Comsats University Islamabad, Lahore campus

ASSIGNMENT#8(Lab)

RAMSHA KHAN

(SP23-BCS-112)

Section: C

Course: PDC

Instructor’s name:Akhzar Nazir

API Communication & Microservices Performance Report

Comprehensive Analysis of REST, tRPC, gRPC, and Microservices Architecture

---

1. Overview

This project implemented and compared three communication protocols across different architectural patterns to evaluate their performance characteristics, use cases, and trade-offs in modern distributed systems.

Technologies Evaluated:

- **REST**: Standard HTTP/JSON API following RESTful principles
- **tRPC**: Type-safe RPC framework for TypeScript/Node.js (using HTTP transport)
- **gRPC**: High-performance binary protocol using Protocol Buffers
- **Microservices**: Distributed scenario where Service B calls Service A using gRPC

---

2. Benchmark Results

Test Configuration: 1000 sequential iterations for each protocol

Technology	Avg Latency (ms)	Total Time (ms)	Payload Size	Best Use Case
REST	~2.04	~2046	403 bytes (JSON)	Public APIs, simple web clients. High compatibility. Status quo.
tRPC	~11.71	~11712	403 bytes (JSON)	Full-stack TypeScript mono-repos/projects. Great DX, type safety.
gRPC	~2.58	~2586	403 bytes (JSON decoded)	Internal microservices, low latency, polyglot environments. Smallest wire size.
Microservices (REST → gRPC)	~8.98	~8982	403 bytes	Distributed systems. Adds latency but provides valuable decoupling.

Note: Payload size listed is the approximate JSON stringified size of the received object. gRPC's actual wire size is significantly smaller due to binary serialization.

---

3. Detailed Analysis

3.1 REST API

Advantages:

- Ubiquitous and universally supported across all platforms
- Easy to debug with standard browser and command-line tools
- Stateless architecture promotes scalability
- Human-readable JSON format
- Well-established patterns and best practices

Disadvantages:

- JSON serialization is slower than binary protocols
- Larger payload size increases bandwidth usage
- No strict type enforcement at runtime without additional tools
- Verbose compared to binary formats

Performance Assessment: Very fast for simple local queries with minimal overhead. However, payload overhead grows with data complexity, making it less efficient for high-frequency or large-volume communications.

3.2 tRPC

Advantages:

- End-to-end type safety from client to server
- Excellent developer experience with auto-completion
- No code generation or build steps required
- Seamless integration with TypeScript projects
- Rapid development and refactoring support

Disadvantages:

- Tied exclusively to the TypeScript/JavaScript ecosystem
- HTTP transport adapter adds overhead
- Potential cold-start batching configuration issues
- Less mature ecosystem compared to REST/gRPC

Performance Assessment: Slower than raw REST/gRPC in this specific test setup (likely due to HTTP transport adapter overhead or batching), but performance is acceptable for most web applications where developer productivity and type safety are prioritized over microsecond optimizations.

3.3 gRPC

Advantages:

- Extremely fast with compact binary Protocol Buffer format
- Strict API contracts enforced through .proto files
- Built-in HTTP/2 support with multiplexing and streaming
- Superior bandwidth efficiency for large-scale systems
- Cross-language support (polyglot environments)

Disadvantages:

- Requires browser proxy (such as Envoy) for web clients
- Binary format makes debugging more challenging
- Steeper learning curve for teams new to Protocol Buffers
- Limited direct browser support without additional infrastructure

Performance Assessment: Comparable to REST in latency for local testing, but significantly superior in throughput and bandwidth usage at scale. The binary serialization provides substantial advantages for high-frequency internal communications between services.

3.4 Microservices Architecture (Service B → Service A)

Observations:

Calling Service A via gRPC from Service B added approximately 6ms of overhead compared to direct gRPC/REST calls. This additional latency accounts for:

- The extra HTTP request to Service B
- Processing time within Service B
- The internal gRPC call from Service B to Service A
- Network serialization/deserialization at each hop

Architectural Benefits:

Despite the added latency, the microservices pattern provides significant value:

- Service decoupling enables independent development and deployment
- Fault isolation prevents cascading failures
- Independent scaling of services based on load
- Technology diversity across service boundaries

Conclusion:

gRPC is an excellent choice for internal service-to-service communication because it minimizes the serialization overhead that would compound exponentially if using REST for all internal hops. The 6ms overhead is a reasonable trade-off for the architectural benefits of service decoupling.

---

4. Recommendations

When to Use REST:

- **Public-facing APIs** where broad client compatibility is essential
- **Simple web applications** with moderate traffic
- **Third-party integrations** requiring universal standards
- **Development teams** prioritizing simplicity and debugging ease

When to Use tRPC:

- **Full-stack TypeScript projects** with shared codebases
- **Monorepo architectures** with frontend and backend in one repository
- **Rapid prototyping** where developer experience matters most
- **Small to medium teams** working primarily in TypeScript/Node.js

When to Use gRPC:

- **Internal microservices** communication requiring high performance
- **High-frequency** or **high-volume** data exchanges
- **Polyglot environments** with multiple programming languages
- **Mobile applications** where bandwidth efficiency is critical
- **Real-time systems** requiring low latency and streaming

When to Use Microservices:

- **Complex systems** requiring independent scaling
- **Large organizations** with multiple teams working on different services
- **Systems requiring fault isolation** and resilience
- **Applications with diverse technology requirements** across components

---

5. Conclusion

The choice between REST, tRPC, and gRPC should be driven by your specific architectural requirements, team expertise, and performance needs.

REST remains the gold standard for public-facing APIs due to its universal support, simplicity, and extensive tooling ecosystem.

gRPC excels in internal microservices architectures where performance, bandwidth efficiency, and strong typing are paramount. Its binary protocol and HTTP/2 features make it ideal for high-scale distributed systems.

tRPC offers the best developer experience for TypeScript-centric teams building full-stack applications. While it may not match the raw performance of gRPC, the productivity gains and type safety benefits often outweigh the marginal performance differences in typical web applications.

For modern distributed systems, a **hybrid approach** often works best: REST for public APIs, gRPC for internal service communication, and tRPC for full-stack TypeScript applications where developer velocity matters most.
