Modern API Protocols - GraphQL & gRPC

Web Services and Middleware Technologies

WSMT Course

The Dominance of REST

- For the last decade, **REST** (Representational State Transfer) has been the undisputed king of web APIs
- It is simple, stateless, cacheable, and uses ubiquitous HTTP methods (`GET` , `POST` , `PUT` , `DELETE`)
- It relies heavily on **JSON (JavaScript Object Notation)** as a human-readable interchange format
- If REST is so good, why do companies like Meta (Facebook), Netflix, and Google build alternatives?

The Shortcomings of REST

- REST is heavily tied to the concept of "**Resources**" (`/users` , `/posts` , `/comments`)
- What happens when a frontend mobile app needs a user's profile, their last 3 posts, and the top comment on each post?
- In REST, you either:
  1. Make a dozen individual HTTP requests. (Too slow on mobile)
  2. Create a custom `GET /mobile-dashboard` endpoint. (Breaks REST philosophy)

Limitation 1: Over-fetching

- **Over-fetching:** The server returns significantly *more* data than the client actually needs
- **Scenario:** A mobile app wants to show a list of usernames
- It calls `GET /users`
- The REST API returns a massive JSON array containing the user's ID, username, hashed password, email, shipping address, date of birth, and credit card token
- The mobile app ignores 95% of the payload

Limitation 2: Under-fetching

- **Under-fetching:** The server returns *less* data than the client needs to render a single view, forcing the client to make subsequent requests. (The N+1 Request Problem)
- **Scenario:** You want a list of Authors and their Books
- Client calls `GET /authors` (1 response)
- Client then loops and calls `GET /authors/1/books`, `GET /authors/2/books` ... `N` times
- 1 initial request + N subsequent requests = N+1

Enter GraphQL

What is GraphQL?

- **GraphQL** is a data query language for APIs, developed by Facebook in 2012 (open-sourced in 2015)
- It is a runtime for fulfilling those queries with your existing data
- **The Core Philosophy:** The *Client* dictates exactly what data it wants, and the server returns exactly that data. Nothing more, nothing less

The Single Endpoint

- Unlike REST (`/users` , `/orders` , `/products`), a GraphQL API typically only has **One Endpoint**:
  - `POST /graphql`
- You send all queries, mutations (writes), and subscriptions (websockets) to this single URL
- The HTTP request body contains the GraphQL Query string

The Power of the Query

Here is a REST request vs a GraphQL request:

REST Requests:

```
GET /users/4
GET /users/4/friends?limit=2
```

GraphQL Request Payload:

```
query {
  user(id: 4) {
    name
    email
    friends(first: 2) {
      name
    }
  }
}
```

The Precise Response

REST Response Payload:

```
// GET /users/4
{
  "id": 4,
  "name": "Jane Doe",
  "email": "jane@example.com",
  "password_hash": "...",
  "address": "...",
  "dob": "1990-01-01",
  "cc_token": "..."
}
// GET /users/4/friends?limit=2
[
  { "id": 7, "name": "Alice", "email": "..." },
  { "id": 9, "name": "Bob",   "email": "..." }
]
```

GraphQL Response Payload:

```
{
  "data": {
    "user": {
      "name": "Jane Doe",
      "email": "jane@example.com",
      "friends": [
        { "name": "Alice" },
        { "name": "Bob" }
      ]
    }
  }
}
```

The GraphQL Schema (SDL)

- How does the frontend know what it is allowed to ask for?
- Every GraphQL server relies on a **Schema**
- The Schema is a strongly-typed contract written in Schema Definition Language (SDL)
- It defines every Object type, field, and relationship in your API

Defining a Schema

```
type User {
  id: ID!
  name: String!
  email: String
  friends: [User]
}

type Query {
  user(id: ID!): User
  allUsers: [User]
}
```

- `String!` means the field is non-nullable (Conceptually similar to `NOT NULL` in SQL)
- `[User]` means an Array of Users

Resolvers: The Backend Engine

- GraphQL is just a specification. How does it fetch data from a Database?
- **Resolvers** are the backend functions that actually do the work
- You must write one Resolver function for every field in your schema (though most GraphQL libraries auto-generate trivial resolvers)

Example Resolver (Node.js / Apollo)

```
const resolvers = {
  Query: {
    // Fetch a user by ID from the database
    user: (parent, args, context) => {
      return db.query(`SELECT * FROM users WHERE id = ${args.id}`);
    }
  },
  User: {
    // How to resolve the 'friends' field for a specific User
    friends: (parentUser, args) => {
       return db.query(`SELECT * FROM users JOIN friendships ... WHERE user_id = ${parentUser.id}`);
    }
  }
};
```

Mutations: Modifying Data

- In REST, we use `POST`, `PUT`, `DELETE` to modify data
- In GraphQL, we use **Mutations** (all sent over `POST /graphql`)
- A Mutation defines an intent to change data, and similarly dictates what data should be returned after the change occurs

Mutation Example

Request:

```
mutation {
  createUser(name: "John Doe", email: "john@doe.com") {
    id
    name
  }
}
```

Response:

```
{
  "data": {
    "createUser": {
      "id": "109",
      "name": "John Doe"
    }
  }
}
```

GraphQL Tools: GraphiQL

- Because of the strongly typed Schema, GraphQL ships with incredible developer tools
- **GraphiQL / Apollo Studio:** An interactive in-browser IDE for exploring APIs
- It gives developers documentation, syntax highlighting, and auto-complete for writing queries directly in the browser

GraphQL Drawbacks 1: Performance (The N+1 Resolver)

- Wait, didn't we say GraphQL fixes N+1 HTTP requests? Yes
- But it can easily create N+1 **Database Queries**
- If we ask for 10 Authors, and their Books, the GraphQL engine might execute 1 SQL query for Authors, and then fire the Book resolver 10 separate times (11 SQL queries total)
- **Solution:** DataLoaders (Batching and Caching utility)

GraphQL Drawbacks 2: Caching

- **REST Caching:** Because REST uses unique URLs (`/users/5`), CDNs (Cloudflare/Akamai) and browsers can automatically cache the HTTP `GET` response perfectly
- **GraphQL Caching:** Because all traffic goes through `POST /graphql` (and POST is non-cacheable by default in HTTP), network-level caching is completely broken out of the box

GraphQL Summary

- **Use REST when:** You have a simple system, standard resources, and want to leverage HTTP caching
- **Use GraphQL when:** You have a massive graph of heavily relational data, a massive frontend team building rapidly changing UI screens (React/React Native), and you want to prevent over-fetching

Part 2: Backend-to-Backend Communication

The Problem with JSON in Microservices

- GraphQL is usually found at the "Edge" (Between the Mobile App and the Gateway)
- But what happens deep inside the datacenter, when Microservice A (Billing) needs to talk to Microservice B (Inventory)?

1. **Parsing Overhead:** JSON is just a string. To read `{"age": 30}`, the CPU must read every character, recognize the string, cast the string "30" into a 32-bit integer. This eats CPU cycles
2. **Size:** JSON transmits massive amounts of redundant text. Every request repeats `"user_id":`, `"user_id":`
3. **No Schema Enforcement:** REST/JSON has no strict contract. If Billing expects an Integer, but Inventory sends a String, Billing crashes at runtime

Enter gRPC

- **gRPC:** A high-performance, open-source universal Remote Procedure Call (RPC) framework developed by Google in 2015
- It runs on **HTTP/2** (not HTTP/1.1)
- It uses **Protocol Buffers (Protobufs)** as its Interface Definition Language (IDL) and its underlying message interchange format

What is a Remote Procedure Call (RPC)?

- We briefly touched on this in SOA (Service-Oriented Architecture)
- Instead of calling `POST /api/sales`, an RPC framework makes a network call look like a local function call in your code

```
// Under the hood, this makes a network request to the Inventory server!
boolean success = inventoryService.reserveItem(itemId, 5);
```

Protocol Buffers (Protobufs)

- Protobufs are Google's language-neutral, platform-neutral mechanism for serializing structured data
- It is a **Binary Format**. (Not human-readable text)
- Think of it like XML or JSON, but vastly smaller, faster, and simpler

The `.proto` File Contract

In gRPC, you start by defining your service and message structures in a `.proto` file.

```
syntax = "proto3";

// Define the RPC Service Operations
service PaymentService {
  rpc ProcessPayment (PaymentRequest) returns (PaymentResponse);
}

// Define the exact message structure
message PaymentRequest {
  string credit_card_number = 1;
  int32  amount_cents = 2;
  bool   is_recurring = 3;
}
```

The Code Generation Magic

- You write the `.proto` file once
- You use the `protoc` compiler right before you compile your code
- `protoc` automatically generates thousands of lines of client and server code in Java, Python, Go, C++, or Node.js

Generated Code Example (Java)

After running the compiler, the Java team implements the server logic:

```
// The base class was auto-generated by protoc!
public class PaymentServiceImpl extends PaymentServiceGrpc.PaymentServiceImplBase {

    @Override
    public void processPayment(PaymentRequest req, StreamObserver<PaymentResponse> responseObserver) {

        System.out.println("Processing: $" + (req.getAmountCents() / 100.0));

        PaymentResponse response = PaymentResponse.newBuilder()
            .setSuccess(true).build();

        // Send the response across the network
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
}
```

Why is gRPC so Fast? (Binary Encoding)

- Because both the Client and the Server possess the `.proto` schema, they know the exact structural shape of the data
- When sending `{ "amount_cents": 500 }`, JSON sends 21 bytes of text characters
- Protobufs only send the field tag `2` and the integer `500`. It might take literally 3 bytes of binary data
- When decoding, there is no string parsing. The CPU rips the bytes directly into memory structures

Why is gRPC so Fast? (HTTP/2)

- While REST usually relies on HTTP/1.1, gRPC mandates **HTTP/2**
- **Multiplexing:** HTTP/1.1 requires a new TCP connection (or serialized requests) for every HTTP call. HTTP/2 allows sending thousands of parallel requests down a single TCP connection concurrently
- **Header Compression:** HTTP/2 compresses massive headers (like JWT tokens)
- **Persistent Connections:** The gRPC client opens a TCP connection to the server and keeps it open for days, entirely eliminating the 3-Way TCP Handshake latency

SOTA gRPC: HTTP/3 & QUIC

- HTTP/2 still suffers from "TCP Head of Line Blocking" in poor networks
- **SOTA:** Modern gRPC implementations can run on **HTTP/3 (QUIC)**
- QUIC operates over UDP rather than TCP
- It provides built-in TLS 1.3 encryption and eliminates purely head-of-line blocking, making gRPC blazing fast even on unstable mobile connections

gRPC Communication Types

gRPC is not just limited to Request/Response! Due to HTTP/2, it natively supports 4 types of RPCs:

1. **Unary RPC:** Standard (1 Request -> 1 Response)
2. **Server Streaming:** Client sends 1 request, Server returns a stream of multiple responses (e.g., Live stock ticks)
3. **Client Streaming:** Client streams huge amounts of data, Server returns 1 response (e.g., Video upload)
4. **Bidirectional Streaming:** Both sides stream data continuously over a single long-lived connection (e.g., Multiplayer game networking)

Bidirectional Streaming `.proto`

Defining a chat stream is as simple as adding the `stream` keyword:

```
service ChatService {
  // Client streams messages up, Server streams messages down
  rpc ChatLive (stream ChatMessage) returns (stream ChatMessage);
}

message ChatMessage {
  string user = 1;
  string text = 2;
}
```

The Drawbacks of gRPC (Browser Limitations)

- If gRPC is so vastly superior, why don't mobile apps and React web apps use it instead of REST/GraphQL?
- **The Browser Trap:** Modern web browsers (Chrome, Safari) do not expose the low-level HTTP/2 framing controls required to act as a native gRPC client
- Web browsers speak HTTP/1.1 or highly restricted HTTP/2

The Workaround (gRPC-Web)

- To talk to gRPC from a web browser, you must use **gRPC-Web**
- It is a Javascript library that wraps the binary payload into an HTTP/1.1 compatible wrapper
- You must place an Envoy Proxy in front of your gRPC server to translate the gRPC-Web HTTP/1.1 requests into native HTTP/2 gRPC traffic

SOTA Solution: ConnectRPC

- gRPC-Web is painful to set up (requires Envoy)
- **SOTA:** The industry is moving to **ConnectRPC** (by Buf)
- ConnectRPC is a family of libraries that work with Protocol Buffers but can seamlessly speak gRPC, gRPC-Web, *and* standard HTTP POST over JSON, right from the browser **without** needing a translation proxy!

Another Drawback: Debugging Binary

- In REST, you can open Postman, type a URL, look at the JSON response, and understand the data
- If you intercept a gRPC packet on Wireshark, you just see a random scramble of unreadable 0s and 1s
- You cannot `curl` a gRPC endpoint directly without specialized tools (like `grpcurl`) and possessing the exact `.proto` file

Handling Errors in gRPC

- REST uses HTTP Status Codes (404 Not Found, 500 Server Error)
- gRPC has its own distinct set of Status Codes
- Examples: `OK (0)`, `INVALID_ARGUMENT (3)`, `NOT_FOUND (5)`, `UNAUTHENTICATED (16)`
- You throw these specific exceptions in your Java/Node code, and gRPC transmits them over the HTTP/2 trailer headers safely

Backward Compatibility

- Systems evolve over years. You add columns to databases. You add fields to API requests
- If you change the `.proto` file, do you break all old clients immediately?
- **Protobuf Rules for Compatibility:**
  1. *Never* change the integer tags of existing fields. (Tag `1` is forever!)
  2. *Never* delete a field. (Just mark it as `deprecated = true`)
  3. You may add *new* fields. Old clients will just ignore them safely

gRPC vs REST vs GraphQL

A quick architectural recap. Which should you choose?

- **REST:** The default standard. Public facing APIs. Integration with external 3rd party companies (Stripe, Twilio)
- **GraphQL:** Mobile Apps and Single Page Web Apps (React/Angular) where minimizing bandwidth and Over-fetching are critical
- **gRPC:** High-performance, low-latency communication deep within a private backend microservice cluster

gRPC Code Deep Dive (Node.js Client)

Let's look at how easy the client code is once the `.proto` is generated.

```
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

// Load the Proto schema dynamically
const packageDefinition = protoLoader.loadSync('payment.proto');
const paymentProto = grpc.loadPackageDefinition(packageDefinition).PaymentService;

// Connect to the server
const client = new paymentProto('localhost:50051', grpc.credentials.createInsecure());

// Execute the Remote Procedure Call
client.ProcessPayment({ credit_card_number: "123", amount_cents: 500 }, (err, res) => {
    if (res.success) console.log("Payment Confirmed!");
});
```

Understanding RPC "Stubs"

- In the previous code, what is `client`?
- It is an **RPC Stub**
- A stub is a local dummy object acting as a representative of the distant server
- It packages the arguments into the protobuf binary format (Marshaling), pushes them to the OS network socket, waits for the response, and un-packages it (Unmarshaling)

gRPC Deadlines (Timeouts)

- In distributed architectures, you cannot wait forever for a response
- If 5 microservices call each other in a chain, and the 5th one freezes, the entire chain freezes
- **gRPC Deadlines:** You can attach a strict timeframe to a request

```
stub.withDeadlineAfter(5, TimeUnit.SECONDS).processPayment(...)
```

If the server takes 5.1 seconds, gRPC aborts the call on the client *and actively cancels the operation on the server*.

Security in gRPC (Authentication)

- gRPC does not use cookies
- **Transport Level:** Production gRPC mandates TLS/SSL (HTTPS) credentials to encrypt the binary stream
- **Call Level:** You can attach Interceptors (Middleware) to inject OAuth2 or JWT tokens into the call metadata (which act exactly like HTTP Headers)

Interceptors (The gRPC Middleware)

- How do you add global Logging or global Metrics to every single gRPC call without rewriting the business logic?
- **Interceptors.** (Like Express.js middleware)
- You intercept the call, start a stopwatch, let the call proceed, stop the stopwatch, and write the latency to Prometheus
- You use Interceptors to centralize authentication token validation

A Warning: The Fallacies of Distributed Computing

- gRPC makes network calls look like local function calls. This is dangerous
- Developers forget that `inventory.reserve()` might take 0ms, or it might take 5000ms if the network cable is cut
- You must *still* engineer around network partitions. You must still implement Circuit Breakers (Resilience4j) around gRPC stubs

When NOT to use gRPC

- Do not use gRPC if your frontend consists primarily of web browsers accessing your service directly. (Due to gRPC-Web proxy complexity)
- Do not use gRPC if your consumers are random 3rd party developers. You cannot expect random internet developers to figure out how to compile `protoc` binaries just to call your API. Give them a simple JSON REST API

Schema Evolution Strategies

- What happens when two microservices fundamentally disagree on the protobuf version?
- Team A updates the `.proto`. Team B doesn't update their code
- As long as backwards compatibility rules were followed (never changing tags), the system works
- If a breaking change *must* occur, you must create a brand new service inside the proto file: `service PaymentServiceV2`

A Real World Architecture (The Hybrid Approach)

1. **Frontend (React):** Makes a **GraphQL** Query to the API Gateway (`POST /graphql`)
2. **API Gateway:** Resolves the GraphQL Query structure. The `friends` resolver determines it needs data from the User Microservice
3. **Gateway to Backend:** The Gateway establishes a persistent **gRPC** HTTP/2 connection to the User Service, passing the fast binary payload
4. **Backend to DB:** The User Service executes standard SQL to fetch data
5. **Response:** User Service returns gRPC binary to Gateway. Gateway serializes it back into JSON and returns it to the React App

GraphQL Federation (Supergraphs)

- If an enterprise has 50 REST microservices, how do they create One Single GraphQL Endpoint?
- They don’t write a massive monolithic Node.js GraphQL gateway
- **SOTA: Apollo Federation**
- Each microservice exposes its own mini-GraphQL schema (Subgraph)
- An **Apollo Router** automatically stitches the 50 mini-schemas together into one unified global Graph (Supergraph)

The Apollo Router (SOTA Gateway)

- Historically, the Supergraph was built using Node.js. It was too slow under massive traffic
- **Apollo Router** is a high-performance GraphQL federation gateway written in **Rust**
- It analyzes incoming global GraphQL queries, calculates the fastest parallel execution plan to fetch data from the microservice "Subgraphs", and executes them concurrently

GraphQL Subscriptions (WebSockets)

- We mentioned gRPC Streaming. Can GraphQL do streaming?
- Yes, through **Subscriptions**
- A client opens a WebSocket to the GraphQL server

```
subscription {
  newMessages {
    id
    text
  }
}
```

Every time a mutation adds a new message, the server pushes the specifically shaped data down the WebSocket.

Authentication in GraphQL

- Where do we authenticate?
- Do not put authentication logic inside your `user` resolver!
- Authentication happens at the **Context** level (before the query is parsed)
- The HTTP Server extracts the JWT token from the `Authorization` header, validates it, and places the `UserContext` object into the global state
- Resolvers simply check: `if (!context.user) throw AuthenticationError;`

GraphQL API Security Risks

- GraphQL gives massive power to the client. This is a security risk
- **The Malicious Query Trap:**

```
query {
  user {
    friends {
      user {
        friends {
          user { friends { name } }
        }
      }
    }
  }
}
```

An attacker just requested an infinitely deep nested join. Your database will crash executing billions of rows.

Mitigating GraphQL Attacks

- **1. Query Depth Limiting:** The server rejects any query nested deeper than e.g. 5 levels
- **2. Query Cost Analysis:** Every field has a "point" cost. Nested relationships cost more. If a query totals > 1000 points, reject it
- **3. Persisted Queries:** The server only accepts a pre-approved exact list of query hashes

Advanced gRPC: Reflection

- We said earlier you cannot use `curl` or Postman on gRPC without the `.proto` file
- Actually, you can enable **gRPC Server Reflection**
- The server exposes a special endpoint that describes its own protobuf schema dynamically
- Tools like `grpcurl` or Postman can hit the Reflection endpoint, download the schema securely, and automatically generate a UI allowing you to interact with the binary API

Tooling: Protocol Buffers vs FlatBuffers

- Are Protocol Buffers the absolute fastest? No
- Google also developed **FlatBuffers**
- FlatBuffers allows you to access serialized data straight in memory *without* parsing or unpacking it first. (Zero-copy deserialization)
- Used heavily in Video Games and mobile devices where memory allocation is critical

The Role of Service Meshes with gRPC

- In Lecture 9 we discussed Service Meshes (Istio, Envoy)
- Service Meshes are exceptionally powerful when paired with gRPC
- Why? Because gRPC load balancing over HTTP/2 is complex. A 1 TCP connection carries 1000 requests. Standard L4 Load Balancers (like AWS ELB) can’t balance them properly
- An Envoy Service Mesh operates at Layer 7 and understands HTTP/2 frames, load-balancing individual gRPC RPCs effortlessly across multiple pods

Migrating from REST to gRPC

- You have a massive existing Node+Express REST application. How do you migrate to gRPC safely?
- Use the **Strangler Fig Pattern** (from Lecture 9)
- Implement the gRPC server inside the exact same Node.js application, running on a different port (e.g., 50051)
- Slowly route internal cluster traffic from the Express HTTP/1.1 port over to the gRPC port. Monitor metrics

The Importance of API Contracts

- Whether using REST (OpenAPI/Swagger), GraphQL (Schema), or gRPC (Protobufs)…
- The fundamental shift in modern software engineering is **Schema-First Design**
- You do not write code first. You write the API Contract first
- You debate the contract with the Frontend and Backend teams. Once the contract is finalized, both teams go build their implementations in parallel

Summary of Modern Communication

- **REST/JSON:** Great for public APIs, highly cacheable, universally understood
- **GraphQL:** Eliminates over-fetching/under-fetching. Puts query control firmly in the hands of the Frontend developers. Excellent for massive relational UI dashboards
- **gRPC/Protobufs:** Lightning-fast, strict binary communication over HTTP/2. Perfect for deep internal datacenter microservice clusters minimizing latency

Q&A

Questions?

Next Steps

- **Lecture 13:** Event-Driven Architecture and Distributed Systems
- We will move completely away from synchronous Request/Response
- We will learn how systems scale to billions of messages using Apache Kafka and Event Sourcing patterns
- **Note:** Lecture 13 will be the **last lecture of this semester**, since June 1 is a public holiday (Children's Day)