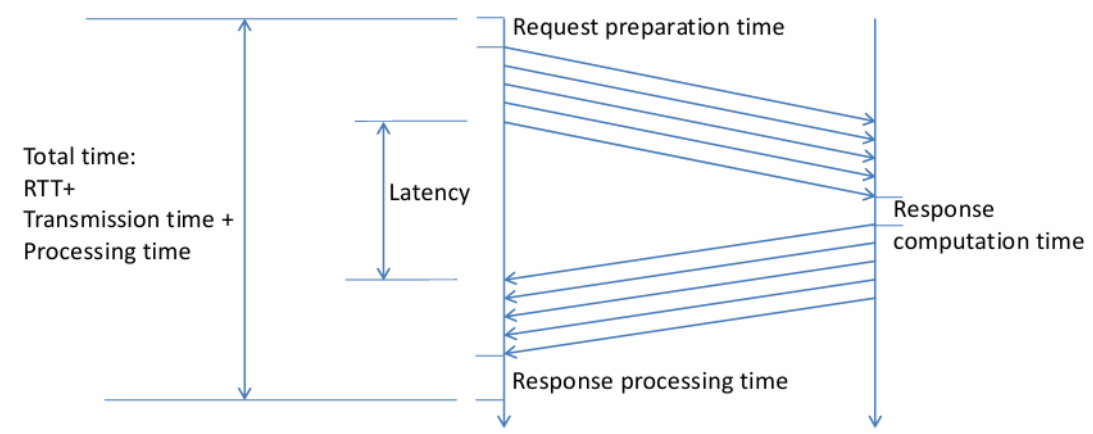
DISTRIBUTED SYSTEMS

0) INTRODUCTION

A **DISTRIBUTED SYSTEM** (**DS**) is a collection of autonomous computing elements that appears as a single coherent system; can include any kind of computer and computing element (**PROCESS/NODE**) that can be interconnected (es. IoT, sensor network, cloud-based system…) [most applications are DS].

Autonomy implies ASYNCHRONOUS (no global clock) and CONCURRENT, HETEROGENEOUS (different HW, OS, programming language, location) and possibility of PARTIAL FAILURE.

Networked interconnections imply VARIABLE TIME and possibility of FAILURE, large attack surface for malicious intruders (so SECURITY is relevant). So, every distributed interaction takes **PROCESSING TIME + COMMUNICATION TIME** (es. UDP):



Processing time << Communication time (RTT: LAN = 1ms, Internet/WAN = 100ms). Here **MINIMUM TIME for typical interactions**:

- for 1-way notification datagram = **RTT/2**
- for opening TCP connection and sending notification = **1.5 RTT**
- for UDP request/response = **1 RTT**
- for opening TCP connection and performing request/response interaction = **2 RTT**

DS General Requirements:

- **DISTRIBUTION TRANSPARENCY** → internal details of a DS should be hidden to users (location of data and computation, data replication, storage/access, failures), but it's not possible to achieve it fully (latency, failures…); this feature is sometimes not convenient/desired
- **DYNAMIC MEMBERSHIP** → there must be ways to add/remove processes and processes should be able to locate other collection elements at runtime
- **SECURITY** → no unauthorized access to data/services (confidentiality, access control); no possibility of interference with the system behavior (integrity, availability)
- **SCALABILITY** → extent to which a system adapts to an increase of its dimensions without losing its performance significantly (size, geographical, administrative…); 2 techniques:

- **Scaling up** = improving the capacity of the computing/storage/communication elements (so only limited)
- **Scaling out** = modifying system's construction or operations:
  - Partitioning and Work Distribution (es. Web, DNS)
  - Replication (es. Cache)
  - Communication Latencies Hiding/Limitation (async communication)

- **OPENNESS** → extent to which a system offers components that can easily be used by or integrated into other systems, using standard/public protocols and APIs (interoperability and portability)

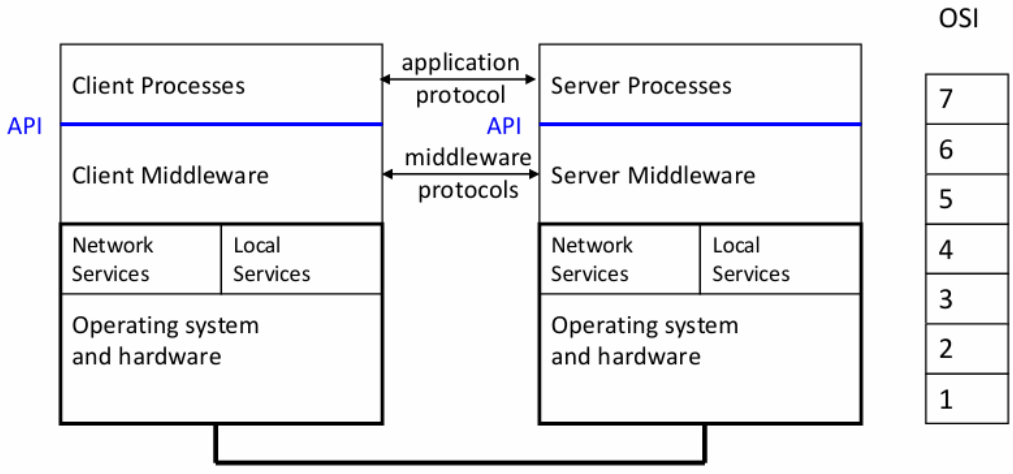
GENERAL SOFTWARE ARCHITECTURE of a DS is based on the **OSI model** and is made of a collection of processes (executed on different network hosts, interacting according to a stack of protocols) [the OSI layers relevant for the DS programmer are the application-oriented layers (5,6,7)]; to reduce complexity, we can use **MIDDLEWARE**s (which stands between application and OS, and solves difficult requirements of distributed software, by providing business-unaware services for coordination and communication [hiding the communication through network, process and host heterogeneity, security issues…]) [examples of middleware are interaction services, services for accessing specific applications, management and administration…]

⚠ Main design goals of DS are **reuse** (use middleware solutions as far as possible) and **information hiding** (build apps relying only on APIs)

Classification of DS Architectures:

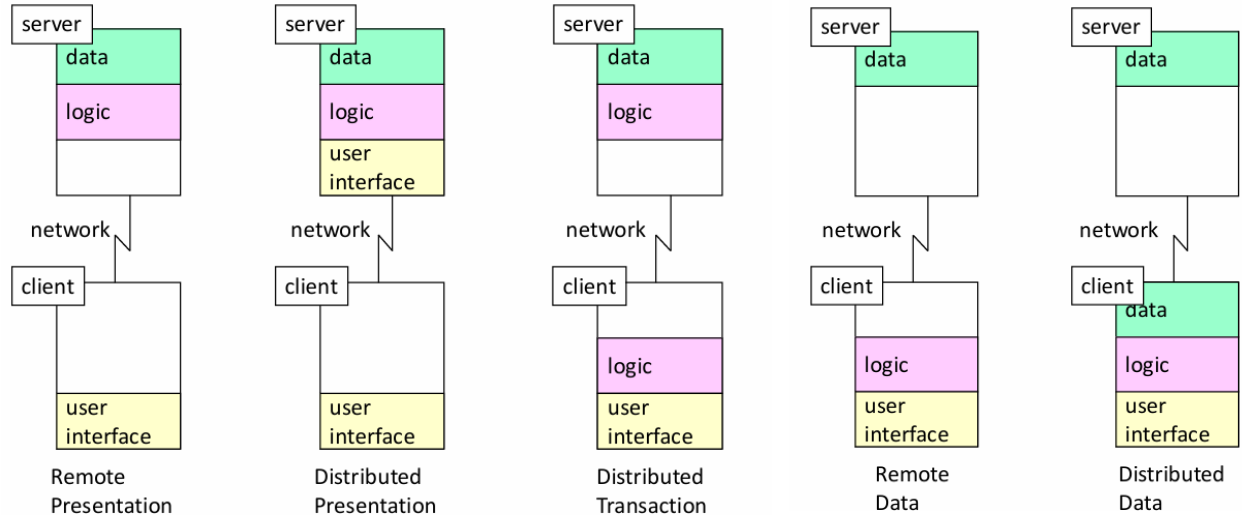
- Client-Server (**C/S**) Interaction Model → centralized organization; each interaction occurs between 2 processes (client and server) and is based on a message exchange (client sends request, server sends back response) [a process can be client in an interaction, and server in another interaction]
  A **Client-Server Architecture** is as described, but the processes are always divided in 2 fixed classes (only client and only server, not mixed; **many clients – 1 server**); reliability, simple
- **Peer-to-Peer** (**P2P**) **Architecture** → all processes identical peers, that can be both client and server; fully decentralized; complex, less control

This is a **C/S Software**:

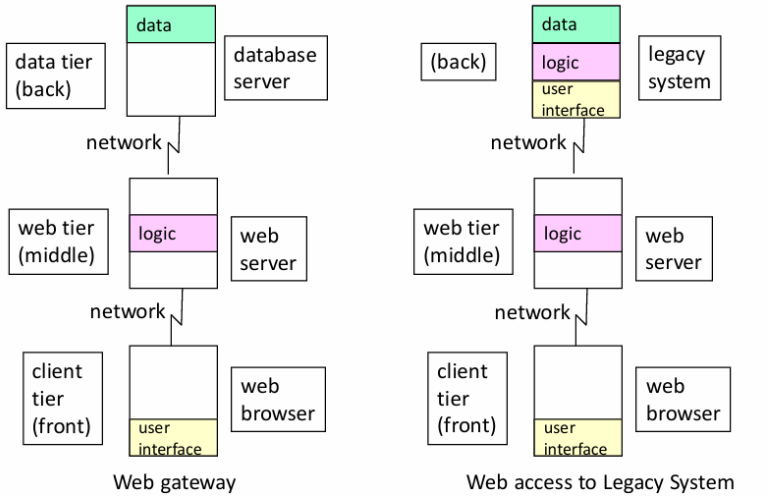


Each application can be **logically divided into 3 main parts** (logical tiers): user interface tier, processing tier, data tier. In **C/S** systems, these logical tiers are **mapped onto different processes running on different hosts** (physical tiers) [**VERTICAL DISTRIBUTION**]:

- **C/S 2-tier Architectures** = only 2 physical tiers (1 client = frontend, 1 server = backend); classification (Gartner Group):



- **C/S 3-tier Architectures** = > scalability, > openness/flexibility thanks to the introduction of an intermediate physical tier (*middle machine*) [that can be used for workload balancing, filtering…]



- **C/S multi-tier Architectures** = for further flexibility and for complex systems

In **P2P** systems, all peers perform the same tasks, but on their own share of the data set [**HORIZONTAL DISTRIBUTION**]. A P2P system is built by its peers which create an **OVERLAY NETWORK** (each peer can communicate only with some other peers); 2 types of overlay networks:

- STRUCTURED → overlay built with a deterministic/regular topology (es. esagono, cubo…)
- UNSTRUCTURED → each peer connects to some other peers in a random way (es. flooding, random walks, policy-based search)

Other P2P systems are (es. Skype and BitTorrent):

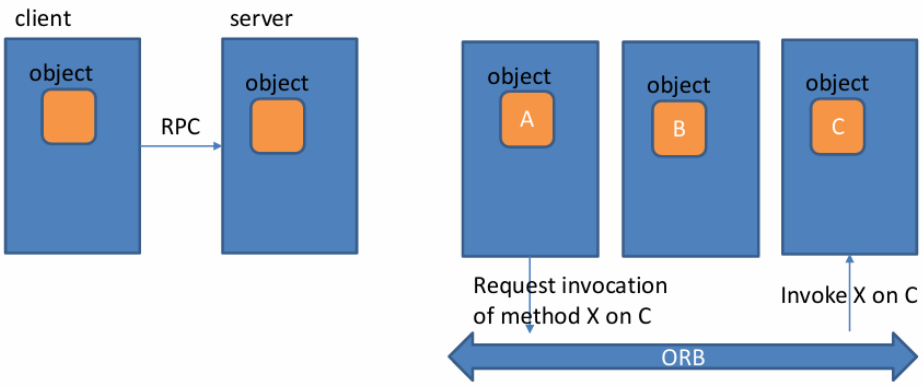
- HIERARCHICAL = peers are not all the same (normal peers connected to super peers)
- HYBRID = combine C/S and P2P

Abstractions for DS (classified in terms of coordination [*coupling*]):

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct (RPC)	Mailbox-based (MOM)
Referentially decoupled	Event-based (MOM)	Shared data

These abstractions are implemented by middlewares/protocols/APIs, but in some cases "**HIGHER-LEVEL abstractions**" are also used:

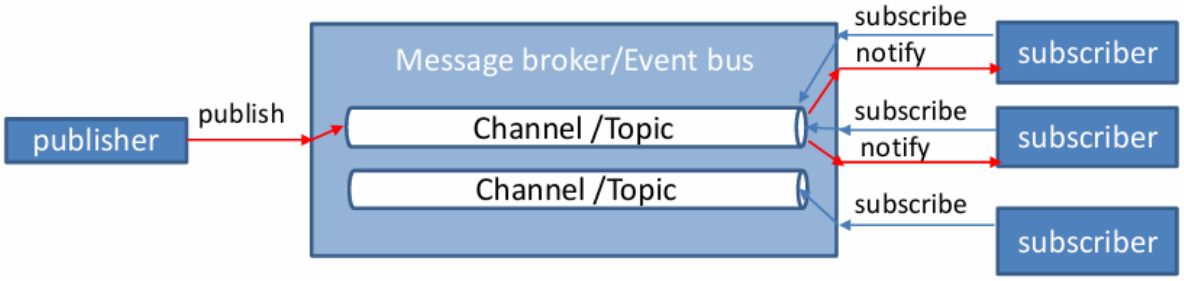
- **RPC-based**
  - Distributed Object Systems = transfer the object model in a distributed environment: objects of a single app can live in multiple hosts connected by a net, but can be used as they were local



• Example: RMI Example: CORBA

  - Distributed Services (and Web Services) = similar to Distributed Objects, but services:
    - have larger grain
    - are autonomous and long-living
    - may be made available on a marketplace (enabling service composition)
  - Distributed Resources = similar to Distributed Services, introduced with REST and HTTP

- **MOM-based**
  - Publish-Subscribe Systems = C/S paradigm that achieves referential decoupling:
    - processes don't know each other
    - clients can only do 2 operations: Publish (generate msg/event) or Subscribe (express interest for a class of msg/events)
    - servers (msg broker/event bus): notify each published msg to all subscribers; can be centralized (MQTT), decentralized (DDS) or cloud-based (Amazon SNS)



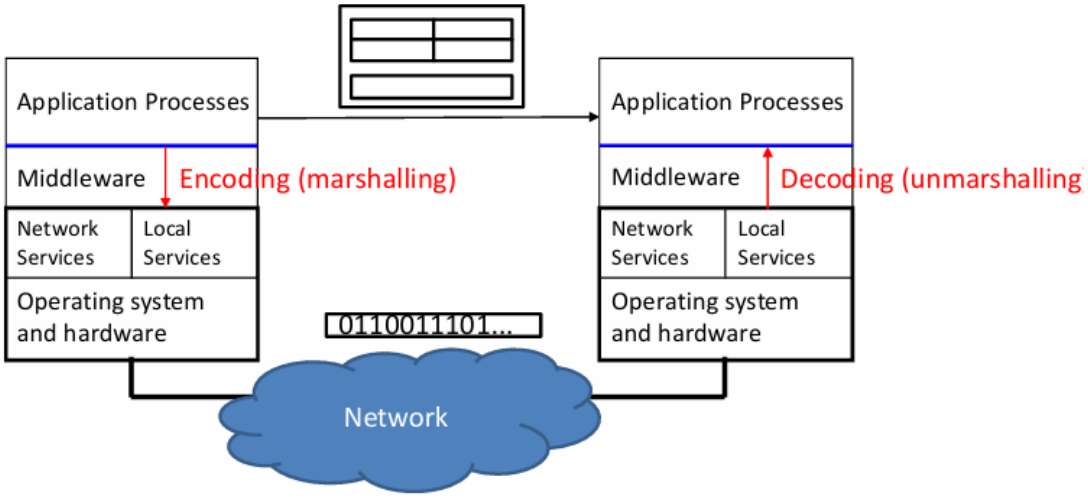
  - Message Queues = distributed implementation of a queue (clients can queue/dequeue msg); asynchronous [es. JMS, AMQP…]

There are also "**LOWER-LEVEL abstractions**", the **SOCKETS** (offer elementary communication primitives and max flexibility [can be used to implement user-defined abstractions and protocols]) [es. TCP/IP Sockets, WebSockets…]

MIDDLEWARE Architecture: middleware can be organized as a set of components, each one providing some services accessed by an API; services offered by a component can be used by other components to build complex services (basic services used by all middleware components are network and transport layer communication facilities). Multiple middleware components can be combined in **FRAMEWORKS**.

1) Abstract Syntaxes and Schemas

DATA TRANSPARENCY = protocols must ensure that data are correctly transferred despite systems are **heterogeneous**, and this is obtained by properly encoding and decoding data, according to a protocol:



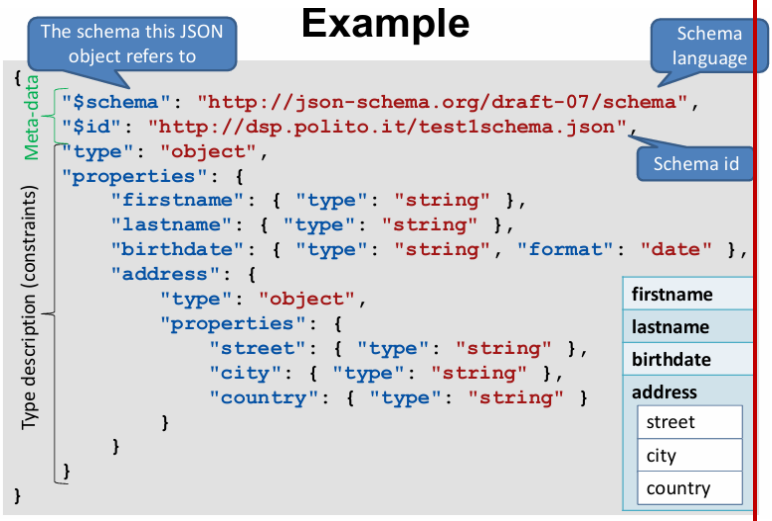
There are different types of codes: **binary** (01010110) and **character-oriented** (GET http://...). Codes are defined so that the receiver can *separate*, *validate* and *decode* messages. The HTTP solution (character-oriented) for example use different codes, initially the code is plain ASCII and the code used in the body is communicated/negotiated in the header.

Applications typically exploit **standard** ways for **system-independent** data representation:
- **Binary** solutions → ASN.1, XDR …
- **Character-oriented** solutions → XML, JSON, YAML …

These standards usually include:
1. a language to define **ADT** (Abstract Data Types) [or "**Abstract Syntax**"]
2. neutral (system-independent) data representations for any ADT that can be defined by the language

⚠ Why Abstract Syntaxes? **Language/system independency** (programmers can use any programming language and libraries translate automatically between language data types) + **independency from binary encoding** [different binary codes can be used for the same abstract data]. Abstract Syntaxes can be also used to specify/validate message syntax (es. programmers can delegate validation to standard validators [es. express-validator WebApp1])

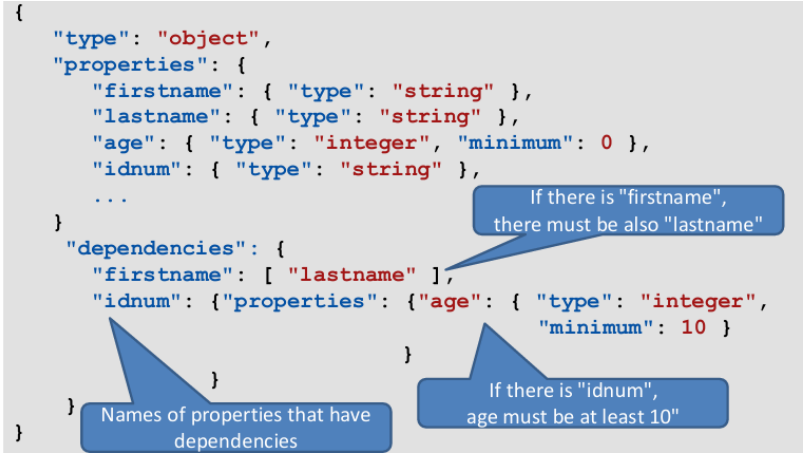
JSON Schema is an Abstract Syntax for character-oriented representations; it's based on JSON itself: in fact, **a schema is a JSON object with special meaning** [machine-readable, serializable and can be validated] **which defines a data type** (cioè a set of valid JSON values) [JSON data types = types available in any programming language (primitive types) → string, boolean, number, null, arrays, objects]



⚠ Quindi in pratica lo schema è una sorta di classe/prototipo che possiamo definire e linkare con "$schema" nelle varie istanze/oggetti dello schema per fare in modo che si possano checkare che seguano lo schema definito

A **schema** is composed by:

- **object** → includes:
  - **meta-data**:
    - Annotations [`title`, `description`, `default`, `examples`]
    - Comments [`$comment`])
  - **constraints** (on what is valid) → **type** of the data structure:
    - **string** (unicode) → additional type-dependent constraints:
      - `minLength`
      - `maxLength`
      - `format` (date, time, email, ipv4…)
      - `pattern` (regexp to follow)
    - **number** (real number) → additional type-dependent constraints (<u>also for `integer`</u>):
      - `multipleOf`
      - `minimum`
      - `exclusiveMinimum`
      - `maximum`
      - `exclusiveMaximum`
    - **integer** → // //
    - **object** (collection of properties) → additional type-dependent constraints:
      - `properties` (**`key: {value}`**)
      - `additionalProperties`: se *non specificata*, ogni additional property della singola istanza rispetto allo schema è valido; se *specificata* come proprietà:
        - se **`true`** → ogni additional property è valida
        - se **`false`** → nessun additional property è valida
        - se **`uno schema`** → ogni additional property deve obbedire allo schema
      - `required`: if *not specified*, all properties are <u>optional</u>; if *specified*, its value is the <u>array</u> with the names of the required properties
      - `propertyNames`: if *not specified*, all property names are admitted; if *specified*, its value is the schema with the names to obey for properties
      - `minProperties`
      - `maxProperties`
      - **`dependencies`**: if *specified*, its value is the object whose properties express the dependencies; 2 types of dependencies:
        - `property dependencies` (presence of a property means presence of some other properties)
        - `schema dependencies` (presence of a property means the schema must be extended with certain additional constraints)



- **array** (ordered list of typed values) → additional type-dependent constraints:
  - minItems
  - maxItems
  - uniqueness (if true, items must be unique)
  - items:
    - if value is **a schema**, each item must obey the schema
    - if value is **an array of schema objects**, each item must obey the corresponding schema
  - additionalItems:
    - if **true** → any additional items allowed after tuple
    - if **false** → no additional items allowed after tuple
    - if **a schema** → additional items obeying it allowed
  - contains = if **a schema**, 1 or more items must obey the schema
- **boolean**
- **null**
- **enum** → specifies the allowed values by enumeration using JSON syntax:

```
{
   "type": "object",
   "properties": {
      "firstname": { "type": "string" },
      "lastname": { "type": "string" },
      "sex": { "enum": [ "male", "female" ] },
   }
}
```

- **true** → everything is valid; **false** → nothing is valid

SCHEMA COMBINATION = a schema can be the result of a combination of schemas:

- **allOf** → requires validity against all sub-schemas
- **anyOf** → requires validity against at least 1 sub-schema
- **oneOf** → requires validity against exactly 1 sub-schema
- **not** → requires not validity

```
{
   "anyOf": [
      { "type": "number", "multipleOf": 3},
      { "type": "number", "multipleOf": 5 }
   ]
}
```

Schemas can be combined in a **conditional** way:

```
{
   "if": { "type": "integer", "multipleOf": 3},
   "then": { "type": "number", "multipleOf": 5},
   "else": { "type": "number", "multipleOf": 7},
}
```

S_{if} S_{then} S_{else}

Schemas can also be reused by **reference** (both internal and external schemas):

- Reusing internal schema using JSON pointer

```
{
  "definitions": {
    "address": {
      "$id": "#definitions/address",
      "type": "object",
      "properties": {
        "street": { "type": "string" },
        "city": { "type": "string" },
        "country": { "type": "string" }
      }
    }
  },
  "type": "object",
    "properties": {
      "orderDate": { "type": "string", "format": "date"},
      "quantity": { "type": "integer" },
      "shippingAddress": { "$ref": "#definitions/address" },
      "billingAddress": { "$ref": "#definitions/address" },
      ...
    }
}
```

- Reusing external schema

```
{
"type": "object",
    "properties": {
      "orderDate": { "type": "string", "format": "date" },
      "quantity": { "type": "integer" },
      "shippingAddress": { "$ref": "addressschema.json" },
      "billingAddress": { "$ref": "addressschema.json" },
      ...
    }
}
```

addressschema.json

```
{
  "$id": "#address",
  "type": "object",
  "properties": {
    "street": { "type": "string" },
    "city": { "type": "string" },
    "country": { "type": "string" }
  }
}
```

⚠ Extension of a schema can be achieved by combining a referenced schema with another

Sintassi di "**Schema Validation** with Express/ajv":

```
var { Validator, ValidationError } = require('express-json-validator-middleware');
var userSchema= JSON.parse(fs.readFileSync(schemaFileName));
var validator = new Validator({ allErrors: true });
validator.ajv.addSchema(userSchema);
var validate = validator.validate;
```

```
app.post('/user', validate({ body: userSchema }) , (req, res) => {
...
}
```

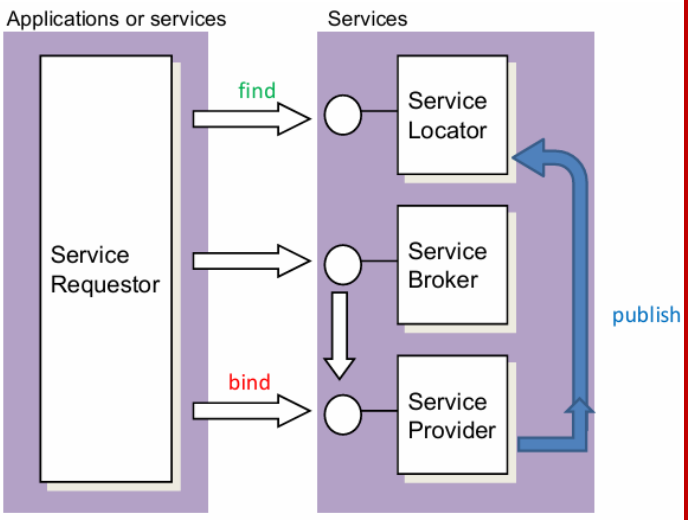
```
app.use(function(err, req, res, next) {
  if (err instanceof ValidationError) {
    res.status(400).send('Invalid request');
    next();
  } else next(err);
});
```

2) Web Services and REST Architecture

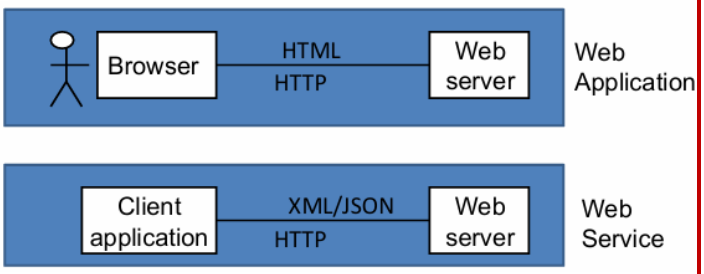
Distributed Services = abstraction similar to distributed objects, but services have larger grain, are autonomous and long-living, are available for use by different clients from different organizations.

SOA (**Service Oriented Architecture**) = software organized as a set of services, each provided through interfaces (that are published, automatically discoverable and machine readable). A SERVICE is:

- described by a contract (made of interfaces)
- implemented by a single instance (always available)
- coarse grained
- loosely coupled (interactions by message exchanges)



Web Services [**WS**] (**API**s) are distributed services based on web (HTTP); can be SOA-based or RESTful (as you see from image, Web Services are different from Web Application, because the Client is a Client Application [so, machine-to-machine interaction]). In fact the RESTful API is an example of web service!

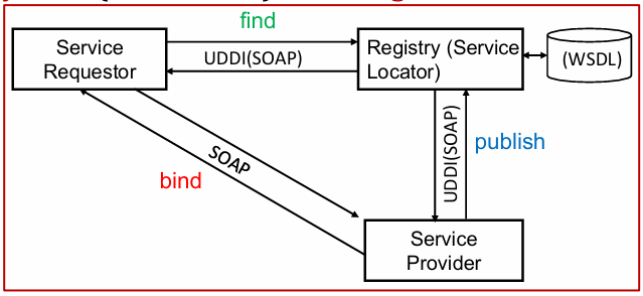


Scenarios:

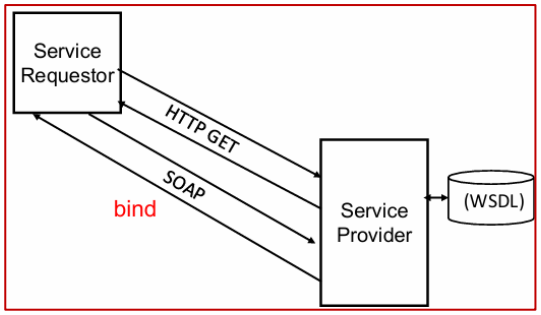
- SaaS (Software as a Service, made available through the network as web services)
- whole applications made available as web services (es. a whole information system process)
- integration of services to create added-value new services
- web services marketplace

WS LifeCycle: Build (design + implementation) → Deployment (installation) → Management → Dismiss

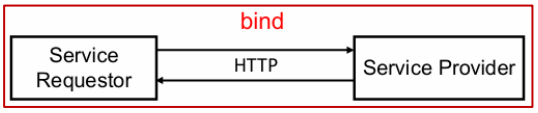
SOA-based WS (SOAP/Standard WS) →



But the **SOAP/Standard WS** "**Actual Simplified Model**" →

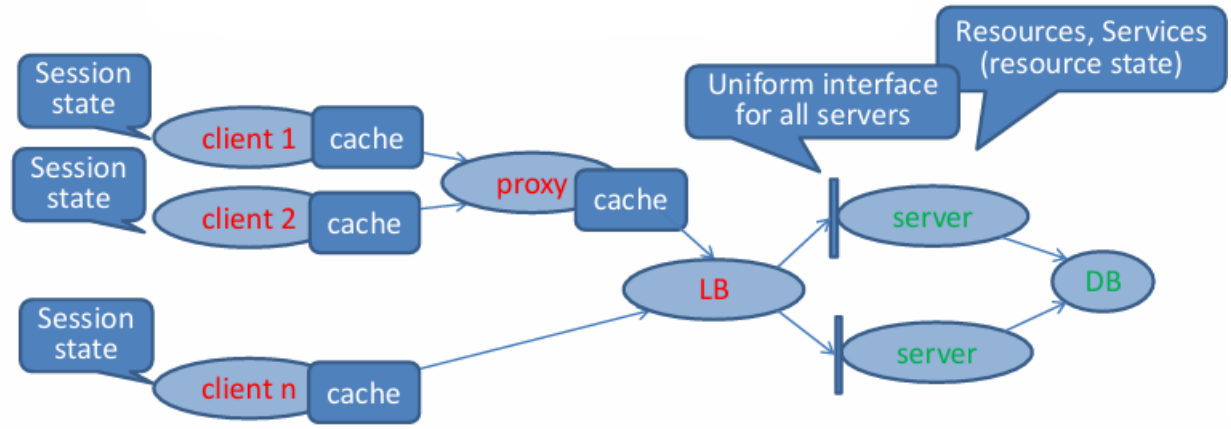


Here instead on **REST Architectural style** (RESTful WS) →

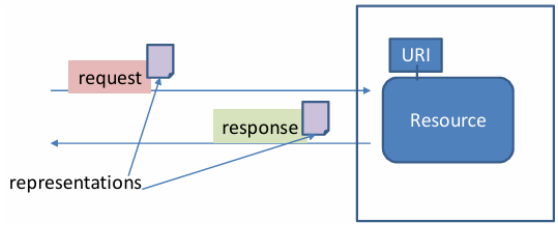


REST (REpresentational State Transfer) is implemented by HTTP protocol and the web; it's based on a typed request/response messaging protocol (es. HTTP); **CONSTRAINTS** (*):

- client/server → separation of roles for simplicity, scalability and security
- stateless → session state entirely on the client (> visibility and monitoring, > reliability [easy recoveries from partial failures] and > scalability [server don't need to store session info])
- cacheable → responses can be labeled as cacheable (can be cached by clients) or non-cacheable [> efficiency and performance]
- layered → enabling multi-tier architectures (vertical layering = each layer interacts only with adjacent layers) [> simplicity to add functions dynamically]
- with fixed (uniform) interface → obtained by setting additional constraints on interface organization; uniform interface means that server interfaces are resource-oriented [key abstraction = resource] and protocol enables transfer of representations with metadata
  Resource Example: il Politecnico di Torino, URI: http://www.polito.it; possible representations are:
  - HTML home page (content-type: text/html)
  - a picture of PoliTO (content-type: image/jpg)



When a client requests an operation on a resource, a representation of the resource may be transferred (but not the resource itself, only a representation of it, because resources stay always in the server side, where their URI points to).



HTTP Protocol offers a fixed uniform interface for **requested operations** on resources, with structure:

- requested method
- request body
- request headers (control metadata)
- target URL (including any query string)

Result of the operation is communicated in the **response** (via status code, response body and response headers).

CRUD Operation	HTTP request	HTTP response (if no error)
Create a new resource under *res* (or send data to *res* for resource-specific processing)	POST *res* (resource or data representation in the body)	URL of new resource created and/or result of processing (in body)
Read resource *res*	GET *res* (no body)	representation of current state of *res* (in the body)
Update resource *res* by replacing its state with a new one (*res* can be created)	PUT *res* (new resource representation in the body)	description of executed operation (200 or 201 or 204 status code)
Delete resource *res*	DELETE *res*	

⚠ GET and DELETE carry parameters in their query string (no body)

Apart from *CRUD* operations, there also other operations:

Operation	HTTP request	HTTP response (if no error)
Read information about resource *res* without transferring its representation	HEAD *res* (no body)	same as for GET but without body (only headers sent)
Read supported methods for a resource	OPTIONS *res* (no body)	list of supported methods (in the Allow header and possibly in the body)
Test the connection to resource *res* (loopback testing similar to echo)	TRACE *res*	the received request (in the body)
Partially Update the resource *res* by applying a patch	PATCH *res* (patch to be applied to *res* in the body)	description of executed operation (200 or 201 or 204 status code)

REST interface is powerful enough to represent any set of operations: REST WS are semantically as expressive as SOAP WS

⚠ Come abbiamo potuto vedere però, REST e SOAP sono opposti:

- REST → few fixed operations, many resources
- SOAP → few fixed resources, many operations

Example (Pizza Shop Service):

Customers can read the menu (pizza types and available toppings)

Customers can submit pizza orders
- pizza types, quantities, toppings,...

Pizza orders are processed by pizza makers

Request	Meaning
GET *menu*	Read the menu(getMenu operation)
POST *submittedOrders* with Order element as body	Submit an order (submit operation) Side effect: order is added to submittedOrders
POST *consumedOrders* with empty body (response includes Order)	Consume next order (getNextOrder operation) Side effect: oldest order is moved from submittedOrders to consumedOrders

⚠ Machine-Readable Description of RESTful WS → not globally accepted standard yet, but examples:

- WADL [description of resources + admitted methods and parameters for each]
- WSDL 2.0 [standard, but more limited]
- ***SWAGGER*** (**OpenAPI**) [quello che abbiamo usato per GeoControl a SW1]

⚠ **HATEOAS**: ideally, a RESTful WS should operate like a **traditional WebApp** (*Un client dovrebbe poter navigare l'intera API seguendo solo i link forniti dal server, senza dover costruire manualmente gli URI*) [es. quando ricevi un oggetto film dal server, invece di dover sapere che per vedere il proprietario devi costruire l'URL /api/users/{owner}, il server ti fornisce direttamente il link] [`self` hyperlink points to resource itself, `next` hyperlink is used when navigating to the next resource in a sequence]

⚠ *Richardson Maturity Model* → ranking WS about how they are compliant to REST constraints (*)[0-3]

3) Service Interface Design

Interface Design is part of the **software design phase** (in the architecture definition, where we define modules and their interfaces); a basic principle is ***Information Hiding*** (separation between interface and implementation → show no more than what is strictly needed by the service client), then we have to answer some questions:

- How to partition the system in modules? We have to maximize internal cohesion and minimize coupling, while balancing the load of the various distributed sub-systems; we have to be open to future changes and to reuse. So we need to find the right trade-off for **GRANULARITY**:
  - **too fine** grain partitioning → < performance, > coupling
  - **too coarse** grain partitioning → < internal cohesion, < reuse and flexibility
- How to design interfaces?

Interface Design must produce a formal description of interfaces; must be coherent with the partitioning principles (scritti sopra) but also with some other best practices:

- foresee all particular cases (exceptions)
- prefer idempotent methods
- limit number of interactions (but also the size of messages)

Interface Design Approaches:

- **Method** centric = fixed endpoint; design is about operations and their input/output arguments
- **Message** centric = fixed, single-operation interface; design is about messages and endpoints
- **Constrained** (***REST***) = fixed, multiple-operation interface; design is about endpoints, allocation of operations and their I/O arguments

ESEMPIO (bank account operations design)

Method-centric Design (2nd try)

```
public interface ImprovedAccountUpdater {
       public void addDeposit(
                     String id,
                     float amount,
                     String description,
                     String transId)
              throws ReplicatedTransIdException,
                     UnknownAccountIdException,;

       public float addWithdrawal(
                     String id,
                     float requestedAmount,
                     boolean reducible,
                     String description,
                     String transId)
              throws ReplicatedTransIdException,
                     UnknownAccountIdException,
                     NoAvailabilityException;
}
```

REST Service Design (2nd try)

Designing Resources

Resources	URLs	Repr	Meaning
* accounts	accounts		set of all accounts
1 {id}	accounts/{id}	account	single account
* operations	accounts/{id}/operations		set of all operations
1 {tid}	accounts/{id}/operations/{tid}	operation	single operation

Designing Operations

Resource	Method	Req. body	status	Resp. body	meaning
accounts					
accounts/{id}	GET		200	account	OK
			404		Not found
accounts/{id}/operations /{tid}	PUT	operation	201	operation	Created
			409	reason	Conflict
			404		id Not found

Design approach influences syntax, but most design principles apply to all design approaches because they are related to the underlying service features (*universality of design principles*).

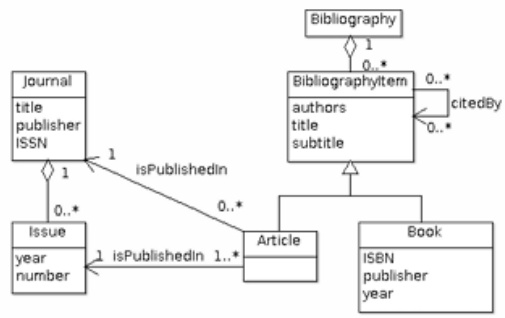
When designing RESTful API, there are some issues to be considered (how to organize resources, how to map operations to methods, how to provide documentation…); there are design patterns to follow.

ESEMPIO → **design a RESTful WS for management of a bibliography**, in which you can:

- Search a bibliography:

- search items by keyword, type and publication year
- for each item, get available data
- navigate through items by citation

- Populate/Update a bibliography:
  - add new items to a bibliography (also related data)
  - modify/delete item in a bibliography

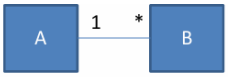


STEP1) **Designing Resources** → start with a **conceptual** resource design (independent of URIs and representations [because it should be possible to remap the resources to different URIs]); resources correspond to single entities (or collections of entities of the same type); resources are unique, but different resources can represent related/overlapped entities [resources = nomi; actions = verbi]

Resources	Meaning
biblio	The bibliography (main resource)
biblio/items	The items in the bibliography
biblio/items/{id}	The item uniquely identified by {id}
biblio/journals	The journals in the bibliography
biblio/journals/{issn}	The journal with ISSN {issn}
biblio/journals/{issn}/issues	The issues of the journal with ISSN {issn}
biblio/journals/{issn}/issues/{iid}	The issue with id {iid} of the journal with ISSN {issn}

STEP2) Represent **Resource Relations**? Different mechanisms:

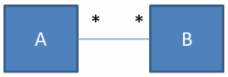
- Example: 1 to many



  - option 1:
    - B data model includes reference to A

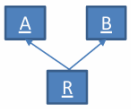
  - option 2 (better when B not autonomous):
    - B instances (or references) nested into A instances

- Example: many to many



  - option 1:
    - B data model includes references to A
    - A data model includes references to B

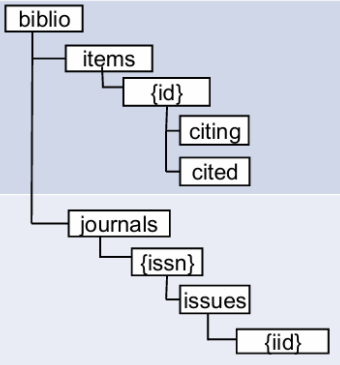
  - option 2:
    - represent the elements of the relation, i.e. (A,B) pairs, as resources



Resources	Meaning	Relative URLs
biblio	The bibliography (main resource)	biblio
biblio/items	The items in the bibliography	biblio/items
biblio/items/{id}	The item uniquely identified by {id}	biblio/items/{id}
biblio/items/{id}/citing	The items that cite the item identified by {id}	biblio/items/{id}/citing
biblio/items/{id}/cited	The items that the item identified by {id} cites	biblio/items/{id}/cited
biblio/journals	The journals in the bibliography	biblio/journals
biblio/journals/{issn}	The journal with ISSN {issn}	biblio/journals/{issn}
biblio/journals/{issn}/issues	The issues of the journal with ISSN {issn}	biblio/journals/{issn}/issues
biblio/journals/{issn}/issues/{iid}	The issue with id {iid} of the journal with ISSN {issn]	biblio/journals/{issn}/issues/{iid}

⚠ Si usano convenzioni per rendere più facile scrivere/comprendere le API: usare nomi plurali per le collezioni, singolari per non-collezioni; usare URL (**uri**) per le relazioni gerarchiche tra risorse (per non mettere ogni risorsa dentro ogni sua reference, ma usare uri per farne riferimento) [**hyperlinks**]

STEP3) **Representing Resource** → define an abstract data model for each resource (in practice, the resource state; representable in UML, schema…) and the different representations of the resource data model (different content types); + hyperlinks as said above (aggiungendo **`"format" : "uri"`**)

Resources	Representation (schema)
	biblio items item items items journals journal issues issue

For each operation to be offered by the service, find resources and HTTP methods to be used; for each resource/method pair, define how the method has to invoked (accepted query parameters, request headers and body contents) and the possible results (status codes) [and for each of them, response headers and body contents]

ESEMPIO (continuazione di BIBLIO):

- Search items by keyword, type (article/book) and publication year
- For each item, get available data

Resource	Verb	Query params	Status		Resp. Body
biblio/items	GET	keyword: string type: { "enum": ["article", "book"] } beforeInclusive:date afterInclusive:date	200	OK	filtered items (items)
biblio/items/{id}	GET		200	OK	item
			404	Not found	

- Navigate through items by citation

Resource	Verb	Query params	Status		Resp. Body
biblio/items/{id}/citing	GET		200	OK	Items that cite {id} (items)
			404	Not Found	
biblio/items/{id}/cited	GET		200	OK	Items that {id} cites (items)
			404	Not Found	

- Add new items to a bibliography
- Modify or delete an item in a bibliography

Resource	Verb	Req. body	Status		Resp. body
biblio/items	POST	item	201	Created	item
			400	Bad Request	string
biblio/items/{id}	PUT	item	204	No Content	
			400	Bad Request	string
			404	Not Found	
biblio/items/{id}	DELETE		204	No Content	
			404	Not Found	

- Modify an item in a bibliography (add citations)

Resource	Verb	Req. body	Status		Resp. body
biblio/items/{id}/citing	POST	item	204	No content	
			400	Bad Request	string
			404	Not Found	
			409	Conflict	string
biblio/items/{id}/cited	POST	item	204	No content	
			400	Bad Request	string
			404	Not Found	
			409	Conflict	string

- Missing operations (e.g., add journals and issues left as exercise)

⚠ Common best practices for responses that may be too large: query parameters (to let users decide what to get), paging and using references (instead of full representations)

⚠ **HTTP CONDITIONAL REQUESTS** (let clients specify that an operation should be done only in some conditions):

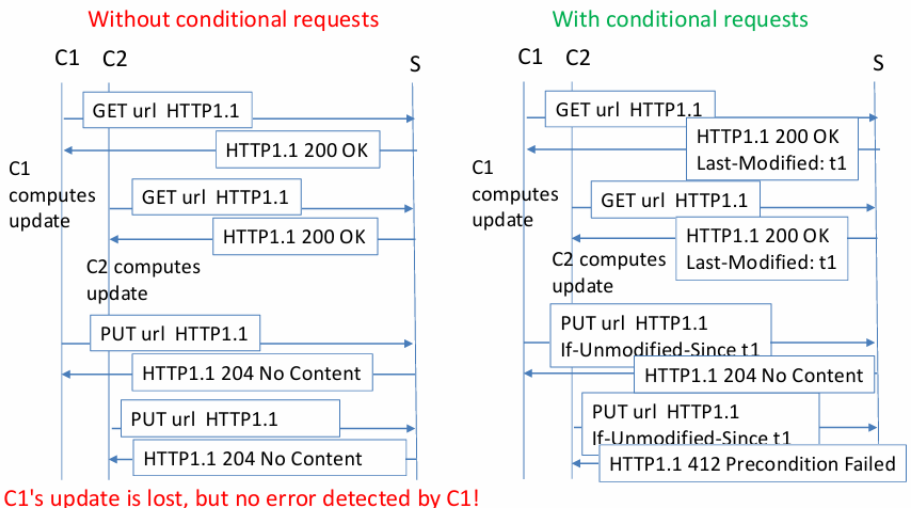
Header	Condition for executing
If-Modified-Since	The Last Modified date of the resource is more recent than the value of this header
If-Unmodified-Since	The Last Modified date of the resource is older than or same as the value of this header
If-Match	The Etag of the resource is equal to one of the values listed in this header
If-None-Match	The Etag of the resource is different from each of the values listed in this header

Queste CONDITIONAL REQUESTS possono essere usate (come abbiamo già visto) in una situazione come quando faccio GET su una risorsa nella CACHE del client: se cambiata la risorsa rispetto la cache,

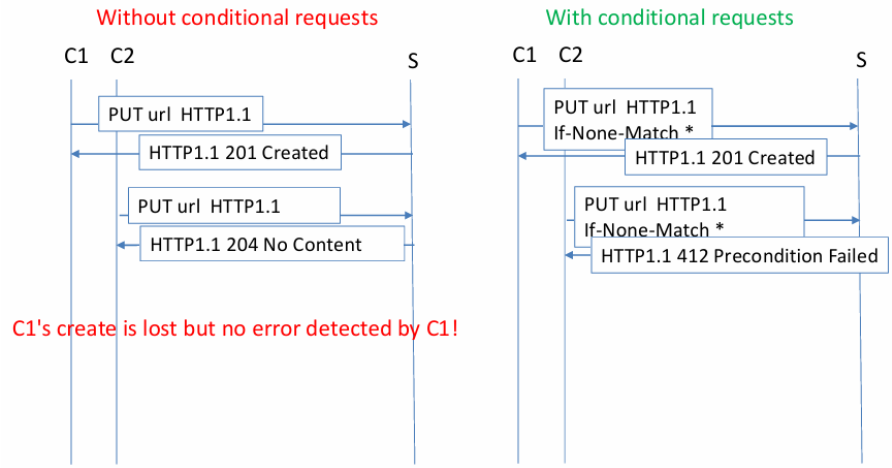
con if-modified-since, faccio **refresh** della risorsa in cache; se non è cambiata, mi **restituisce** la risorsa in cache.

È importante però evitare **RACE CONDITIONS** di tipo:

- GET-PUT (read-update):

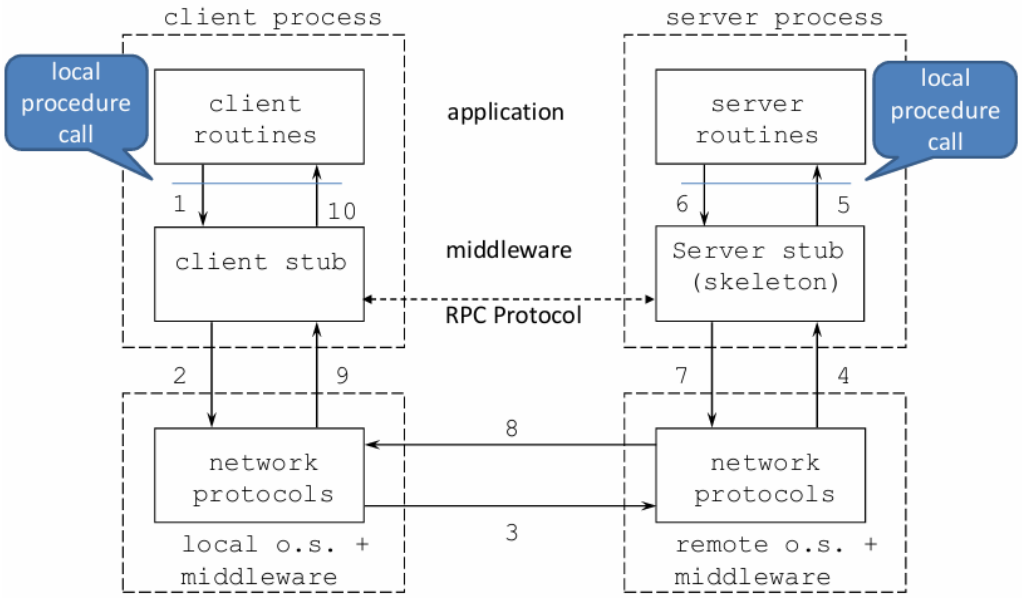


- PUT (create):

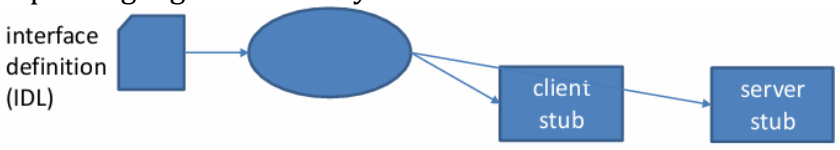


4) RPC (Remote Procedure Call), HTTP/2 and gRPC

RPC is the transfer of the local procedure call mechanism to a distributed environment (es. JAVA RMI for Java, CORBA [distributed objects], WS [distributed services], REST [distributed resources], **gRPC**):



RPC Middleware: RPC is offered as a feature of a programming language (es. JAVA RMI), exploiting language-specific serialization; it has high transparency, but limited to single-language apps. Stubs can be generated for multiple languages in this way:



Issues to be handled by the RPC Middleware:

- caller and callee are heterogeneous (different languages/sw/hw...) → procedure arguments and return values must be converted (*marshaling/unmarshaling*)
- caller and callee run on different processes and hosts:
  - server must be located before issuing the call (*dynamic linking*)
  - multiple incoming calls must be *synchronized* on the server
  - the way arguments can be passed by reference must be defined (*address spaces are disjoint*)
  - *partial failure* possible, so must be managed

Remote call semantics differ from local call semantics (as we said above, a remote call may fail [server crashes, network outages] or may be executed more than once [msg duplications]), but **RPC middleware can handle these problems only partially** (quindi attenzione programmatori!!!) [usually, only 2 kinds of return are possible → call *successful*, call *unsuccessful*]
According to what RPC protocol is used, different call semantics can be obtained → **call semantics**:

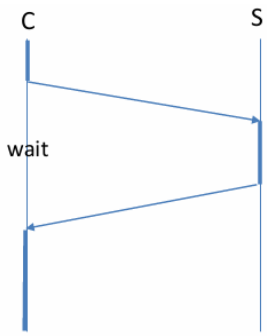
- **`at least once`**:
  - upon *successful* return = at least once semantics
  - upon *unsuccessful* return = no guarantee given (procedure may been executed 0 or + times)
  - procedures must be idempotent (same results independent of times of execution)
  - transport = UDP → RPC protocol includes response-request association + retransmission management
- **`at most once`**:
  - upon successful return = exactly once semantics
  - upon unsuccessful return = at most once semantics
  - transport:
    - TCP → RPC protocol associates response to request via the connection
    - UDP → RPC protocol includes response-request association + retransmission management + duplicate checking at server
- **`exactly once`**

Other RPC issues:

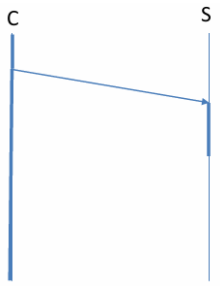
- SECURITY → access control, authN, confidentiality, integrity
- PERFORMANCE: remote call << speed than local call

Types of RPC:

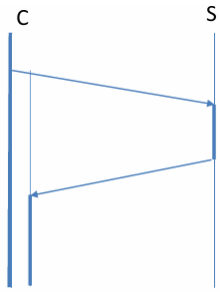
- **Classical** (**Synchronous**) RPC
  - *request*:
    - procedure identification
    - input parameters
  - *response*:
    - success/failure indication
    - return value + output parameters (or exceptions)



- **1-Way** (**Asynchronous**) RPC
  - *request* = // //
  - *response* = no output (no return, output parameters or exceptions)



- **2-Way** (**Asynchronous**) RPC = as Synchronous RPC, but client doesn't wait and callback manages the response



⚠ Tutti i tipi di RPC possono usare "*streaming*" per request/response (ovvero una request/response può essere uno stream di messaggi piuttosto che 1 singolo messaggio)

Developing application on top of:

- **Socket APIs** → sockets provide communication services; the implementation of the interaction protocol is up to the programmer (so emphasis is more on communication than on application logic) [normally sockets are used to implement middleware and application-level protocols]
- **RPC** → application can be developed as a normal monolithic code (but keep in mind that interaction will be distributed!); then app modules are distributed on multiple hosts (also testing is in distributed environment) (so programmer can focus more on app logic than on communication). Applications are conceived as **combinations of services**: services are designed/developed as reusable components that interact via RPC (> scalability and > reusability)

HTTP/2 is the evolution of HTTP for better efficiency/performance, optimized for networks with low error rates (while **HTTP/3** is optimized to reduce latency for long distance or high error rates). Same semantics as HTTP/1.1 (REST APIs are the same), but different data encoding and use of transport protocols: HTTP/1.1 has high latency for establishing connections (*syn* → *syn+ack* → *ack+req* → *resp* [or even the *clienthello* and *serverhello* for HTTPS]), but also these problems (comparison):

	HTTP/2	HTTP/1.1
data encoding	binary	character-oriented
multiplexing	on single TCP connection	on multiple TCP connections
compression	header and body	only body
push possibility	yes	no

In HTTP/2 we distinguish:

- **Frame** = binary message exchanged by client and server (HTTP msg/operation); main **frame types**:
  - `HEADERS`
  - `DATA`
  - `SETTINGS`:
    - if the client knows the server supports HTTP/2: after initiating the TCP connection, sends the preface + SETTINGS; then server responds with its preface + SETTINGS
    - else: **HTTP/2 Negotiation** → 2 types:
      - TLS-based (HTTPS) [via ALPN = Application Layer Protocol Negotiation]
      - HTTP-based (HTTP) [via HTTP/1.1 Upgrade mechanism]. Example:

– The client starts an HTTP/1.1 connection with upgrade header:

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64 encoded HTTP/2 SETTINGS>
```

– The server responds with:

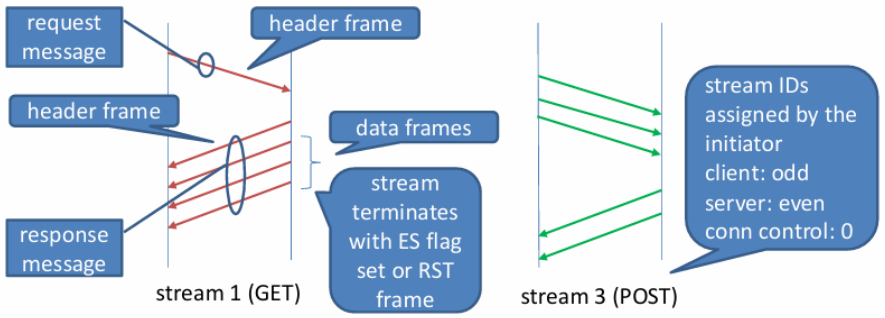
```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
```

h2c: HTTP/2, no TLS
Similar procedure for h2: HTTP/2 over TLS

– The conversation continues as in the previous cases (preface + SETTINGS exchange)

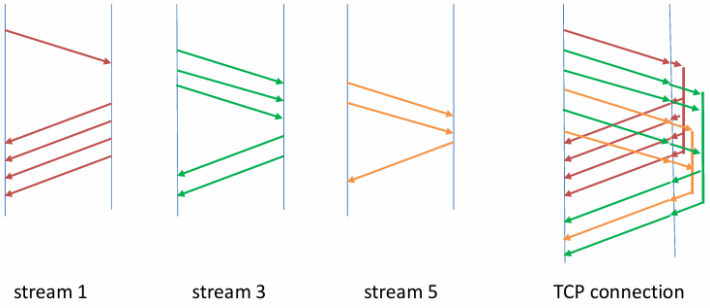
- `RST_STREAM` (signal termination of a stream)
- `PUSH_PROMISE`
- `GOAWAY` (graceful connection termination)
- `WINDOW_UPDATE` (flow control msg)

- **Stream** = independent bidirectional ordered sequence of frames

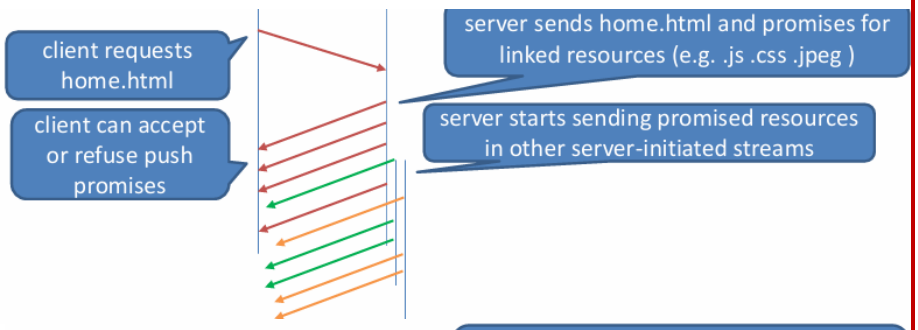


⚠ Frame Layout = length + type + flags + R + stream identifier + frame payload

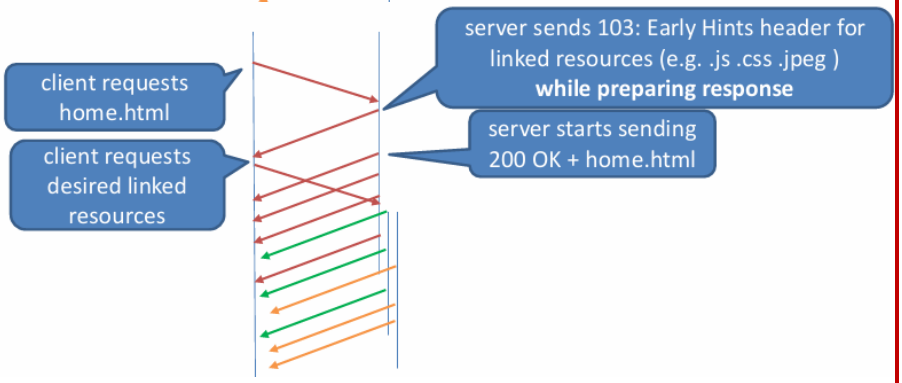
Stream Multiplexing → streams are multiplexed on a single TCP connection. Ordering of the frames of each stream is preserved and window-based flow control is added to regulate stream rates



Push Mechanism → a server can push additional representations, anticipating client's requests (trade bandwidth for latency)



An alternative to push mechanism are **Early Hints** (< bandwidth)



Error Management (**HTTP/2**) = generates a precise report; same as HTTP/1.1 + other mechanisms:

- Stream-level = errors occurring in the management of single streams → **`RST_STREAM`**
- Connection-level = errors occurring in the management of TCP and HTTP/2 connection → **`GOAWAY`**

Main error codes:

- NO_ERROR (0x0)
- PROTOCOL_ERROR (0x1) — generic protocol error
- INTERNAL_ERROR (0x2)
- FLOW_CONTROL_ERROR (0x3) — flow control protocol violation
- SETTINGS_TIMEOUT (0x4)
- STREAM_CLOSED (0x5)
- FRAME_SIZE_ERROR (0x6)

Other ways to learn about status of Non-completed Requests are:

- GOAWAY frame indicates the highest stream number that might have been processed (requests on stream with higher numbers are therefore guaranteed to be safe to retry)
- REFUSED_STREAM error code can be included in a RST_STREAM frame to indicate that the stream is being closed prior to any processing having occurred (requests that was sent on the reset stream can be safely retried)

gRPC is RPC mechanism that works on HTTP/2: exploits performance-oriented HTTP/2 features (binary msgs, header compression…), uses HTTP/2 as transport (like SOAP) and has as typical data representation the **protocol buffers**. It's multi-language, has at most once semantics, does security management (via TLS) and has multiple ways of operation (synchronous, asynchronous, stream-oriented).

Protocol buffers are system-independent data representation and IDL (*abstract* C-like language in protocol buffers IDL v3; binary representation).

```
syntax = "proto3";

service HelloService {
    rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
    string greeting = 1;
}

message HelloResponse {
    string reply = 1;
}
```

These protocol buffers have **scalar** data types (as float, double, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64, bool, bytes, string), but also:

- **`message`** types → user-defined named data structures; each field can be optional **`singular`** (default; 0 or 1 repetitions) or optional **`repeated`** (0 or more). If a field is absent, its default value will be returned when the msg will be read

```
message Book {
    string isbn = 1;
    string title = 2;
    repeated string authors = 3;
}
```

- **`enum`** types:

```
enum ItemType {
    ARTICLE = 0;
    BOOK = 1;
}

message BiblioItem {
    ItemType type = 1;
    string title = 2;
    repeated string authors = 3;
}
```

first value must be 0 (it is the default)

⚠ Default values are:

Type	Default Value
numeric types	0
bool	false
bytes	the empty sequence of bytes
string	the empty string
enum types	the first value (0)
message types	unset (language dependent)

- **map** types → shortcut syntax for associative maps; fields can't be repeated; field **map<int, float> map_field = N** is equivalent to the right image:

```
message Biblio {
    map<string, BiblioItem> items = 1;
    string description = 2;
}
```

```
message MapFieldEntry {
  key_type key = 1;
  value_type value = 2;
}

repeated MapFieldEntry map_field = N;
```

- **oneof** → a set of fields such that at most 1 of them can be set to a non-default value (fields in the set share space on the wire); so set 1 field will automatically unset the others. Fields can't be repeated (and can't be maps)

```
message BiblioItem {
    string title = 1
    repeated string authors = 2;
    oneof extension {
        Article article = 3;
        Book book = 4;
    }
}
```

Managing larger specifications:

- **import library_name.proto**
- **package package_name**

⚠ Name problems can be avoided using the full package name (not only the `message` name, but `package.message`)

message and **enum** declarations can be **nested** inside other **message** types; nested declarations can be referred by *dot notation*:

```
message BiblioItem {
    message Article {...}
    message Book {...}

    string title = 1
    repeated string authors = 2;
    oneof extension {
        Article artcle = 3;
        Book book = 4;
    }
}
message Articles {
    BiblioItem.Article article;
    ...
}
```

⚠ gRPC supports these RPC types: classical (synchronous), 2-way asynchronous, streaming mode

A **RPC interface** can be defined by a **service** block: each RPC procedure has its **rpc** declaration (request msg type + response msg type) [**stream** can precede request/response msg type]

```
service UpdateService {
 rpc update(UpdRequest) returns (UpdResponse);
 rpc updateCS(stream UpdRequest) returns (UpdResponse);
 rpc updateSS(UpdRequest) returns (stream UpdResponse);
 rpc updateDS(stream UpdRequest) returns (stream UpdResponse);
}
```

gRPC PROGRAMMING

gRPC supports client and server stub generation for more programming languages (C++, Java, Node…); clients can use stub instances and can open gRPC channels (**mapped to HTTP/2 connections**) [channels are *bidirectional*, *1 channel can carry several RPC interactions*, *each channel side* can be closed *independently and gracefully*]

Lets see:

- **gRPC programming in Java** → static automated generation of stub classes from proto file: 1 class per **`service`**, 1 class per **`proto file`**. How to:
  - server-side = extend server stub (implement methods, 1 per **`rpc`**) + create server
  - client-side = create channel (open HTTP/2 connection) + instantiate client stub and associate it with channel + call methods on the stub
  - Managing Streaming:
    - **response streaming** in *server-side* programming [**C** ← **S**]:
      - programmer develops stream producer (*uses* **`observer`**)
      - library provides consumer (*creates* **`observer`**)
      - used to return response in all methods (even in those that don't use server streaming)
    - **request streaming** in *server-side* programming [**C** → **S**]:
      - programmer develops stream consumer (*provides* **`observer`**)
      - library produces data (*uses* **`observer`**)

    ⚠ Opposite roles in *client-side* programming

- **gRPC programming in Node** → usando **JS**, possiamo generare uno stub runtime (questo è il vero potenziale del generare lo stub, la sua vera funzione). Vediamo un **esempio di proto file dynamic parsing e stub constructor generation**:

```
const PROTO_PATH = __dirname + '/../proto/bank.proto';
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

let packageDefinition = protoLoader.loadSync(
    PROTO_PATH,
    {keepCase: true,
     longs: String,
     enums: String,
     defaults: true,
     oneofs: true
    });
let protoD = grpc.loadPackageDefinition(packageDefinition);
let bank = protoD.it.polito.dsp.bank;
```

proto file descriptor

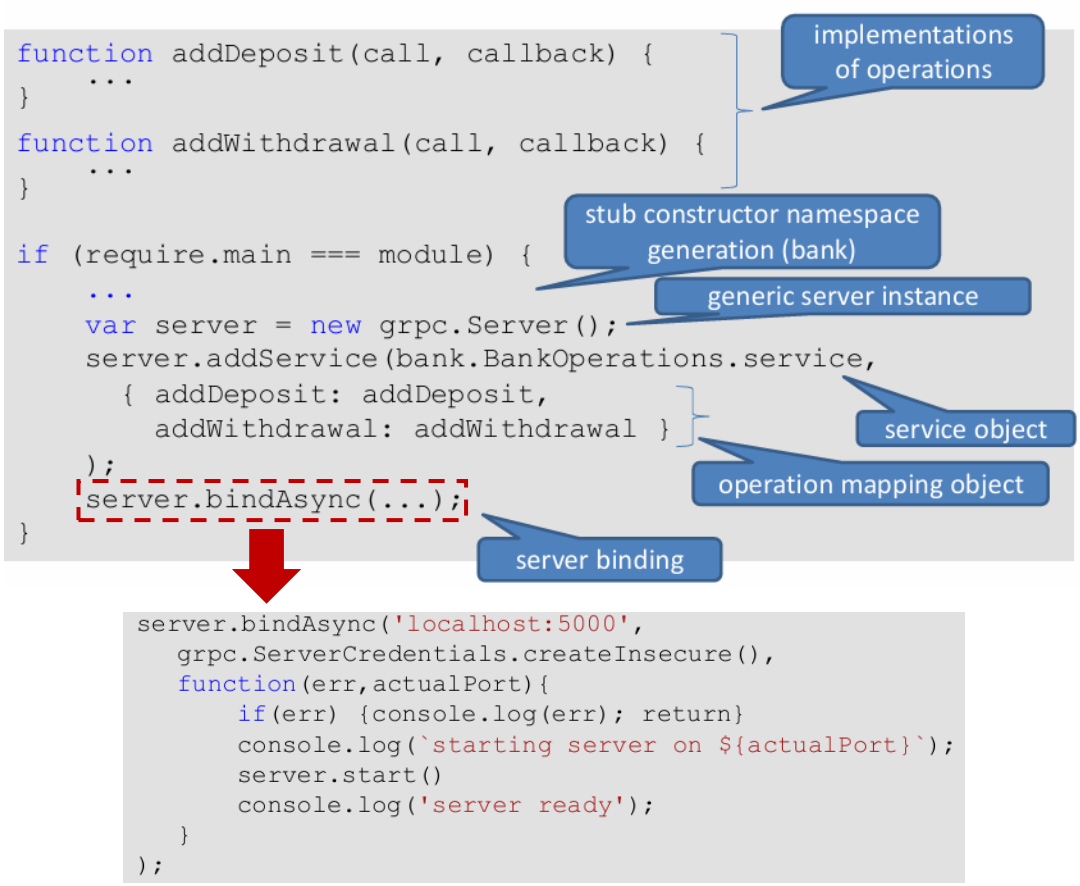
namespace containing stub constructors

Steps

  - server-side:
    - load proto file (create **proto file descriptor**)
    - create **generic Server instance** (constructor from gRPC library)
    - add **service object** (obtained from stub constructor) + **implementation mapping** to server
    - bind the server to and unused port and start it
  - client-side:

- load proto file (create **proto file descriptor**)
- instantiate **stub** by calling **stub constructor** (which creates HTTP/2 channel to server)
- call methods on the stub

Bank example (**server creation + server binding**):



- Managing Streaming:
  - **Server-side**
    - no streaming → **`function name(call, callback)`**; *request* available as **`call.request`**, while *response* returned by calling **`callback(err, response)`**
    - request streaming → **`function name(call, callback)`**; call implements a **`Readable`** to read *request* (events: data, error, end)
    - response streaming → **`function name(call)`**; call implements a **`Writable`** to write *response* (methods: write, end)
    - request + response streaming → **`function name(call)`**
  - **Client-side**
    - no streaming → `stub.fname(request,function(err,response) {…});`
    - request streaming → `var call = stub.fname(function(err,response) {…});` call implements a **`Writable`** to write *request* (methods: write, end)
    - response streaming → `var call = stub.fname(request);` call implements a **`Readable`** to read *response* (events: data, error, end)
    - request + response streaming → `var call = stub.fname();`

⚠ Guarda esempio come codice di gRPC Bank in Node

⚠ So **gRPC main features** are: simple service definition (*protocol buffer*), high-performance and scalability, bidirectional streaming support, multi-language and multi-platform → ***M2M*** communication

5) Distributed Algoritms (Synchronization and Coordination)

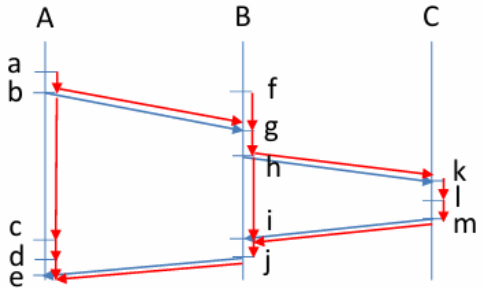
❖ SYNCHRONIZATION

Processes in a DS are **asynchronous**, so sometimes we need to synchronize them (ordering and timing requirements must be satisfied). 2 approaches of synchronization:

- Physical clocks:
  - **UTC Receivers** (accurate synchronization because really *precise*, but *expensive*)
  - **NTP** (*cheaper*, but *not so accurate* → LAN < 1ms; public internet under congestion > 100ms)

- Logical clocks → for synchronization, **agreement on ordering of events** is sufficient: how to do it?
  Happens-before Relation in a DS, ovvero $x \rightarrow y$ significa "**event x happens before event y**". In a DS based on message exchange, $x \rightarrow y$ can be observed in these cases:
  - x happens before y in the same process
  - x and y are the events of sending and receiving the same msg (obviously by different processes)

  ⚠ Ovviamente vale la **proprietà transitiva** (x prima di y, z prima di x, quindi z prima di y)

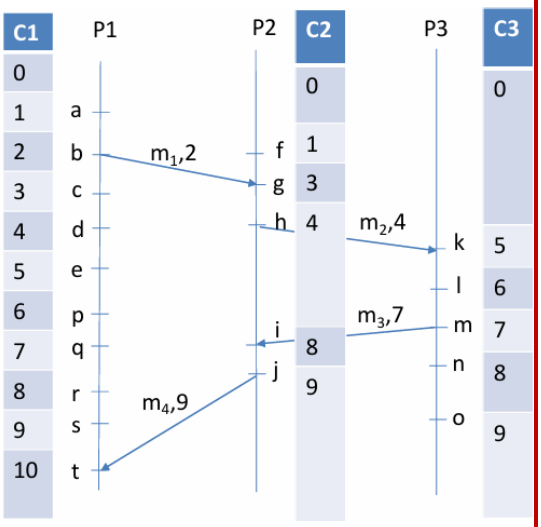


  Happens-before relation captures the possibility of "**causal relation**" (causality) between 2 events:
  - if $x \rightarrow y$, then x and y may be causally related (x = cause, y = effect; or y depends on x)
  - if $x \nrightarrow y$ and $y \nrightarrow x$, then x and y are not causally related (they are concurrent)

  Lamport Clocks are used to assign timestamps (*clock values*) to events so that:

$$\text{if } x \rightarrow y \text{ then } C(x) < C(y)$$

  Come funziona? Algoritmo:
  - each process P_i keeps a local time counter C_i
  - P_i apply a timestamp to each local event x with the current C_i value (that is $C_i(x)$) after having **incremented** C_i
  - P_i apply a timestamp to each msg it sends with the timestamp $C_i(s)$ assigned to the send event s
  - when P_i receives a msg with timestamp C_r, $C_i = \max(C_i, C_r)$ and then it apply a timestamp to the receive event r as usual

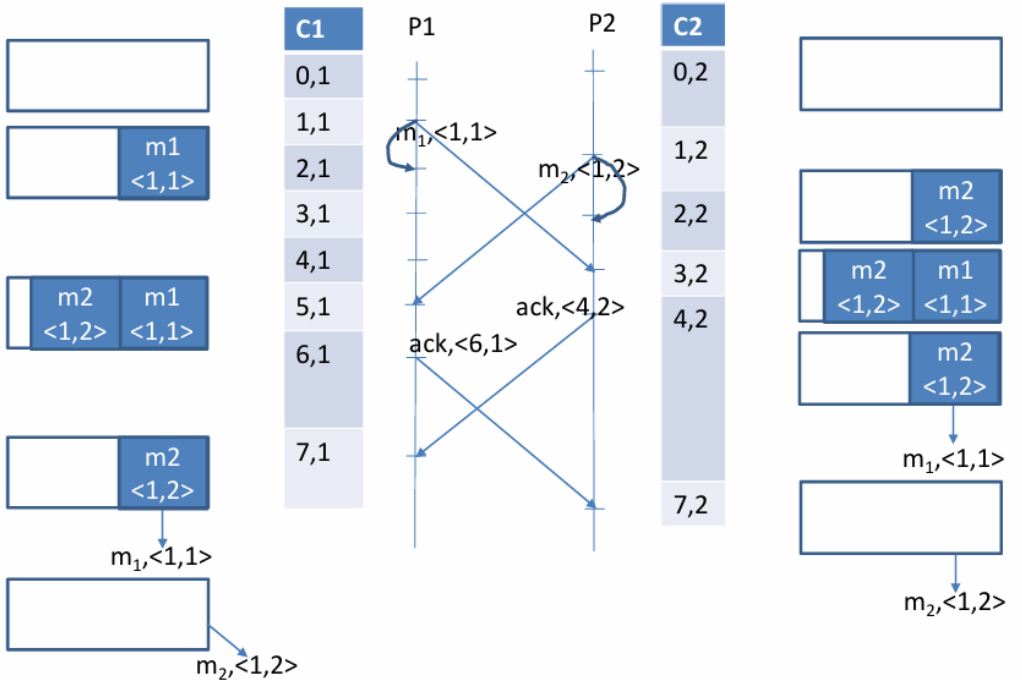


  ⚠ 2 problemi:
  - alcuni eventi in processi diversi hanno lo stesso timestamp (quindi dovremmo aggiungere il **PID** al timestamp per differenziarli → $C(x) =< C_i(x), i >$); questo è detto "total ordering" e nell' esempio sopra avremmo $m_1, < 2,1 >$, poi $m_2, < 4,2 >$, poi $m_3, < 7,3 >$, poi $m_4, < 9,2 >$
  - inoltre se $x \rightarrow y$ sicuro $C(x) < C(y)$, ma non viceversa [*]

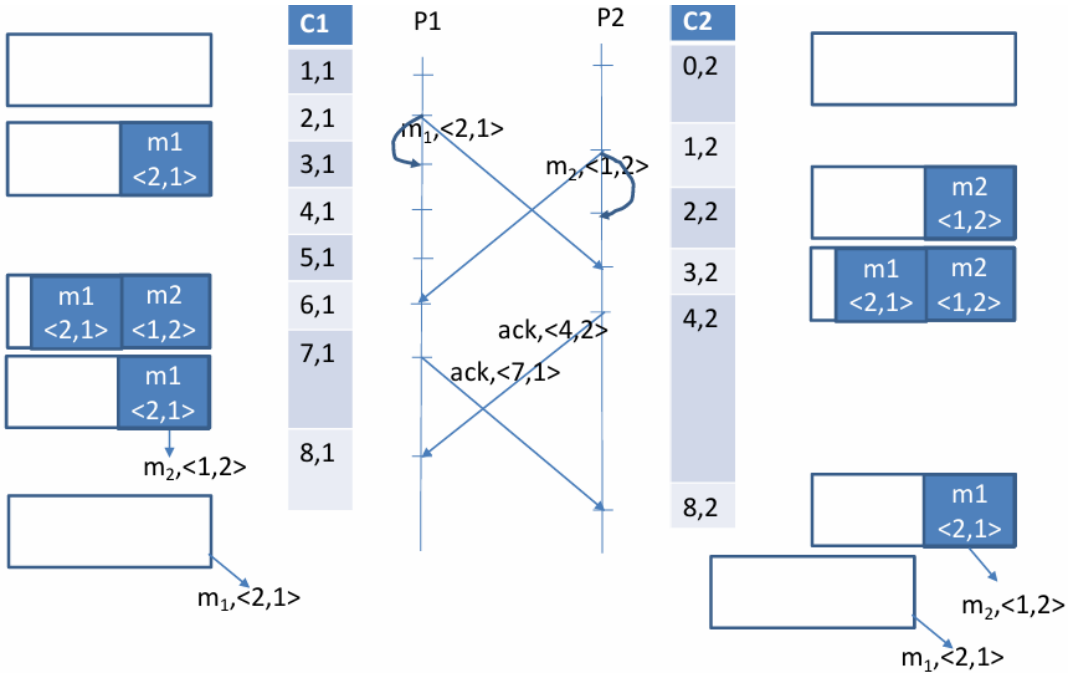
⚠ Può essere implementato con una funzione di timestamp che incrementa e ritorna il contatore oppure con hooks nelle funzioni di send/receive che applicano il timestamp agli eventi di send/receive e ai messaggi, gestendo il contatore

An application of Lamport Clocks is the **TOTAL-ORDERED MULTICAST** (situazione = un set di processi manda msg multicast; tutti i msg multicast devono essere mandate nello stesso ordine ad ogni receiver) [es. usato in 2 server replicati]. Come funziona quindi il **Total-ordered Multicast Algorithm**:

- Lamport clocks are maintained in each process (each multicast msg is timestamped by sender with the send event timestamp)
- Received messages are queued (ordered by timestamp)
- Each receiver acknowledges msg reception to the other receivers
- When acknowledgements for the msg at the head of the queue have been received from all other receivers, that msg is dequeued and handed to the application



Mentre invece, stesso esempio, ma con m_1 che parte al timestamp 2 (nel processo 1), avremo che m_1 sarà messo incodato a m_2 (opposto dell'esempio prima):



⚠ Per far funzionare il total-ordered multicast, dobbiamo fare 2 assunzioni (*TCP* garantisce):
- no messages are lost
- messages from the same sender are received in the same order they were sent

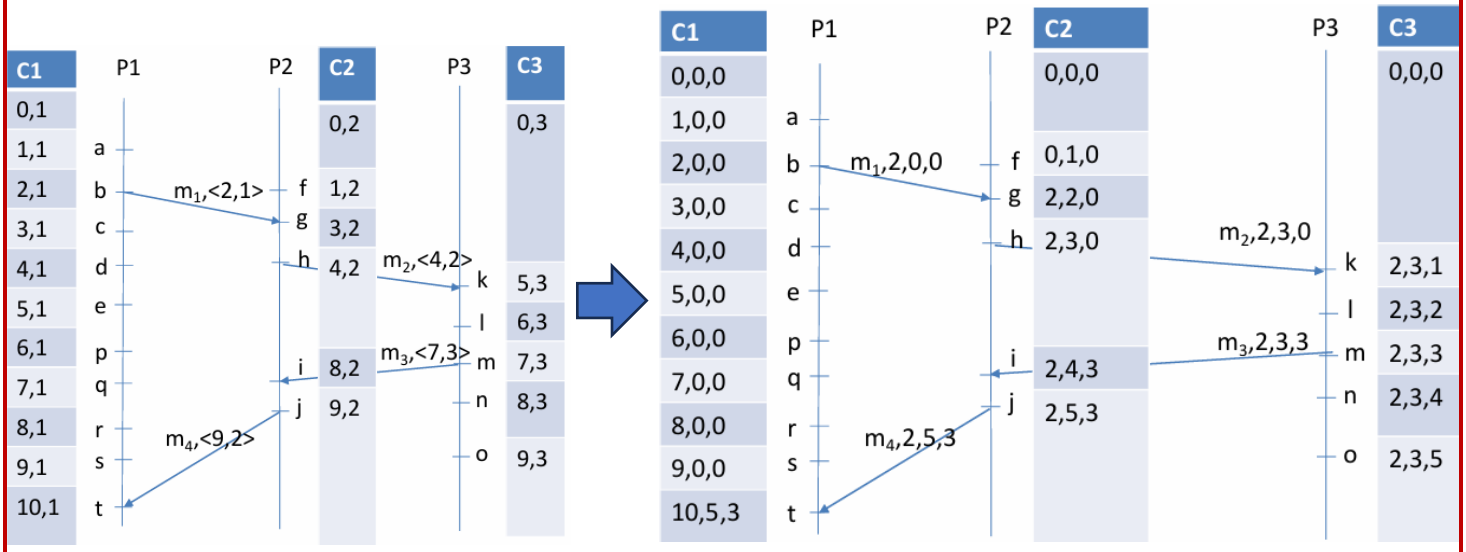
⚠ Se i messaggi fossero operazioni da eseguire nello stesso ordine su dati replicati, l'algoritmo implementa la **state machine replication**

So, Lamport clocks provide a total ordering of events, but this ordering doesn't necessarily imply **causality** (as we said before, if we have $C(x) < C(y)$, then not necessarily $x \rightarrow y$ [*]).
Vector clocks are a way to provide a partial ordering of events that captures causality:

$$\boldsymbol{C(x) < C(y) \Leftrightarrow x \rightarrow y}$$

Come funziona? Algoritmo:
- Each process P_i keeps a local vector of time counters `VCi[]`
  - `VCi[i]` = local event counter (as with Lamport)
  - `VCi[j]` = P_i's knowledge of the local time at P_j
- P_i apply a timestamp at each local event ***x*** with the current `VCi[]` value (ovvero `ts(x)`) after having incremented `VCi[i]`
- P_i apply a timestamp at each msg it sends with the timestamp `ts(s)` assigned to the send event ***s***
- when P_i receives a msg with timestamp `tsr`, P_i sets `VCi[k] = max(VCi[k], tsr[k])` for each ***k*** (vector clock adjustment) and then it apply a timestamp to the receive event as usual

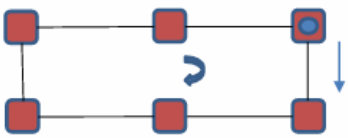
Esempio (sx l'esempio di prima, dx con Vector clocks):



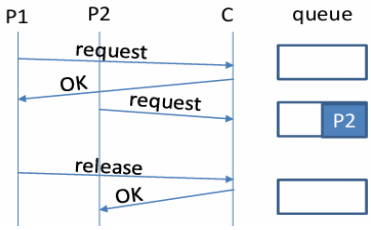
⚠ Applicazioni d'uso dei Vector clocks → **casual ordered multicast**: each multicast msg must be delivered after the delivery of all other causally related messages that were sent before (so non-causally related messages that were sent before may be received after)

❖ COORDINATION

How to guarantee **Mutual Exclusion** on access to shared resources by multiple processes in a DS? 2 types of algorithms:
- **Token-based** → Token Ring Mutual Exclusion: processes organized in a *ring* overlay; *1 token* in the system continues circulating on the ring. How it works?
  A process *P* waits for ther token to access the resource; when arrives:
  - *P* starts accessing the resource and keeps token until access finished
  - then *P* passes the token to the next process in the ring



- **Permission-based**:
  - Centralized → central manager (C). How it works?
    A process P sends a request to C to access a shared resource and waits for permission response; C delays permissions while the resource is engaged



  - Decentralized → based on Lamport clocks (totally ordered multicast). Requester sends request to all processes (including itself) and waits for permission from every process. Responses to requests received while accessing the resource are delayed. In case of conflict (cioè receiver also wants to access the resource), the request with the lower timestamp wins (la più vecchia)

Performance comparison (N = n° of processes):

Algorithm		#Messages/access	delay before entry (# messages)
Permission based	Centralized	3	2
	Decentralized	2 (N-1)	2 (N-1)
Token based	Token ring	1...	0 ... N-1

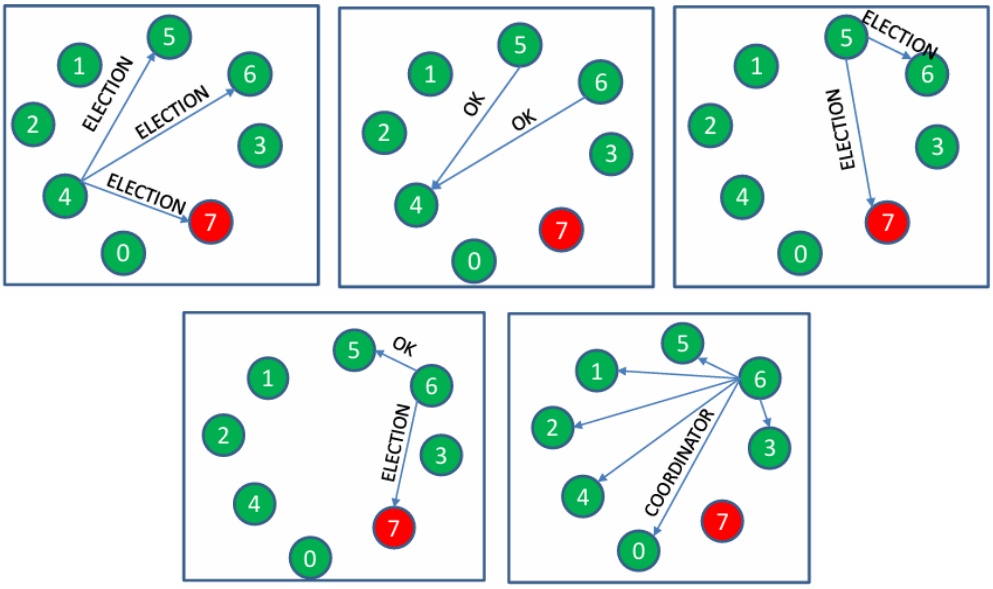
How to do an **ELECTION** of a process (*coordinator*) in a group of processes?

Assumptions:

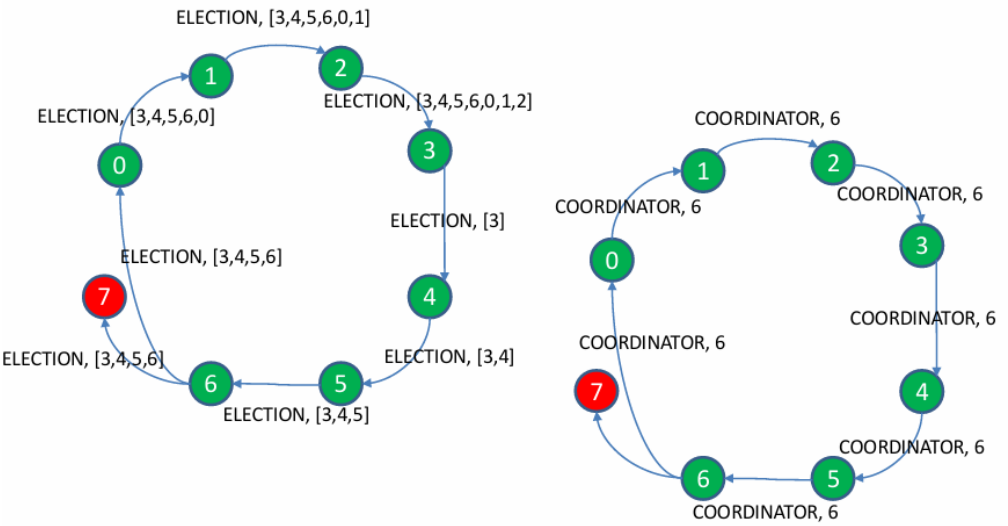
- each process has a unique PID (P)
- each process knows all the other processes in the group
- processes in the group can be ***up and running*** or ***down***, but channels are reliable

To perform an election, the algorithm must **elect the *up* process having the highest pid**; at the end of the algorithm, all processes agree about who is the elected process. 2 algorithms:

- **Bully Algorithm** → $pid(P_k) = k$. How it works?
  - Starts when a process P_k detects *coordinator is missing* and decides to hold an election:
    - P_k sends **`ELECTION`** msg to $P_{k+1}, P_{k+2}, \dots, P_{N-1}$
    - if nobody responds, P_k wins the election; else P_k gives up
  - Whenever P_i receives an ELECTION msg:
    - P_i responds with **`OK`** (means "*alive*")
    - P_i holds an election (if it wasn't already holding one)
  - All processes give up except from the 1 who wins; the winner finally informs the other processes



- **Ring Algorithm** → processes are ordered in a logical ring (es. by *pid*). How it works?
  - Starts when a process P_k detects *coordinator is missing* and decides to hold an election:
    - P_k sends **`ELECTION`** msg to its successor in the ring
    - if successor doesn't respond, P_k sends `ELECTION` msg to the next successor etc…
  - Each `ELECTION` msg carries the list of senders
  - When eventually `ELECTION` msg gets back to the starter P_k:
    - P_k stops the circulation of the msg
    - P_k computes the winner and circulates a **`COORDINATOR`** msg on the ring (same way as for `ELECTION`) to inform about the winner



Another (more general) coordination problem is **CONSENSUS** (how n processes, each one proposing an input value, agree on the same output value?), but it can be solved using Mutual Exclusion and Election algorithms.

6) Data Replication and Consistency

DATA REPLICATION is used to improve reliability and performance (we focus on performance-related data replication [scaling out technique for size/geographical scalability]). Ideally data replication should be hidden to the user (transparency), but because of **consistency** (when replicated data change, inconsistent states arise because replicas become temporarily not identical) we can't totally do it. So, keeping replicas consistent has a performance cost (so **we have to balance the performance and scalability obtained using data replication, but also the disadvantage of performance to keep replicas consistent**).

Different types of consistency:

- **STRICT** = each data change is *propagated to all replicas* before executing the next operation on the data store (nella pratica: each read returns the last written data; data changes can't run concurrently with other operations). **Easy** to use by programmers, but **high performance cost** (so to improve we have to relax the consistency constraints, but so it's less easy to use by programmers) [tradeoff]

- **CONTINUOS** = each read returns a value close to the last written one; consistency is measured as how much deviation from strict consistency is **tolerated** (3 assi di deviazione: numerical deviation, staleness deviation [difference of last update t between replicas] and deviation in ordering of write operations [max n° W not yet made permanent])

  ⚠ Usually it measured in data units (***conits***) rather than to the whole data store [large conits = large inconsistencies; small conits = large n° of conits to manage]

- **SEQUENTIAL** (**Lamport**) = result of any execution is the same as if:
  1. R/W operations by all processes on the data store were executed in some sequential order
  2. operations of each single process appear in sequence in the order specified by its program

  Any interleaving of R/W operations that satisfies the property **2** is possible, but all processes see the **same interleaving**

		✓
P1	W(x)a	
P2	W(x)b	
P3	R(x)b R(x)a	
P4	R(x)b R(x)a	

		✗
P1	W(x)a	
P2	W(x)b	
P3	R(x)b R(x)a	
P4	R(x)a R(x)b	

  In pratica, each read returns the same value that would be returned if all the operations of the system were executed sequentially in 1 of the possible sequential orders

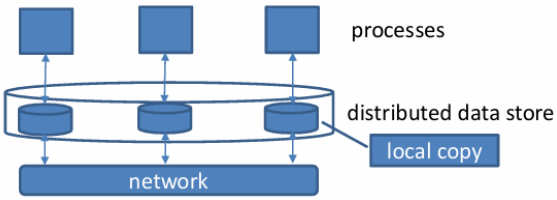
- **CAUSAL** = writes that may be causally related must be seen by all processes in the same order (so respecting causality), while concurrent writes may be seen in a different order on different machines
  ⚠ Weaker than sequential consistency

		✓
P1	W(x)a W(x)c	
P2	R(x)a W(x)b	
P3	R(x)a R(x)c R(x)b	
P4	R(x)a R(x)b R(x)c	

		✗
P1	W(x)a	
P2	R(x)a W(x)b	
P3	R(x)a R(x)b	
P4	R(x)b R(x)a	

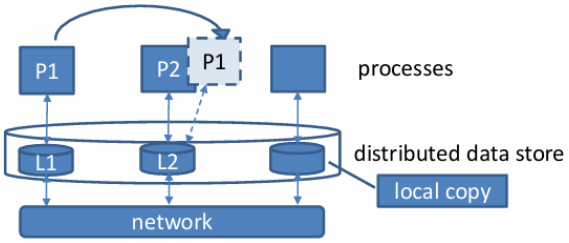
- **EVENTUAL** = each read return the last written data or an older version; in absence of W-W conflicts, if no updates take place for a long time, all replicas become **identical**
  ⚠ It's used in some application where it's tolerated to have weaker consistency guarantees (es. data store that have many R and few W [es. static WebPages])
  ⚠ It works well if processes always access the same replica (no *process mobility*) [*]

CONSISTENCY MODEL = contract between processes (so, the *programmer*) and data store (es. "if programmers respect certain rules, data store guarantees consistency properties in R/W operations")



Types of consistency models:

- **Data-Centric** → consistency properties are expressed in terms of how R/W operations behave on the **global data store** (es. *strict consistency*: each R returns the last written data); provides **system-wide** data consistency properties
- **Client-Centric** → provide consistency properties for each **single process** (*client*). These models can be added to limit the issues related to *process mobility* with eventual consistency [*]:

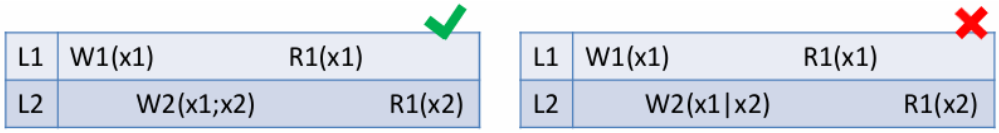


Definendo:

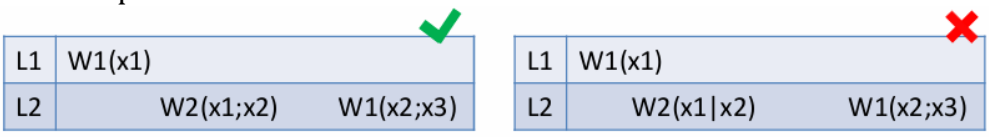
xj	**version** j of variable x
Wk(xj)	Pk writes xj
Wk(x1;x2)	Pk writes x2 which follows from x1
Wk(x1|x2)	Pk writes x2 concurrently with x1 (potential write-write conflict)
Rk(xj)	Pk reads xj

Definiamo:

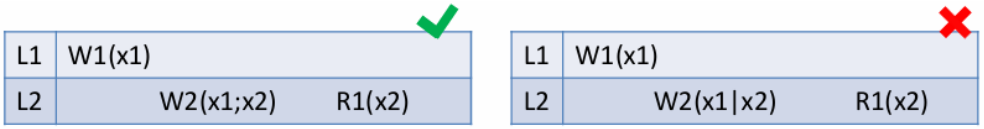
- **Monotonic Reads** → if a process reads the value of a data item x, any successive R operation on x by that process will always return the same value or a more recent value



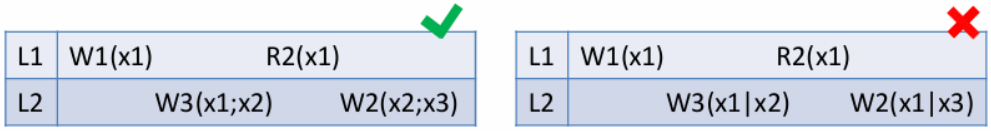
- **Monotonic Writes** → a W operation by a process on a data item x is completed before any successive W on x by the same process



- **Read Your Writes** → the effect of a W operation by a process on data item x will always be seen by a successive R operation on x by the same process



- **Writes Follow Reads** → a W operation by a process on a data item x (following a previous R on x by the same process) is guaranteed to take place on the same or a more recent value of x that was read



REPLICA MANAGEMENT:

- Server Replication → what servers to replicate and where to put servers? (*placement problem*)
- Content Replication → what contents to replicate and where to put replicas?

How to update propagation to replicas? **Update Propagation Strategies**:

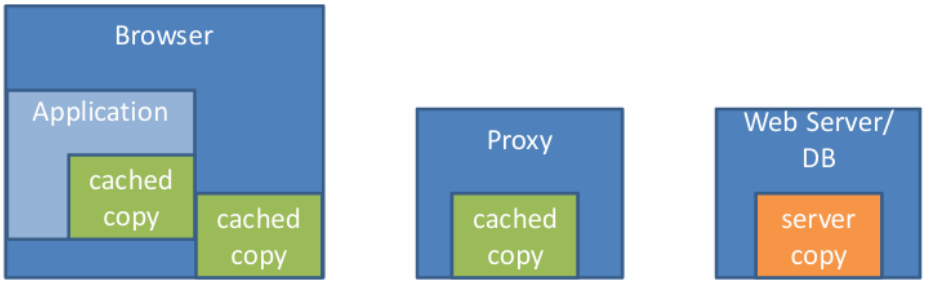
- update notification propagation [buono con < R/W]
- data propagation [buono con > R/W]
- update operation propagation [not always possible/convenient]

We distinguish:

- **Push** protocols (*server-based*) → strong consistency [buono with > R/W]
- **Pull** protocols (*client-based*) → weak consistency [buono con < R/W]; response time ~ cache misses
- **Continuos** Consistency protocols → processes modifies local copies; local writes are propagated to other processes, which update their local copy (so processes monitor consistency)
- **Sequencial** Consistency protocols
  - Primary Based Protocols → for each data item x, a primary process is responsible for coordinating write operation on x; we distinguish:

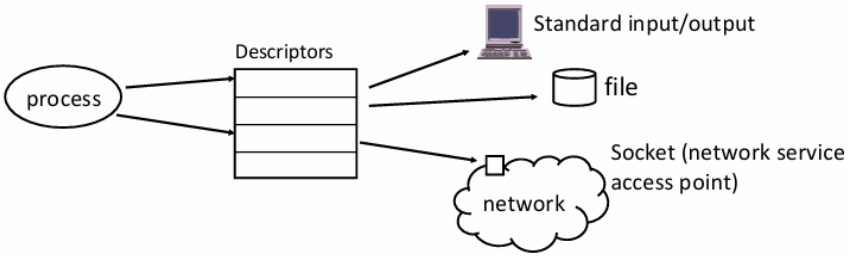
        - Remote Write protocols = primary for x is fixed (es. MongoDB, Azure SQL Database…)
        - Local Write protocols = primary for x migrates to the process that need to write (es. Hazelcast…)
    - Replicated Write Protocols → write operations can be initiated at different replicas; we distinguish:
        - Active Replication protocols = global ordering of operations can be kept consistent by performing ordered multicast or by using centralized sequencers
        - Quorum-based protocols = versions of data are recorded; each process that wants to R/W must start a voting procedure (R/W operations need agreement with other processes)

- **Cache Coherence** protocols → protocols specific for caches (es. used in WebApp). 2 strategies:
    - Coherence (**when** inconsistencies are detected)
        - client validates consistency ***before*** transaction
        - client validates consistency ***while*** transaction and aborts if validation fails
        - client validates consistency ***after*** transaction and aborts if validation fails
    - Enforcement (**how** consistency is enforced)
        - ***read-only*** caches with push/pull mechanisms
        - ***read-write*** caches with primary-based local-write protocols; concurrent W are admitted



7) TCP/IP Sockets

Socket API = standard API for accessing network services of layer **L4-L3-L2** (was born in C language with UNIX; initial idea ↓)



A ***socket*** is the abstraction of an inter-process communication *channel endpoint* or SAP; sockets live in **domains**, each one characterized by its protocol family and address family (communication between 2 different domains is impossible). Domain examples:

Domain	Protocol Family	Address Family
ARPA Internet	PF_INET	AF_INET
Internet with IPv6	PF_INET6	AF_INET6
ISO/OSI	PF_ISO	AF_ISO
Unix pipes	PF_UNIX	AF_UNIX

Socket characteristics:

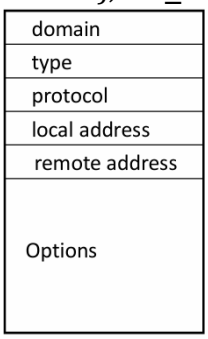
- **TYPE** → specifies the `service type` accessible through the socket; main types:
    - `SOCK_STREAM` = continuos bidirectional byte stream (no delimiters), transmitted reliably; it'a *connection-oriented* service, offered by L4 [**TCP**]

- **`SOCK_DGRAM`** = bidirectional message (datagram) delivery without reliability guarantee; it'a *connectionless* service, offered by L4 [**UDP**]
- **`SOCK_RAW`** = direct access to the services provided by L2-L3 protocols

- **PROTOCOL** → protocol used; es. if `PF_INET` domain, following protocol choices are possible:
  - `SOCK_STREAM` type → TCP protocol (**`IPPROTO_TCP`**)
  - `SOCK_DGRAM` type → UDP protocol (**`IPPROTO_UDP`**)
  - `SOCK_RAW` type → ICMP protocol (**`IPPROTO_ICMP`**) or IP protocol (**`IPPROTO_RAW`**)

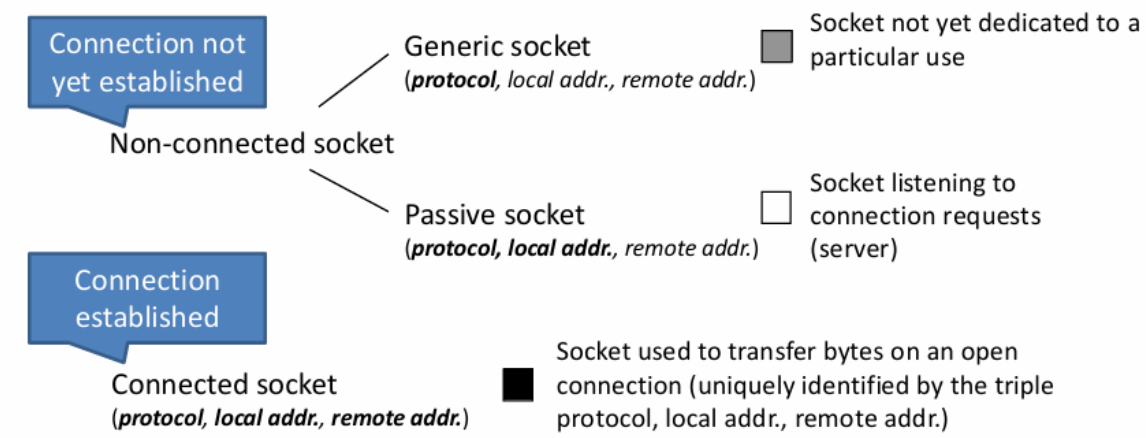
- **OPTIONS**→ specify other socket features (es. `SO_RCVBUF` (size of receive buffer), `SO_SNDBUF`, `SO_LINGER`, `SO_KEEPALIVE`)

This is the data structure associated with a generic socket →



⚠ **Connected sockets** = legato ad un peer, trasporta i dati dell'app; **Passive sockets** = sta in ascolto e genera socket per-client

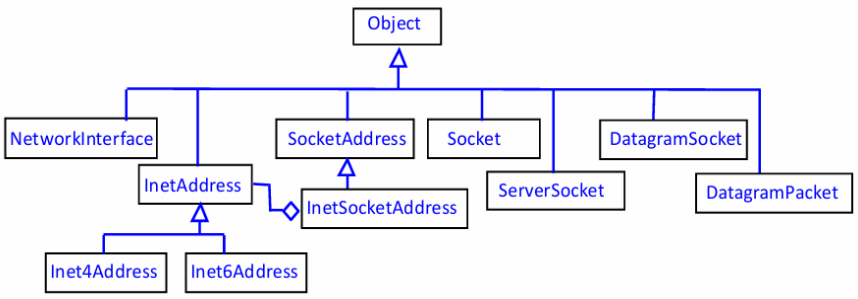
Using **TCP** (**`STREAM`**) **sockets** (endpoints can take different states/forms, according to the role they are playing in the communication):



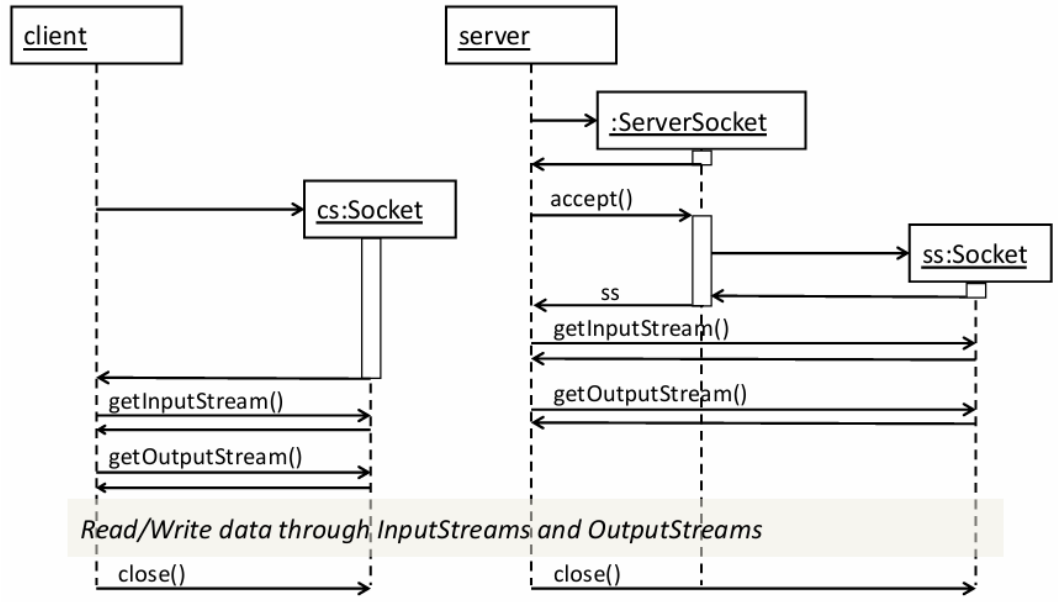
Operations offered by the **Socket API for Accessing L4** (**TCP**):

- allocate local resources for communication
- specify endpoints
- open a connection (*client-side*)
- wait for the establishment of a connection (*server-side*)
- send/receive data on a connection (also *urgent* data)
- be notified when data arrive
- gracefully terminate a connection or abort a connection
- release resources when a connection terminates

Ora che abbiamo avuto un'introduzione generale ai socket, vediamo l'implementazione in **Java** della **Socket API** (**`java.net`** package) →



Basic example of **use of connected sockets TCP**:



Operations TCP nel dettaglio:

- **ESTABLISH** a connection:
  - **server-side**
    - create ServerSocket → **`ServerSocket (int port, int backlog, InetAddress bindAddr)`**; all parameters are optional
    - call **`accept()`** to wait for connection establishment
    - return **`Socket`** when connection established
  - **client-side** → **`Socket(InetAddress addr,int port,InetAddress localAddr,int localPort)`** oppure **`Socket (String host, int port, InetAddress localAddr, int localPort)`**

⚠ Possible Exceptions:

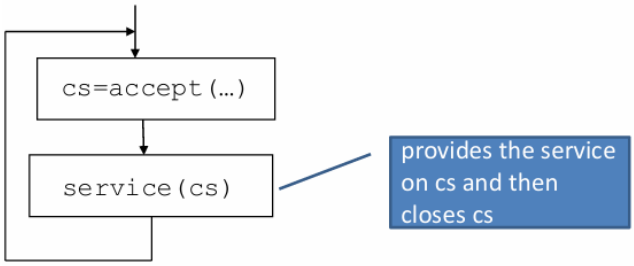
```
java.io.IOException
|-- java.net.SocketException
|   |-- BindException,
|   |-- ConnectException,
|   |-- NoRouteToHostException,
|   |-- PortUnreachableException
|
|-- java.net.UnknownHostException

java.lang.SecurityException
```

- **CLOSING** a connection:
  - `Socket`**`.close()`** = terminates both sides of the connection (FIN sent by TCP, FIN sent by peer; all buffered data must be delivered; no more R/W)
  - `Socket`**`.shutdownOutput()`** = terminates output side (FIN sent by TCP; all buffered data must be delivered; no more R/W)
  - `Socket`**`.shutdownInput()`** = terminates input side (undelivered data are discarded; FIN sent by peer; no more R/W)

  How to detect closing? A read operation on the socket input stream returns **-1** if closed by peer
- **BLOCKING operations and timeouts** → some operations are blocking (es. **`accept()`**, **`new Socket`**, **R** on **`InputStream`** [block until 1 byte available], **W** on **`OutputStream`** [block until space available]); so when blocking in not wanted, the operation can be called by a dedicated thread or can be used TIMEOUTs (for limiting R blocking using **`SO_TIMEOUT`** socket option)

SEQUENTIAL SERVERS = sequential model is the simplest: the server serves requests sequentially in FIFO order (so that only when request completed, goes to next request); TCP sequential servers has this algorithm structure:

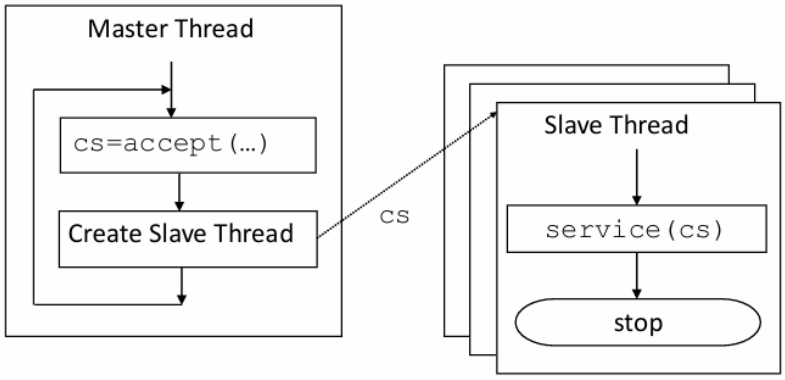


But sequential servers have some problems:

- each request must wait until all previous requests have been served
- waiting occurs event when CPU is idle
- when high request rate, it's likely that a request arrives while server is busy (so requests may be discarded or clients may timeout)

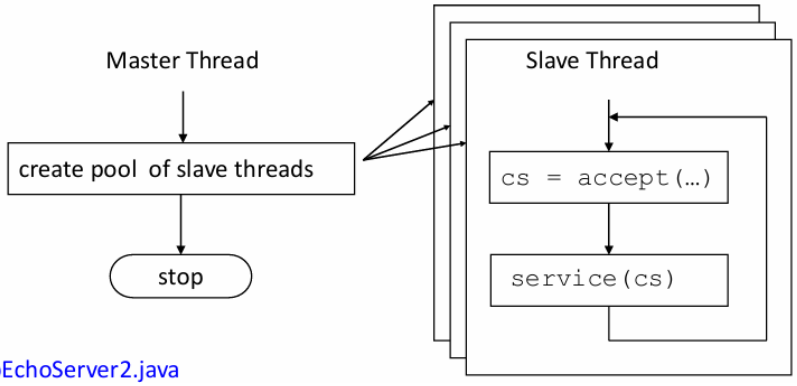
So a possible solutions is **CONCURRENT SERVERS**, where requests are served concurrently (so probability that a request arrives when server can't accept it's reduced). 2 possible implementation of concurrent server:

- by **assigning each request to a different thread** → **THREAD CREATION ON DEMAND**: a master thread continuously accepts connections; as a new connection is accepted, a slave thread is created to serve it



TcpEchoServer1.java

- by **simulating concurrency within a single thread** → **PRE-CREATED THREAD POOL**: to reduce response time without giving up the advantages of concurrency, a pool of threads can be created at startup and allocated to requests when they arrive



TcpEchoServer2.java

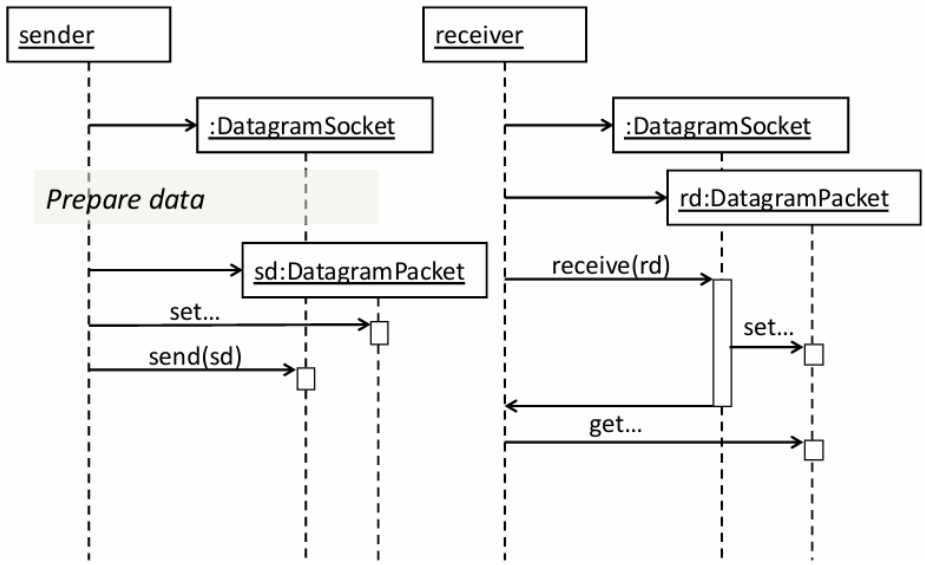
⚠ Java supports thread pool management through `Executors` (`java.util.concurrent library`)

Operations offered by the **Socket API for Accessing L4** (**UDP**):

- allocate local resources for communication

- specify endpoints
- send/receive datagram
- release allocated resources

Basic example of **use of connected sockets UDP**:

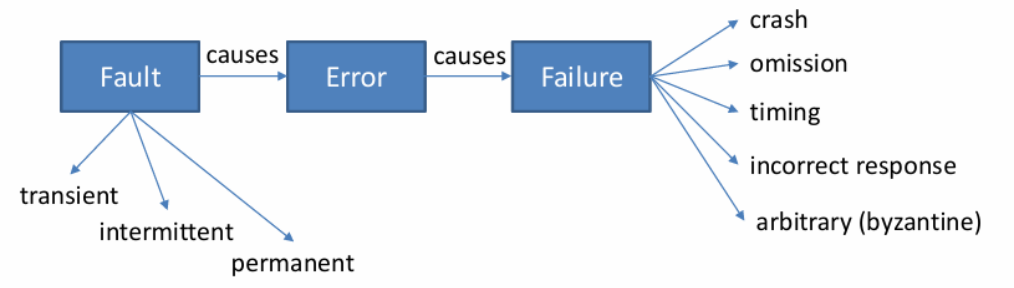


Operations UDP nel dettaglio → only 1, **USING** UDP Sockets in Java:

- `DatagramSocket (int localPort, InetAddress localAddr)`; all parameters are optional; then:
  - `connect(InetAddress localAddr)`
  - `close()` → destroys socket and releases resources
- `DatagramPacket (byte[] buf, int length, InetAddress destAddr, int destPort)`; only `destAddr` is optional

⚠ Encoding/Decoding data → TCP and UDP provide transport of bytes, but data encoding/decoding (*marshalling/unmarshalling*) is up to the programmer (we can use libraries for XML, JSON… or library functions of some java libraries)

→ Fault Tolerance



Dependability (availability, reliability, safety, security…) is threatened by faults and can be achieved by combination of fault prevention, fault tolerance, fault removal and fault forecasting.

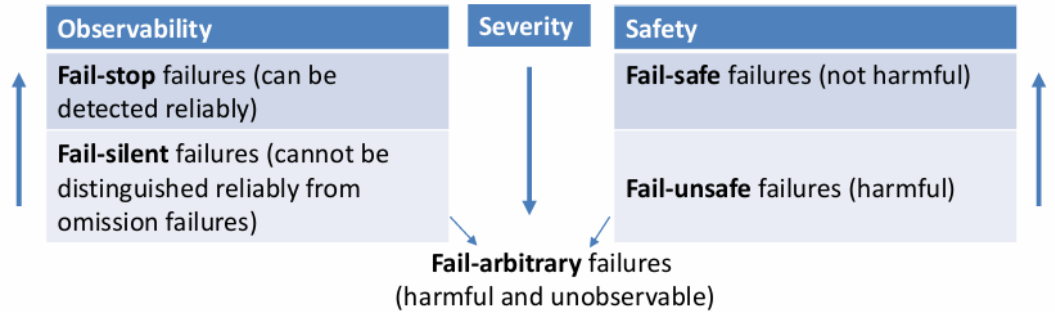
Fault tolerance = ability to continue delivering service in presence of partial failures (with degradation of service, but without seriously affecting overall performance); consist of automatic recover from partial failures (**detection + recovery**) and continue operation in presence of parial failures (while waiting for repair), rather than stopping

Detection + Recovery + Repair techniques:

Failure Type	Detection Techniques	Recovery/Repair Techniques
crash	timeout	replacement, restart, redundancy
omission	timeout	redundancy (including retransmission)
timing	timeout	redundancy
incorrect response	input validation /comparison	redundancy
arbitrary	input validation /comparison	redundancy

Redundancy techniques → *time* redundancy (retransmit msg, redo operations that fail) and *physical* redundancy (adding more equipment or data copies, process replication)

How to **DETECT process crash**? DS are asynchronous so a process P can detect the crash failure of another process Q by timing out when no data arrive from Q; but detections can also be ***erroneous*** and ***delayed***. Another problem is that **not** all crash failures have the same observability/safety/severity:

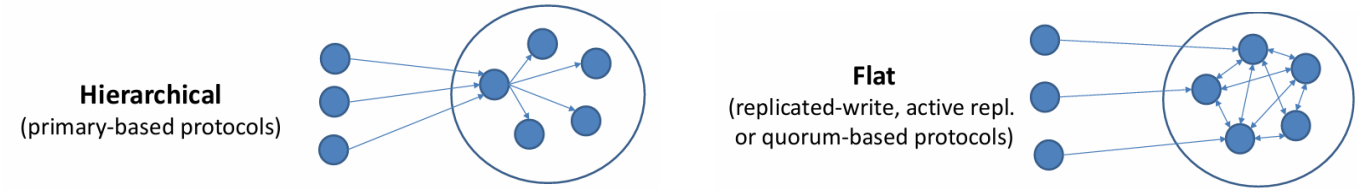


Example = **detecting** crash of process **connected via TCP** → TCP layer can know if a connection has been closed or reset by the peer (*FIN or RST segment received*) or can guess that communication with the peer has been lost (*max # of retransmissions reached*). Also process is informed about these conditions by the Socket API when R/W:

	Read	Write
connection closed	returns -1	successful
connection reset	throws SocketException	throws SocketException

⚠ **Possible crash** scenarios:
- process crash (OS detects → TCP layer sends RST to connected peers)
- host crash with restart (TCP respond to segments belonging to pre-crash connections with RST)
- host crash without restart (peers try to send timeout; programmers should use their timeout)

We can use **PROCESS GROUPS** (group of processes perceived as a single dependable process) to achieve dependability through redundancy; implementing process groups require distributed algorithms (protocols) to manage groups and to make the group fault-tolerant. **Process Groups Organizations**:



How to **obtain the desired fault-tolerance**? A process group is *k*-fault tolerant if it can resist faults in ***k*** members. The consensus protocols and the total number of processes necessary to achieve k-fault tolerance depend on the types of failures that are possible:

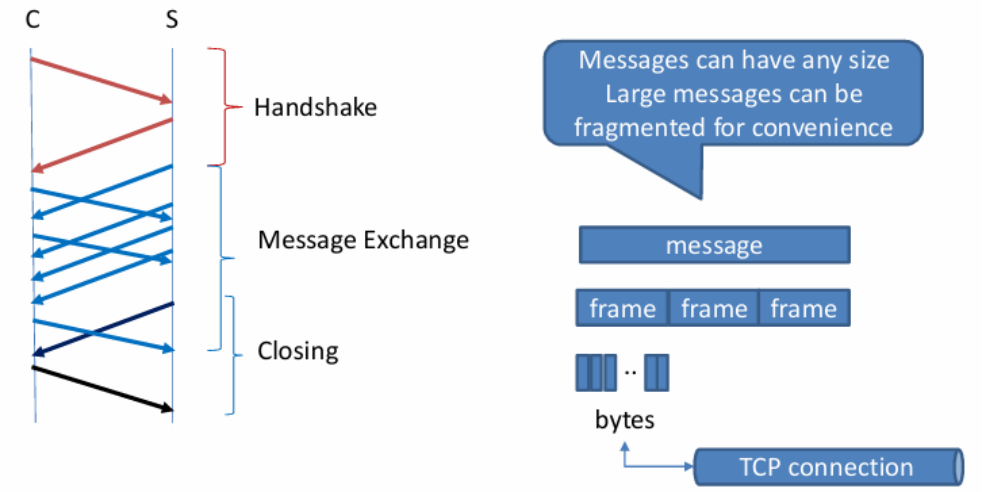
Failure type	#Processes needed for k-fault tolerance	Consensus protocol Examples
crash, omission, timing, network	k+1	flooding
+incorrect response	2k+1	voting, Paxos
+arbitrary (byzantine)	3k+1	PBFT, Blockchain

⚠ **CAP Theorem** (*Brewer's theorem*) → only 2 of **C**onsistency, **A**vailability and **P**artition Tolerance can be obtained at the same time in a distributed systems

8) WebSockets

Modern webapp are becoming more interactive (gaming, video-conference, push notifications…). For example we can implement push notifications with ***request-response*** (through polling, HTTP streaming and subscription) but not fully satisfactory for bidirectional low-latency communication.

To achieve low-latency general-purpose bidirectional channels we use **WebSockets**, ovvero TCP-like communication service, but with some differences. Websockets reuse the web infrastructure (same HTTP ports, compatible with proxies, can work side-by-side with regular HTTP-based communications and SOP-based security model). Websockets are a **message-based communication over a binary framing structure layered upon TCP**. WebSocket protocol (RFC 6455):



WebSockets offer TCP connections with additional features:

- web origin-based security model for browsers
- naming mechanisms (multiple endpoints on 1 port or multiple host names on 1 IP address)
- framing and messaging mechanism with no msg length limit
- in-band additional closing mechanism that works with proxies and intermediaries

WebSocket endpoints are uniquely identified by **URI**s (like the HTTP URIs, but with ws or wss scheme [es. **`ws://host :port path`** (on *TCP*) or **`wss://host :port path`** (on *TLS* on TCP)]).

Example of handshake:

- Client (upgrade) request:

WS endpoint

```
GET ws://foo.com:80/webso HTTP/1.1
Host: foo.com
Connection: Upgrade
Upgrade: websocket
```

```
Sec-WebSocket-Key:dGhlIHNhbXBsZSBub25jZQ==
Origin: http://foo.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

subprotocol negotiation see https://www.iana.org/assignments/websocket/websocket.xhtml

- Server response:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

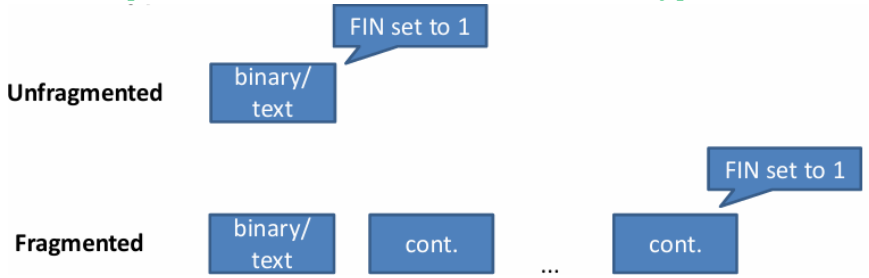
The **frames** types of websockets frames are:

- **control** (connection close or ping/pong)
- **data** (binary, text)

2 types of **messages**:

- **Binary**
- **Text** (UTF-8)

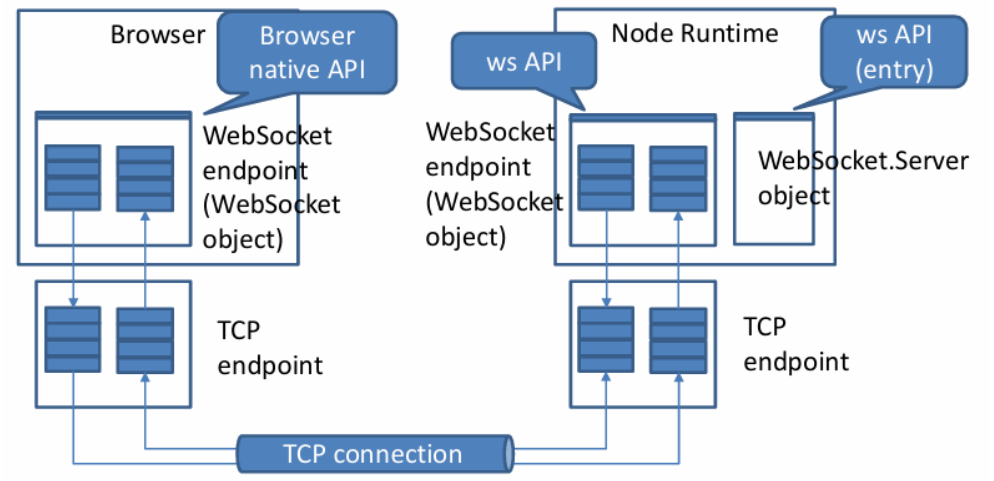
Messages are encoded as sequences of data frames, all of the same type:



How to **close** a connection? A closing operation can be initiated by either endpoint. After a close frame, no more data frames are sent. An endpoint responds to a close frame with a close frame, if one wasn't already sent. After the *closing handshake*, TCP connection is closed

WebSockets provide minimum data types and control procedures; applications using websockets implement higher-level protocols (msg types, metadata, procedures…)

WebSocket Programming with **Javascript**:



WebSocket Programming in the **browser** (client-side) [**native JS API**]:

- constructor → `WebSocket(url [, protocols])`
- constants → `CONNECTING`, `OPEN`, `CLOSING`, `CLOSED`
- methods → **`send(data)`** and **`close([statuscode [, reason]])`**
- events:
  - **open** = a connection has been opened

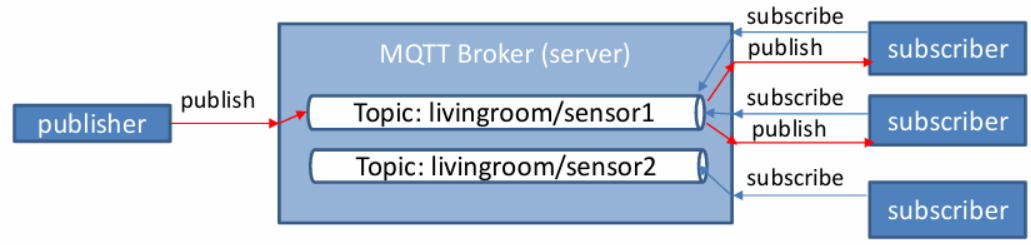
    - **`close`** = a connection has been closed
    - **`message`** = msg received
    - **`error`** = error occurred
- properties:
    - `onopen`, `onclose`, `onmessage`, `onerror`
    - **`url`** (URL of the server endpoint)
    - **`protocol`** (subprotocol selected by server)
    - **`readyState`** (current state of connection)
    - `binarytype` (blob | arraybuffer)
    - `bufferedamount` (outstanding data to be sent)

Programming WebSocket servers in Node with **ws** library → a WebSocket.Server object represents a WebSocket server and has:
- constructor → **`WebSocket.Server`**`(options [, protocols])`
- properties and methods → **`clients`** (all connected clients) and **`close([callback])`** (close)
- events:
    - **`connection`** (WebSocket, http.IncomingMessage)
    - **`error`** (Error)

9) MQTT

MQTT (MQ Telemetry Transport) is a client-server publish-subscribe (≠ from msg queue) messaging transport protocol, usually based on TCP (or *TCP+TLS*). It's open, lightweight, simple and designed for constrained environments as **IoT M2M**, where constraints are:
- need for small code footprint (***simple***)
- restricted network bandwidth and low-power (***efficient***)
- unreliable communication channels (***reliable*** and ***fault-tolerance***)

MQTT **Architecture**:



Come si vede dall'immagine, non c'è un effettivo "canale fisico" tra Broker e subscribers, ma ci sono delle **liste di subscription**; notifications normally are delivered synchronously by Broker to connected subscribers.

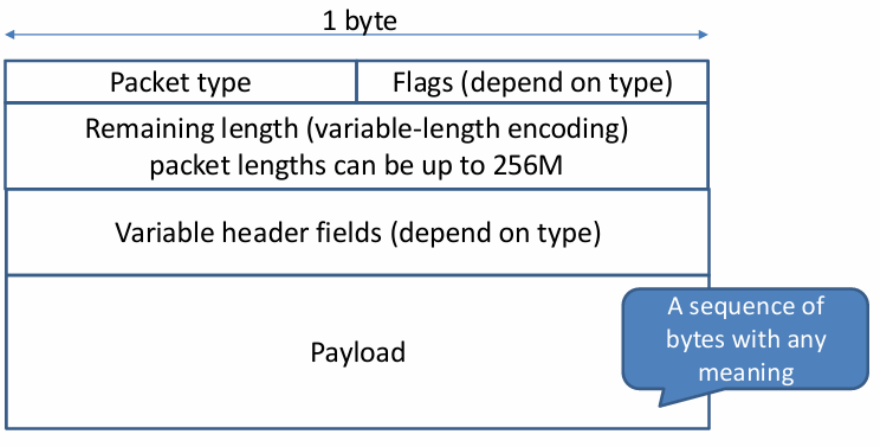
In publish-subscribe systems (as MQTT), clients don't know each other (no referential coupling) and the addressable units are the channels (**Topics**); in fact clients can communicate provided they know shared topics.

These **MQTT Topics** are uniquely identified (within a broker) by hierarchical path-like names (es. apartment **`1/livingroom/temperature`**) [**`/`** = *level separator*]. Topics are never created/destroyed, because they exist yet: clients just need to reference them

⚠ Topic names starting with **`$`** are reserved, while the ones starting with **`$SYS`** provide system info (es. **`$SYS/broker/version`**, **`$SYS/broker/clients/connected`** …)

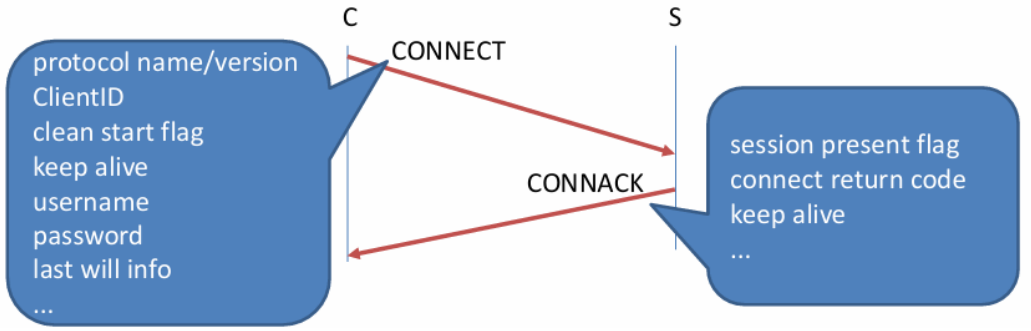
Topic filters can be used to refer collections of topics using *wildcards* (es. **`apartment 1/+/temperature`** → **`+`** means single level; **`apartment 1/#`** → **`#`** means multilevel [in fact must be at the end])

MQTT Control Packets:



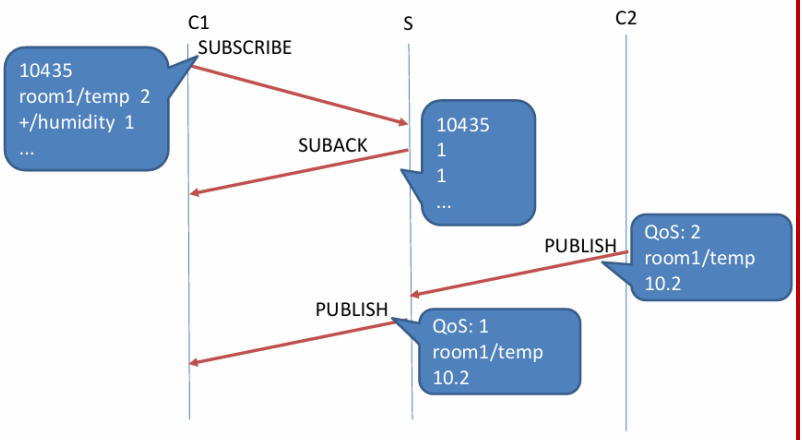
Handshake → when connection is established, a MQTT request/response handshake happens:

- client identification and authN
- association with client session state (if any)
- possibility to set/negotiate client preferences



Operations = after the initial handshake, a client can perform a sequence of operations by exchanging control packets with the broker:

- **Publish** → starts with a publish request (topic + data to be published); 3 possible QoS:
  - 0 = message delivered ***at most once*** (best effort); fire and forget (nothing more than what is guaranteed by TCP about deliver) [PUBLISH]
  - 1 = message delivered ***at least once*** (may deliver multiple copies); receiver acknowledges each delivery. Sender waits for acknowledgment and retransmits after a timeout (sender forgets message only when it receives acknowledgment) [PUBLISH → PUBACK]
  - 2 = message delivered ***exactly once***; in addition to delivery acknowledgment (as withQoS1) a duplication avoidance mechanism is introduced [PUBLISH → PUBREC → PUBREL → PUBCOMP]
- **Subscribe** → one subscription can subscribe a client to one or more topics (expressed by one or more topic filters). For each topic filter, a **max QoS is requested**. The server can grant subscriptions selectively, and downgrade the max QoS for each topic filter
- **Unsubscribe** → UNSUBSCRIBE → UNSUBACK
- **Ping**

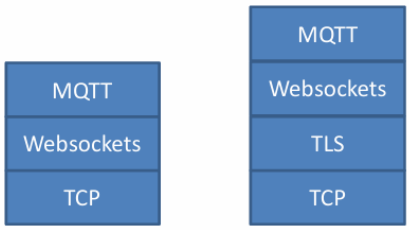


A client terminates a session cleanly by sending a "Disconnect" control packet!

Persistent Storage = broker mantains some state info even for disconnected subscribed clients (persistent sessions [started with clean start flag unset], possibility to mark a publish as RETAINED [boolean flag]). Clients need to keep state information as well (in progress QoS 1 and QoS 2 publish)

⚠ **Last Will** (testament) = ungraceful disconnections of clients can be detected by the broker; if the client that gets disconnected had a will, the broker will execute the will (a will can be sent when connecting to broker [Will topic, Will message, Will QoS, Will retain flag])

⚠ MQTT over Websockets → feature introduced to enable browsers to behave as MQTT clients (because browser APIs don't offer direct access to TCP, but they offer access to websockets)



Each MQTT packet is encapsulated into one or more websocket frames

We program MQTT in JavaScript using **MQTT-js library** (use MQTT over websockets). Vediamo l'API:

- entry point → `mqtt` → connection: `mqtt.connect(url, options)` which returns `mqtt.Client`
- client connection: `mqtt.Client`; events:

event name	meaning	callback parameters
connect	a connection has been opened	(connack)
close	a connection has been closed	()
message	a published message has been received	(topic, message, packet)
error	an error has occurred	(error)

- methods:
  - `publish (topic, message, [options], [callback])`
  - `subscribe (topic, [options], [callback])`
  - `unsubscribe (topic, [options], [callback])`
  - `end([force],[options], [callback])`

Interface Design Approaches (riproposto):

- METHOD centric:
  - Fixed endpoint (programming language approach)
  - Design is about operations and their input/output arguments
- MESSAGE centric (**MQTT**)
  - Fixed, single-operation interface (e.g. send(Message))
  - Design is about messages and endpoints
- CONSTRAINED (hybrid) [es. REST]

⚠ **Granularity level** of msg and topics: fine granularity of topics reduces the load on the broker. We can use error topics (or error msg) to return errors; in fact computing load on the broker →

- Frequency of publish operations: f
- Average number of topic subscribers: n
- Rate of messages (messages per second): f (n+1)
- Reducing topic granularity, n tends to decrease

⚠ **MQTT guidelines**: use plain ASCII in topic names, use topic names ad id to carry part of msg information, leave topic structure open to extension