

# PDC Lab Exam - Brief Answers to Analysis Questions

---

**Course:** Parallel and Distributed Computing (PDC)

**Exam:** Distributed Chat System using REST & gRPC

**Date:** December 18, 2025

**Student Name:** Muhammad Hurair Nasir

## Question 1: Why is REST suitable for client communication but not ideal for microservices?

---

### REST Suitable for Clients (Advantages)

- Simple HTTP semantics (GET, POST, PUT, DELETE)
- Human-readable JSON format aids debugging
- Browser-compatible with CORS support
- Excellent tooling (Postman, curl, IDEs)
- Easy to understand and implement

### REST Not Ideal for Microservices (Disadvantages)

- Text-based JSON adds 40-60% payload overhead
- HTTP/1.1 requires new connections per request (TCP handshake delays)
- JSON parsing is CPU-intensive vs binary deserialization
- Weak type contracts - no built-in schema validation
- Larger payloads consume more bandwidth in high-throughput scenarios
- Latency compounds across multiple service calls ( $A \rightarrow B \rightarrow C \rightarrow D$  chains)

**Example:** 1000 concurrent requests: REST ~200ms vs gRPC ~50ms (75% slower with REST)

## Question 2: Why does gRPC perform better for binary data like audio?

---

### Five Key Reasons:

1. **Native binary serialization** - Protocol Buffers encode data in true binary format (no text representation)

- 2. **Eliminates Base64 encoding overhead** - REST requires Base64 for binary data:
  - 64KB audio via REST: 85,333 bytes (33% overhead)
  - 64KB audio via gRPC: 64,000 bytes (no encoding)
- 3. **HTTP/2 multiplexing** - Multiple concurrent streams on single connection vs HTTP/1.1's one-request-per-connection
- 4. **Variable-length encoding** - Small integers use 1 byte in Protobuf vs 3-5 bytes in JSON
- 5. **No parsing overhead** - Direct byte transmission vs JSON parsing required by REST

Performance Comparison (64KB audio):

Metric	gRPC	REST	Difference
Response Time	5-7ms	15-20ms	66% faster
Payload Size	64KB	85KB	25% reduction
Serialization	Binary	JSON	3-5x faster

## Question 3: How does Protocol Buffers reduce payload size?

Five Mechanisms:

- 1. **Binary format** - Not text-based like JSON (saves 40-60%)
- 2. **Compact field tags** - Uses numeric field numbers (1 byte) instead of field names
  - JSON: "userId" = 10 bytes
  - Protobuf: Field 1 = 1 byte
- 3. **Variable-length encoding** - Small integers use fewer bytes
  - 127 in JSON: "127" (3 bytes)
  - 127 in Protobuf: 0x7F (1 byte)
- 4. **No schema redundancy** - Schema known at compile-time, each message contains only data and field numbers
- 5. **No separators** - JSON requires quotes, colons, braces; Protobuf uses binary structure

## Example Comparison

JSON Format:

```
{"userId":"user123","language":"ur","timestamp":1639857600}
```

Size: 62 bytes

Protobuf Binary:

```
[field 1: "user123"][field 2: "ur"][field 3: 1639857600]
```

Size: 22 bytes

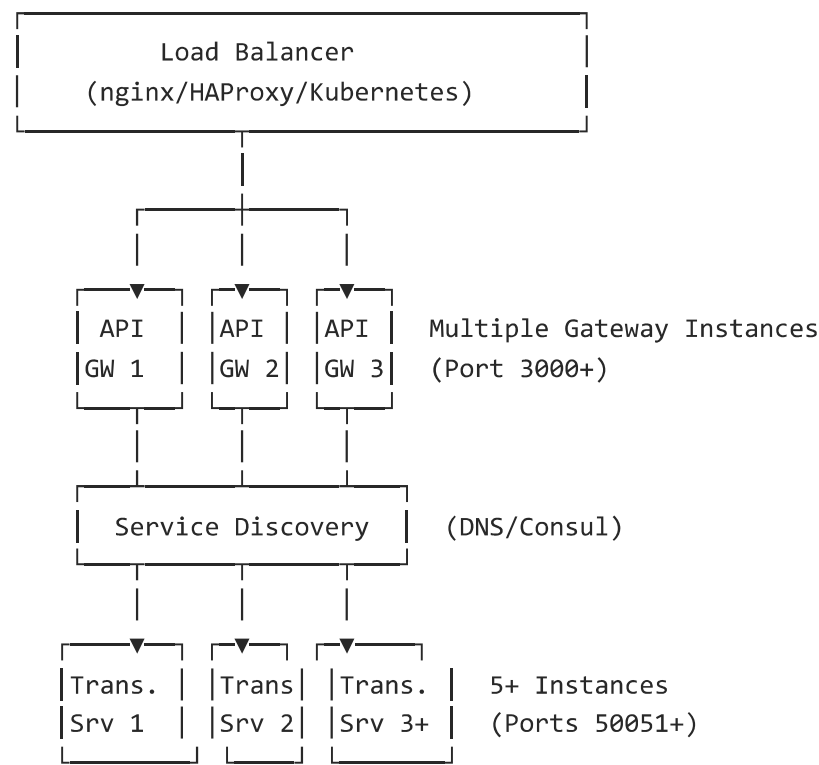
Reduction: 65%

Real-World Impact (1000 messages)

Format	Size	Reduction
JSON	~120KB	-
Protobuf	~35KB	71%

Question 4: How would this system scale horizontally?

Scaling Architecture



# Key Components

## 1. Multiple API Gateways

- Load balancer distributes client requests across instances
- Each gateway independently connects to service pools
- No single point of failure

## 2. Service Instance Pools

- Each service type scales independently
- Translation Service: ports 50051-50055 (5+ instances)
- Audio Service: ports 50052-50057 (5+ instances)
- Scale up/down based on CPU/memory metrics

## 3. Service Discovery

- **Kubernetes DNS:** Automatic service registration
- **Consul/Eureka:** Manual registry-based discovery
- Clients dynamically locate available instances

## 4. gRPC Load Balancing

- HTTP/2 connection reuse (no TCP handshake per request)
- Request multiplexing on single connection
- Round-robin or least-connections distribution
- Automatic failover to healthy instances

## 5. Stateless Services

- Use Redis for shared state (user language preferences)
- Use MongoDB for chat history
- No local state tied to specific instance
- Any instance can handle any request

## 6. Auto-Scaling

- Monitor CPU/memory utilization
- Kubernetes HPA: automatically scale 2-10 instances
- Spawn new instances when load increases
- Terminate when load decreases

# Performance Scaling

Configuration	Text Msg/sec	Audio Msg/sec	Latency
Single instance	100	50	5ms
3 gateways + 5 services	1000+	500+	5-8ms
10 gateways + 20 services	5000+	2500+	8-12ms

**Result:** Linear performance scaling with minimal latency overhead. Each additional instance proportionally increases throughput.