Distributed Systems - Programming Assignment 2

Darshan Vijayaraghavan *Umar Ahmed Thameem Ahmed*

Objective

Build a marketplace system where the client communicates with the server using REST, and the server communicates with the database using gRPC. Evaluate and compare their performance by measuring latency and throughput under different load scenarios.

Overview

We implemented the assignment using Python with FastAPI, gRPC, and MySQL. REST APIs are used for communication between the client and the server, while gRPC is used for communication between the server and the database to enable faster internal calls. The client interacts with the system through a CLI interface. Users must create an account and log in before performing any actions. MySQL is used as the database for storing and managing data.

Setup

For the evaluation setup, we wrote a Python script that simulates buyers and sellers with unique IDs. Each client logs in, performs a few selected operations for 1000 iterations while we record the latency and throughput. The same experiment is then run for 10 iterations to calculate the average latency and average throughput. We have taken 2 operations per each buyer and seller client and a comparison between these two in different scenarios.

For running concurrent users, the simulation setup creates threads for each client. We ran into problems when running 100 concurrent processes, and had to increase the database connection pool range in the database configuration to tackle this. We implemented a threading barrier to ensure all threads complete account creation and login, then wait until every thread has reached the barrier before starting operations simultaneously.

For GCP deployment, we created a VPC and a subnet where all the different components will be deployed. The database is deployed using MySQL on CloudSQL, and separate VMs run the frontend and the gRPC database for buyer and seller services. We set firewall rules to allow public access to the server components and the VMs communicate with the database over Google's private network. We have performed initial testing and have discussed it in the observation section.

Experiment 1 (1 seller + 1 buyer)

Iterations	get_seller_rating		search_items	
	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)
1	36.5	27.39	36.34	27.52
2	35.61	28.08	36.7	27.25
3	36.65	27.28	38.14	26.22
4	35.98	27.41	36.92	26.84
5	36.42	27.65	37.15	27.13
6	36.17	27.82	37.48	27.42
7	35.84	27.33	36.75	26.71
8	36.71	28.14	37.84	27.05
9	36.29	27.59	36.88	26.96
10	36.36	27.14	36.4	26.87
Average	**36.253**	**27.583**	**37.06**	**26.997**

Iterations	display_items_for_sale		register_item_for_sale	
	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)
1	61.41	16.28	65.32	15.31
2	67.05	14.91	70.45	14.19
3	68.81	14.53	69.59	14.37
4	64.12	15.02	67.81	14.58
5	65.48	15.67	68.94	14.72
6	66.73	14.88	69.12	14.91
7	63.95	15.31	66.73	14.05
8	67.26	14.76	70.08	14.63
9	68.14	15.49	68.55	14.88
10	64.59	15.55	67.94	14.59
Average	**65.754**	**15.24**	**68.453**	**14.623**

Experiment 2 (10 sellers + 10 buyers)

Iterations	get_seller_rating		search_items	
	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)
1	81.11	123.3	86.86	115.14
2	82.52	121.2	90.27	110.79
3	82.83	120.75	89.27	112.03
4	81.94	121.34	87.95	112.18
5	82.37	122.18	88.63	113.74
6	82.68	120.96	89.14	111.92
7	81.75	121.72	88.27	114.05
8	82.49	122.05	89.72	110.88
9	81.88	121.11	87.48	112.67
10	81.96	122.89	90.41	113.13
Average	**82.153**	**121.75**	**88.8**	**112.653**

Iterations	display_items_for_sale		register_item_for_sale	
	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)
1	227.79	43.9	195.77	51.08
2	277.54	36.03	194.88	51.31
3	312.6	31.99	199.19	50.2
4	268.74	36.72	196.24	50.72
5	273.92	37.48	197.53	51.14
6	276.38	38.15	195.91	50.39
7	271.55	35.94	198.02	51.02
8	274.11	37.83	194.76	50.84
9	269.83	36.61	196.88	50.63
10	273.97	38.42	196.95	51.3
Average	**272.643**	**37.307**	**196.613**	**50.863**

Experiment 3 (100 sellers + 100 buyers)

Iterations	get_seller_rating		search_items	
	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)
1	756.82	109.84	818.47	101.92
2	771.45	111.27	834.92	103.45
3	762.19	110.53	822.63	102.38
4	768.73	108.91	840.15	100.84
5	759.64	111.02	829.71	104.11
6	774.21	109.47	836.44	101.56
7	763.58	110.18	821.39	103.02
8	770.92	108.76	845.08	102.71
9	757.31	111.39	824.76	104.36
10	766.84	109.95	832.57	100.97
Average	**765.169**	**110.132**	**830.612**	**102.532**

Iterations	display_items_for_sale		register_item_for_sale	
	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)
1	2386.74	32.84	1846.92	46.72
2	2421.35	33.91	1756.34	47.15
3	2468.92	34.27	1898.41	45.98
4	2504.18	32.76	1815.49	46.84
5	2534.69	33.58	1903.49	47.33
6	2562.44	34.12	1789.72	45.76
7	2584.18	32.95	1863.18	46.59
8	2598.73	33.44	1828.67	47.02
9	2621.57	34.03	1874.55	46.41
10	2664.82	33.67	1830.75	45.89
Average	**2534.762**	**33.557**	**1840.752**	**46.569**

Performance Analysis

Scenario 1: Single Client Performance (1 Seller + 1 Buyer)

In the single-client scenario, the latency reflects the normal overhead of our layered system design. The get_seller_rating operation averages 36ms with throughput of 28 ops/sec, while search_items shows similar performance at 37ms latency and 27 ops/sec throughput. For seller operations, display_items_for_sale averages 66ms (15 ops/sec) and register_item_for_sale shows 68ms (15 ops/sec).

These results demonstrate the protocol overhead introduced by the REST/gRPC architecture. Each request must traverse multiple layers: HTTP request parsing at the REST server, JSON deserialization, gRPC call marshalling, Protocol Buffer encoding, network transmission to the database layer, database query execution, and the reverse path for responses. This multi-hop communication introduces significant latency compared to direct socket communication. Additionally, the stateless design requires session token validation on every request, adding authentication overhead. Write operations show slightly higher latency than read operations due to database transaction commits and bidirectional serialization costs.

Scenario 2: Moderate Concurrency (10 Sellers + 10 Buyers)

With 10 concurrent clients, throughput improves significantly compared to the single-client case. The get_seller_rating operation reaches about 121 ops/sec with an average latency of 82ms, which is roughly 4x higher throughput than the single-client scenario. Similarly, search_items achieves around 113 ops/sec with about 89ms latency. Seller operations also show clear improvement. display_items_for_sale reaches about 37 ops/sec with an average latency of 273ms, while register_item_for_sale achieves around 51 ops/sec with about 197ms latency.

This scenario represents the optimal operating point for our REST/gRPC architecture. The system effectively amortizes protocol overhead across concurrent requests through HTTP keep-alive connections, gRPC channel reuse, and database connection pooling. FastAPI's asynchronous request handling allows efficient multiplexing of concurrent operations, while the gRPC layer maintains persistent connections to database services, reducing connection establishment overhead. The stateless frontend design enables horizontal scaling without coordination overhead. However, latency increases by 2-3x compared to single-client due to request queuing and resource constraints at both the REST and gRPC layers.

Scenario 3: High Concurrency (100 Sellers + 100 Buyers)

At 100 concurrent clients, the system shows significant performance degradation. The get_seller_rating operation reaches 765ms latency (110 ops/sec), while search_items shows 830ms (103 ops/sec). Seller operations experience severe degradation: display_items_for_sale averages 2535ms (34 ops/sec) and register_item_for_sale shows 1840 (46 ops/sec).

This scenario shows the scalability limits of our system under very high load. When 100 buyers and 100 sellers run at the same time, many shared resources become bottlenecks. The REST server has limits on HTTP connections, the gRPC layer can become saturated, and the database connection pool may run out of available connections. Thread pools at different layers also compete for CPU time. As concurrency increases, the cost of converting data (JSON and Protocol Buffers) and handling many requests at once puts heavy pressure on the CPU. Network limits and TCP congestion also slow down communication between services. Write operations are affected more than read operations because they require database locks and transaction commits, which add extra waiting time under contention. Even though latency becomes very high in this scenario, throughput remains relatively steady. This shows that the system continues to process requests correctly under heavy load, but individual response times increase significantly due to queuing and resource contention.

Comparison Across Scenarios

The three scenarios show clear changes in performance as concurrency increases. When moving from 1 client to 10 clients, throughput increases by about 4.4x, while latency increases by about 2–3x. This means the system uses resources efficiently at moderate concurrency. It is able to handle many more requests per second with only a reasonable increase in response time.

However, when moving from 10 clients to 100 clients, throughput remains the same, but latency increases sharply (around 9x). This shows that the system is reaching its limits. At this stage, requests begin to wait in queues because shared resources such as CPU, threads, network bandwidth, and database connections are heavily contested.

The reason for this behavior is how the system handles requests at each layer. At low concurrency, most of the delay comes from protocol overhead (REST, gRPC, serialization). At moderate concurrency, the system performs well because it can reuse connections and efficiently share pooled resources. At high concurrency, resource exhaustion becomes the main issue. Since the system has multiple layers (REST server, gRPC layer, database), delays at one layer add up with delays at others, causing a large overall increase in latency.

PA1 vs PA2 Performance Comparison

Comparing PA2 with PA1 reveals significant performance trade-offs introduced by the architectural changes:

Single-Client Scenario: PA1 achieved much lower latency than PA2. For example, get_seller_rating averaged around 1–2ms in PA1, compared to about 36ms in PA2, which is roughly a 20x increase. Similarly, register_item_for_sale increased from about 0.75ms in PA1 to around 68ms in PA2, nearly a 90x increase. This large difference is mainly due to the architectural design of PA2. In PA1, the system used direct TCP socket communication, which involved fewer processing steps and a single communication path. In PA2, each request passes through multiple layers: HTTP request handling, JSON processing, gRPC calls, Protocol Buffer encoding, and additional network hops between services. Each of these steps adds extra processing time, resulting in higher overall latency compared to the simpler socket-based approach in PA1.

Moderate Concurrency (10 Clients): PA1 achieved much better raw performance, with get_seller_rating averaging about 7ms latency and 1602 ops/sec throughput. In comparison, PA2 shows around 82ms latency and 122 ops/sec throughput. This means PA2 has roughly 13x higher latency and about 13x lower throughput in absolute terms. However, when we look at scalability, PA2 performs better. In PA2, throughput increases by about 4.4x when moving from 1 client to 10 clients, whereas PA1 only shows about a 2.5x improvement. This indicates that PA2's architecture scales more efficiently as concurrency increases. The use of connection pooling, persistent gRPC channels, and asynchronous request handling helps distribute overhead across multiple requests. Although PA2 has higher base latency due to REST and gRPC layers, it utilizes resources more effectively under moderate load.

High Concurrency (100 Clients): PA1 still shows much better absolute performance. For get_seller_rating, PA1 averages about 69ms with 1469 ops/sec, while PA2 averages around 765ms with 110 ops/sec. This clearly shows that PA1 handles heavy load with much lower latency and higher throughput overall. However, when we look at how performance degrades from low to high concurrency, PA2 behaves more predictably. In PA1, latency increases about 40x when moving from 1 client to 100 clients. In PA2, latency increases about 21x over the same range. This means that although PA2 starts with higher baseline latency due to its multi-layer architecture, its performance degradation under load is more gradual and stable. The use of connection pooling, asynchronous request handling, and structured service layers helps PA2 manage concurrency in a more controlled way, even if the absolute numbers are lower than PA1.

Architectural Trade-offs: The performance differences between PA1 and PA2 highlight important architectural trade-offs. PA1 uses a simple socket-based design, which keeps communication direct and lightweight. This results in very low latency and high throughput. However, this tightly coupled approach makes the system less flexible, harder to scale independently, and more difficult to maintain as it grows.

In contrast, PA2 uses a layered REST and gRPC architecture. This design adds extra processing overhead, which increases latency and reduces raw performance. However, it provides clear benefits: stateless server design, standardized communication protocols, better separation between services, and the ability to scale components independently. It also aligns well with cloud-native deployment models and makes debugging and maintenance easier. Overall, PA2 trades some performance for improved modularity, scalability, and long-term maintainability.

For production deployments, PA2’s architecture is more practical even though it has lower raw performance. The stateless frontend allows easy horizontal scaling and load balancing across multiple servers. The gRPC layer uses efficient binary communication for service-to-service calls, which is better suited for structured, distributed systems. The modular design also makes it possible to scale, update, and deploy each component independently.

While PA2 introduces additional latency compared to PA1, this overhead is acceptable for most real-world applications. In production environments, factors such as scalability, reliability, maintainability, and ease of deployment are often more important than achieving the lowest possible latency.