

JSON - JavaScript Object Notation

- It is a very popular language-independent format used to exchange data in interoperable systems.
- It is a *textual* format, written according to the object notation of JavaScript. In JS: `JSON.parse(text)`, `JSON.stringify(obj)`. Many languages have libraries to manipulate JSON data.

JSON rappresenta i dati come oggetti

- It considers data objects consisting of attribute–value pairs and array data types
- It supports the possibility to define a schema (as XSD in XML)

La sua popolarità deriva dalla sua semplicità, leggibilità e dalla facilità con cui i dati in formato JSON possono essere generati e analizzati.
JSON è basato sulla notazione degli oggetti di JavaScript, ma è indipendente dal linguaggio: può essere generato e interpretato da molti linguaggi di programmazione.

JSON Data Types

Un oggetto JSON è una raccolta non ordinata di coppie nome-valore racchiuse tra parentesi graffe { }.

- *Object*: unordered collection within { } of comma-separated pairs made of *name (string)* : *value*.
  It is intended to represent an associative array.
- *Array*: ordered list within [ ] of 0+ values, each of any type.
- *Number*: no distinction in the format between integer and floating-point.
- *String*: Unicode chars within “”
- *Boolean*: `true` and `false`.
- `null`: empty value

An Example of JSON

```
{
  "firstname":"Tom",
  "lastname":"Smith",
  "age":20,
  "exams":[ "Calculus","Algebra","Algorithms"]
}
```

As it can be seen, JSON as XML is self-describing and easy to understand, but supports arrays as well.

JSON is easier to parse than XML, and much shorter.

Not surprising that JSON has gained widespread popularity.

ProtoBuf and gRPC - Say something, TODO

```
{
  "firstname":"Tom",
  "lastname":"Smith",
  "age":20,
  "exams":[ "Calculus","Algebra","Algorithms"]
}
```

As it can be seen, JSON as XML is self-describing and easy to understand, but supports arrays as well.

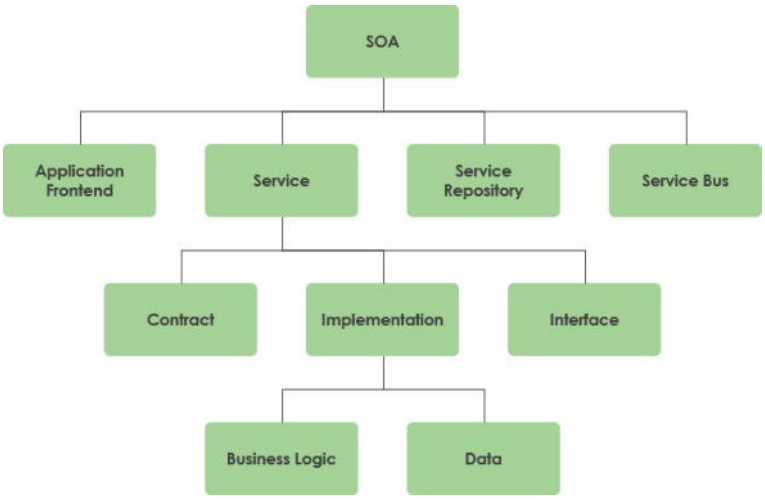
JSON is easier to parse than XML, and much shorter.

Not surprising that JSON has gained widespread popularity.

"The possibility to have pieces of code shaped as services provides the opportunity to build complex applications out of code scattered across the internet. However, assembling all these elements is not a simple task, both in a proper and effective way. From a software engineering perspective, a defined approach has been introduced to support the structure of such complex applications, known as Service-Oriented Architecture (SOA).
In SOA, there are several services, but we also need a service repository to store them, an application front-end, and to support communication in a coordinated way, we may need additional middleware support represented by the so-called service bus. This serves as another abstraction layer for communication between services belonging to the same overall application. When we talk about services, we have to specify: the contract, the implementation, and the interface."

SOA - Service Oriented Architecture (I)

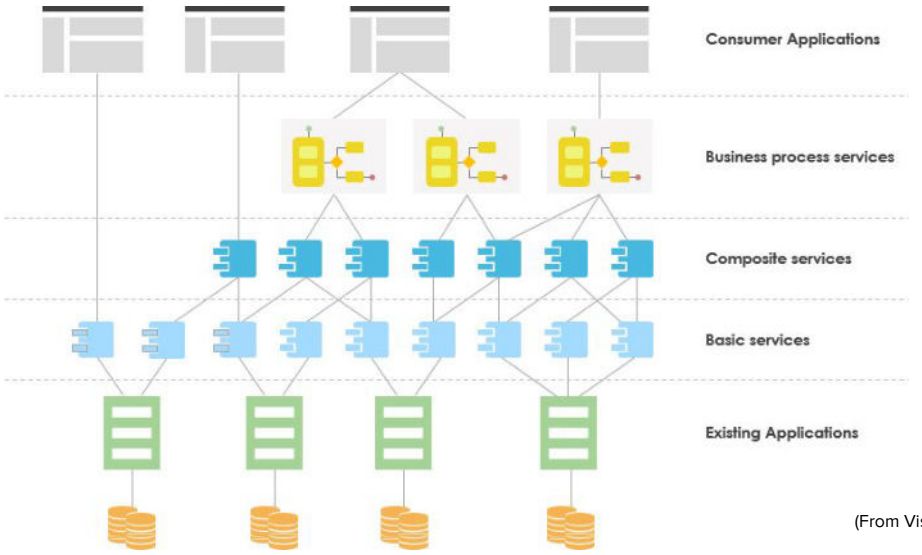
- The interface defines how a service provider will perform requests from a service consumer
- The contract defines how the service provider and the service consumer should interact
- The implementation is the actual service code itself



(From VisualParadigm website)

Il contratto va oltre la semplice interfaccia del servizio, includendo aspetti legali, operativi e di conformità.

SOA - Service Oriented Architecture (II)



(From VisualParadigm website)

"This vision is very flexible, and in this framework, several applications could make use of shared services. There could be multiple existing applications on different nodes, each accessing its own backend data. They may expose basic services in the way we have already seen, and these services can be used to compose further services that exploit other minor services. This approach can be used to obtain what is referred to as business process services: services that can be directly used by an enterprise application. Upper-level applications can make use of services at any layer, from basic services up to business process services."

Microservices

"What we have described is indeed a very flexible approach, but for very big services, there are some problems related to the complexity of the overall system. Most of the complexity arises from the need to provide special support for communication across services, typically named a 'service bus' and sometimes also referred to as an 'enterprise service bus' to underline the fact that it is intended to connect enterprise services (services that can be used to support the building of other services).

The result is that depending on how specific service-oriented architectures have been used in developing software projects, the final result can be a success or, in other cases, a disaster. Different approaches have been proposed, and a successful proposal was to avoid using very complex services but instead using smaller services to standardize the way to access them, maybe just by a RESTful way.

The same single service can be exposed by one single interface, but we may provide many different interfaces for the same service to be used for different purposes. This context has led to the development of the so-called microservices, one of the most popular architecture paradigms used nowadays."

Microservices

- TODO

Towards JEE Architecture

Lo sviluppo di Java EE (Enterprise Edition), una piattaforma estesa della Java Standard Edition (Java SE), è stato una risposta diretta alle crescenti esigenze delle applicazioni aziendali in termini di funzionalità, sicurezza, gestione delle transazioni, e scalabilità. La piattaforma Java EE fornisce gli strumenti e le specifiche necessarie per affrontare queste sfide in modo efficiente e standardizzato.

Distributed Applications: More and More Complex

The increasing complexity of distributed applications has lead to the definition, in the Java community, of Java EE (Enterprise Edition).

It is a set of specifications to define a *whole framework* for the support of a wide variety of functionalities for distributed applications, relying on proper runtime (web servers, application servers, etc.).

Final Thought: The More Complex The App, The more Support We Need...