Service-Oriented Communication Technologies: A Comparative Study

TELECOM SudParis — CSC8603 IT Architecture Course

Authors: Salim LAKHAL, Nossa IYAMU

Date: February 2026

---

1. Introduction

Modern software systems rely on Service-Oriented Architecture (SOA) where applications are split into independently deployable services that communicate over a network. This report compares four major API communication paradigms — SOAP/WSDL, REST, GraphQL, and gRPC — analysing their architecture, strengths, weaknesses, and appropriate use cases.

Each technology was designed to solve different problems. SOAP provides strict contracts and enterprise-grade security. REST offers simplicity by building directly on HTTP. GraphQL lets clients request exactly the data they need. gRPC prioritises performance through binary encoding and HTTP/2. Understanding when each approach is the right choice is the central question of this study.

We implemented working code examples for each technology as part of this project, using the same technology stacks as the TELECOM SudParis course labs (Python for SOAP/REST/gRPC, Node.js + Express for GraphQL).

---

2. SOAP / WSDL

Architecture

SOAP (Simple Object Access Protocol) is an XML-based messaging framework standardised by W3C. Every message is wrapped in an **Envelope** with an optional Header, a mandatory Body, and a Fault element for errors.

WSDL (Web Services Description Language) is a machine-readable XML contract describing the service. It separates *what* (abstract interface via `<portType>`) from *how* (protocol binding) and *where* (endpoint address), allowing multiple transport bindings from a single definition.

Key Characteristics

- **Contract-first**: the WSDL file defines every operation, message type, and data schema before any code is written. Code generators (e.g. `wsimport`, `zeep`) produce client stubs automatically.
- **WS-* Extensions**: WS-Security provides message-level encryption and digital signatures. WS-ReliableMessaging guarantees delivery. WS-AtomicTransaction supports distributed transactions across services.
- **Transport independence**: although HTTP is most common, SOAP can operate over JMS, SMTP, or TCP.

Strengths and Weaknesses

Strengths	Weaknesses
Formal contract guarantees interoperability	Verbose XML increases payload size
Enterprise security (WS-Security)	Complex setup and steep learning curve
ACID transaction support	Poor browser/mobile support
Language-neutral code generation	Slower parsing than binary formats

Best Suited For

Banking, healthcare, government systems, and enterprise integrations where strict contracts, compliance, and reliable messaging are mandatory.

---

3. REST

Architecture

REST (Representational State Transfer), defined by Roy Fielding in 2000, is an architectural style built on HTTP. Everything is a **resource** identified by a URL, manipulated through standard HTTP verbs: GET (read), POST (create), PUT (replace), PATCH (update), DELETE (remove).

Key constraints: **statelessness** (each request is self-contained), **uniform interface** (standard verbs and URLs), **client-server separation**, and **cacheability**.

Key Characteristics

- **JSON payloads**: lightweight and human-readable, supported natively by browsers and all programming languages.
- **HTTP status codes** convey semantics: 200 (OK), 201 (Created), 400 (Bad Request), 404 (Not Found), 422 (Unprocessable Entity).
- **OpenAPI/Swagger** provides optional machine-readable documentation, though it is not required by the architecture.
- **Stateless**: no server-side sessions, which simplifies horizontal scaling.

Strengths and Weaknesses

Strengths	Weaknesses
Simple, widely understood	Over-fetching or under-fetching data
Native HTTP caching	No built-in schema validation
Excellent tooling and ecosystem	Multiple round trips for related resources
Works everywhere (browsers, mobile, IoT)	No standard for real-time updates

Best Suited For

Public APIs, web and mobile backends, CRUD applications, and microservices where simplicity and broad compatibility are priorities.

---

4. GraphQL

Architecture

GraphQL is a query language for APIs developed by Facebook (2012, open-sourced 2015). It exposes a **single endpoint** where clients send queries describing exactly which fields they need. The server responds with precisely that data — nothing more, nothing less.

A **typed schema** defines all available types, queries, and mutations. The schema acts as both documentation and runtime validation.

Key Characteristics

- **Client-driven queries**: the client decides what data to fetch, solving REST’s over-fetching and under-fetching problems.

- **Single endpoint**: all operations go through one URL (typically `/graphql`), unlike REST which uses many URLs.
- **Introspection**: clients can query the schema itself to discover available types and operations at runtime.
- **Mutations** handle write operations; **subscriptions** (via WebSocket) handle real-time updates.

Strengths and Weaknesses

Strengths	Weaknesses
No over/under-fetching	Complex server-side implementation
Strong typing with schema validation	N+1 query problem without DataLoader
Single endpoint simplifies routing	HTTP caching is harder (single URL)
Self-documenting via introspection	Query complexity attacks possible

Best Suited For

Mobile applications with limited bandwidth, dashboards aggregating data from multiple sources, and APIs serving diverse frontend clients with different data needs.

---

5. gRPC

Architecture

gRPC is a high-performance RPC framework built by Google (2015). It uses **Protocol Buffers (Protobuf)** for service definition and message serialisation, with **HTTP/2** as the transport layer. Services are defined in `.proto` files, and code generators produce typed client/server stubs for many languages.

Key Characteristics

- **Binary encoding**: Protobuf messages are 5-10x smaller than equivalent JSON, and faster to serialise/deserialise.
- **HTTP/2 multiplexing**: multiple RPCs share a single TCP connection with header compression.
- **Four communication patterns**: unary (request-response), server streaming, client streaming, and bidirectional streaming.

- **Deadlines and cancellation**: built-in timeout propagation across service chains.
- **Reflection**: servers can expose their service definitions at runtime for tools like `grpcurl`.

Strengths and Weaknesses

Strengths	Weaknesses
Highest performance (binary + HTTP/2)	Not browser-friendly (needs grpc-web proxy)
Streaming support (server, client, bidi)	Binary payloads are not human-readable
Strong typing via Protobuf schema	More complex setup than REST
Code generation in 10+ languages	Limited tooling compared to REST ecosystem

Best Suited For

Internal microservice communication, real-time streaming (IoT, chat), latency-sensitive systems, and polyglot architectures where type-safe contracts across languages are important.

---

6. Comparative Analysis

Summary Table

Dimension	SOAP	REST	GraphQL	gRPC
Protocol	SOAP over HTTP/SMTP/JMS	HTTP	HTTP	HTTP/2
Data Format	XML	JSON (typically)	JSON	Protobuf (binary)
Contract	WSDL (mandatory)	OpenAPI (optional)	Schema (mandatory)	.proto (mandatory)
Typing	XSD (strong)	None built-in	Strong (schema)	Strong (Protobuf)
Caching	Not natively	HTTP caching (GET)	Difficult (single URL)	Not natively
Streaming	No	No	Subscriptions	Full (4

Dimension	SOAP	REST	GraphQL	gRPC
		(polling/SSE)	(WebSocket)	patterns)
Browser Support	Limited	Excellent	Excellent	Requires proxy
Security	WS-Security (message-level)	TLS + OAuth2	TLS + auth middleware	TLS + interceptors
Payload Size	Large (XML verbosity)	Medium (JSON)	Medium (JSON)	Small (binary)
Learning Curve	Steep	Low	Medium	Medium
Best For	Enterprise/ compliance	Public/web APIs	Flexible frontends	Internal services

Key Observations

Performance: gRPC consistently outperforms the others in throughput and latency due to binary encoding and HTTP/2 multiplexing. In our code examples, the Protobuf-serialised message was roughly 7x smaller than its JSON equivalent for the same data.

Developer Experience: REST has the lowest barrier to entry — any HTTP client works. GraphQL requires learning a query language but rewards frontend teams with flexible data fetching. SOAP has the steepest learning curve due to XML namespaces and WS-* specifications.

Contract Strength: SOAP, GraphQL, and gRPC all enforce typed contracts, which catches errors at compile time rather than runtime. REST's lack of a mandatory contract makes it flexible but less safe for large-scale systems.

When to Mix Technologies: In practice, most non-trivial systems use more than one paradigm. A common pattern is REST for external APIs, gRPC for internal service-to-service calls, and GraphQL as a gateway aggregating multiple backends. This is exactly the approach we took in Project 2 (Insurance Claim System).

---

7. Conclusion

Through this comparative study, we found that no single technology is universally superior. Each paradigm reflects a different set of priorities:

- **SOAP** prioritises reliability and formal contracts at the cost of complexity and verbosity.
- **REST** prioritises simplicity and universality at the cost of data efficiency.
- **GraphQL** prioritises client flexibility at the cost of server complexity.
- **gRPC** prioritises raw performance at the cost of ecosystem maturity and browser support.

The most practical takeaway is that the choice depends on the specific context: who is the consumer (browser, mobile app, internal service)? What are the performance requirements? Is a strict contract needed? Is real-time streaming required?

In our Project 2 implementation, we used all four technologies in a single insurance claim processing system, which gave us hands-on experience with the trade-offs described in this report. REST was the natural choice for most services due to its simplicity. SOAP was appropriate for identity verification where a formal WSDL contract adds trust. gRPC made sense for fraud detection where low latency matters. GraphQL was well-suited for document review and claim tracking where clients need flexible queries.

This project helped us understand that choosing an API technology is not a theoretical exercise — it is an engineering decision that depends on concrete requirements, team expertise, and the broader system architecture.

Appendix: Code Examples Overview

As part of this project, we implemented working examples for each technology. All examples follow a self-contained pattern: running a single script starts a local server, runs demo operations, and shuts down cleanly.

SOAP Example (Python + zeep)

Our SOAP example implements a calculator service. The server exposes a WSDL contract at `/calculator?wsdl`, and the client uses the `zeep` library to automatically generate Python proxy objects from the WSDL. We demonstrate four operations: addition, division, fault handling (divide by zero triggers a SOAP

Fault), and raw envelope inspection. The key learning: zeep handles all XML serialisation automatically — the developer works with Python objects, not XML strings.

REST Example (Python + Flask)

The REST calculator uses Flask with JSON payloads. We expose endpoints following REST conventions: `POST /calculate` for operations, `GET /calculate/history` for retrieving past results, and `GET /health` for a health check. HTTP status codes carry meaning: 200 for success, 400 for malformed requests, 422 for semantic errors like division by zero. The key learning: REST's simplicity means any HTTP client (curl, browser, Postman) works immediately without special libraries.

GraphQL Example (Node.js + Express + express-graphql)

Following the TELECOM SudParis course lab, we used Node.js with `express-graphql`. The schema defines a `Query` type (calculate, history) and a `Mutation` type (clearHistory). The client demonstrates field selection — requesting only the fields it needs — and error handling, where GraphQL returns HTTP 200 with an `errors[]` array rather than HTTP 4xx codes. We also demonstrate introspection: querying the schema itself with `{ __schema { types { name } } }`. The key learning: GraphQL gives the client full control over the response shape.

gRPC Example (Python + grpcio)

Our gRPC example defines a calculator service in a `.proto` file and uses `grpc_tools.protoc` to generate Python stubs. We demonstrate all three patterns: unary RPC (single request/response), server streaming (server sends multiple responses), and bidirectional streaming (both sides stream simultaneously). We also compare binary size — the Protobuf-encoded message was roughly 7x smaller than its JSON equivalent. The key learning: gRPC's code generation and binary format provide both type safety and performance, but at the cost of needing compiled stubs.

References

1. Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.
2. W3C. (2007). *SOAP Version 1.2*. https://www.w3.org/TR/soap12/

3. W3C. (2001). *Web Services Description Language (WSDL) 1.1*. https://www.w3.org/TR/wsdl
4. GraphQL Foundation. (2021). *GraphQL Specification*. https://spec.graphql.org/
5. Google. (2015). *gRPC Documentation*. https://grpc.io/docs/
6. Swagger/OpenAPI Initiative. *OpenAPI Specification 3.1*. https://spec.openapis.org/oas/v3.1.0