Scalability & Real-Time Communication

This section deals with handling growth and enabling modern, interactive user experiences.

WebSockets for Real-Time Communication

Traditional HTTP is a request-response protocol.[28] The client asks for something, and the server responds. For real-time features like chat applications, live notifications, or collaborative editing, this is inefficient as the client would have to constantly poll the server for updates.

WebSockets solve this by establishing a **persistent, full-duplex** (two-way) communication channel between the client and the server over a single TCP connection.[29] Once the connection is established, both the client and the server can send data to each other at any time without needing a new request.[30] This is far more efficient and enables true real-time functionality.

Scalable Architecture Decisions

This refers to the core principles of building systems that can grow. As discussed in the infrastructure section, key decisions include:

- **Horizontal Scaling:** Adding more machines rather than making one machine more powerful (vertical scaling).
- **Stateless Services:** Ensuring your application servers don't store any client-specific data.[31] This makes it trivial to add or remove servers behind a load balancer.
- **Asynchronous Processing:** Using message queues (like RabbitMQ or AWS SQS) to decouple long-running tasks. For example, when a user uploads a video, the server can quickly respond "Upload received" and place a "process video" job in a queue. A separate fleet of worker services can then pick up these jobs and process them without blocking the main application servers.

---

5. Efficient Data Handling

This is about how your application and its clients exchange data, focusing on speed and efficiency.

GraphQL for Specific Data Fetching

In a traditional **REST API**, you often face the problems of **over-fetching** or **under-fetching**.[32]

- **Over-fetching:** An endpoint gives you more data than you need. For example, a `/user/123` endpoint returns the full user object with 20 fields when you only needed the `name`.

- **Under-fetching:** You have to make multiple API calls to get all the data you need. For example, fetching a blog post and then making separate calls for its comments and the author's details.

GraphQL is a query language for your API that solves this. The client specifies exactly what data it needs in a single request, and the server returns a JSON object with precisely that data, nothing more and nothing less. This is particularly powerful for mobile applications where bandwidth is a concern.

REST vs. gRPC

Feature	REST (Representational State Transfer)	gRPC (Google Remote Procedure Call)
Paradigm	Based on resources and standard HTTP verbs (GET, POST, PUT, DELETE).	Based on services and functions (Remote Procedure Calls). You call a function on a remote server as if it were local.
Protocol	Typically uses HTTP/1.1.	Built on **HTTP/2**, which allows for multiplexing (sending multiple requests over one connection), making it much faster.
Data Format	Uses human-readable text formats, primarily **JSON**.	Uses **Protocol Buffers (ProtoBuf)**, a binary format.
Best For	Public-facing APIs where broad compatibility and human readability are important.	High-performance internal communication between microservices where speed is critical.

JSON vs. ProtoBuf (Speed and Efficiency)

- **JSON (JavaScript Object Notation):**
    - **Format:** Text-based and human-readable.
    - **Schema:** Schema-less. It's flexible but requires both the sender and receiver to agree on the structure.

    - **Performance:** Slower and larger. Being text, it takes up more space and requires more CPU time to parse.
- **ProtoBuf (Protocol Buffers):**
    - **Format:** Binary. It is not human-readable.
    - **Schema:** Requires a predefined schema in a `.proto` file. This schema is used to generate code for serializing and deserializing data, ensuring type safety.
    - **Performance:** Significantly faster and smaller. The binary format is highly compact, and the serialization/deserialization process is extremely efficient. This is why gRPC uses it for high-performance communication.

https://github.com/codekarle/system-design/blob/master/system-design-prep-material/architecture-diagrams/Whatsapp%20System%20design.png

https://www.codekarle.com/system-design/Whatsapp-system-design.html