Research Document

ChatsApp

Nhat Nam Ha

Table of content

Description

Chat applications have become an integral part of modern communication, offering users a platform to interact, share information, and stay connected in real-time. However, developing a chat application comes with its own set of challenges, which need to be addressed to ensure a seamless user experience. The target audiences are for the small-medium groups in the organizers, companies for private usage.

My main research questions focus on the optimizing topics, from the low-latency approaches and minimizing operation costs if the project being commercialized. The expected benefit from this research is better user experience minimizing the disruption, the business gains more profits.

Research question and sub questions

What are the straggles to support users chatting effectively in ChatsApp ?

- Which databases will be suitable for storing and retrieving chat messages in ChatsApp ?
- Which protocol provides low-latency chat transmission between users?
- What are other techniques can be used for reducing the latency between users?

Research Plan

Research Questions	Research methods	Expected Deliverables
Which databases will be suitable for storing and retrieving chat messages in ChatsApp ?	Library: • Community research • Literature study Lab: • Computer simulation Workshop: • Prototyping	A document describes the number of choices of protocols, highlights how it works based on the implementation.
Which protocol provides low-latency chat transmission between users?	Library: • Community research • Literature study Lab: • Computer simulation	A document describes common metrics available and implementation for visualizing them, the chosen metrics will be used for the next main research question.
What are other techniques can be used for reducing	Library: • Community research	A document describes the practices available from the internet and benchmark tests for comparing the

the latency between users?	• Literature study • Best good and bad practices Showroom: • Benchmark tests	efficiency from these approaches and the results will be used in this chat project.

Research results

Which databases will be suitable for storing and retrieving chat messages in ChatsApp ?

Research methods: Community research, Literature study

When considering databases for storing and retrieving chat messages in an application like ChatsApp, I have to look for databases that can efficiently handle high-volume, real-time data and support the necessary querying and indexing for message retrieval.

From database of databases, with the criteria supports high-performance, MongoDB, Scylla, PostgreSQL (Postgres), and MariaDB are popular choices in industry, each with its own strengths.

Database	Description
MongoDB	MongoDB is a NoSQL database that excels in handling unstructured or semi-structured data like chat messages. Its flexible schema allows for easy adaptation to evolving data structures. MongoDB's document-oriented storage is well-suited for storing chat messages where each message can be represented as a document.
PostgreSQL	PostgreSQL is a robust relational database system that provides powerful querying capabilities. It is suitable for scenarios where complex queries or transactions are required on chat data. Postgres support for JSONB (JSON Binary) data type allows for efficient storage and querying of semi-structured data like chat messages.
MariaDB	MariaDB is a fork of MySQL and offers similar relational database features. It provides reliable performance and scalability for handling chat message data. With appropriate indexing and schema design, MariaDB can efficiently manage chat message storage and retrieval.
Scylla	Scylla: a high-performance NoSQL database that is compatible with Apache Cassandra but offers significant performance improvements. It is designed to handle large-scale workloads with low latency and high throughput.

Test scenarios

Research methods: Computer simulation, Prototyping.

To choose the suitable databases, I have made test scenarios which applied to all 4 databases mentioned above. The first one is Inserting 1000 Chat Messages and Fetching All Chat Messages using JMeter, which only one client backend making queries to database.

Insert 1000 Chat Messages:

- Configure JMeter to execute database queries to insert chat messages into the database.

- Measure the throughput (messages inserted per second), response times, and any errors encountered during the insertion process.

Fetch All Chat Messages in 5 seconds:

- After inserting the chat messages, simulate the scenario of fetching all chat messages from the database.
- Define a thread group in JMeter with multiple concurrent threads representing users.
- Use JMeter to send database select query to retrieve all chat messages.
- Configure JMeter to handle the response and validate the correctness of retrieved messages.
- Measure the response times, throughput, and resource utilization (CPU, memory, network) during the fetching operation.

Insertion Metrics:

- Throughput: Number of chat messages inserted per second. (requests per second)
- Response Time: Average, minimum, and maximum time taken to insert a chat message (millisecond).
- Error Rate: Percentage of failed insertions (% error rate).

Fetching Metrics:

- Response Time: Average, minimum, and maximum time taken to fetch all chat messages (millisecond).
- Throughput: Number of chat messages fetched per second. (requests per second)
- Validation: Ensure all chat messages retrieved are correct and complete. (% error rate)

Expected Outcomes:

- Identify performance bottlenecks related to chat message insertion and retrieval.
- Determine the scalability and responsiveness of the database under the simulated load.
- Validate the system's ability to handle concurrent chat interactions effectively.

The second extreme load test has been made with inserting 100_000 messages and fetching all chat messages. This one is applied when the error rate from the first test is 0% or must not produce errors.

MariaDB load tests

The first one below is MariaDB, a SQL database, insertion throughput is around 1712 requests per second, reading throughput is around 6478 requests per second, and did not produce errors while in loading tests.

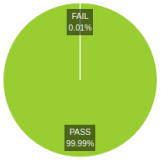
Continuing with second load tests with higher number of messages have been inserted, insertion throughput is around 2100 requests per second, but reading throughput is decrease around 100 requests per second, which is dramatically.

Source file	"mariadb.csv"
Start Time	"5/3/24, 12:37 PM"
End Time	"5/3/24, 12:37 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
0.000	500 ms	1 sec 500 ms	Setup Request
0.999	500 ms	1 sec 500 ms	JDBC insert Request
1.000	500 ms	1 sec 500 ms	Get Request
1.000	500 ms	1 sec 500 ms	clear Request

Requests Summary



Statistics

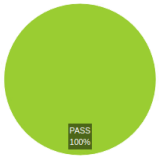
Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**33397**	**2**	**0.01%**	**0.16**	**0**	**105**	**0.00**	**1.00**	**1.00**	**1.00**	**5863.24**	**79.30**	**0.00**
Get Request	32395	0	0.00%	0.15	0	10	0.00	1.00	1.00	1.00	6477.70	88.56	0.00
JDBC insert Request	1000	1	0.10%	0.50	0	46	0.00	1.00	1.00	2.00	1712.33	15.05	0.00
clear Request	1	0	0.00%	3.00	3	3	3.00	3.00	3.00	3.00	333.33	2.93	0.00
Setup Request	1	1	100.00%	105.00	105	105	105.00	105.00	105.00	105.00	9.52	0.08	0.00

Source file	"mariadb.csv"
Start Time	"5/3/24, 12:55 PM"
End Time	"5/3/24, 12:56 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
1.000	500 ms	1 sec 500 ms	Get Request
1.000	500 ms	1 sec 500 ms	JDBC insert Request
1.000	500 ms	1 sec 500 ms	Setup Request
1.000	500 ms	1 sec 500 ms	clear Request

Requests Summary



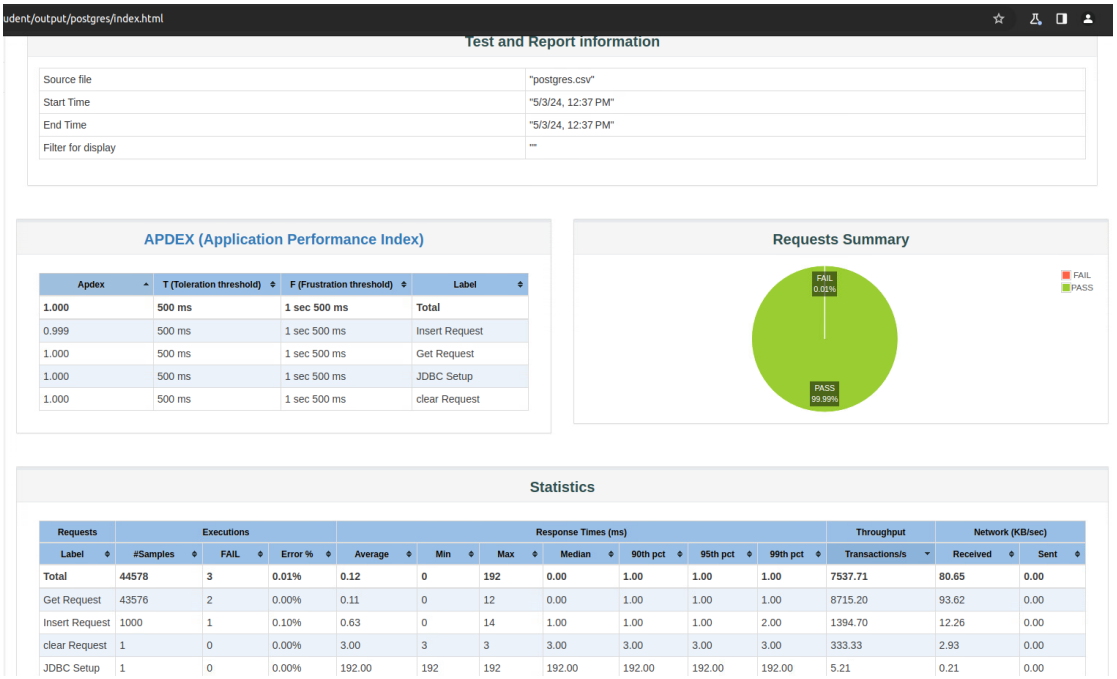
Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**101042**	**0**	**0.00%**	**0.56**	**0**	**209**	**0.00**	**1.00**	**9.00**	**10.00**	**1763.26**	**15.62**	**0.00**
JDBC insert Request	100000	0	0.00%	0.46	0	209	0.00	1.00	1.00	2.00	2118.96	18.62	0.00
clear Request	1	0	0.00%	2.00	2	2	2.00	2.00	2.00	2.00	500.00	4.39	0.00
Get Request	1040	0	0.00%	9.59	9	20	10.00	10.00	10.00	11.00	103.97	1.62	0.00
Setup Request	1	0	0.00%	104.00	104	104	104.00	104.00	104.00	104.00	9.62	0.08	0.00

PostgreSQL load tests

The second one is PostgreSQL, another SQL database, well-known in industry. Insertion throughput is around 1314 requests per second, reading throughput is around 8715 requests per second, which is higher than MariaDB above and did not produce errors while loading tests.

Continuing with second load tests with higher number of messages have been inserted, insertion throughput is around 1873 requests per second, but reading throughput is decrease around 263 requests per second, which is dramatically.


udent/output/postgres/index.html

Test and Report information

Source file	"postgres.csv"
Start Time	"5/3/24, 12:37 PM"
End Time	"5/3/24, 12:37 PM"
Filter for display	""

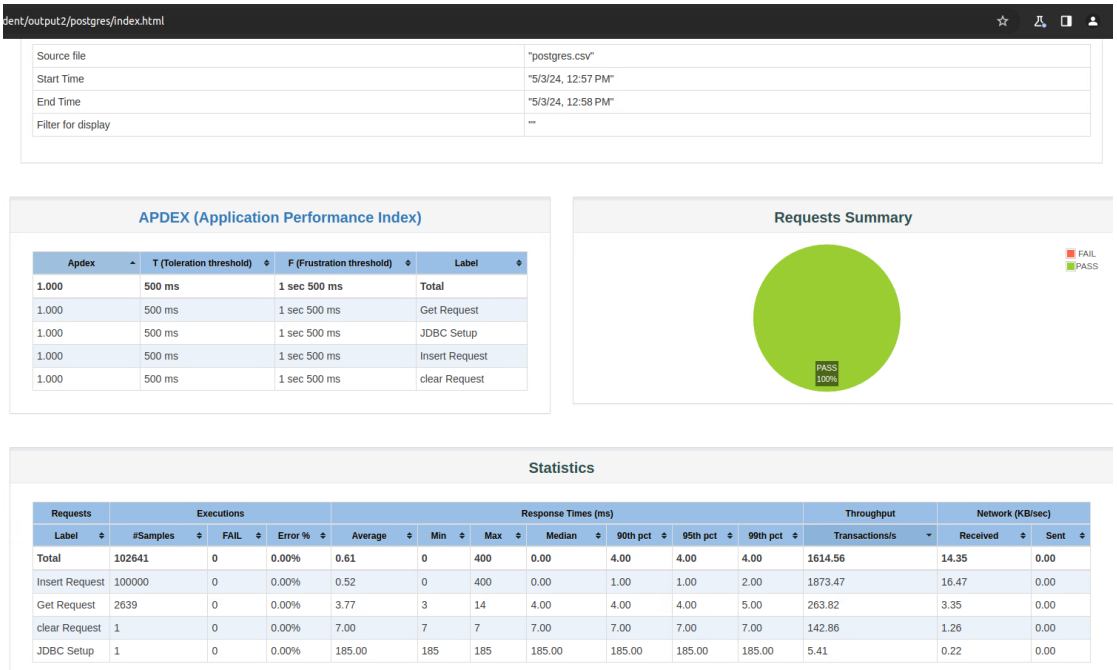
APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
0.999	500 ms	1 sec 500 ms	Insert Request
1.000	500 ms	1 sec 500 ms	Get Request
1.000	500 ms	1 sec 500 ms	JDBC Setup
1.000	500 ms	1 sec 500 ms	clear Request

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**44578**	**3**	**0.01%**	**0.12**	**0**	**192**	**0.00**	**1.00**	**1.00**	**1.00**	**7537.71**	**80.65**	**0.00**
Get Request	43576	2	0.00%	0.11	0	12	0.00	1.00	1.00	1.00	8715.20	93.62	0.00
Insert Request	1000	1	0.10%	0.63	0	14	1.00	1.00	1.00	2.00	1394.70	12.26	0.00
clear Request	1	0	0.00%	3.00	3	3	3.00	3.00	3.00	3.00	333.33	2.93	0.00
JDBC Setup	1	0	0.00%	192.00	192	192	192.00	192.00	192.00	192.00	5.21	0.21	0.00


dent/output2/postgres/index.html

Source file	"postgres.csv"
Start Time	"5/3/24, 12:57 PM"
End Time	"5/3/24, 12:58 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
1.000	500 ms	1 sec 500 ms	Get Request
1.000	500 ms	1 sec 500 ms	JDBC Setup
1.000	500 ms	1 sec 500 ms	Insert Request
1.000	500 ms	1 sec 500 ms	clear Request

Requests Summary

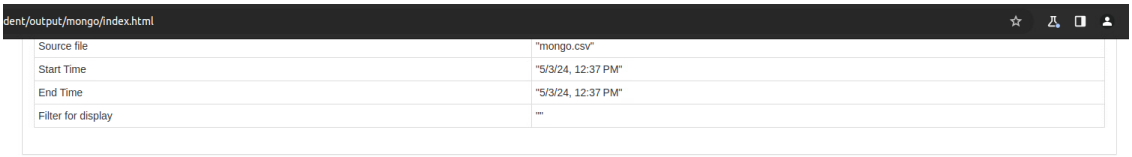
Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**102641**	**0**	**0.00%**	**0.61**	**0**	**400**	**0.00**	**4.00**	**4.00**	**4.00**	**1614.56**	**14.35**	**0.00**
Insert Request	100000	0	0.00%	0.52	0	400	0.00	1.00	1.00	2.00	1873.47	16.47	0.00
Get Request	2639	0	0.00%	3.77	3	14	4.00	4.00	4.00	5.00	263.82	3.35	0.00
clear Request	1	0	0.00%	7.00	7	7	7.00	7.00	7.00	7.00	142.86	1.26	0.00
JDBC Setup	1	0	0.00%	185.00	185	185	185.00	185.00	185.00	185.00	5.41	0.22	0.00

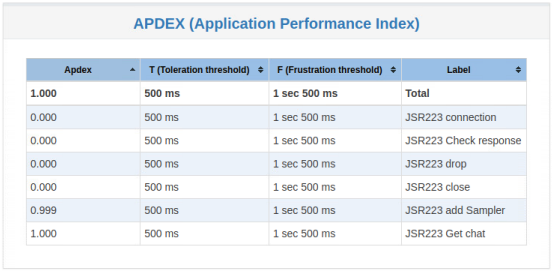
MongoDB load tests

The third one is MongoDB, a No-SQL database, also well-known in industry. Insertion throughput is around 1187 requests per second, reading throughput is around 8715 requests per second, which is surprisingly the highest read-speed and did not produce errors while in loading tests.

Continuing with second load tests with higher number of messages have been inserted, insertion throughput is around 7361 requests per second, but reading throughput is increase around 43000 requests per second, which is significant.

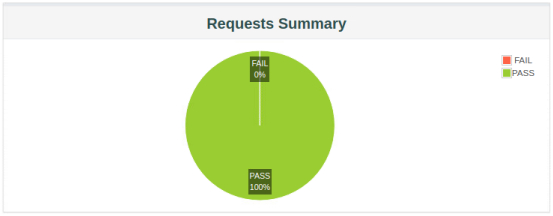


Source file	"mongo.csv"
Start Time	"5/3/24, 12:37 PM"
End Time	"5/3/24, 12:37 PM"
Filter for display	""



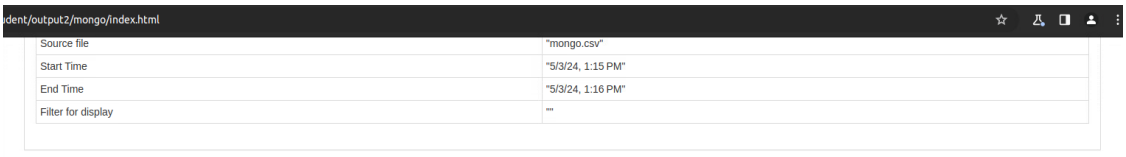
APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
0.000	500 ms	1 sec 500 ms	JSR223 connection
0.000	500 ms	1 sec 500 ms	JSR223 Check response
0.000	500 ms	1 sec 500 ms	JSR223 drop
0.000	500 ms	1 sec 500 ms	JSR223 close
0.999	500 ms	1 sec 500 ms	JSR223 add Sampler
1.000	500 ms	1 sec 500 ms	JSR223 Get chat

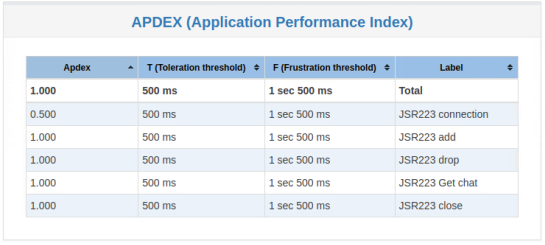


Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**190027**	**6**	**0.00%**	**0.03**	**0**	**749**	**0.00**	**0.00**	**0.00**	**1.00**	**28674.66**	**0.00**	**0.00**
JSR223 Get chat	189023	1	0.00%	0.02	0	16	0.00	0.00	0.00	1.00	38481.88	0.00	0.00
JSR223 add Sampler	1000	1	0.10%	0.73	0	74	1.00	1.00	1.00	2.00	1187.65	0.00	0.00
JSR223 close	1	1	100.00%	16.00	16	16	16.00	16.00	16.00	16.00	62.50	0.00	0.00

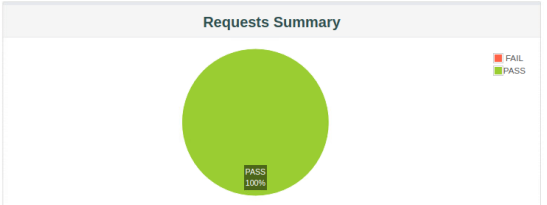


Source file	"mongo.csv"
Start Time	"5/3/24, 1:15 PM"
End Time	"5/3/24, 1:16 PM"
Filter for display	""



APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	**500 ms**	**1 sec 500 ms**	**Total**
0.500	500 ms	1 sec 500 ms	JSR223 connection
1.000	500 ms	1 sec 500 ms	JSR223 add
1.000	500 ms	1 sec 500 ms	JSR223 drop
1.000	500 ms	1 sec 500 ms	JSR223 Get chat
1.000	500 ms	1 sec 500 ms	JSR223 close

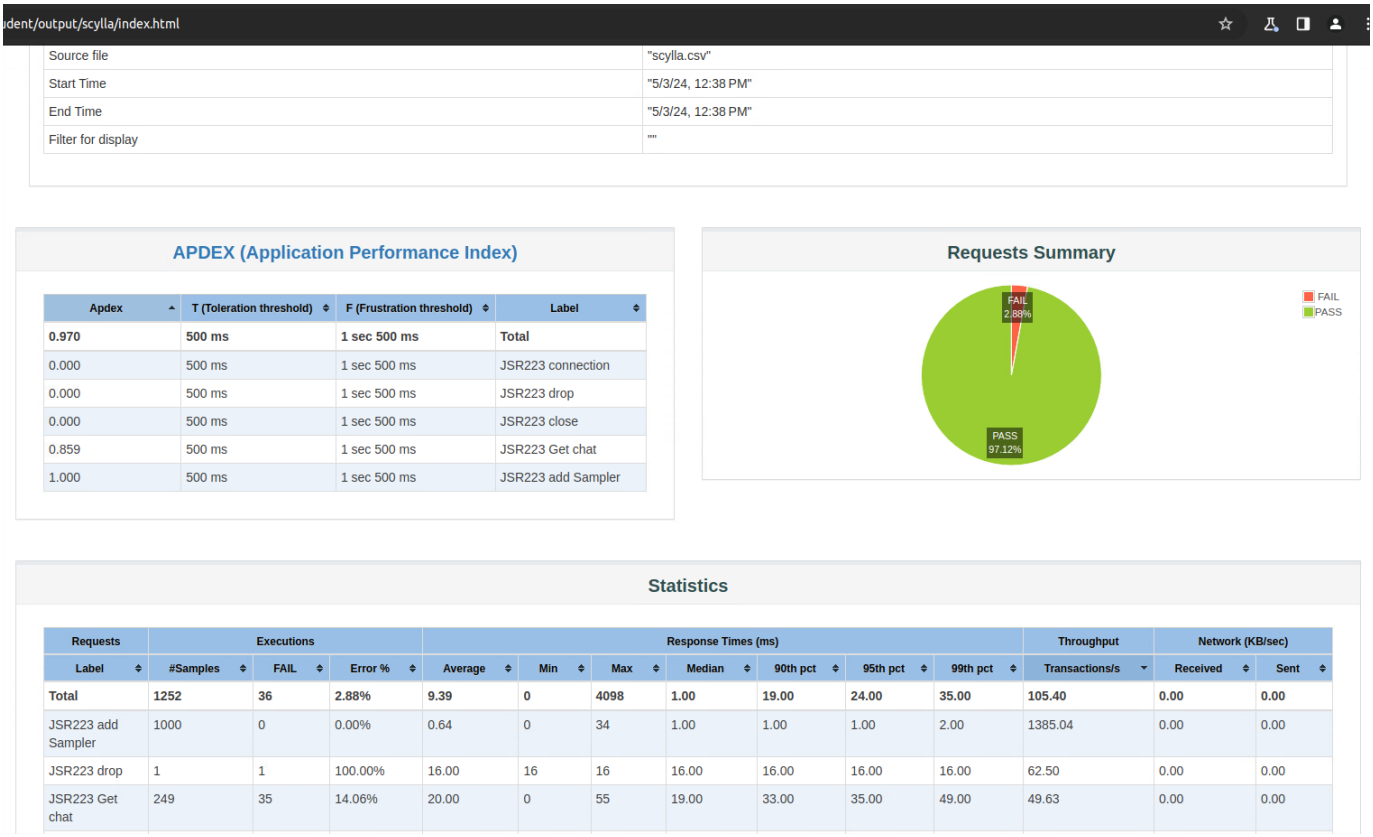


Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**533992**	**0**	**0.00%**	**0.04**	**0**	**764**	**0.00**	**0.00**	**0.00**	**1.00**	**21878.64**	**0.00**	**0.00**
JSR223 Get chat	433989	0	0.00%	0.02	0	26	0.00	0.00	0.00	1.00	43398.90	0.00	0.00
JSR223 add	100000	0	0.00%	0.12	0	72	0.00	0.00	1.00	1.00	7361.06	0.00	0.00
JSR223 close	1	0	0.00%	17.00	17	17	17.00	17.00	17.00	17.00	58.82	0.00	0.00
JSR223 drop	1	0	0.00%	38.00	38	38	38.00	38.00	38.00	38.00	26.32	0.00	0.00

Scylla load tests

The last one is Scylla, a No-SQL database, famous for powering Discord. This database provides error rate when running in load-testing for fetching chat messages, whereas the other three did not show unstable in load-test.



dent/output/scylla/index.html

Source file	"scylla.csv"
Start Time	"5/3/24, 12:38 PM"
End Time	"5/3/24, 12:38 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.970	**500 ms**	**1 sec 500 ms**	**Total**
0.000	500 ms	1 sec 500 ms	JSR223 connection
0.000	500 ms	1 sec 500 ms	JSR223 drop
0.000	500 ms	1 sec 500 ms	JSR223 close
0.859	500 ms	1 sec 500 ms	JSR223 Get chat
1.000	500 ms	1 sec 500 ms	JSR223 add Sampler

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**1252**	**36**	**2.88%**	**9.39**	**0**	**4098**	**1.00**	**19.00**	**24.00**	**35.00**	**105.40**	**0.00**	**0.00**
JSR223 add Sampler	1000	0	0.00%	0.64	0	34	1.00	1.00	1.00	2.00	1385.04	0.00	0.00
JSR223 drop	1	1	100.00%	16.00	16	16	16.00	16.00	16.00	16.00	62.50	0.00	0.00
JSR223 Get chat	249	35	14.06%	20.00	0	55	19.00	33.00	35.00	49.00	49.63	0.00	0.00

From the tests above, Scylla is not the suitable choice for ChatsApp due to unstable and inconsistency in load-test, produce percentage of error rate, the other three did not produce error rate in all load tests, are suitable candidates working in stress environment, receiving requests concurrently.

Conclusion choosing database

Research methods: Computer simulation and Benchmark tests.

PostgreSQL and MariaDB show the declined results in fetching requests, the throughput around 200 requests per second. Mongo keeps the speed around 40000 requests per second, even it needs fetching 100_000 messages in one request.

So, from the load test above, using MongoDB is a suitable choice for ChatsApp database storage.

Which protocol provides low-latency chat transmission between users?

Research methods: Community research, Literature study

For low-latency chat transmissions and handling concurrent users at same time, there are 2 types of technologies are WebSocket and SignalR.

WebSocket (WS):

- Protocol: WebSocket is a standardized protocol that provides full-duplex communication over a single TCP connection. It is designed for low-latency, real-time communication between clients and servers.
- Low-Latency Chat Transmission: WebSocket is well-suited for low-latency chat transmission because it establishes a persistent connection that allows messages to be sent and received instantly without the overhead of traditional HTTP requests.
- Handling Concurrent Users: WebSocket can efficiently handle concurrent users by leveraging its event-driven, asynchronous nature. The server can manage multiple WebSocket connections simultaneously, making it scalable for supporting many concurrent chat users.
- Scalability: WebSocket is a foundational protocol and can be integrated into various backend technologies to build scalable and efficient chat applications.

SignalR:

- Framework: SignalR is a library/framework built on top of WebSocket and other transport mechanisms (like Server-Sent Events, Long Polling) to simplify real-time web applications.
- Low-Latency Chat Transmission: SignalR leverages WebSocket (when available) for low-latency communication, making it suitable for real-time chat applications. However, SignalR adds an abstraction layer over WebSocket, which might introduce some overhead compared to raw WebSocket usage.
- Handling Concurrent Users: SignalR abstracts away the complexities of managing WebSocket connections and provides features like automatic connection management, reconnection, and fallback to other transports. This simplifies handling concurrent users but may add some performance overhead compared to a direct WebSocket implementation.
- Additional Features: SignalR offers features like hub-based communication, client and server method invocation, and group messaging, which can enhance the development experience but may impact pure low-latency performance.

Test scenario

Research methods: Computer simulation, Prototyping.

Testing scenario is how many users connected concurrently by an instance of server, as a load test. The objective of this load test is to determine the maximum number of concurrent users that can be connected to a server instance before performance degradation occurs.

Define Test Plan in JMeter:

- Create a new test plan in JMeter for the concurrent user connection load test.
- Configure JMeter Thread Group:
- Define the number of concurrent users (threads) to simulate. Start with a reasonable number (e.g., 100) and gradually increase.
- Set up ramp-up period and loop count based on the desired testing duration and intensity.

Execute Load Test:

- Start the load test in JMeter to simulate concurrent user connections to the server instance.
- Monitor the server's performance metrics such as response time, throughput, and error rate during the test execution.
- Gradually increase the number of concurrent users in subsequent test iterations until the server's performance metrics indicate degradation (e.g., increased response time, reduced throughput, error responses).

Test Metrics to Capture:

- Concurrency Level: Maximum number of concurrent users successfully connected to the server without significant performance degradation. (Total samples subtract Fail samples)
- Response Time: Measure the response time of the server under increasing load. (milliseconds)
- Throughput: Determine the throughput (requests per second) the server can handle at different concurrency levels.
- Error Rate: Monitor error responses or failed connections as concurrency increases. (%)

WebSocket results

Test and Report information

Source file	"ws.csv"
Start Time	"5/8/24, 2:17 PM"
End Time	"5/8/24, 2:17 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.958	**500 ms**	**1 sec 500 ms**	**Total**
0.002	500 ms	1 sec 500 ms	WebSocket Open Connection
1.000	500 ms	1 sec 500 ms	WebSocket Single Write Sampler

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**706200**	**23238**	**3.29%**	**250.83**	**0**	**10848**	**0.00**	**5005.00**	**5006.00**	**5553.00**	**31463.58**	**37.95**	**546.93**
WebSocket Open Connection	30000	23238	77.46%	5904.32	11	10848	5223.00	9578.00	9678.00	9886.00	1336.60	37.95	46.78
WebSocket Single Write Sampler	676200	0	0.00%	0.01	0	863	0.00	0.00	0.00	0.00	39662.15	0.00	658.45

By testing 30_000 user requests concurrently, there are more than 23000 failed samples connecting to server, the WebSocket handles around 7000 users at the same time.

SignalR results

Test and Report information

Source file	"signalr.csv"
Start Time	"5/8/24, 2:20 PM"
End Time	"5/8/24, 2:20 PM"
Filter for display	""

APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
0.998	**500 ms**	**1 sec 500 ms**	**Total**
0.766	500 ms	1 sec 500 ms	WebSocket Open Connection
1.000	500 ms	1 sec 500 ms	WebSocket Single Write Sampler

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**2856700**	**1733**	**0.06%**	**5.01**	**0**	**3536**	**0.00**	**0.00**	**0.00**	**4.00**	**166106.52**	**293.73**	**2995.11**
WebSocket Open Connection	30000	1733	5.78%	476.05	0	3536	130.00	2258.00	2760.00	3280.99	1744.39	293.73	266.45
WebSocket Single Write Sampler	2826700	0	0.00%	0.01	0	33	0.00	0.00	0.00	1.00	165691.68	0.00	2750.74

By testing 30_000 user requests concurrently, there are around 1700 failed samples connecting to server, the SignalR handles around 29000 users at the same time.

Conclusion choosing protocols

Research methods: Computer simulation and Benchmark tests.

From the test report below, SignalR can handle almost 30000 users at the same time, which is more than 4 times than WebSocket, which handles 700 users at the same time.

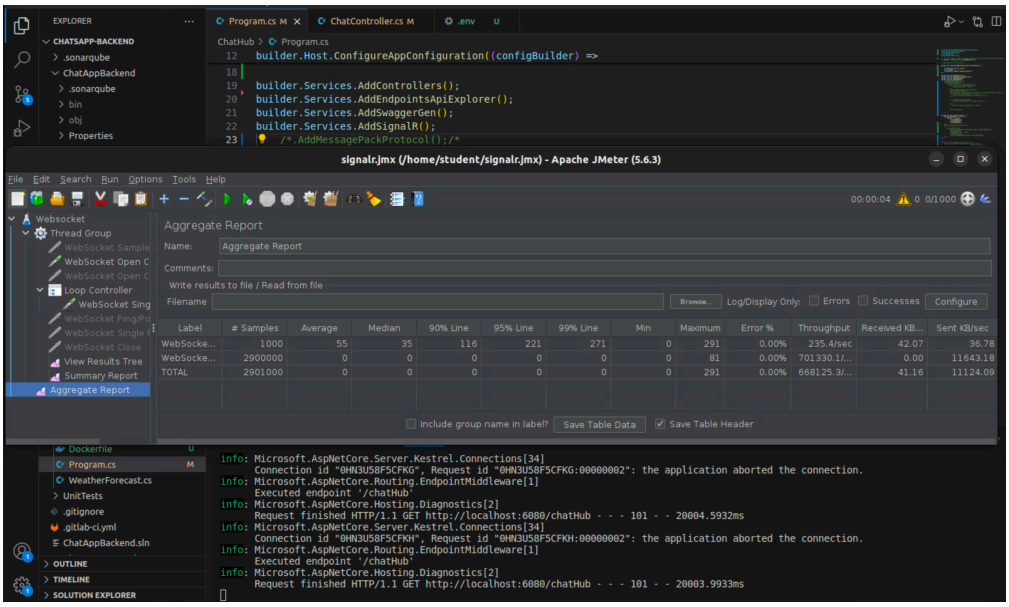
So, from the load test above, using SignalR is a suitable choice for ChatsApp transmission protocol.

What are other techniques can be used for reducing the latency between users?

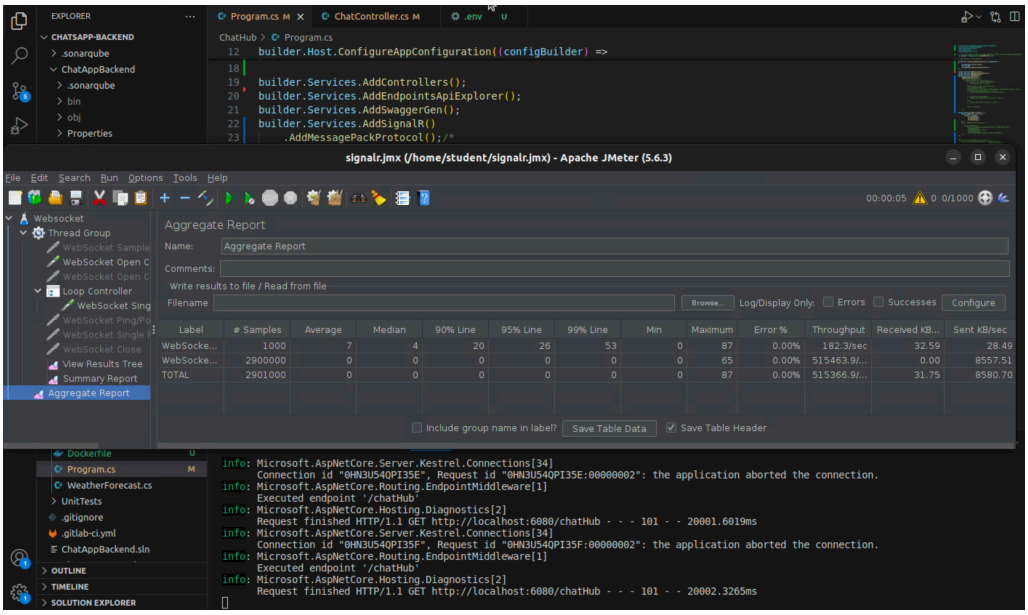
Reducing latency in chat transmission is crucial for ensuring a seamless and real-time communication experience. Here are several techniques that can be employed to achieve this.

Binary Data Formats

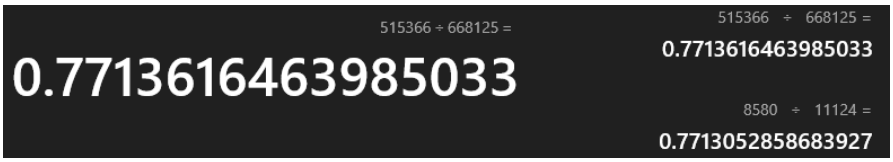
Employ efficient data formats like Protocol Buffers or MessagePack instead of JSON or XML.



From the benchmark above, average payload size for sending it is around more than 11000 KB for transmission and the average throughput is around 668_000 requests per second.



From the benchmark above, average payload size for sending it is around 8580 KB for transmission and the average throughput is around 515_000 requests per second, which is normal for compression need more computing for compressing.



From the measurement, the MessagePack reduces the payload size and throughput around 22%.