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 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

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.

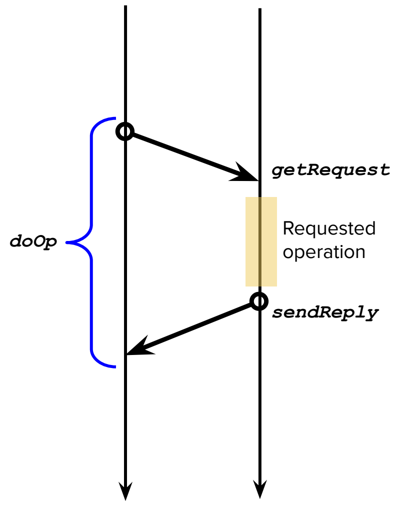
General Primitives (I)

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

1. `public byte[] doOp(RemoteRef s, int opId, byte[] args)`
   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.

General Primitives (II)

2. `public byte[] getRequest()`
   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)

Message type: int (REQ/REPLY)

RequestId: int

Remote Reference: dedicated type

OperationID: int, or dedicated type

Arguments: array of bytes

WHY?
Here, just for the sake of generality

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).

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

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?

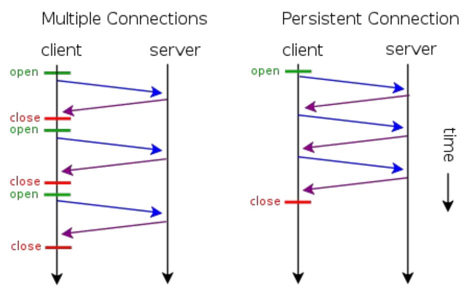
- We refer to **idempotency** of a function f **w.r.t. sequential composition**, i.e. two subsequent calls of f: $f(\cdot)\,;f(\cdot)$
- Formally, idempotency of f means that,

  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.
- Of course, this is a significant property for functions *with* side effects: pure functions are idempotent by nature!

Safe to retry!

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.



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

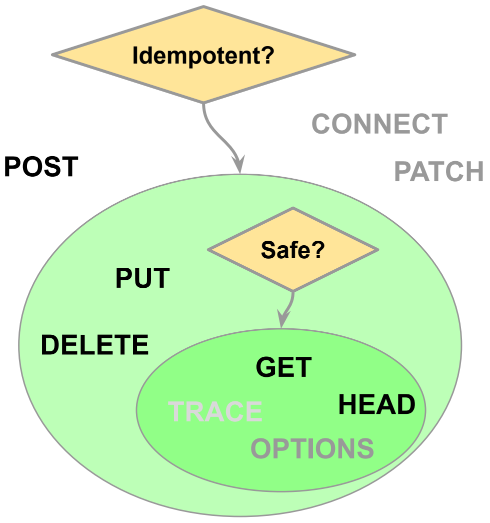
```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

HTTP “Methods”

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

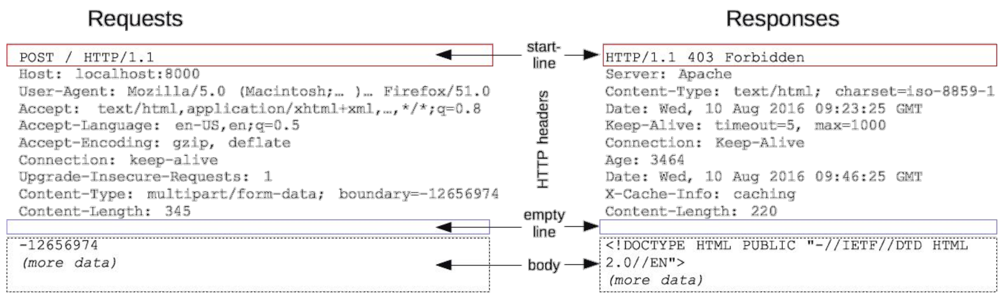
Idempotence: the side-effects of N > 0 identical requests are the same as for a single request ➜ caching of results is *possible*!

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



Structures of HTTP Messages

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



```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;… )… Firefox/51.0
Accept: text/html,application/xhtml+xml,…,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

```
HTTP/1.1 403 Forbidden
Server: Apache
Content-Type: text/html; charset=iso-8859-1
Date: Wed, 10 Aug 2016 09:23:25 GMT
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
Age: 3464
Date: Wed, 10 Aug 2016 09:46:25 GMT
X-Cache-Info: caching
Content-Length: 220

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML
2.0//EN">
(more data)
```

From developer.mozilla.org

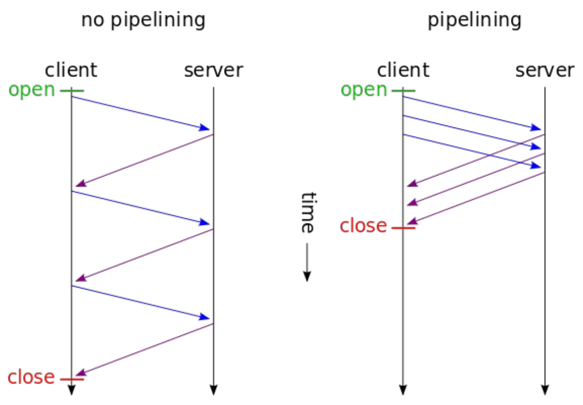
The same fields have possible different meanings (or can be used differently), depending on the type/method

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.



One step further:
HTTP/2 multiplexing + QUIC

Marshalling? No, Content Specification!

Approach to the identification of type of data contents:

data is prefixed by **MIME specification**,

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

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…)

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.

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

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!

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)

RPC Semantics wrt Faults

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

What about remote calls, in presence of faulty behaviours?

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

How to Deal with Transparency in RPC

Two possible points of view:

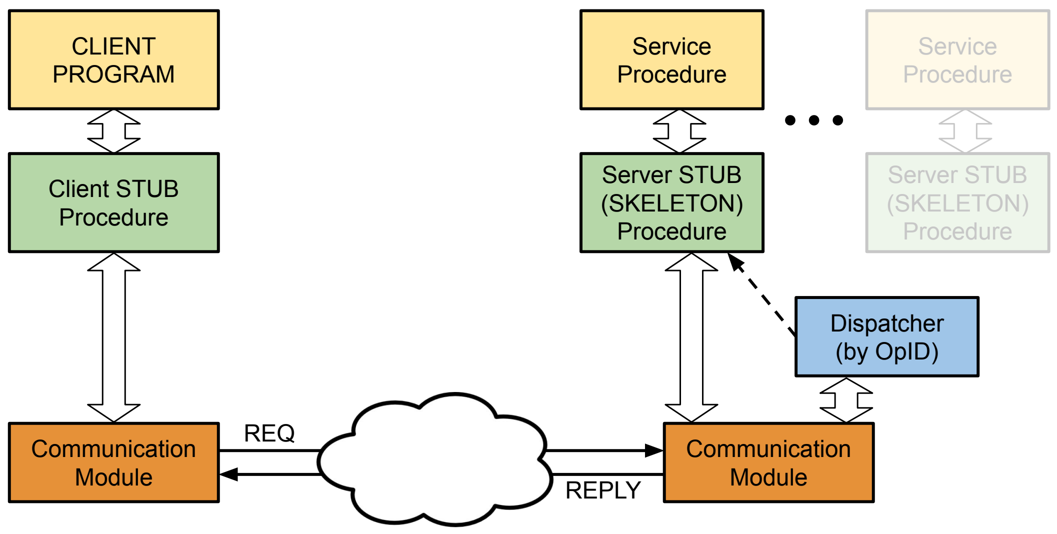
- Local calls *syntactically identical* to remote ones
- Local calls ***not*** *syntactically identical* to remote ones

Remote calls yield 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.

RPC Implementation



RMI - Remote Method Invocation

RMI: Towards OO, and More

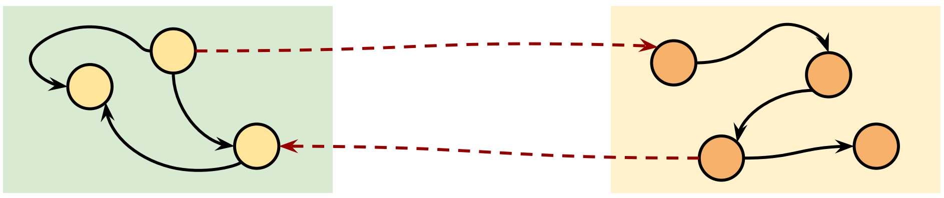
Influenced by CORBA (interoperable OO RPC)
Similar to RPC wrt:
programming by interfaces, *call semantics*, *transparency issues*.

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...).



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.

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*

Remote Interfaces for Remote Objs

An object is a remote object if it implements a *remote interface*,
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).

Each method declaration in a remote interface
(or its super-interfaces) must be a *remote method* declaration, i.e. …

Remote Method Declaration

… must:

- 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.

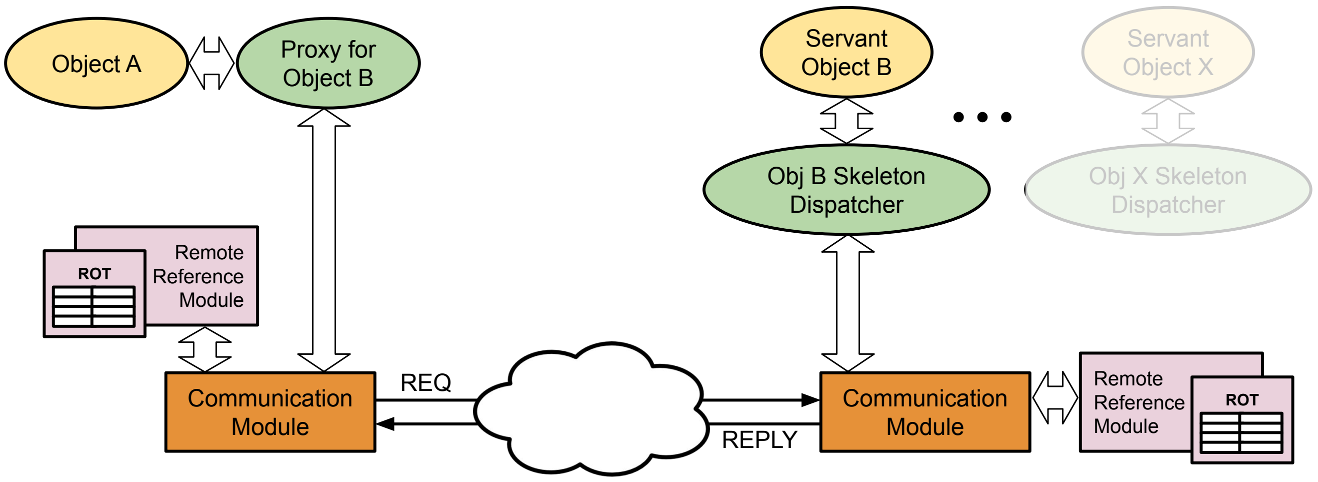
Summing Up: Parameter Passing in RMI

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

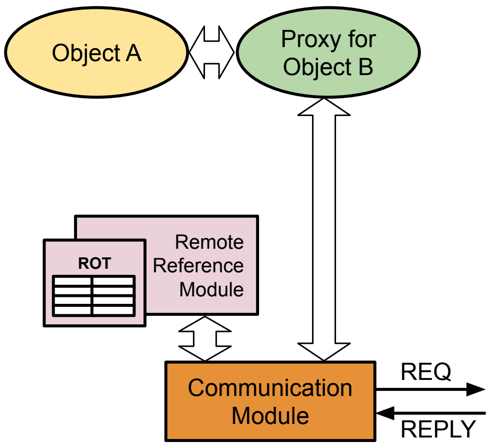
Parameters

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

RMI Implementation



On Client Side: Proxy (Stub)



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

The proxy can be constructed on the basis of the relative remote interface (requires `rmic` for Java < 5)

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

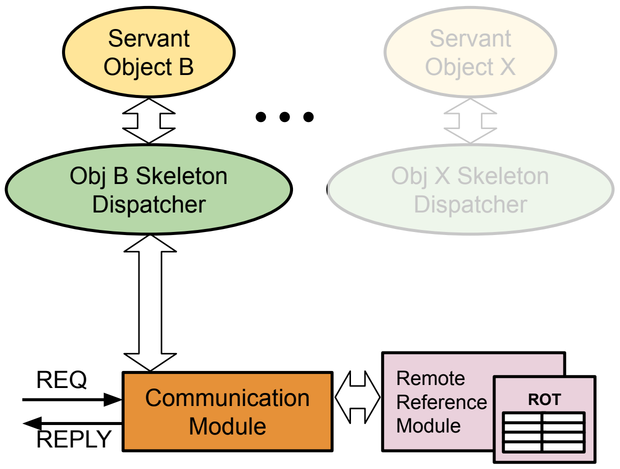
On Server Side: Dispatcher and Skeleton

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*, determines the method to invoke on the skeleton.

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



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 locally.

Building Up Servants (II)

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

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`

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*).

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

Misc on rmiregistry

- 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.

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"

In practice:

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

Class Loading

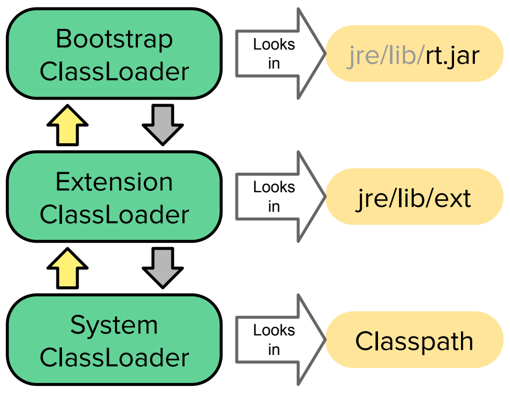
Class loading is performed by special components: *Class Loaders*.

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(...)`



Distributed Dynamic Class Loading

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?

1) A remote reference can be annotated with this information (URL)
2) The object sent within a remote method call
   is annotated with the URL.

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

Distributed Dynamic Class Loading

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

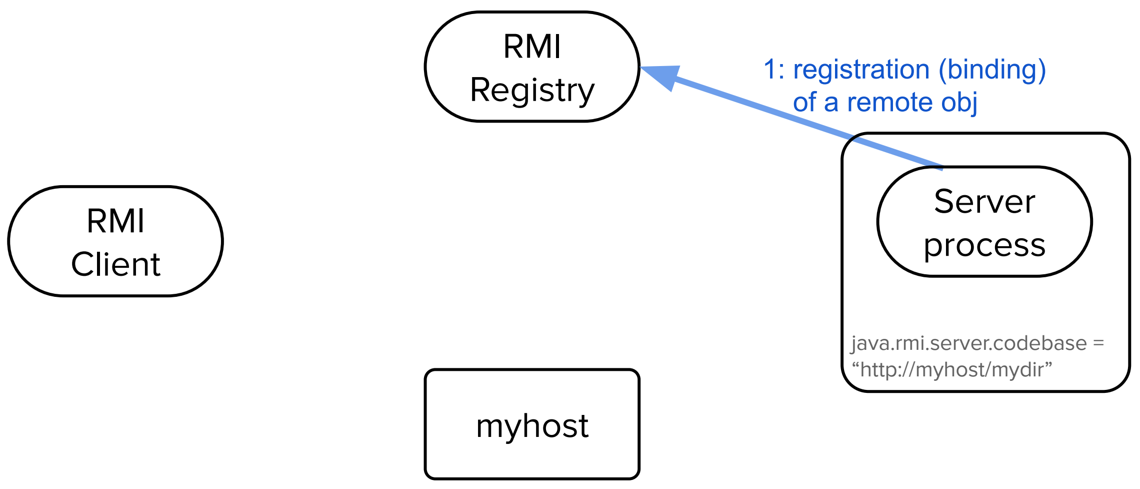
How to know about the (http/file) server to download a class from?

1) A remote reference can be annotated with this information (URL)
2) The object sent within a remote method call
   is annotated with the URL.

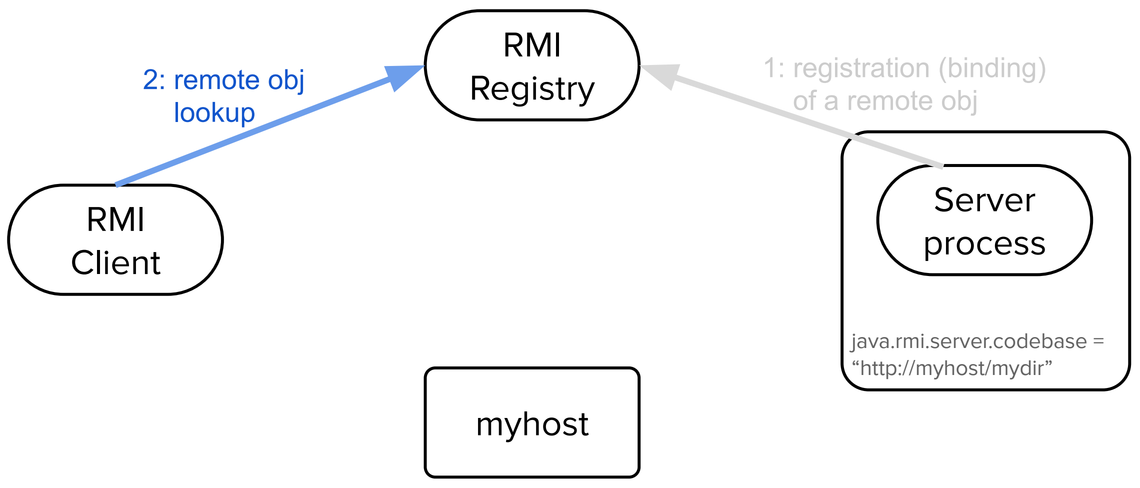
Classloaders are informed on where to load classes
by the *codebase* property, i.e. for RMI: java.rmi.server.codebase

What do you know
about
Java properties?

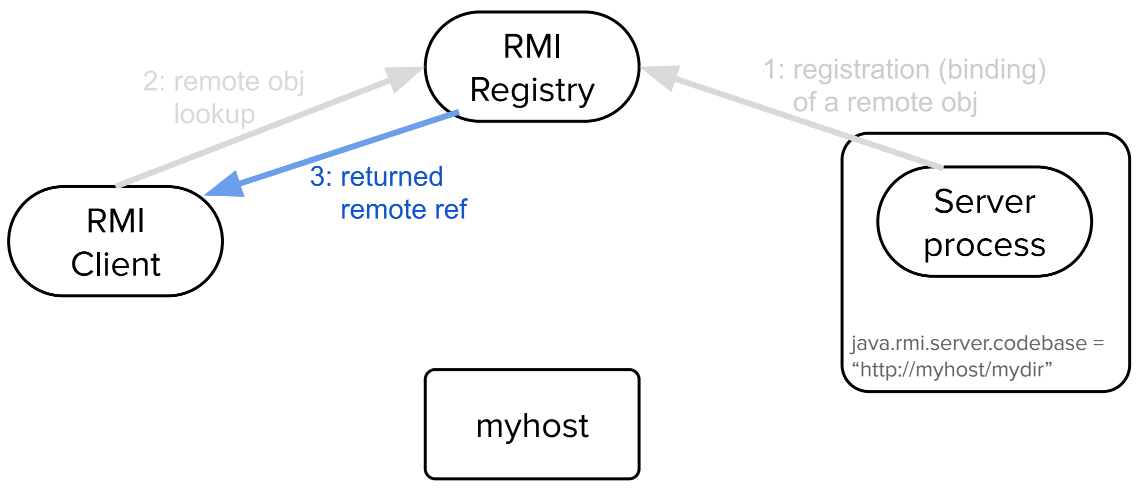
Distributed Class Loading - Scenario


RMI Registry
1: registration (binding) of a remote obj
RMI Client
Server process
java.rmi.server.codebase = "http://myhost/mydir"
myhost

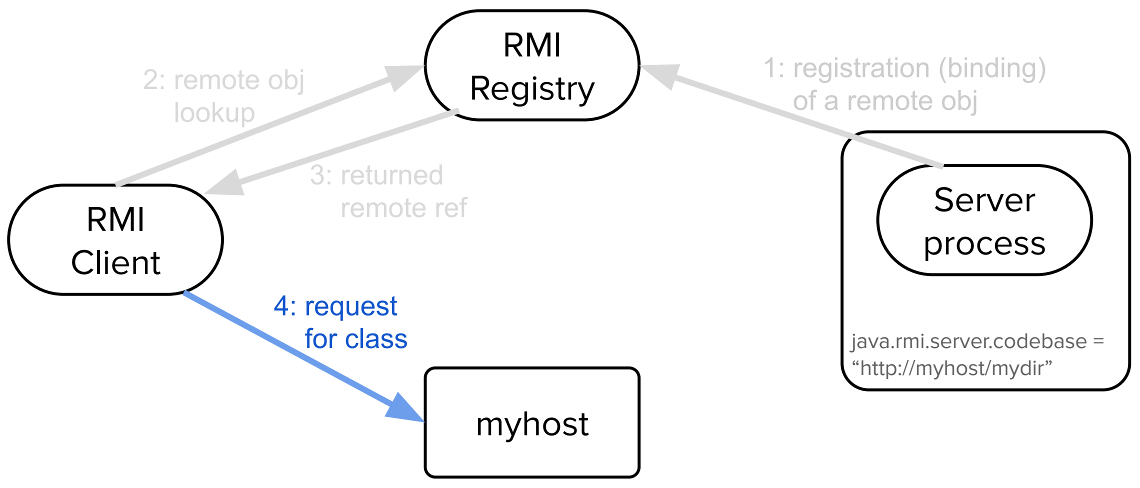
Distributed Class Loading - Scenario


2: remote obj lookup
RMI Registry
1: registration (binding) of a remote obj
RMI Client
Server process
java.rmi.server.codebase = "http://myhost/mydir"
myhost

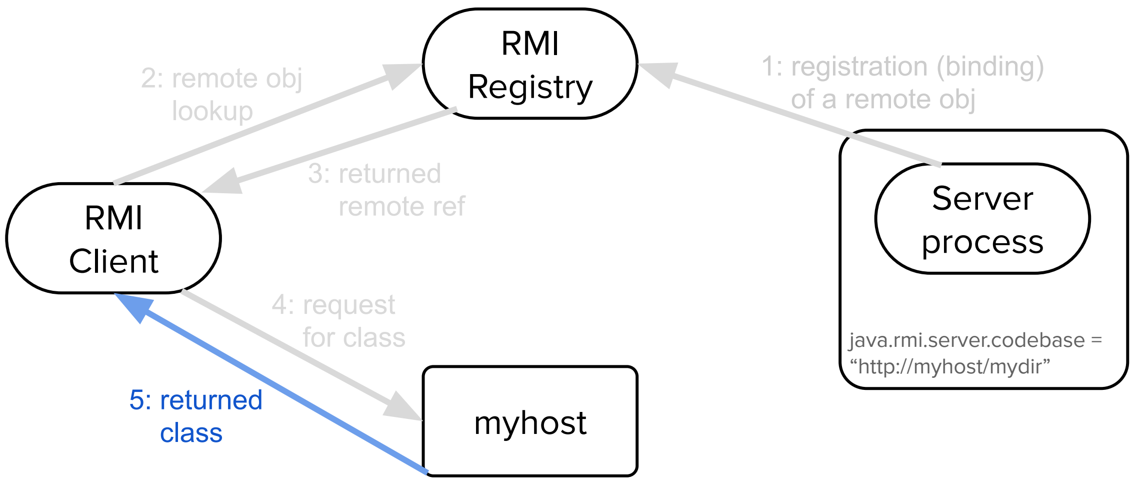
Distributed Class Loading - Scenario


RMI Registry
2: remote obj lookup
1: registration (binding) of a remote obj
3: returned remote ref
RMI Client
Server process
java.rmi.server.codebase = "http://myhost/mydir"
myhost

Distributed Class Loading - Scenario



Distributed Class Loading - Scenario



java.rmi.server.codebase = "http://myhost/mydir"

Garbage Collection and DGC

Garbage Collection: Mark and Sweep

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

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

BTW: Tri-Color Marking (I)

This algorithm does not require “freezing” the system.

Three sets of objects:

- **White**, with candidates to be swept;
- **Grey**: objs reachable from roots,
  not scanned yet for refs to White
- **Black**: not candidates for collection;
  reachable from roots, and *no refs to objs in White*;

BTW: Tri-Color Marking (II)

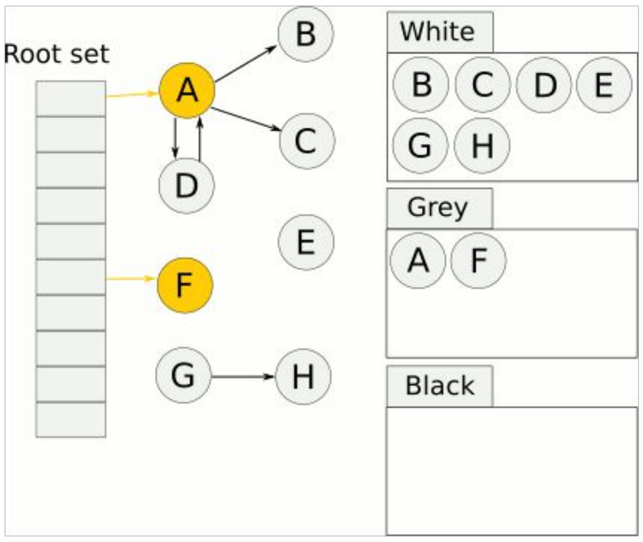
Structure of marking algorithm:
At init, Black is empty, Grey contains the objs directly reachable by roots, White has all the rest.

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.

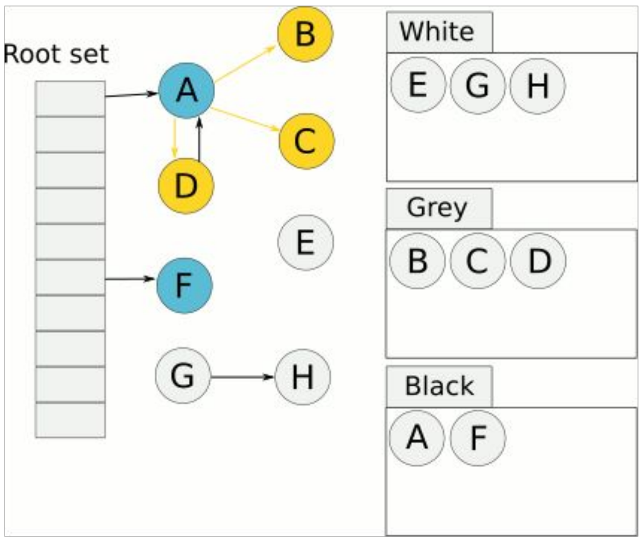
Property: no obj in Black has a ref to objs in White
Thus, at the end Grey is empty → all the objs in White can be swept.

Tri-Color Marking GC - Init



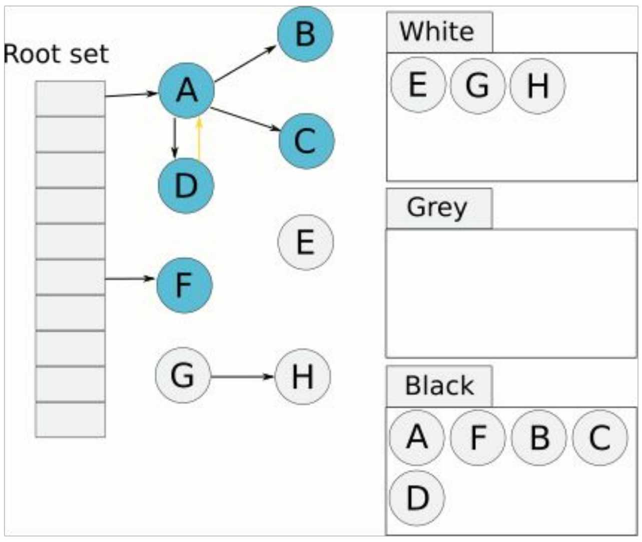
(from WikiCommons)

Tri-Color Marking GC - Iteration 1



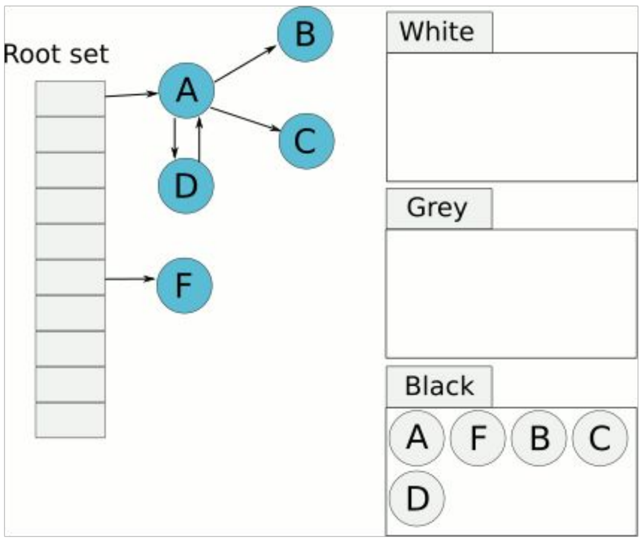
(from WikiCommons)

Tri-Color Marking GC - Iteration 2



(from WikiCommons)

Tri-Color Marking GC - Sweeping!



(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.

Local GC in Java

Local GC is usually implemented using Mark&Sweep approaches.

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

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>)`

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.

… 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".

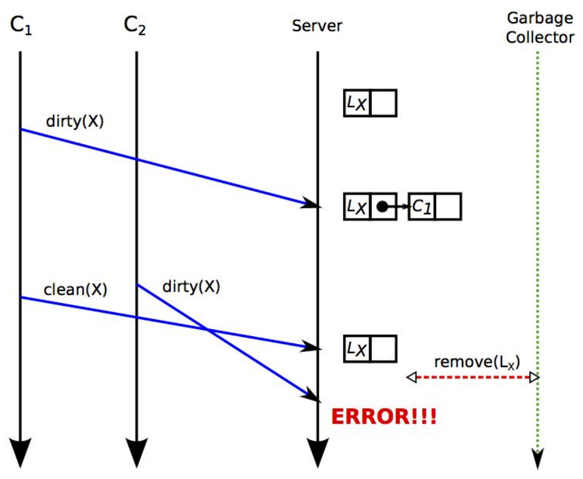
As one list becomes empty and also no local reference is present for the object ➔ it can be removed from the heap.

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.

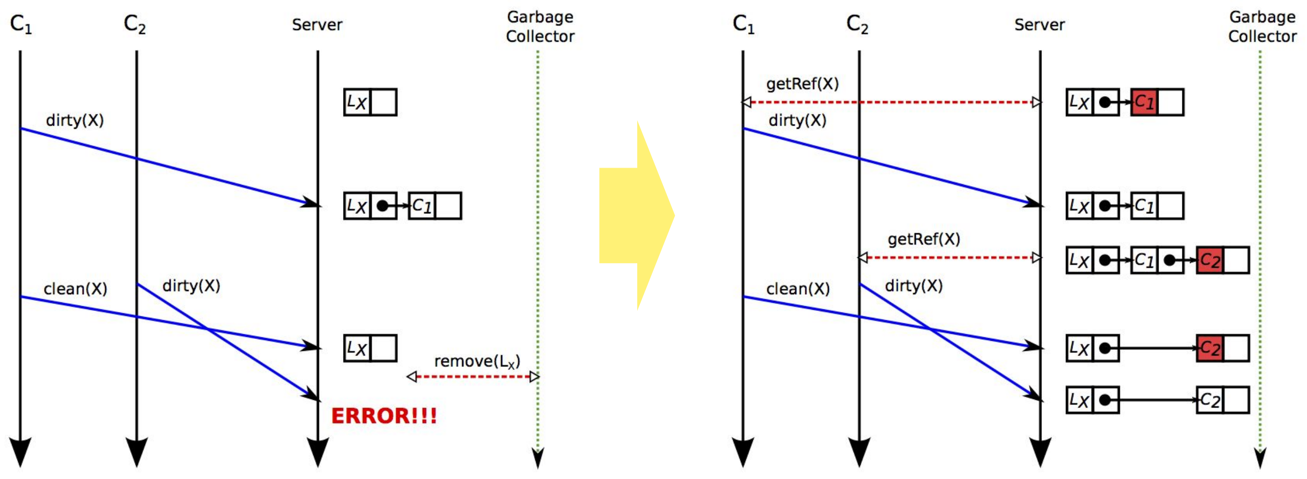
Clean/Dirty Race



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.

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

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

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

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**.

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.

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.

Big Web Services: Architecture

- 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

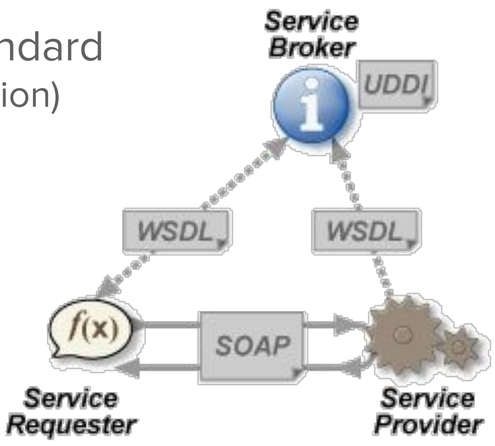


Image from Wikimedia Common

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)

Architectural REST Constraints

- Client-server Architecture
- *Stateless*-ness, i.e. no client context stored on server side.
- Cacheability - clients and intermediaries can cache responses.
- Layered System - no way for a client to tell apart the server and an intermediary.
- Code on demand - possibility to transfer code (e.g. JavaScript...)
- 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.

REST Uniform Interface

Resou operations,
speci

- Pl deletes one
- G e
  in ecoupled
  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

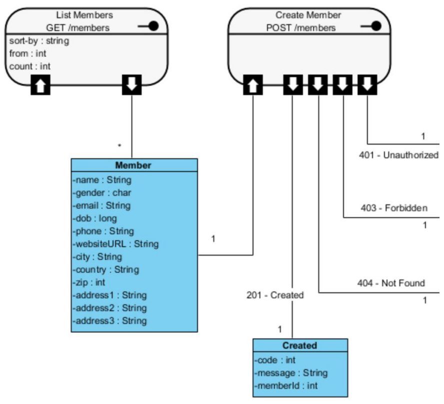
RESTful APIs/interfaces

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



Java RESTful WSs with JAX-RS

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.

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

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.
- 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)

JSON Data Types

- *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"]
}
```

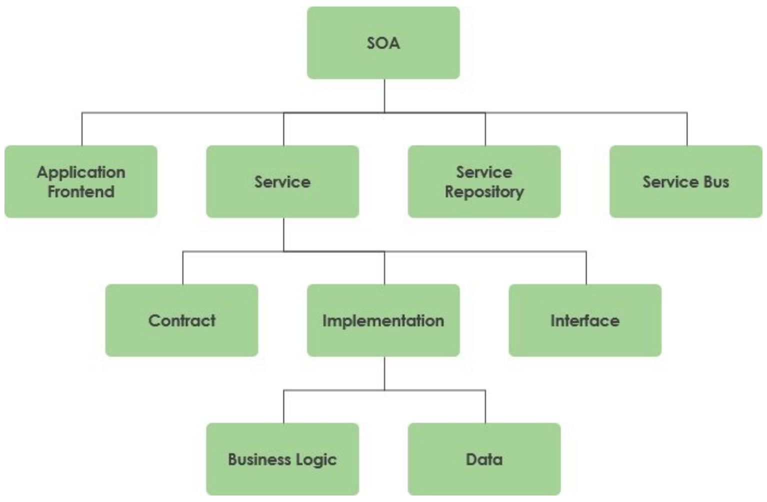
As it can be seen, JSON as XML is self-describing and easy to understand, but supports arrays as well.

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

Not surprising that JSON has gained widespread popularity.

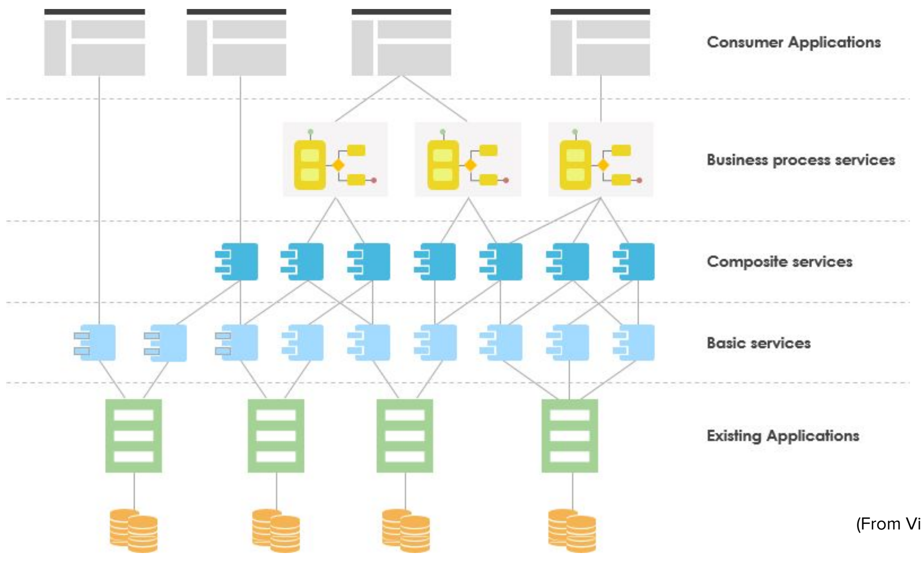
SOA - Service Oriented Architecture (I)

- 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)

SOA - Service Oriented Architecture (II)



(From VisualParadigm website)

Microservices

Microservices

- TODO

Towards JEE Architecture

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...