# REST Architectural Style

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

# Communication Architecture

- Saying that we have, e.g., distributed objects, does not tell much.
- It describes a programming paradigm oriented mostly towards remote invocation.
- Says nothing about actual communication architecture:
  - a few monolithic distributed objects? many small objects?
  - chatty RPC's with lots of ping-pong or bulky RPC's?
  - do we maintain session-state or are servers stateless?
  - do objects form a broker network to relay messages?
  - do we use any form of caching by intermediate proxies?

# Architectural styles in distributed systems

- Communication patterns between distributed components define architectural styles of distribution.
- An architectural style is defined as a set of constraints (of what is allowed to happen).
- Roy Fielding (one of the HTTP/1.1 authors) wrote the dissertation *Architectural Styles and the Design of Network-based Software Architectures*.
- Defines a taxonomy of architectural styles:
  - data-flow styles;
  - replication styles;
  - hierarchical styles;
  - mobile code styles;

# Data-flow styles

Pipe and Filter (PF): components form a *pipeline*, each acting as a filter; each reads a stream of data, produces stream of data; no state sharing between components;

Uniform Pipe and Filter (UPF): special case of PF; added constraint that all filters have the same interface; example: classic Unix pipes over byte streams;

# Replication styles

- Replication improves availability and scalability;
- Examples: *domain name system* (DNS), USENET news (nntp);

**replicated repository (RR):** several servers provide same service, providing ilusion that there is only one; example: distributed file systems, remote versioning systems;

**cache ($):** variant in which result of individual request is kept to be reused by later requests, possibly from other clients; important for scalability, e.g., of WWW;

# Hierarchical styles

client-server (CS): basic model from which others derive;

layered-client-server (LCS): a server is client of other servers; introduces *overhead* and latency; allows intermediaries like *gateways* and *proxies*; style of classic n-tier systems;

client-stateless-server (CSS): special case of CS in which there is no session state about clients in servers;

client-cache-stateless-server (C$SS): combines $ with CSS; improves performance; example: Sun NFS;

layered-client-cache-stateless-server (LC$SS): example: DNS;

remote session (RS): client initiates session in server, invokes commands and exits; state is kept on server;

remote data access (RDA): client invokes series of operations that manipulate state on server or return parts of state; e.g., SQL commands;

# Mobile code

Brings processing closer to data, improving efficiency and remote resource utilization.

virtual machine (VM): same execution environment in each node; adds security and management problems; examples: applets in JVM, scripts in Javascript, PERL, Python.

remote evaluation (REV): combines *client-server* and *virtual machine*; client passes code and data to server, which performs computation and returns result;

code on demand (COD): client ask server for code and executes it locally; facilitates extensibility/upgrading of clients;

layered-code-on-demand-client-cache-stateless-server (LCODC$SS): example: Web browser that allows applet downloading and protocol extensions;

mobile agent (MA): derivation of REV and COD, where an entire component, with code and data is moved to another site, possibly in the middle of a computation;

# REST: the architectural style of the WWW

- The most successful distributed service has been the WWW.
- Roy Fielding characterizes WWW architecture as REST.
- REST: Representational State Transfer
  - uniform interface, with a fixed set of operations;
  - resources and uniform resource identifiers (URIs);
  - hypermedia as the engine of application state (HATEOAS);
  - *layered client cache stateless server (LC$SS)* style, with optional *code on demand (COD)*;
- These architectural constraints allowed extreme scalability, being important towards success of WWW.

# REST: uniform interface

- In REST, all components present same interface, with fixed set of operations, with universal semantics, regardless of application;
- E.g., in HTTP we have: GET, PUT, POST, DELETE, ...
- Contrasts with RPC/RMI systems, where diferent interfaces keep being defined for diferent distributed objects that may co-exist;
- Fixed interface important for long term evolution of large scale independently managed systems without global coordination;
- Universal semantics allows using generic intermediaries (like *proxies* and *gateways* in WWW), important for scalability;

# REST: resources and uniform resource identifiers

- REST is resource-oriented: a distributed system is organized as a set of resources;
- Each resource has a unique identifier (URI), in a single global name-space;
- All resources are accessed in a uniform way, through the same fixed interface;
- An application is designed in terms of resource design (definition, relationships); not in terms of interface definition;

# REST: hypermedia as the engine of application state (HATEOAS)

- Operations manipulate resources through *representations*;
- Example: an HTTP GET on a resource *transfers* a representation of that resource;
- Representations are manipulated through preferably standardized *media-types*;
- A representation may contain *links* to other resources;
- A client can progress by using one of the links in an obtained representation to invoke an operation on the respective resource;
- Hypermedia defines possible state transitions of clients;

# REST: layered client cache stateless server (LC$SS)

- Interactions are stateless, with no conversational state:
  - a request must contain all information to be understood;
  - a component responds to a request without having to know previous requests from the client;
  - session state is kept in the client;
- Server state is only resource-related, not client-session related, leading to greater scalability and fault-tolerance;
- Intermediaries can understand each request separately;
- Clients or intermediaries can cache state when appropriate, improving scalability;

# Non-RESTful Web Serviços

- Distributed objects middleware (DCOM, CORBA, RMI) was not successful for interaction between independent organizations;
- Given the success of WWW, standards were defined using XML and working over HTTP: the *Web Services* (WS-*);
- Those standards focus over aspects such as:
  - remote invocation (SOAP);
  - service definition: Web Services Description Language (WSDL);
  - discovery: Universal Description Discovery and Integration (UDDI);

# Non-RESTful Web Services critique

In the beginning of 2000s there was considerable polemic over WS-*:

- WWW characteristics, present in REST architectural style, that made it successful are not respected;
- Example: interface definition and RPC orientation;
- Instead of embracing WWW, WS-* merely use HTTP as a transport protocol, being unusable from a plain web application;
- Example: tunneling all requests through POST, with huge XML marshalling overhead, and not exploting safety or idempotency;
- The use of HTTP as a transport protocol is formally incorrect;

*“HTTP is not a Transport Protocol” [Roy Fielding]*

# WS-* versus RESTful use of HTTP + XML/JSON

- Oponents of WS-* defend that:
  - instead of using a stack isolated from the web (SOAP, WSDL, UDDI) and interface-oriented;
  - WWW integration should be promoted, through the use of HTTP + XML/JSON according to REST principles (resource orientation);
- WWW integration and correct HTTP use brings advantages:
  - reutilization of existing mechanisms such as intermediaries (e.g., proxies), authentication, or access control;
  - allowing standard web clients (e.g., browsers, libcurl);
  - exploiting HTTP semantics (safety, idempotency) for scalability;

# HTTP methods

Main methods:

| | |
|---|---|
| GET | retrieves resource representation; |
| PUT | updates resource or creates new resource; |
| POST | appends to resource or creates subordinate resource; |
| DELETE | removes resource; |

Other methods:

| | |
|---|---|
| PATCH | modifies an existing resource; |
| HEAD | retrieves metainformation about resource; |
| OPTIONS | retrieves communication options for resource; |
| TRACE | traces request along proxies; |

# HTTP safe and idempotent methods

- A *safe* method should not have a visible effect upon resources (modify resource representation or create/remove resources)
- Method GET should be safe; never modify resources;
- An *idempotent* method should have the same effect if invoked several times (ignoring concurrent invocations by others);
- Methods GET, PUT and DELETE should be idempotent;
- POST has a “create new or append” semantics, being neither safe nor idempotent;

# HTTP safe and idempotent methods

| Method | Safe | Idempotent |
|---|---|---|
| GET | yes | yes |
| HEAD | yes | yes |
| OPTIONS | yes | yes |
| TRACE | yes | yes |
| PUT | no | yes |
| DELETE | no | yes |
| POST | no | no |
| PATCH | no | no |

# HTTP Status-Codes

- HTTP defines status-codes to be returned, to represent outcome;
- Most common fall in four classes: 2XX, 3XX, 4XX, 5XX;
- 2XX means successfully received, understood and accepted;
- 3XX means redirection; repeat request to new URI;
- 4XX means client error (e.g., not found, authorized or allowed);
- 5XX means server error, possibly temporary;

# Some common status-codes

| |
|---|
| 200 OK<br>201 Created<br>204 No Content |
| 301 Moved Permanently<br>304 Not Modified |
| 400 Bad Request<br>401 Unauthorized<br>404 Not Found<br>405 Method Not Allowed<br>409 Conflict<br>412 Precondition Failed<br>415 Unsupported Media Type |
| 500 Internal Server Error<br>501 Not Implemented<br>503 Service Unavailable |

# GET method

- Should simply retrieve representation;
- Should be safe and idempotent;
- Should not modify, create or remove resources;
- Can be *conditional GET* using headers like If-Modified-Since;
- Conditional GET reduces network usage, refreshing client/cache without transfering representation;
- May be a *partial GET* using Range header, allowing partial data transmission; e.g. to restart transmission after network error;

# PUT method

- Sends entity to update resource of given URI;
- Should be idempotent;
- May also be used to create new resource with a given client defined URI (returning “201 Created” in this case);
- Invalidades cache entry for URI that may exist; response is not cacheable;

# POST method

- Sends entity to append to resource or create subordinate;
- Used to update or create resource in a non-idempotent way;
- Can add information to existing resource; examples:
  - increment counter;
  - add item to list;
- Can create subordinate resource with server-defined URI:
  - create new thread in forum;
  - create reply comment as child of another in a forum thread;
  - create new record in database table with server-defined key;
- When creating new resource should:
  - return “201 Created”;
  - return URI of created resource in “Location” header;

# DELETE method

- Removes resource;
- Is naturally idempotent;
- Invalidades cache entry for URI; response is not cacheable;

# Pointers

- Roy Fielding dissertation:
  `www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`
- HTTP/1.1:
  `www.w3.org/Protocols/rfc2616/rfc2616.html`
- Webmachine:
  `www.github.com/basho/webmachine`
- Dropwizard:
  `dropwizard.io`
- Example: Amazon S3 REST API:
  `docs.aws.amazon.com/AmazonS3/latest/API/APIRest.html`
- Example: last.fm REST API:
  `www.last.fm/api/rest`