Introduction to Distributed Computing

Programació Multiplataforma i Distribuïda

Grau d'informàtica. EPSEVG

Novembre 2019

Jordi Esteve jesteve@lsi.upc.edu

Distributed computing

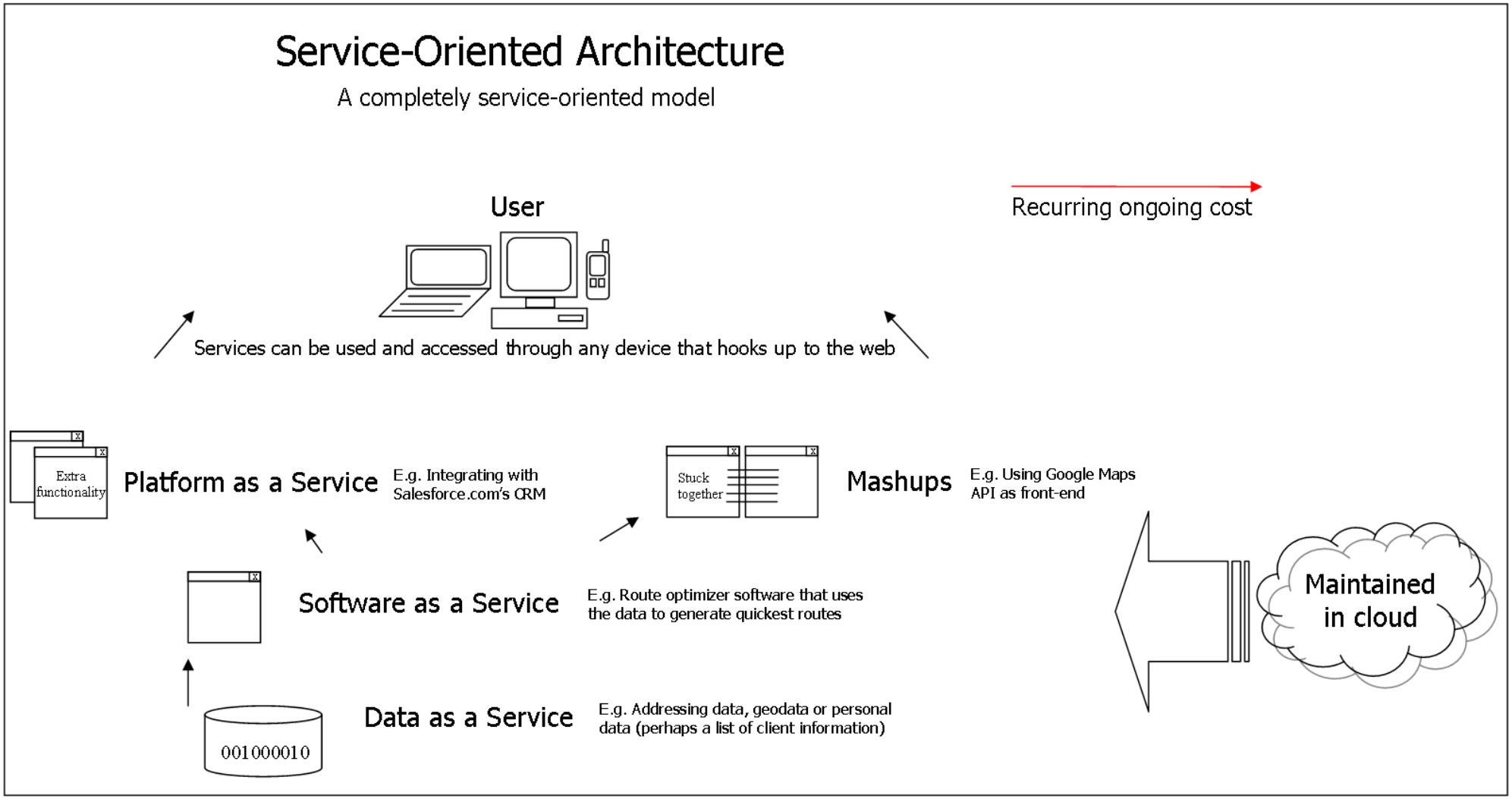
A **distributed system** [1] is a software system whose components are located on **different network computers** which communicate and coordinate their actions by passing **messages** to one another.

The components interact with one another in order to achieve a **common goal**.

Examples of distributed systems:

- Servers and web browsers
- SOA-based systems (Service Oriented Architecture)
- Massively multiplayer online games
- Peer-to-peer applications

Distributed computing: SOA example



Distributed computing: Message mechanisms and Characteristics

Several alternatives to the message passing mechanism:

- **RPC** (Remote Procedure Calls) connectors
- **Queues** of messages

Characteristics of distributed systems:

- **Transparency** of the location of the components
- **Concurrence** of components
- Lack of a global clock (**asynchronous**)
- Be **independent of the failure** of some component

Distributed programming is the process of writing programs that run on distributed systems.

Distributed computing: Advantages

It's often convenient to split applications into several independent parts [2]. This normally leads to several advantatges:

- Cleaner design
- Reduces the complexity
- Improves maintainability
- Enhances security (often)

Especially for web-applications, it's sensible to split the frontend (data-presentation, user-I/O) from the "real" application (multi-tier architecture). This

- Cleanly separates the presentation from the logic
- Simplifies the creation/use of alternative frontends
- Improves security because the frontend (usually running with webserver-rights) doesn't have direct access to the data, but may only call some functions of an other process

Example of a distributed system: A three-tiered application

A **three-tiered** application [3]:

Presentation tier

This is the topmost level of the application. The presentation tier displays information related to such services as browsing products and shopping cart contents. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network. (In simple terms it's a layer which users can access directly such as a web page, or an operating systems GUI)

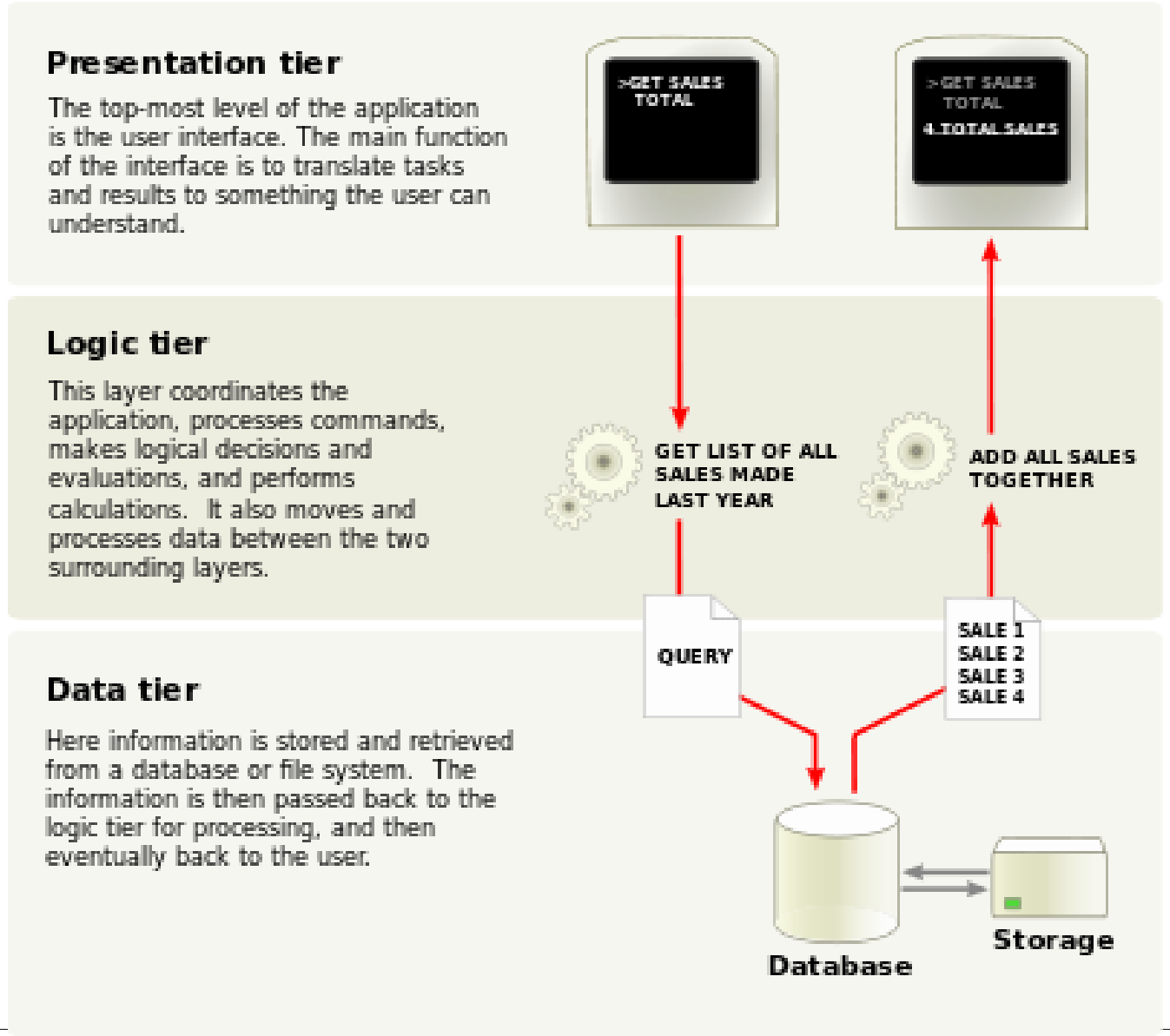
Application tier (buiness logic, logic tier, data access tier, or middle tier)

The logical tier is pulled out from the presentation tier and, as its own layer, it controls an application’s functionality by performing detailed processing.

Data tier

This tier consists of database servers. Here information is stored and retrieved. This tier keeps data neutral and independent from application servers or business logic. Giving data its own tier also improves scalability and performance.

Example of a distributed system: A three-tiered application



Distributed computing. Remote Procedure Call (RPC)

To really separate the parts from each other, we probably want to run the parts in several processes (with different users/rights). But this means that the processes have to communicate with each other in some way. This is called inter-process communication (IPC), and one way of doing this is by using remote-procedure calls (RPC).

Remote Procedure Call (**RPC**) [4] is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called **remote invocation** or **remote method invocation**.

Distributed computing. Events in a RPC

Sequence of events during a RPC

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the server stub.
5. The server stub unpacks the parameters from the message . Unpacking the parameters is called unmarshalling.
6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

Distributed computing. RPC technologies

RPC-system consists of several *independent* parts [2]:

1. **Data structure** (how requests/responses/errors look like)
2. **Serializer** (i.e. JSON, XML, URI, ...)
3. **Transport** (i.e. Unix Domain Socket, TCP/IP, HTTP)
4. **Proxy/dispatcher** (map function-calls to RPC and vice versa)

Many different (often incompatible) technologies have been used to implement the concept:

- XML-RPC is an RPC protocol that uses XML to encode its calls and HTTP as a transport mechanism.
- JSON-RPC is an RPC protocol that uses JSON-encoded messages.
- SOAP is a successor of XML-RPC and also uses XML to encode its HTTP-based calls.
- CORBA provides remote procedure invocation through an intermediate layer called the *object request broker*.
- Pyro object-oriented form of RPC for Python.
- Java's Java Remote Method Invocation (Java RMI) API provides similar functionality to standard Unix RPC methods.

Distributed computing: XML-RPC

XML-RPC [5] is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism.

XML-RPC was created in 1998. As new functionality was introduced, the standard evolved into what is now SOAP.

How XML-RPC works?

- By sending a HTTP request to a server implementing the protocol.
- The client is typically software wanting to call a single method of a remote system.
- Multiple input parameters can be passed to the remote method, one return value is returned.
- The parameter types allow nesting of parameters into maps and lists, thus larger structures can be transported. Therefore XML-RPC can be used to transport objects or structures both as input and as output parameters.

Distributed computing: XML-RPC

Data types: http://en.wikipedia.org/wiki/XML-RPC#Data_types

Examples: http://en.wikipedia.org/wiki/XML-RPC#Examples

Disadvantages:

- RPC calls can be made with plain XML, and that XML-RPC does not add any value over XML. Both XML-RPC and XML require an application-level data model, such as which field names are defined in the XML schema or the parameter names in XML-RPC.
- XML-RPC uses about 4 times the number of bytes compared to plain XML to encode the same objects

Distributed computing: JSON-RPC

JSON-RPC [6] is a remote procedure call protocol encoded in JSON. It is a very simple protocol (and very similar to XML-RPC), defining only a handful of data types and commands. JSON-RPC allows for notifications (data sent to the server that does not require a response) and for multiple calls to be sent to the server which may be answered out of order.

JSON-RPC was created in 2005. The last version 2.0 was revised in 2010.

How JSON-RPC works?

- By sending a request to a server implementing this protocol.
- The client in that case is typically software intending to call a single method of a remote system.
- Multiple input parameters can be passed to the remote method as an array or object, whereas the method itself can return multiple output data as well.
- A remote method is invoked by sending a request to a remote service using HTTP or a TCP/IP socket (starting with version 2.0). When using HTTP, the content-type may be defined as `application/json`.

Distributed computing: JSON-RPC

All transfer types are single objects, serialized using JSON. A request is a call to a specific method provided by a remote system. It must contain three properties:

- `method` - A String with the name of the method to be invoked.
- `params` - An Array of objects to be passed as parameters to the defined method.
- `id` - A value of any type, which is used to match the response with the request that it is replying to.

The receiver of the request must reply with a valid response to all received requests. A response must contain three properties:

- `result` - The data returned by the invoked method. If an error occurred while invoking the method, this value must be null.
- `error` - A specified Error code if there was an error invoking the method, otherwise `null`.
- `id` - The id of the request it is responding to.

Since there are situations where no response is needed or even desired, notifications were introduced. A notification is similar to a request except for the id, which is not needed because no response will be returned. In this case the `id` property should be omitted.

Distributed computing: JSON-RPC examples

Examples for version 2.0

Procedure call with positional parameters:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}
<-- {"jsonrpc": "2.0", "result": -19, "id": 2}
```

Procedure call with named parameters:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3}
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}

--> {"jsonrpc": "2.0", "method": "subtract", "params": {"minuend": 42, "subtrahend": 23}, "id": 4}
<-- {"jsonrpc": "2.0", "result": 19, "id": 4}
```

Notification:

```
--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}

--> {"jsonrpc": "2.0", "method": "foobar"}
```

Procedure call of non-existent procedure or invalid JSON:

```
--> {"jsonrpc": "2.0", "method": "foobar", "id": 10}
<-- {"jsonrpc": "2.0", "error": {"code": -32601, "message": "Procedure not found."}, "id": 10}

--> {"jsonrpc": "2.0", "method": "foobar", "params": "bar", "baz"]
<-- {"jsonrpc": "2.0", "error": {"code": -32700, "message": "Parse error"}, "id": null}
```

Distributed computing: JSON-RPC advantatges

Advantages of JSON-RPC

- Small and simple.
- Lightweight because it uses a simple and compact format for data-serialization: JSON.
- Transport-independent: JSON-RPC can be used with *any* transport, e.g. Unix domain sockets, TCP/IP, http, https, avian carriers, ...
- Unicode: JSON and JSON-RPC support unicode out-of-the-box.
- JSON directly supports Null/None.
- Supports named/keyword parameters.
- Notifications.
- Built-in request-response-matching ("id"-field).
- Integrates extremely easily into e.g. Python or JavaScript.

Distributed computing: SOAP

SOAP (Simple Object Access Protocol) [7] is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on XML Information Set for its message format, and usually relies on other Application Layer protocols, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

SOAP can form the foundation layer of a web services protocol stack, providing a basic messaging framework upon which web services can be built. This XML based protocol consists of three parts:

- an envelope, which defines what is in the message and how to process it
- a set of encoding rules for expressing instances of application-defined datatypes
- a convention for representing procedure calls and responses.

SOAP has three major characteristics:

- **Extensibility** (Security and WS-routing are among the extensions under development).
- **Neutrality** (SOAP can be used over any transport protocol: HTTP, SMTP, TCP, or JMS).
- **Independence** (SOAP allows for any programming model).

Distributed computing: SOAP

Example message: http://en.wikipedia.org/wiki/SOAP#Example_message

Comparison to JSON/XML

- SOAP is capable of representing general graph structures, not just tree structures, of objects
- SOAP messages can be sent to multiple recipients
- SOAP has the ability to encrypt parts of the message so that some recipients but not others can see those parts. (This ability is standardised in XML too but not JSON).
- SOAP has guaranteed message delivery - if a connection fails, it will try to re-send the message.
- Naturally this all comes at a cost of increased complexity.

Advantages

- SOAP is versatile enough to allow for the use of different transport protocols: HTTP, SMTP, ...
- Since the SOAP model tunnels fine in the HTTP post/response model, it can tunnel easily over existing firewalls and proxies, without modifications to the SOAP protocol.

Disadvantages

- When using standard implementations and the default SOAP/HTTP binding, the XML infoset is serialized as XML. Because of the verbose XML format, SOAP can be considerably slower than competing middleware technologies such as CORBA or ICE.

Web Service

A **Web service** [8] is a method of communication between two electronic devices over World Wide Web. A **web service** is a software function provided at a network address over the web or the cloud; it is a service that is "always on".

It is a software system designed to support machine-to-machine interaction over a network typically using HTTP or HTTPS with an XML or JSON serialization in conjunction with other Web-related standards.

Classes of Web services:

- *REST-compliant Web services*, in which the primary purpose of the service is to manipulate XML or JSON representations of web resources using a uniform set of "stateless" operations. They use operations (GET, POST, PUT, ...) and other existing features of the HTTP protocol.
- *Arbitrary Web services*, in which the service may expose an arbitrary set of operations. These application-specific operations supplant HTTP operations. For example the previous XML-RPC, JSON-RPC or SOAP over HTTP protocol could be consider arbitrary Web services.

RESTful Web Services

Representational state transfer (**REST**) [9] is a software architectural style that defines a set of constraints to be used for creating Web services.

Web services that conform to the REST architectural style, called ***RESTful* Web services**, provide **interoperability** between computer systems on the Internet.

RESTful Web services allow the requesting systems to access and manipulate textual representations of **Web resources** by using a uniform and predefined set of **stateless operations**:

- **Web resources** were first defined on the WWW as documents or files identified by their **URLs**, but now can be a response formatted in HTML, XML, JSON, …
- The **stateless operations** (HTTP methods) available are GET, HEAD, POST, PUT, PATCH, DELETE, CONNECT, OPTIONS and TRACE.

RESTful Web Services constraints

The constraints of the REST architectural style are:

- **Performance** in component interactions.
- **Scalability** allowing the support of large numbers of components and interactions among components.
- **Simplicity** of a uniform interface.
- **Modifiability** of components to meet changing needs (even while the application is running).
- **Visibility** of communication between components by service agents.
- **Portability** of components by moving program code with the data.
- **Reliability** in the resistance to failure at the system level in the presence of failures within components, connectors, or data.

RESTful Web Services specific constraints (I)

1. **Client-server architecture**: The separation between client and server improves the portability across multiple platforms and scalability by simplifying the server components. And the separation allows the components to evolve independently (modificability).
2. **Statelessness**: The client-server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and the session state is held in the client. This improves scalability and performance.
3. **Cacheability**: Clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data in response to further requests. This improves scalability and performance.
4. **Layered system**: A client doesn't know if it's talking with an intermediate or the actual server. Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches.

RESTful Web Services specific constraints (II)

5. **Code on demand** (optional): Servers can temporarily extend or customize the functionality of a client by transferring executable code: for example client-side scripts such as JavaScript.
6. **Uniform Resource Interface** (URI): It simplifies and decouples the architecture, which enables each part to evolve independently. It has four constraints:
   - **Resource identification in requests**: Individual resources are identified in requests, for example using URIs in RESTful Web services.
   - **Resource manipulation through representations**: When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.
   - **Self-descriptive messages**: Each message includes enough information to describe how to process the message. For example, which parser to invoke can be specified by a media type (HTML, XML, JSON, ...).
   - **Hypermedia as the engine of application state** (**HATEOAS**): Having accessed an initial resource with a URI, a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs.

RESTful Web Services: RESTful APIs

Web service APIs that adhere to the REST architectural constraints are called RESTful APIs.

HTTP-based RESTful APIs are defined with the following aspects:

- a **base URI**, such as http://api.example.com/collection/.
- **standard HTTP methods** (e.g., GET, POST, PUT, PATCH and DELETE).
- a **media type** that defines state transition data elements (e.g., application/json). The current representation tells the client how to compose requests for transitions to all the next available application states.

RESTful Web Services: Common URIs (in gray the lest used ones)

HTTP METHODS	Collection resource, such as `http://api.example.com/tasks/`	Member resource, such as `http://api.example.com/tasks/item3`
GET	*Retrieve* the URIs or representations of the member **resources** of the collection.	*Retrieve* representation of the member **resource**.
POST	*Create* a member **resource** in the collection.	*Create* a member **resource** in the collection.
PUT	*Replace* all the representations of the member **resources** of the collection, or *create* the collection if it does not exist.	*Replace* all the representations of the member **resource** or *create* the member resource if it does not exist.
PATCH	*Update* all the representations of the member **resources** of the collection, or *may create* the collection if it does not exist.	*Update* all the representations of the member **resource**, or *may create* the member resource if it does not exist.
DELETE	*Delete* all the representations of the member **resources** of the collection.	*Delete* all the representations of the member **resource**.

RESTful Web Services: Common URIs

The GET method is **safe**, meaning that applying it to a resource does not result in a state change of the resource (read-only). Also HEAD, TRACE and OPTIONS are safe methods.

The GET, PUT and DELETE methods are **idempotent**, meaning that calling them multiple times to a resource we get the same result as calling them once, though the response might differ. POST and PATCH are not idempotent (PATCH can be idempotent, but not always).

Example of URIs for managing tasks:

Method	**URL**	**Action**
GET	/tasks	Retrieve all tasks
GET	/tasks/1	Retrieve task with id == 1
POST	/tasks	Add a new task
PUT	/tasks/1	Update task with id == 1
DELETE	/tasks/1	Delete task with id == 1

References

[1] http://en.wikipedia.org/wiki/Distributed_computing

[2] http://www.simple-is-better.org/rpc/

[3] http://en.wikipedia.org/wiki/Multitier_architecture

[4] http://en.wikipedia.org/wiki/Remote_procedure_call

[5] http://en.wikipedia.org/wiki/XML-RPC

[6] http://en.wikipedia.org/wiki/JSON-RPC

[7] http://en.wikipedia.org/wiki/SOAP

[8] http://en.wikipedia.org/wiki/Web_service

[9] https://en.wikipedia.org/wiki/Representational_state_transfer