Services and communications

Davide Rossi
Dipartimento di Informatica – Scienze e Ingegneria
Università di Bologna

Topics

- Communication styles
- REST
- Binary messaging

Communication styles

- Explicit addressing
  - Request/response (a.k.a. RPC)
    - Sync: the response uses the request’s channel
    - Async: the response uses a different channel (the caller must expose a response endpoint)
  - Streaming
- Implicit addressing
  - Queues
  - Publish/subscribe channels (topics)
  - Logs

Communication styles

- Explicit addressing
  - Request/response (a.k.a. RPC)
    - Sync: the response uses the request’s channel
    - Async: the response uses a different channel (the caller must expose a response endpoint)
  - Streaming
- Implicit addressing
  - Queues
  - Publish/subscribe channels (topics)
  - Logs

Message brokers

Distributed event logs / streaming platforms

Communication styles

- Explicit addressing
  - Request/response (a.k.a. RPC)
    - Sync: the response uses the request’s channel
    - Async: the response uses a different channel (the caller must expose a response endpoint)
  - Streaming
- Implicit addressing
  - Queues
  - Publish/subscribe channels (topics)
  - Logs

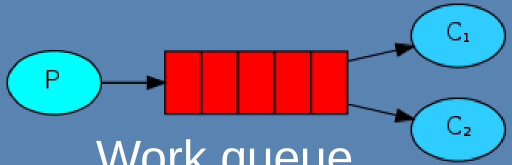
Event-driven architectural style

Event-driven architectural style

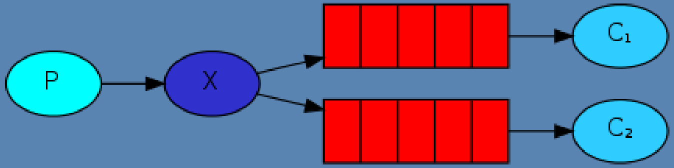
- The main components are event **producers** and event **consumers**, which are decoupled (producers are unaware of consumers and vice versa)
- **Pub/sub model**: consumers register (subscribe) with a message infrastructure. Generated (published) events are dispatched to registered consumers. Subscribers only receive messages published after their subscription
- **Event streaming model**: generated events are persisted in a ***log*** where they are stored in order (typically within a *partition*). Clients can process the events by retrieving them from the stream and are responsible for positioning (which means that they can replay events generated at any time)

Message brokers

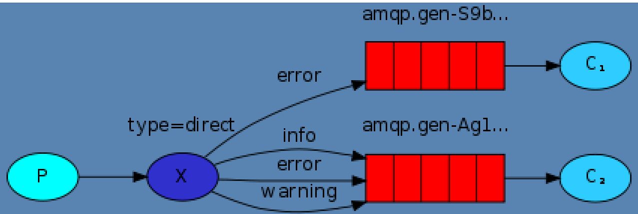
Queue



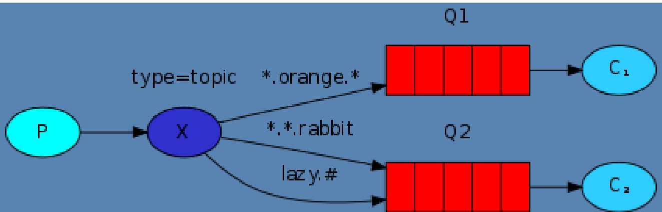
Work queue
(competing consumer group)



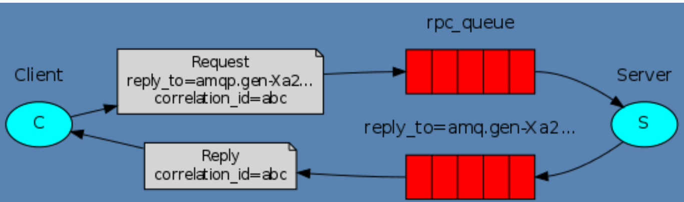
Publish/subscribe



Routing (queue filter)



Topics (pattern matching filter)



RPC

REST

REST is an architectural style.

The largest known implementation of a system conforming to the REST architectural style is the World Wide Web.

REST exemplifies how the Web's architecture emerged by characterizing and constraining the macro-interactions of the four components of the Web, namely origin servers, gateways, proxies and clients, without imposing limitations on the individual participants. As such, REST essentially governs the proper behavior of participants.

Architectural Style

A set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global **constraints** on the way the composition is done.

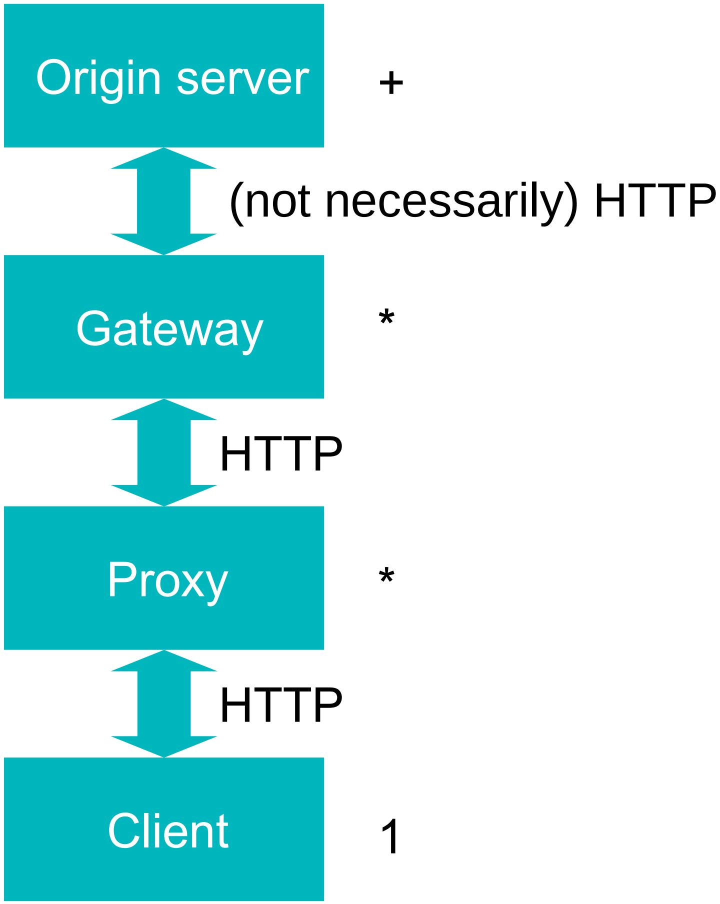
[Shaw & Clements, 1996]

REST constraints

The REST architectural style describes the following six constraints applied to the architecture, while leaving the implementation of the individual components free to design:

- Client–server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand (optional)

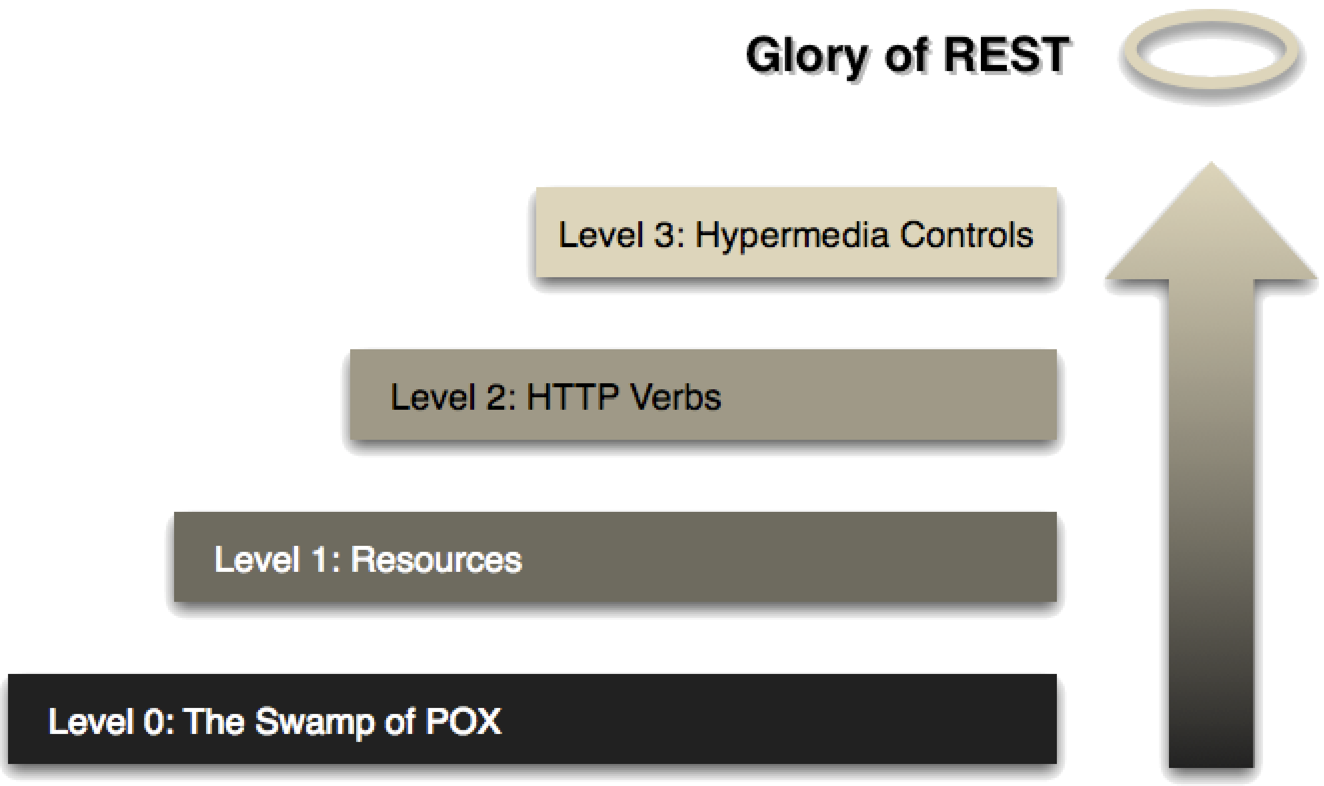
(Simplified) Web architecture



REST and the Web

REST	Web
Client–server	HTTP
Stateless	HTTP
Cacheable	HTTP/Web architecture
Uniform interface	HTTP/URI/HATEOAS/ content negotiation
Layered system	Web architecture
Code on demand	Javascript (?)

Richardson Maturity Model



HATEOAS

- Hypermedia as the engine of application state (HATEOAS): resource representations contain links to other resources.
- Links make interconnected resources navigable. Identifying new resources is service-specific. RESTful applications navigate instead of calling. Representations contain information about possible traversals. The application navigates to the next resource depending on link semantics. Navigation can be delegated since all links use identifiers.

REST and CRUD

Create	POST	Create a resource
Read	GET	Retrieve the current state of a resource
Update	PUT (PATCH)	Initialize or update the state of a resource
Delete	DELETE	Remove a resource

REST factory pattern

If we want to create a resource we could use PUT but we have to make sure that the new URI we provide is not already allocated and that the server is not planning on using it in the future.

The POST factory pattern can help in this context:

```
→ POST /resource

← 201 Created
      Location: /resource/<newid>
```

Example: Doodle

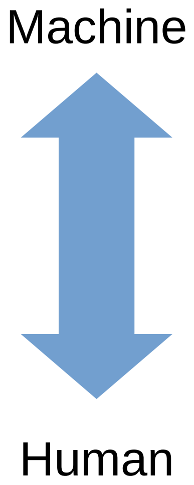
	GET	PUT	POST	DELETE
/poll	V	X	V	X
/poll/{pid}	V	V	X	V
/poll/{pid}/vote	V	X	V	X
/poll/{pid}/vote/{vid}	V	V	X	?

REST interface

- Within RMML2 resource paths are contracts
- If a server moves a resource to another path the clients break
- Thus, **paths are interfaces**
- Yet there is no standard notation for REST interfaces
  - Usual approach: read the docs. Well, docs are not machine processable!

Describing REST API

- WADL (W3C)
- RAML
- OAS - OpenAPI Specification (formerly Swagger)
- API Blueprint



All formats are machine processable and can be used to generate documentation and development artifacts (skel/proxy, ...)

OAS Petstore

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: A paged array of pets
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
    post:
      summary: Create a pet
      operationId: createPets
      tags:
        - pets
      responses:
        '201':
          description: Null response
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
  /pets/{petId}:
    get:
      summary: Info for a specific pet
      operationId: showPetById
      tags:
        - pets
      parameters:
        - name: petId
          in: path
          required: true
          description: The id of the pet to retrieve
          schema:
            type: string
      responses:
        '200':
          description: Expected response to a valid request
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pet"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
components:
  schemas:
    Pet:
      type: object
      required:
        - id
        - name
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
    Pets:
      type: array
      items:
        $ref: "#/components/schemas/Pet"
    Error:
      type: object
      required:
        - code
        - message
      properties:
        code:
          type: integer
          format: int32
        message:
          type: string
```

Application state

- Problem: how does a client know that the path for adding a vote is `/poll/{pid}/vote`?
- REST has no standard service contract
- HATEOAS to the rescue

→ GET /poll/12345
← 200 OK
Content-Type: application/xml
…
<poll>
…
    <link rel=”vote” href=”/poll/12345/vote” />
…
</poll>

RMML3 interfaces


```json
{
    "_links": {
        "self": { "href": "/orders/523" },
        "warehouse": { "href": "/warehouse/56" },
        "invoice": { "href": "/invoices/873" },
        "redo": { "href": "/invoices/873/redo" }
    },
    "currency": "USD",
    "status": "shipped",
    "total": 10.20
}
```

RMML3 interfaces


```json
{
    "_links": {
      "self": { "href": "/orders/523" },
      "warehouse": { "href": "/warehouse/56" },
      "invoice": { "href": "/invoices/873" },
      "redo": { "href": "/invoices/873/redo" }
    },
    "currency": "USD",
    "status": "shipped",
    "total": 10.20
}
```


The contracts in RMML3 are *links' names*

RMML3 interfaces

- **Ontologies are the interfaces** in RMML3 REST APIs
- They can be OWL ontologies, JSON-LD vocabularies, folksonomies, it does not matter. But these are your interfaces.

REST and SOA

- The promise of a lightweight approach to (web) services with diminished burden for the clients holds only within the framework of a specific business model:
  - Large asymmetry between providers and consumers
  - Best effort is a good-enough QoS guarantee

REST and SOA

- Request/reply model only, no asynchronous calls, no server-initiated messages, no publish/subscribe
- No application level encryption
- No reliable messaging (which plays badly with non-idempotent requests)

REST and SOA

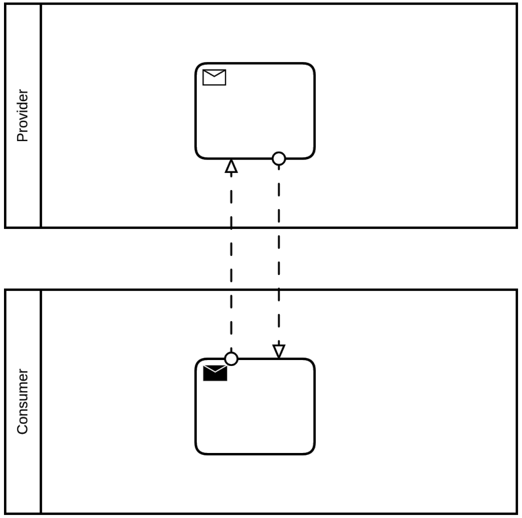
- No distributed transactions
  - See, for example, “A TS-Based 2PC for Web Services Using Rest Architectural Style” by Luiz A. Hiane S. Maciel and Celso M. Hirata or “Atomic distributed transactions: a RESTful design” by Guy Pardon and Cesare Pautasso.
- No concurrency control
- No contract (no service description language)
  - WSDL 2.0, WADL, OAS, RAML, …

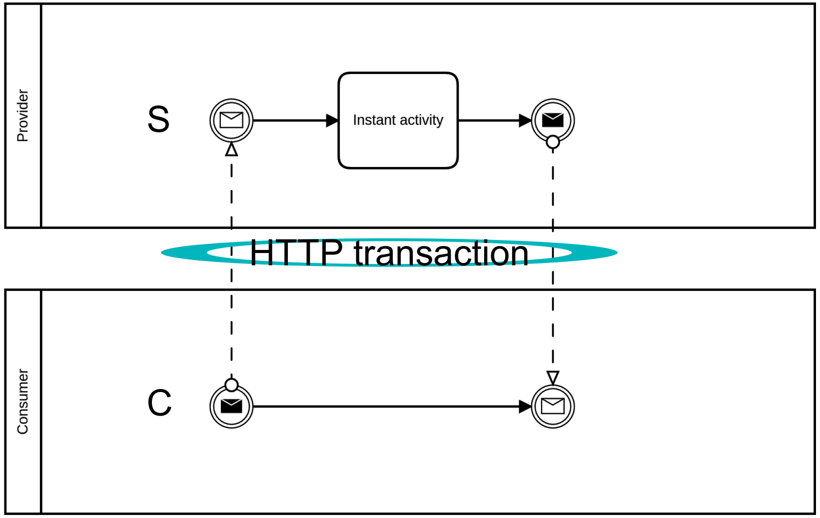
REST and SOA

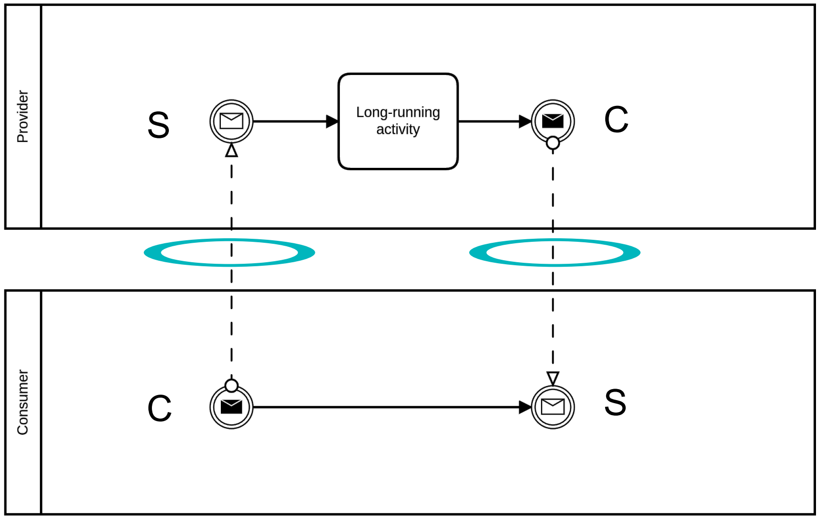
- When QoS is relevant, REST is essentially moving concerns from servers/infrastructure to clients
- More burden for clients developers
- BUT: more awareness too, think, for example, about *graceful degradation*

Request/reply model only

- Issues
  - HTTP is client/server and client-initiated
  - To receive messages/events/notifications you have to be (1) addressable and (2) able to receive (and process) HTTP requests
- Solutions/workarounds
  - REST (Web) Hooks (requires 1 and 2)
  - Long polling (no requirements)
  - Server-Sent Events tunneling (hackish)
  - Notice that WebSockets are not RESTful


Provider
Consumer


Provider
S
Instant activity
HTTP transaction
Consumer
C


Provider
S
Long-running activity
C
Consumer
C
S

Long polling

- Issues with polling
  - Short time between retries
    - Network and server overload
  - Long time between retries
    - Clients are not notified promptly
- Long polling
  - When the resource is not available, do not immediately return but keep the HTTP connection open until the resource is available (or until a timeout, in which case the client should retry).
  - See RFC 6202 for known issues

HTTP methods properties

	SAFE	IDEMPOTENT
GET	YES	YES
PUT	NO	YES
POST	NO	NO
DELETE	NO	YES

POST Once Exactly

- Use GET to retrieve the URL of a one-shot endpoint
- POST to the endpoint
- In case of failure retry the POST
  - `202 – Accepted`
    it had not been processed before
  - `405 – Operation Not Permitted`
    the endpoint has been used already

POST/PUT creation

- A more intuitive (and safer) way of dealing with POST failures when creating resources
- `POST /factory`
  - Returns the URI of a new empty resource `/resource/xyz`
- `PUT /resource/xyz`
  - Initializes the resources - this includes whichever state-changing operation on the server side

Slow creation

```
→ POST /job
← 202 Accepted
  Location: /job/42

→ GET /job/42
← 200 OK
```

or

```
← 303 See Other
  Location: /job/42/output
```

Optimistic concurrency control

- Add €100 to your bank account
  - `GET /account/xyz/amount → amount`
  - `newAmount = amount + 100`
  - `PUT newAmount → /account/xyz/amount`
- What if, in the meanwhile, somebody else tries the same?

Optimistic concurrency control

Use ETags and HTTP conditional requests

```
GET /account/xyz/amount
200 – OK
ETag: qwerty

PUT /account/xyz/amount
If-Match: qwerty
```

Try-Confirm/Cancel

Distributed transactions the REST way

- Create explicit representations of to-be-confirmed operations as resources (**Try**)
- **Confirm** by PUTting on these resources (PUT is idempotent, you can use retry)
- **Cancel** is performed by the services themselves after a timeout
- GET on an intermediate resource returns time left before cancel

Binary messaging

- JSON is designed to be human-readable
- JSON is the lingua franca in the API world
- JSON parsing can be the largest workload of a REST API

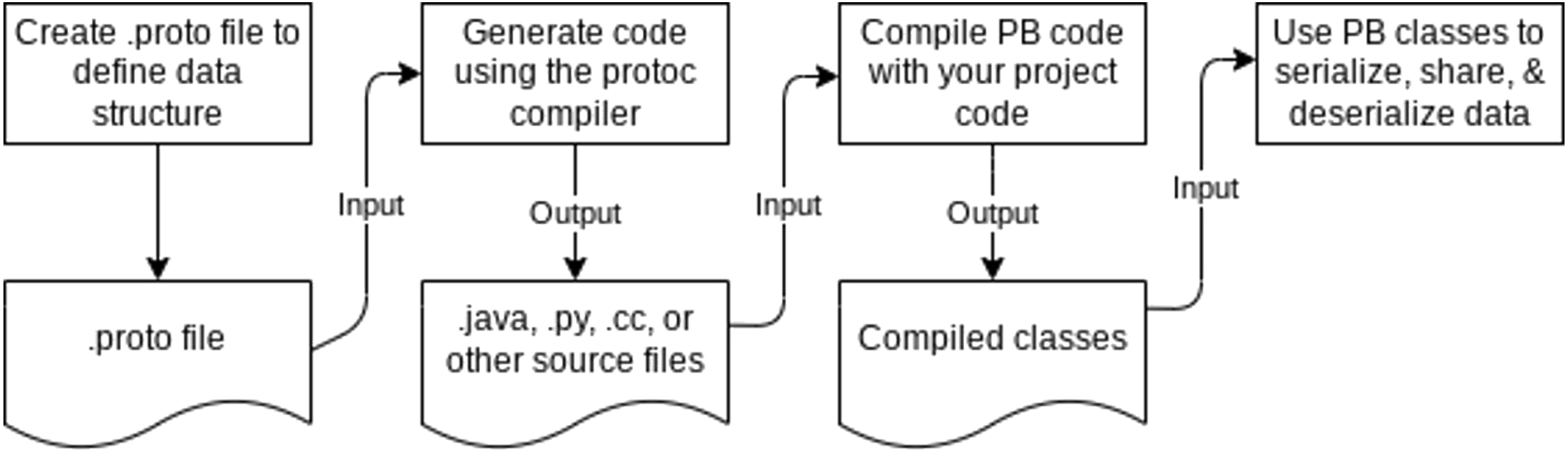
Binary alternatives

- Several binay formats have been proposed, only a few are designed to really promote interoperability
- Usual approach
  - Define a schema with a type definition language
  - Map types to specific programming languages structures

Binary serialization

- CORBA (1991 - serialization is just a part of the whole standard)
- Apache Thrift (originally a FB project)
- Protocol Buffers (originally a Google project)
- Apache Avro (from the Hadoop project)
- ...

Protocol buffers workflow



.proto file example

```
message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;
}
```

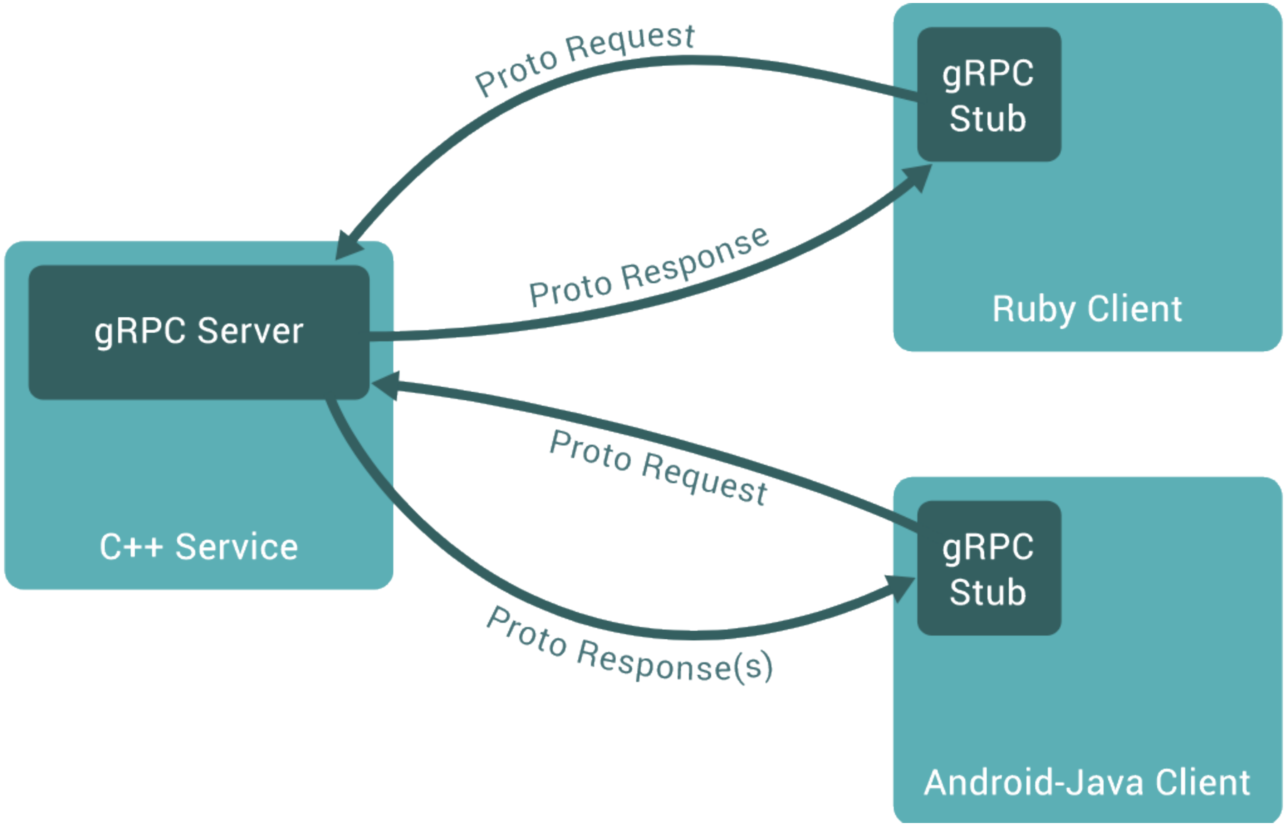
RPC vs binary serialization

- Binary serialization does not imply the use of a specific communication infrastructure
- You can do REST using a binary format to transfer resources
- Some binary serialization systems include an RPC stack (Thrift) or have a preferred RPC stack (like gRPC and Protocol Buffers)

gRPC

- gRPC is an RPC framework that uses protocol buffers to specify and (de)serialize data
- It’s implemented on top of HTTP/2
- Its main distinctive feature is the support for **bi-directional streaming**
- It directly supports C#, C++, Dart, Go, Java, Kotlin, Node, Objective-C, PHP, Python and Ruby

gRPC



Proto Request
gRPC Stub
Proto Response
Ruby Client
gRPC Server
C++ Service
Proto Request
gRPC Stub
Proto Response(s)
Android-Java Client