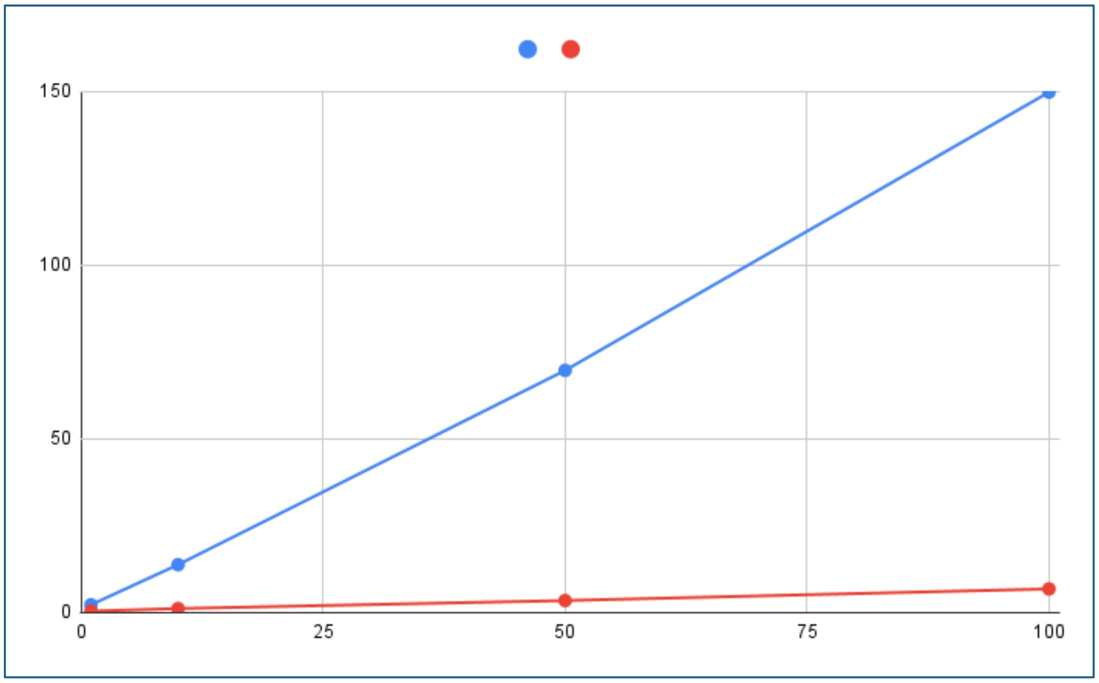
REST vs gRPC Performance Comparison



CMPT453

December 4, 2025

Abdul Rehman Ismaeel

Experiment Setup

All tests were run on a single machine using two separate processes: a load generator and a service host. The load generator executed Python clients that produced concurrent REST or gRPC requests for a fixed duration. The service host ran either a Flask-based REST server or an asynchronous gRPC server. Both exposed matching echo endpoints so that only the communication protocol differed. Each experiment varied one parameter at a time while keeping duration, machine, and server logic constant. The setup allowed consistent measurement of throughput, latency, and error behavior under changing loads.

Description of Experiments

The first experiment tested concurrency scaling by increasing the number of simultaneous clients (1, 10, 50, 100) while keeping payload size and duration constant. This showed how each protocol handles higher request loads.

The second experiment tested data-volume scaling by keeping concurrency fixed and increasing payload size (128 B to 128 KB). This measured how message size affects throughput and latency.

The third experiment tested node-failure behaviour by introducing a fixed 20% server-side failure rate and running at different concurrency levels (20, 50, 100). This revealed how each protocol degrades under partial failure. Together, these tests evaluate scalability, overhead, and resilience for both REST and gRPC.

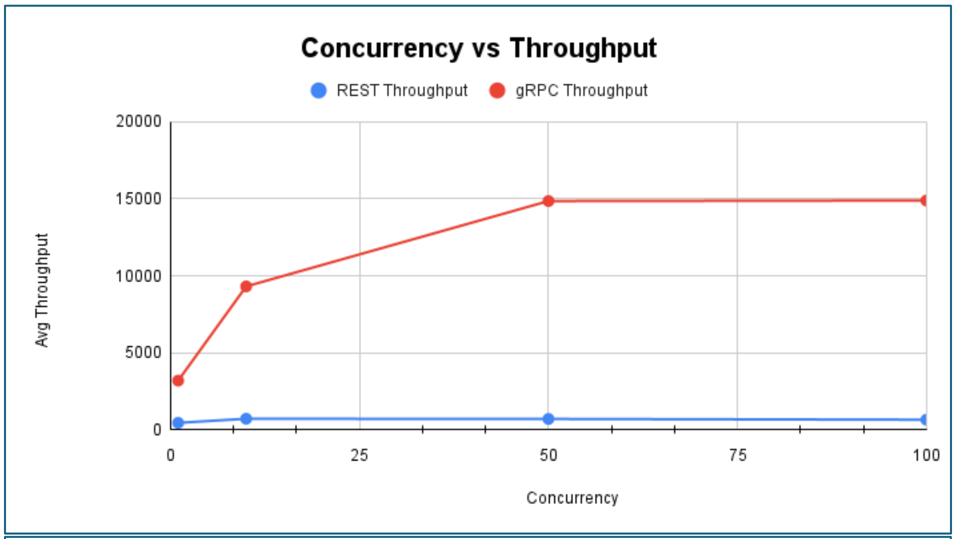
Key Results & Conclusions

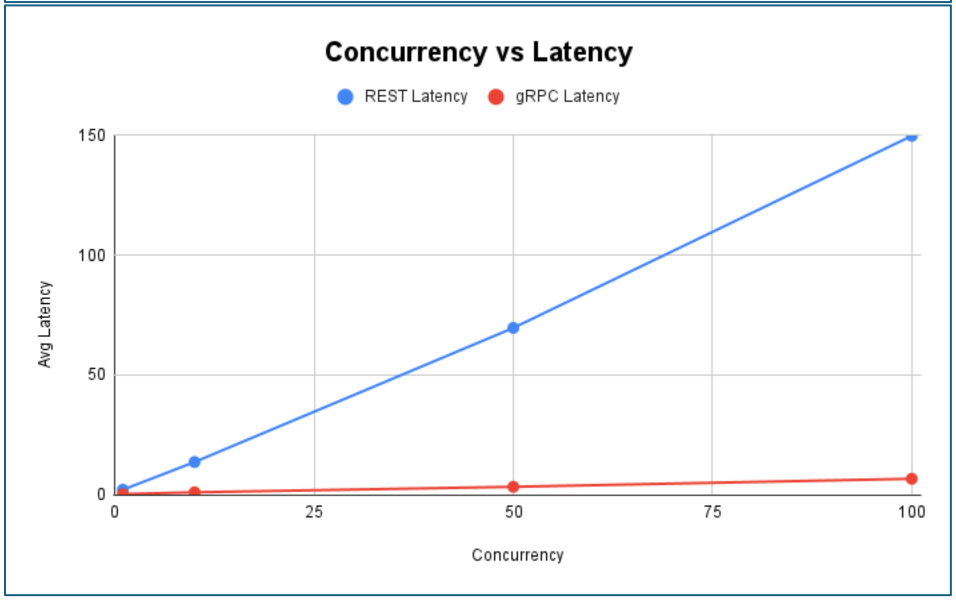
Experiment 1: This experiment measured how throughput and latency changed as concurrency increased from 1 to 100 clients.

Throughput Results
- REST stayed between 465–730 req/s, showing early saturation.
- gRPC increased from ~3,200 req/s at 1 client to ~14,800 req/s at 100 clients.
- gRPC consistently achieved 10–20× higher throughput.

Latency Results
- REST latency rose sharply: 2 ms to 150 ms as load increased.
- gRPC remained low, rising only from 0.3 ms to ~6 ms.

Conclusion: gRPC scales far better than REST under increasing concurrency.





Experiment 2: This experiment tested how REST and gRPC respond when the payload size increases from 128 bytes to 128 KB, while keeping concurrency fixed at 10 clients.

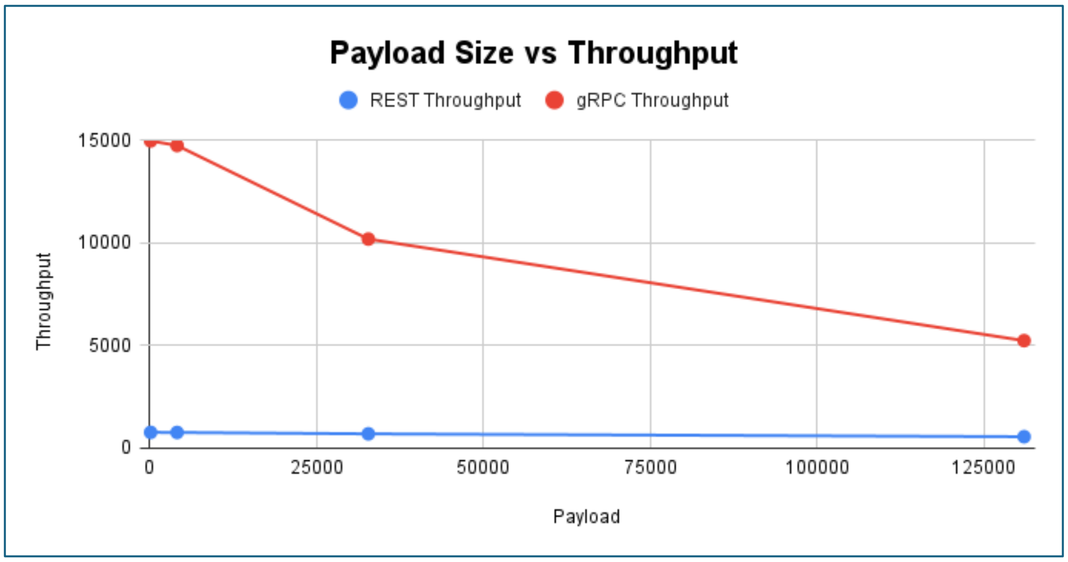
Throughput Results

- REST throughput decreased gradually from ~755 req/s (128 B) to ~542 req/s (128 KB).
- gRPC throughput decreased more sharply: ~15,000 req/s (128 B) → ~5,200 req/s (128 KB).
- Even at the largest payload, gRPC stayed ~10× faster than REST.

Latency Results

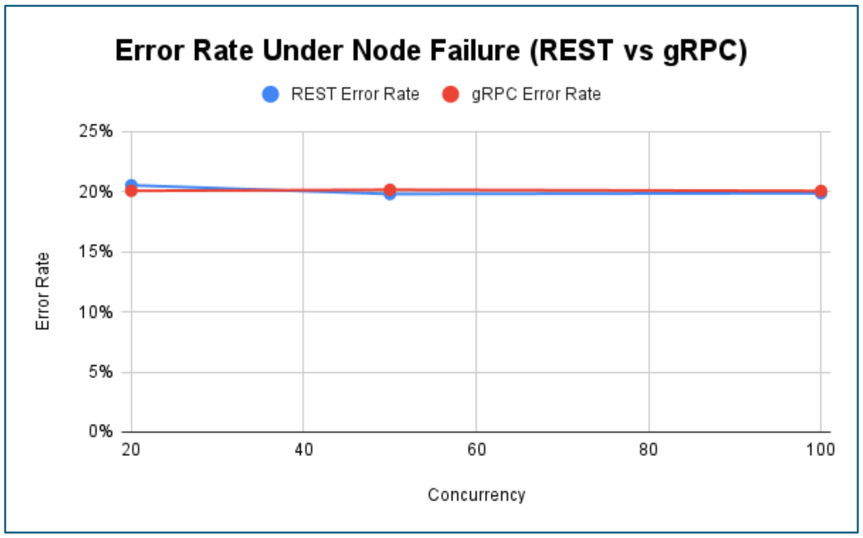
- REST latency grew from 13 ms to18 ms as payload increased.
- gRPC latency increased from 0.67 ms to~1.9 ms, remaining far lower than REST at all sizes.

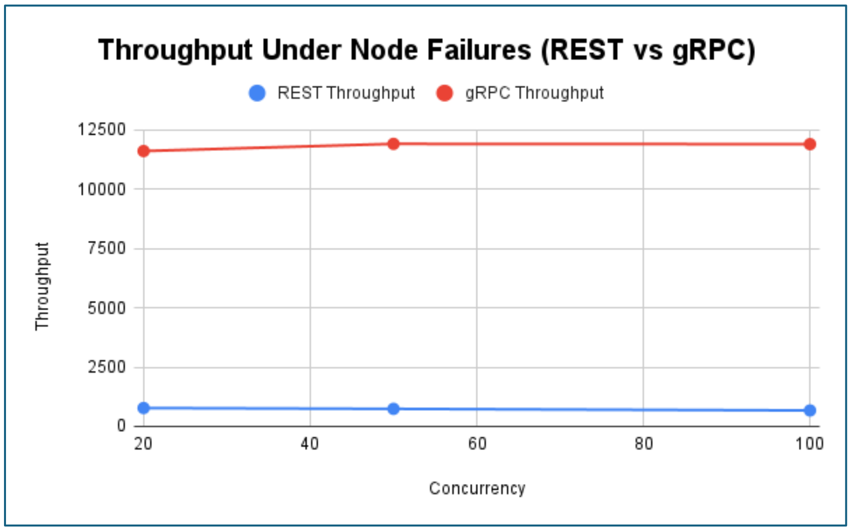
Conclusion: Both protocols slow down as message size grows, but gRPC maintains a significant advantage. REST suffers from JSON encoding/decoding overhead, while gRPC's binary protocol stays efficient even with large messages. Data volume affects both, but gRPC remains the better choice for high-throughput, large-payload communication.



Experiment 3: When a 20% failure probability was introduced, both protocols produced almost identical error rates, staying close to the injected failure level regardless of concurrency. The difference appeared in how much useful work each system could still deliver under failure. REST throughput dropped as load increased, falling from about 775 req/s at 20 concurrency to around 670 req/s at 100. gRPC, on the other hand, remained above 11k req/s in every test, showing only a small decline as concurrency rose. Latency followed the same pattern—REST slowed down significantly under failure, while gRPC kept response times in the single-digit millisecond range.

Conclusion: even when both systems experience the same failure rate, gRPC maintains far higher effective throughput and far lower latency. REST degrades quickly under partial node failure, while gRPC continues to operate near full performance, demonstrating better resilience under fault conditions.





Strengths and Weaknesses of REST and gRPC (Based on Experimental Results)

REST Strengths:

- Stable behaviour at small workloads: At low concurrency (1–10), REST latency remained low (2–13 ms) and throughput stayed consistent around ~700 req/s.
- Predictable error patterns under node failure: Even with forced failure rates, REST stayed near a stable ~20% error rate and behaved consistently across runs.

REST Weaknesses:

- Throughput collapses as concurrency increases. This indicates REST becomes slower and overloaded under parallel load.
- Sensitive to payload size: Throughput dropped from ~753 req/s (128 B) to ~542 req/s (128 KB).

gRPC Strengths:

- Massive throughput advantage.
- Minimal latency.
- Better scalability.

gRPC Weaknesses:

- Higher error counts during failure simulation. Although REST and gRPC *both* showed ~20% error rate under failure injection, gRPC produced a much larger number of raw failed requests simply because it sends more total requests per second.

Use REST when:

- The service will run at relatively low concurrency levels.
- Payload sizes are small and stable.
- Wide compatibility and ease of client testing are required.

Use gRPC when:

- High performance is important (your results show *10–20*× higher throughput).
- Services must handle large concurrency or heavy traffic.
- Payload sizes may grow.
- Node failures must be tolerated while keeping throughput high.