التجاري وفا بنك
Attijariwafa bank

OMNISHORE

The Evolution of Web Communication: Understanding gRPC, REST, and Diving Deep into HTTP Protocols

BRAHIM MANNETA

IT Tricks

Introduction

Web communication has evolved dramatically over the past decades, driven by increasing demands for:

- Real-time interactions
- Lower latency
- Higher efficiency
- Better resource utilization

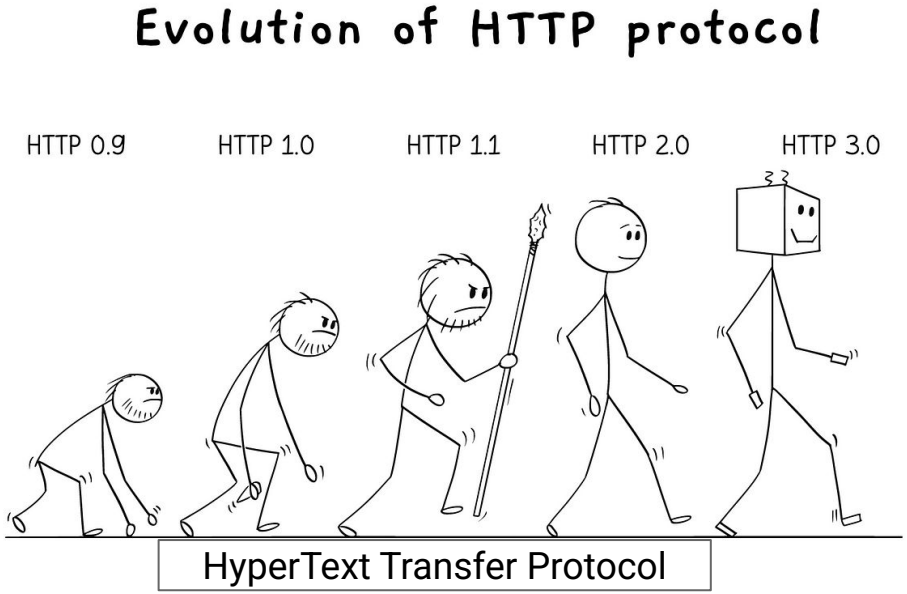
Historical Context & Current Landscape

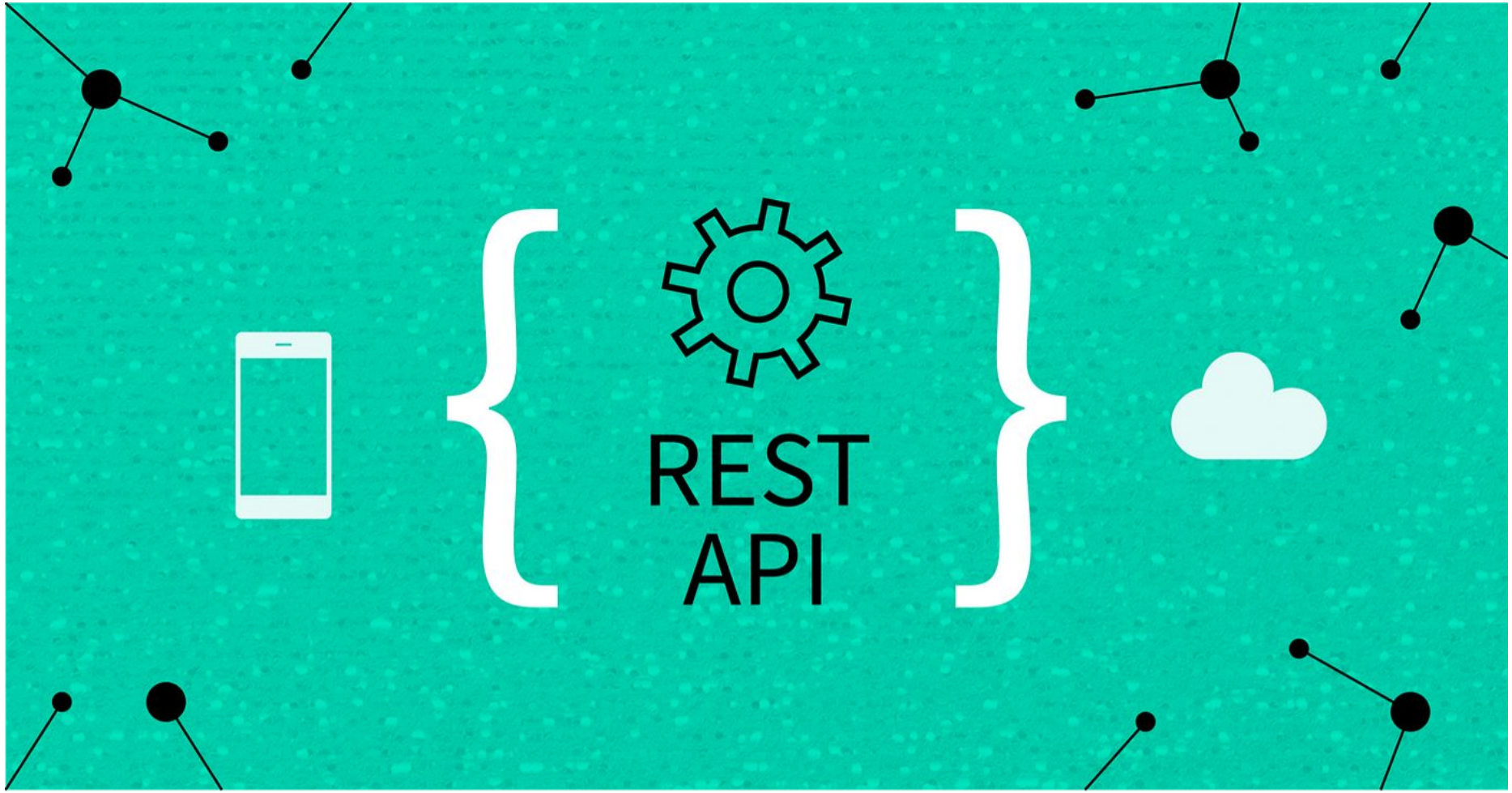
The Journey of Web APIs and Protocols:

- Early 2000s: SOAP and XML-RPC
- 2000s: Rise of REST and JSON
- 2015: HTTP/2 standardization
- 2015: gRPC introduction by Google
- 2022: HTTP/3 (QUIC) standardization

Today's Challenges:

- Microservices architecture complexity
- Mobile-first applications
- Real-time streaming needs
- Global scale deployment
- Low-latency requirements




REST
API

REST Core Concepts

Understanding REST (Representational State Transfer)

Core Principles:

- Uniform Interface
  - Resources identified through URIs
  - Resource manipulation through representations
  - Self-descriptive messages
  - HATEOAS (Hypertext As The Engine Of Application State)

- Stateless Communication
  - Each request contains all necessary information
  - No client context stored on server
  - Improves scalability

```
# REST over HTTP/1.1
GET /api/users/123 HTTP/1.1
Host: api.example.com
Accept: application/json
Authorization: Bearer eyJhbGc...

# REST over HTTP/2
:method: GET
:path: /api/users/123
:scheme: https
:authority: api.example.com
accept: application/json
authorization: Bearer eyJhbGc...

# Response (same format for both, but HTTP/2 offers advantages)
# - Multiplexing (multiple requests over single connection)
# - Header compression (HPACK)
# - Server push capability
# - Binary protocol instead of text-based
HTTP/2 200 OK
content-type: application/json
{
    "id": 123,
    "name": "John Doe",
    "email": "john@example.com",
    "created_at": "2024-01-17T10:30:00Z"
}
```

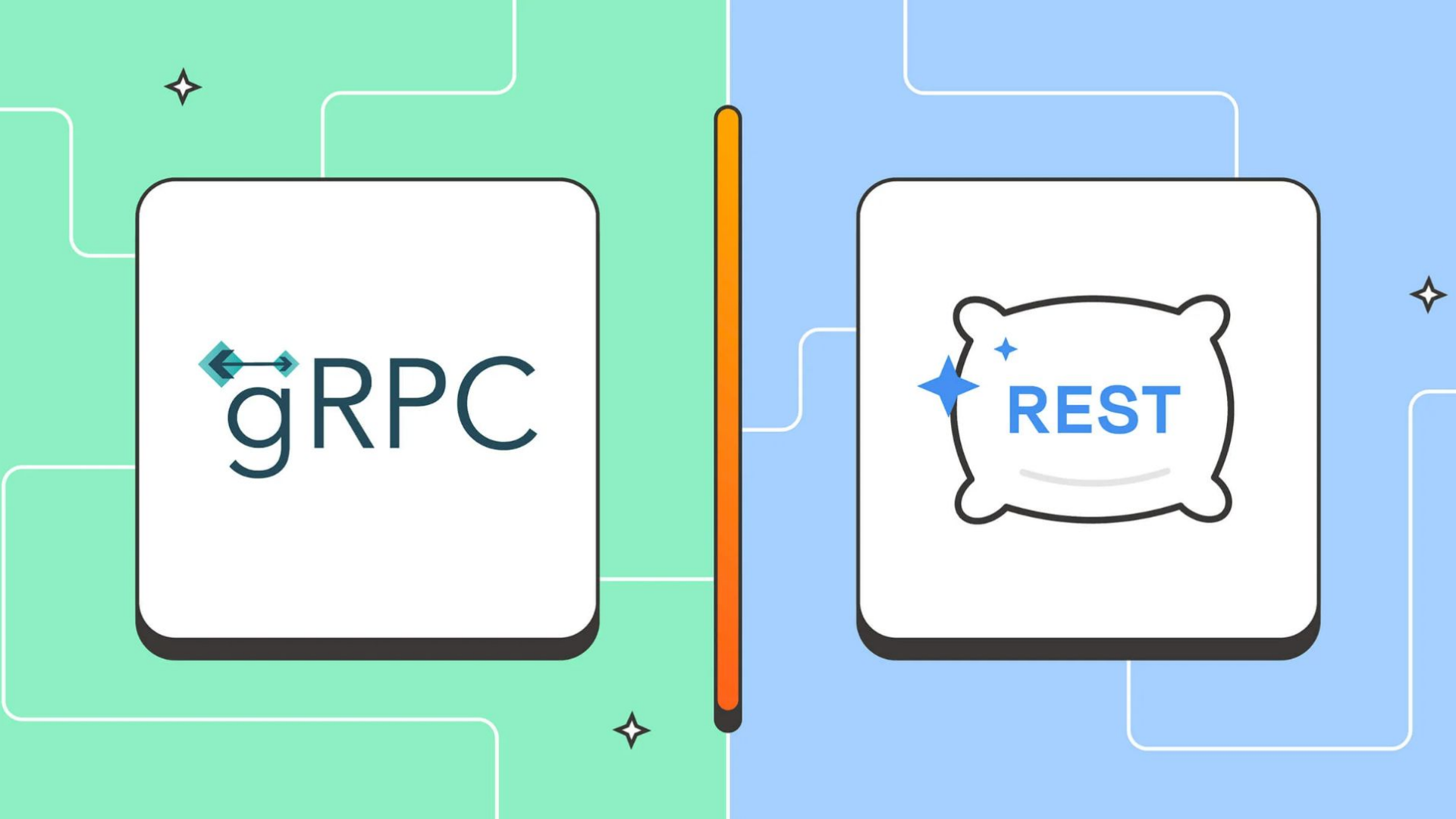

gRPC

gRPC Introduction

What is gRPC?

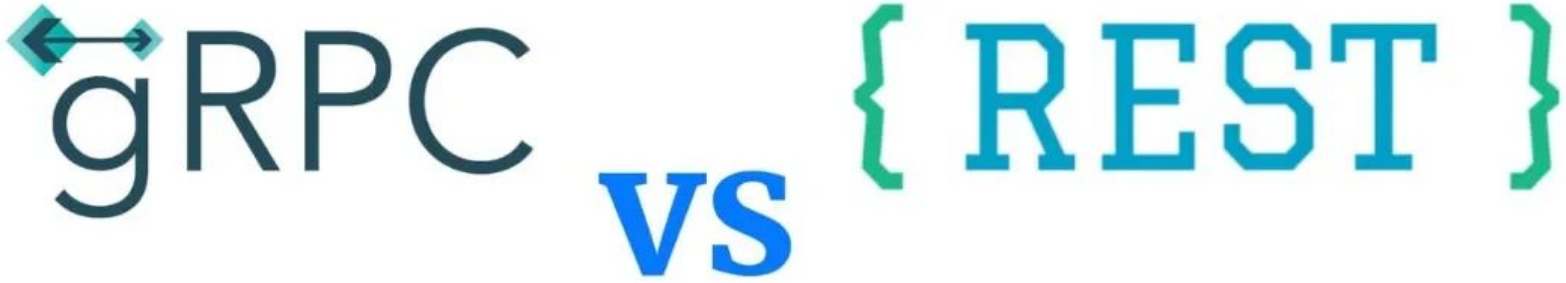
- High-performance Remote Procedure Call (RPC) framework
- Developed by Google, now open source
- Uses HTTP/2 for transport
- Protocol Buffers for serialization
- Built-in code generation for multiple languages
- Supports 4 types of service methods:
  - Unary (request/response)
  - Server streaming
  - Client streaming
  - Bi-directional streaming

```
syntax = "proto3";

package user;

// Service definition
service UserService {
    // Get user by ID
    rpc GetUser (GetUserRequest) returns (User) {}

    // Create a new user
    rpc CreateUser (CreateUserRequest) returns (User) {}
}

// Request message for getting a user
message GetUserRequest {
    int32 id = 1;
}

// Request message for creating a user
message CreateUserRequest {
    string name = 1;
    string email = 2;
}

// User message containing user details
message User {
    int32 id = 1;
    string name = 2;
    string email = 3;
    string created_at = 4;
}
```


gRPC
REST

gRPC vs REST Feature Comparison

Features	gRPC	REST
Messaging Format	gRPC leverages the Protocol Buffer (Protobuf) to transfer messages.	REST API is slower in comparison as it uses the JSON/XML format for message transfer.
HTTP Protocol	gRPC fosters bidirectional communication, transporting data using the HTTP/2 protocol.	REST utilizes the HTTP/1.1 protocol, with methods like GET, POST, PUT, and DELETE.
Code Generation	gRPC makes use of the inbuild protoc compiler to generate code.	With REST APIs, one must use third-party tools like Swagger Codegen and Open API for code generation.
Speed	gRPC is 7 times faster in receiving and almost 10 times faster in sending data.	Due to its use of HTTP 1.1 protocol, REST is slower in receiving and sending data.
Implementation	gRPC API takes about 45 minutes to implement.	REST API is lightning-fast and takes only 10 minutes to implement.
Code Writing	gRPC uses less code.	REST API's code is much more verbose and complicated to use.
Debugging	As gRPC uses the binary format, it's harder to learn the data passed over the wire using protocol buffers.	As JSON objects use the plaintext format, they are easy for developers to work with.

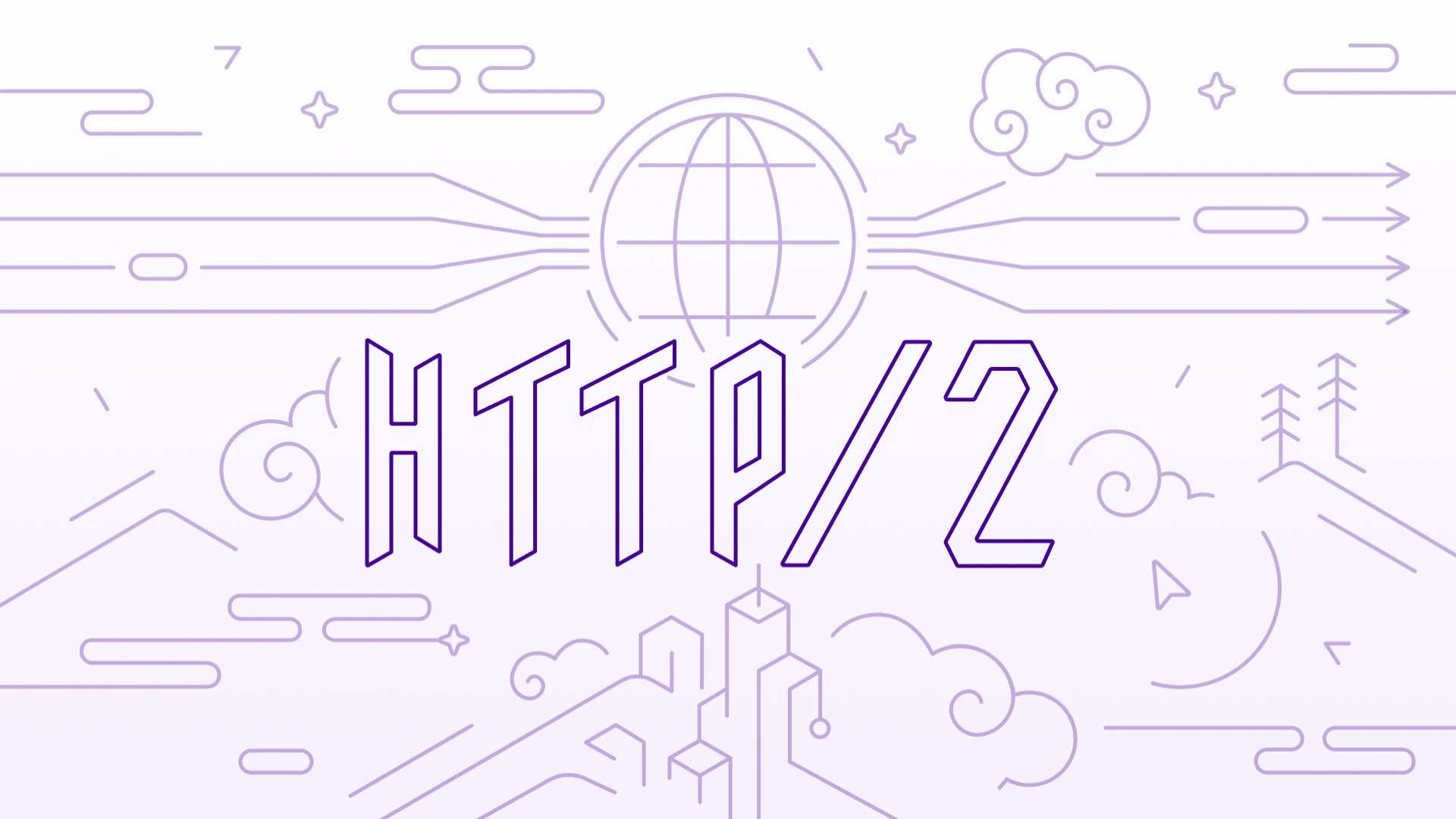

gRPC
vs
{ REST }

gRPC vs REST: Performance Benchmark

Metrics	gRPC	REST
Requests per second	48.00	4.00
Request latency (ms)	6.15	8.00
CPU ms/s	832.00	404.00
CPU/request (ms)	17.33	101.00

Key Findings:

- **Throughput:** gRPC handles 48 requests/sec, which is >10X higher than REST (4 req/sec)
- **CPU Efficiency:** gRPC uses significantly less CPU time per request (17.33ms) compared to REST (101ms)
- **Latency:** Both show similar request latency with gRPC at 6.15ms and REST at 8ms


HTTP/2

Key Components of HTTP/2 Architecture

Connection Establishment (TCP + TLS)

- HTTP/2 starts with a standard TCP three-way handshake
- Followed by TLS 1.3 handshake with ALPN (Application-Layer Protocol Negotiation)
- ALPN allows the client to indicate HTTP/2 support ("h2") during the TLS handshake
- This combined handshake reduces connection setup time compared to HTTP/1.1

HTTP/2 Connection Initialization

- After secure connection establishment, both parties exchange SETTINGS frames
- SETTINGS frames contain protocol-specific parameters (window size, max frame size, etc.)
- Each party acknowledges the other's settings with SETTINGS ACK
- This creates a single, long-lived connection for all subsequent requests

Multiplexing and Streams

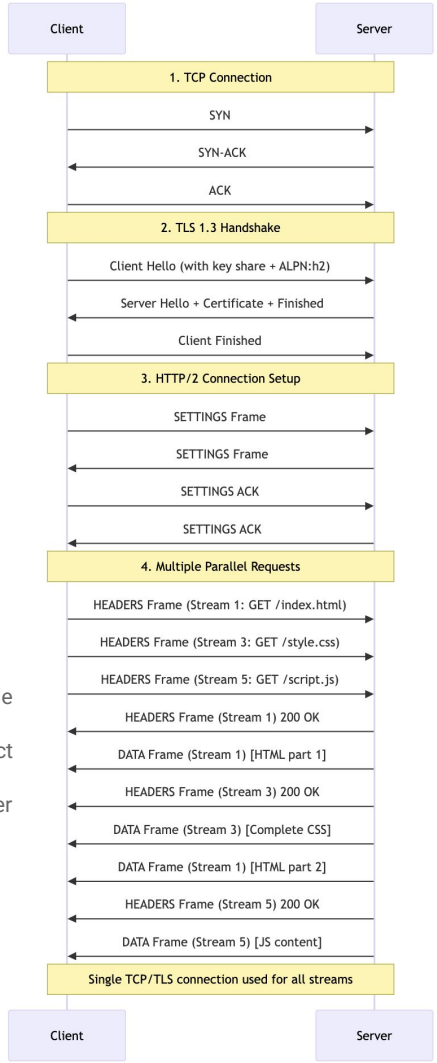
- Multiple requests can be sent concurrently over the same TCP connection
- Each stream has a unique ID (odd numbers for client-initiated streams)
- Streams are independent and can be processed in any order
- Eliminates the "head-of-line blocking" problem from HTTP/1.1

Binary Framing Layer

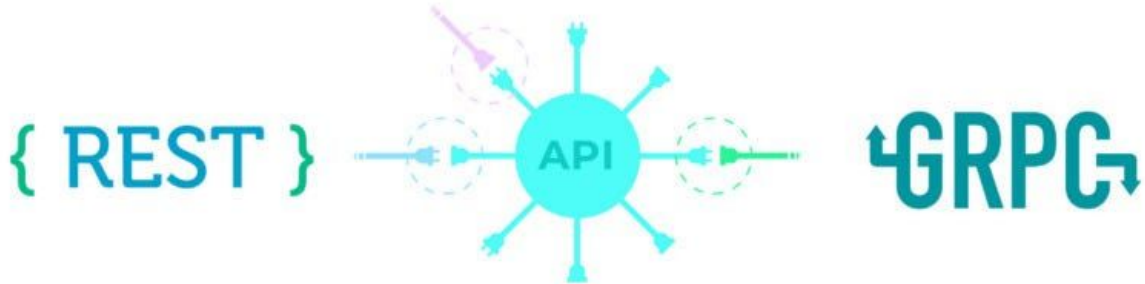
- All communication is divided into frames
- Common frame types:
  - HEADERS: Contains HTTP header information
  - DATA: Contains the actual payload
  - SETTINGS: Protocol configuration
  - RST_STREAM: Terminates a stream
  - PUSH_PROMISE: Server push notification
  - PRIORITY: Stream prioritization
  - PING: Connection keep-alive
  - GOAWAY: Connection termination

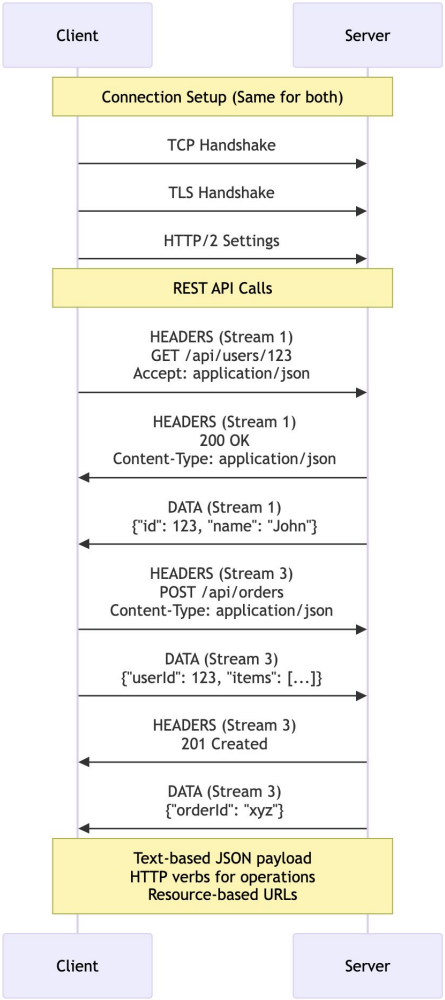
Key Benefits:

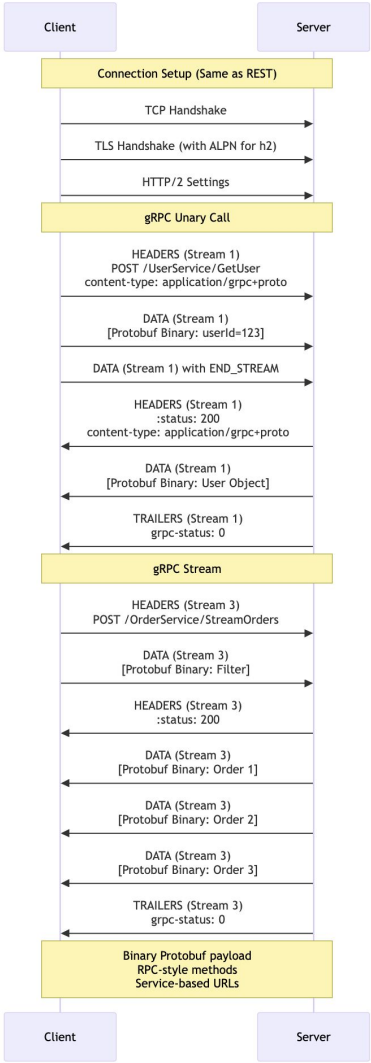
- Multiplexing: Multiple requests/responses in parallel over a single connection
- Binary Protocol: More efficient parsing, fewer errors, and compact representation
- Stream Prioritization: Resources can be delivered in optimal order
- Header Compression: Reduces overhead using HPACK compression
- Server Push: Server can proactively send resources to client cache



Understanding REST and gRPC Communication over HTTP/2




Client
Server
Connection Setup (Same for both)
TCP Handshake
TLS Handshake
HTTP/2 Settings
REST API Calls
HEADERS (Stream 1)
GET /api/users/123
Accept: application/json
HEADERS (Stream 1)
200 OK
Content-Type: application/json
DATA (Stream 1)
{"id": 123, "name": "John"}
HEADERS (Stream 3)
POST /api/orders
Content-Type: application/json
DATA (Stream 3)
{"userId": 123, "items": [...]}
HEADERS (Stream 3)
201 Created
DATA (Stream 3)
{"orderId": "xyz"}
Text-based JSON payload
HTTP verbs for operations
Resource-based URLs
Client
Server


Client
Server
Connection Setup (Same as REST)
TCP Handshake
TLS Handshake (with ALPN for h2)
HTTP/2 Settings
gRPC Unary Call
HEADERS (Stream 1)
POST /UserService/GetUser
content-type: application/grpc+proto
DATA (Stream 1)
[Protobuf Binary: userId=123]
DATA (Stream 1) with END_STREAM
HEADERS (Stream 1)
:status: 200
content-type: application/grpc+proto
DATA (Stream 1)
[Protobuf Binary: User Object]
TRAILERS (Stream 1)
grpc-status: 0
gRPC Stream
HEADERS (Stream 3)
POST /OrderService/StreamOrders
DATA (Stream 3)
[Protobuf Binary: Filter]
HEADERS (Stream 3)
:status: 200
DATA (Stream 3)
[Protobuf Binary: Order 1]
DATA (Stream 3)
[Protobuf Binary: Order 2]
DATA (Stream 3)
[Protobuf Binary: Order 3]
TRAILERS (Stream 3)
grpc-status: 0
Binary Protobuf payload
RPC-style methods
Service-based URLs
Client
Server

Next Session: Deep Dive into Implementation and HTTP/3

Next

- **Hands-on Implementation**
  - Live coding of REST and gRPC services
  - Real-world application examples
  - Best practices for implementation
  - Common pitfalls and how to avoid them
- **Performance in Action**
  - Live benchmarking demonstrations
  - Real-time performance monitoring
  - Load testing scenarios
  - Performance optimization techniques
- **HTTP/3 (QUIC) Deep Dive**
  - Understanding QUIC protocol
  - Key improvements over HTTP/2:
    - Built-in encryption
    - Reduced latency (0-RTT handshakes)
    - Enhanced multiplexing
    - Better mobile network performance
  - Migration considerations
  - Current adoption status


Q&A