International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V2I3P107
Eureka Vision Publication | Volume 2, Issue 3, 60-69, 2021

Original Article

A Hybrid WebSocket-REST Approach for Scalable Real-Time API Design

Varun Kumar Chowdary Gorantla
Independent Researcher USA.

Abstract *- The rapid increase of real-time web applications and scalable systems during recent years demands sturdy architectural systems that enable effective data exchange. Representational State Transfer APIs operate traditionally through stateless communication methods that are widely adopted thanks to their basic design and compatibility features. The real-time bidirectional communication capabilities of REST APIs come short when several applications, such as chat systems, multiplayer games, and financial trading platforms, require efficient functionality. WebSockets conduct real-time data exchanges through full-duplex communication across a single TCP connection, although they necessitate higher complexities during deployment and when expanding system capacity. A new architectural design unites WebSocket and REST capabilities for building web APIs, which provide scalability and real-time functionality. The hybrid design architectural model implements REST throughout initial state retrieval and client control tasks and then utilizes WebSocket functions for live time communications, status synchronization, and event dissemination. The proposed architecture performance analysis under different test loads shows results against implementations that use pure WebSocket and REST interfaces. Experimental studies show that implementing hybrid technology results in substantial enhancements of latency combined with improved throughputs and lowered server resource consumption, especially within highly concurrent systems. The architecture evaluation takes place through testing on an e-commerce simulation platform, which shows its effectiveness in meeting concrete business needs. Security implications alongside fault tolerance and scalability enhancements form the main topics of this research paper. A combination of WebSocket and REST architecture creates a feasible platform that substantially benefits modern applications needing real-time capabilities without compromising scalability.*

Keywords *- WebSocket, REST API, Real-Time Communication, Scalability, Hybrid Architecture, Client-Server Model, Low Latency, API Design*

1. Introduction

1.1 Background

Web development has experienced an evolution from basic static development to highly interactive dynamic applications. [1-4] Multiple sectors, including IoT, social media, collaborative platforms, and online gaming, must currently use real-time data delivery.

1.2 Emergence of WebSocket

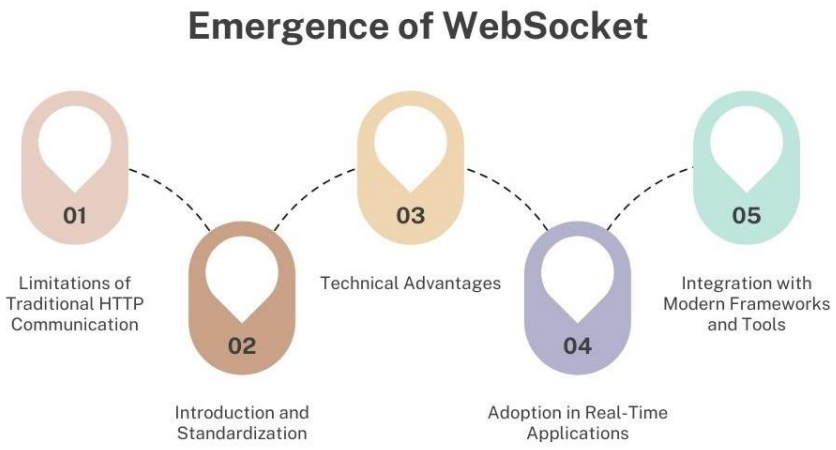

Figure 1: Emergence of WebSocket

- **Limitations of Traditional HTTP Communication:** Before WebSocket's existence became a possibility, web applications had a way of enabling communication between the client and server through HTTP. As would be expected, since HTTP is stateless and relies on poll-based or long-poll-based mechanisms, it introduced certain

limitations in real-time environments. This was due to repeated HTTP requests, which wasted bandwidth time and added more load to the client and server. This pointed to the significance of having a faster and low-latency system for the kind of applications that would be frequently engaged.

- **Introduction and Standardization:** WebSocket originated from the HTML 5 standard and was standardized by the IETF under the RFC 6455 in 2011. To solve HTTP's flaws, WebSocket facilitates bi-directional communication through a single TCP connection. This shifted the model from the request/response model, where systems have a one-time complete conversation, to what we now call or need for always-on two-way communications, well-suited for real-time applications such as live chats or games and financial tickers.
- **Technical Advantages:** WebSocket delivers numerous advantages against HTTP-based communication techniques because of its technical aspects. The principal benefit of WebSocket technology arises from its instantaneous connection setup, which scrapes the need to create new paths per interaction. The network overhead decreases because the header format remains concise. Through its continuous open connection, servers can deliver fresh data to clients at once without polling. WebSocket has become the preferred communication choice in environments where performance needs to be maximized because of its advantages.
- **Adoption in Real-Time Applications:** WebSocket technology has expanded its usage throughout various industrial sectors at a fast pace. By implementing the WebSocket platform, Slack provides real-time capabilities at scale, together with Twitter and Trello, as well as stock trading systems. WebSocket proves its effectiveness in specific applications that require instant messaging and collaborative document editing, as well as live notifications, multiplayer gaming, and IoT data streaming. Modern web applications are experiencing a transformation in data distribution because of escalating WebSocket implementation.
- **Integration with Modern Frameworks and Tools:** Since WebSocket became popular, its functionality has been incorporated into contemporary development frameworks and tools. Implementing WebSocket becomes simpler through tools like Socket.IO, which serves Node.js frameworks and WS and Channels designed for Django. Cloud providers and container orchestration tools now support WebSocket-compatible infrastructure, which promotes its application in distributed systems that scale according to need. The advancements made it easier for developers to use WebSocket and helped it spread quickly through multiple application domains.

1.3 REST and Its Limitations

Web service design established Representational State Transfer (REST) as its foundation when Roy Fielding published the concept in his doctoral dissertation in 2000. Standard HTTP request methods GET, POST, PUT, and DELETE enable its operation while conforming to statelessness requirements, uniform interface rules, and layered system architectural constraints. REST principles drive the high popularity of building scalable and maintainable APIs. Each client request carries all needed information since REST is stateless, which allows easy scalability and simplifies server-side implementation. REST presents high interoperability with web infrastructure, which enables server accessibility to multiple platforms and devices. A straightforward approach, extensive tool and framework support, and documentation have made REST the prime architecture selection for servicing web resources. [5,6] The major shortcomings of REST become apparent in real-time and event-based application development scenarios. Its main drawback emerges from its pull-based architecture, which requires clients to use fresh HTTP requests to acquire updated information.

The immediate need for data delivery creates issues for live chats, collaborative tools, financial dashboards, and IoT telemetry since pull-based communication is a drawback. Developers overcome real-time simulations through inefficient methods such as polling and long-polling while making repeated HTTP requests when no new data is available. Server performance decreases while bandwidth utilization grows alongside this practice, which ultimately causes display delays that affect user comfort. REST does not contain an internal feature to support server-triggered data transmissions. Clients must adopt cumbersome solutions because REST lacks native methods to transfer data from server endpoints, which results in large-scale infrastructure requirements. The restrictions of REST have motivated the implementation of WebSocket as an alternative protocol because it establishes full-duplex communication and event-driven data transfer.

1.4 Need for Hybrid Architecture

Web applications need communication frameworks to achieve reliable performance and fast response times while sustaining scalability for current and future needs. Combining REST and WebSocket protocols within a single architecture provides an excellent solution for these communication needs. A stateless REST framework operates best for executing standard application operations consisting of user authentication, data retrieval, configuration updates, and batch process execution. The system uses REST methodology for tasks that do not need real-time interactions while maintaining high scalability because it works with HTTP tools and stateless infrastructure. WebSocket is an ideal addition to REST APIs by establishing permanent full-duplex connections that facilitate essential real-time communication between clients and servers. Applications like collaborative editing systems and live messaging need WebSocket for instant feedback. Users demand seamless real-time interaction for notification systems, gaming, and Internet of Things data streaming. The server can enhance performance because WebSocket provides instant push notification services that market REST-based polling systems as inefficient legacy practices.

The hybrid method uses both technologies to their full potential since it allocates a suitable protocol for each task. REST maintains control of initializations through structured requests and WebSocket, which focuses on handling continuous event-driven updates. Using this combination enables better efficiency in application performance, user experience, and resource optimization. Developers can create scalable and dynamic systems using a combination of REST and WebSockets, which avoids unnecessary system complexity. The implementation of hybrid architecture receives assistance from contemporary development frameworks, which enables developers to construct modular buildings and maintain their work efficiently. Web applications benefit substantially from using the hybrid REST-WebSocket model since it delivers a practical and future-proof approach to developing high-performance applications that require scalability and responsiveness.

2. Literature Survey

2.1 RESTful API Design Principles

RESTful APIs are preferred as the most suitable models for web services because of their compatibility with HTTP. As has been acknowledged by many authors, REST is well-appealed for implementation of Create, Read, Update, and Delete (CRUD) due to its statelessness and methods, namely, GET, POST, PUT, and DELETE. Scalability and performance benefit from built-in HTTP features such as caching, proxying, and REST's layered system architecture in distributed systems. [7-11] However, a weakness that becomes apparent when it comes to real-time applications is that the REST operation requires frequent updates. In such cases, this can result in poor realization of the available network bandwidth and latency due to the constant polling in waiting for updated data or events.

2.2 WebSocket Implementations

WebSocket, on the other hand, provides bidirectional communication of full-duplex connections over a single connection using a single, long-lived TCP connection, which will minimize the latency and overhead associated with real-time data transfer. Previous works suggest using it in contexts like online gaming, chat-related services, and live data streaming since it is vital to have frequent updates. However, using WebSocket for production involving large applications brings some issues. Sustaining persistent connections costs more in terms of server resources, and load redistribution can become bulky and complicated when there are many connections at a time. Furthermore, error handling and remaking mechanisms also add to the complexity of system design for the WebSocket interface.

2.3 Hybrid Approaches

Therefore, recent studies have looked at combining REST or WebSocket since their independent utilization has drawbacks. Some authors recommend models that involve API gateways, including either REST or WebSocket, based on the context or the client's request. This strategy is an attempt to take the best from the REST and WebSocket approaches and mixes them in such a way that would create a great usage of the features offered by both. Other works have used fallback to HTTP polling since WebSocket is the default communication protocol over which the application can switch if a persistent connection cannot be made. These mixed strategies proved somewhat effective, especially if the network stability was an issue or the clients were a mixed bunch.

2.4 Industry Trends

This argument is supported by the fact that hybrid communication strategies have been embraced in the current working world. Some examples of businesses, including Slack, Trello, and Facebook Messenger, use Rest and WebSocket integrations to provide users with a responsive experience despite dealing with high traffic volumes. For instance, REST can be utilized for original data pull or as a mechanism to handle minimal updates, while WebSocket is used to publish real-time updates and notifications. These implementations show that system designs are shifting to make the communication process dependent on the operation and interaction practiced.

3. Methodology

3.1 System Architecture

Some of the components that make up the hybrid architecture are as follows:

- **Client:** In the case of the client being a web or a mobile application, it uses RESTful APIs to establish a connection with the server to request the necessary data when creating the client or during its infrequent operations. At the start, the client connects a WebSocket to receive real-time data. [8-12] This approach allows the client to gain easy data modeling with REST for structured information and use WebSockets to support efficient real-time information such as notifications, messages, or dashboards.
- **API Gateway:** A distinct module resides at the external interface as the leading point of access to the system where all clients' requests are channelled. Depending on the operation type, it is designed to dynamically send requests to either REST or WebSocket. Thus, the communication logic is abstracted with the help of the gateway, and such control options as authentication, rate limiting, and protocol changing are carried out on the client side. This layer improves security and scalability and propagates hybrid interactions between client and server.

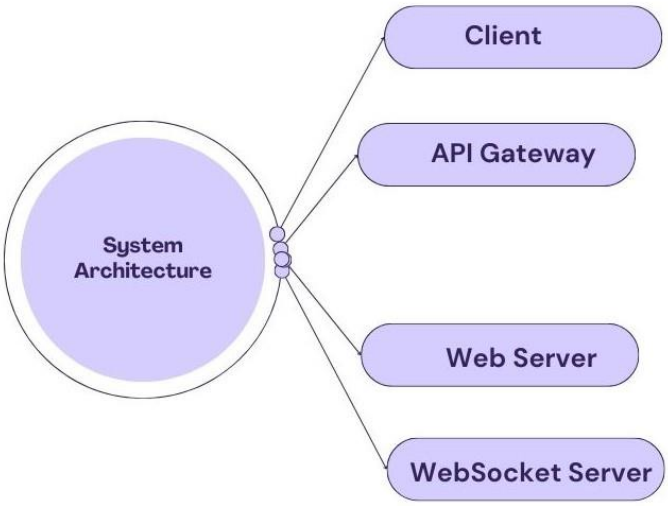

Figure 2: System Architecture

- **Web Server:** Most RESTful services are served on the web server platform; the application development is done using different web frameworks such as Express.js in the Node.js environment or Django environment for Python programming language. Such services manage database operations, user identification, configuration retrieving, and call-and-response work. Net API being stateless lends itself to being cacheable; thus, the web server as a component is very predictable and stable.
- **WebSocket Server:** WebSocket is to have a long-lived, bi-directional socket connection to clients to be able to provide real-time delivery of updates and messages. It is popularly used through libraries like Socker.IO for Node.js or WS, which carry most of the complexities of connection, multicast, and client' sessions. This server is fairly suitable for concurrency and swift data throughput, which means it is ideal for live chat, real-time analysis, or collaborative functionalities.

3.2 Communication Workflow:

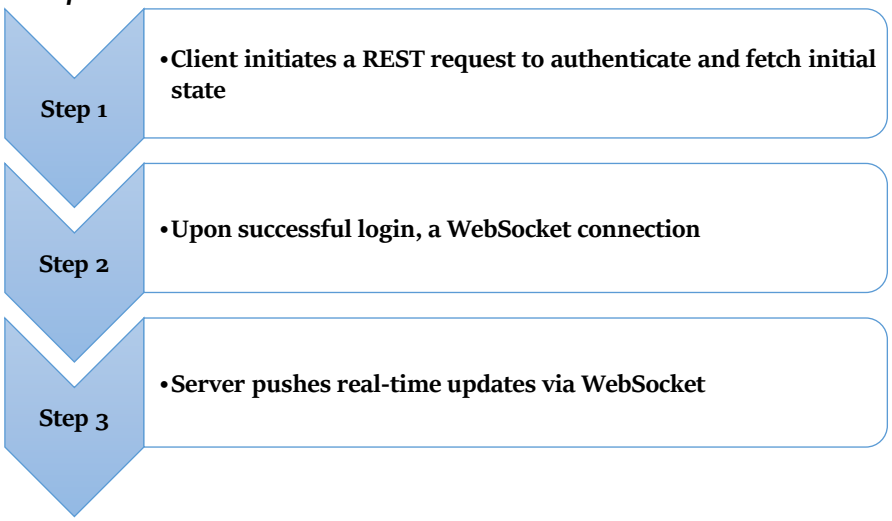

Figure 3: Communication Workflow

- **Step 1: The client initiates a REST request to authenticate and fetch the initial state:** The client initiates communication with the client by making a RESTful request to the server. This is where the client will log in and possibly obtain the first state of the application. This may include user account details, data, user preferences, and settings or other data with which the application has recently interacted. REST is perfect for this step since it does not keep any session information and returns structured responses in a predefined format. This makes the user validation process secure, and the beginning of the session is set out well and properly.

- **Step 2: Upon successful login, a WebSocket connection:** Then, if the client is successfully authenticated and receives some basic data at the start, the client creates the WebSocket to the server. It supports continuous and bidirectional access to data, as is the case with the full-duplex data communication channel. Opening the WebSocket connection at this stage ensures that certain sections of live information can only be accessed by authorized personnel while at the same time ensuring that the system prepares for constant data feed, which would be efficient.
- **Step 3: Server pushes real-time updates via WebSocket:** With the WebSocket connection active, the server can now send data other than a welcome message to the client. These include chat messages, notifications, presence updates, feeds, and other real-time information. Unlike other protocols, such as REST, which requires the client to check for data availability, WebSockets enables the server to alert the client when data is available, enhancing performance and relieving some network burden.
- **Step 4: REST is used for further non-real-time interactions:** The system utilizes the existing RESTful endpoints for operations that do not require real-time response rates or frequent rates, such as storing the user settings or preferences, providing the forms, or requesting multiple pages of historical data. This way of segregating the concerns ensures the WebSocket channel can only be used for real-time flow and the REST for transactional or less frequent requests and responses.

3.3 Mathematical Model

It's about studying and comparing the efficiency of the hybrid communication model [13-17] systems that simple mathematical models can present.

$$R(t) = R\,rest(t) + R\,ws\,(t)$$

Let R(t) be the total count of clients' requests or the number of interactions at a certain time t. It remains to be noted that there are two major contributions to this total: (t) is the random number of REST-based requests, and (t) stands for the number of WebSocket events or messages. Hence, it is possible to define the total request load at any moment as follows: This equation enables us to monitor how the system load is split in the two communications over some period of time. Normally, in the first stages of the session, I/ O, R rest (t) is high because most of the time is spent on authentication and data retrieval. Such an engagement translates into the fact that as the session progresses and transitions into a real-time and multimodal interface like messaging and notifications, R ws (t) tends to dominate. To measure the reactivity, we use the clients' latency L(t) when they get their response or update at time t. Since it's based on HTTP and Stateless, REST-based operations are slightly slower because of the required overheads and frequent polling, particularly under load. WebSocket connections are persistent and are much faster for transmitting real-time event parameters. An ideal system for web applications will ensure that L(t) is minimized most of the time while R(t) is fairly split between REST and WebSocket protocols. This can be done statically with simple load distribution algorithms or dynamically through spectacular flow rerouting gateways that change the channels depending on certain contexts observed behavior or the state of the running system. This way, these futures can be quantified mathematically and help to create models that will allow for the simulation of the performance and consequent bottlenecks, as well as factors that will, therefore, affect scalability and the user experience in the best possible way.

3.4 Security Mechanisms

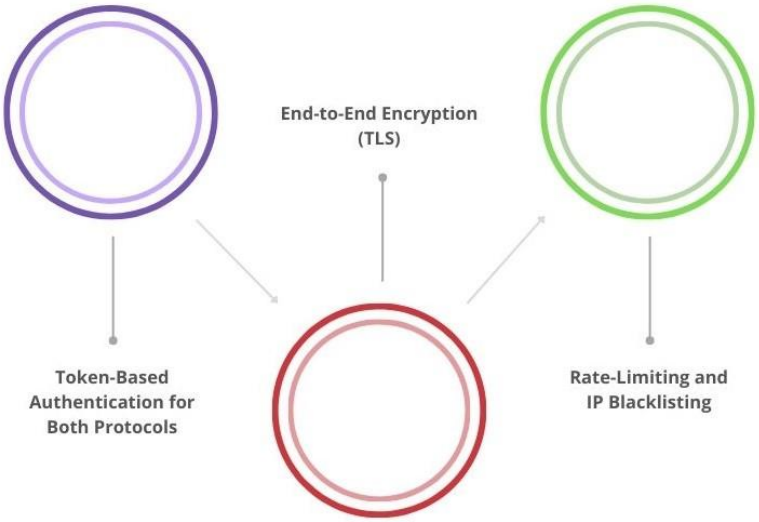

Figure 4: Security Mechanisms

- **Token-Based Authentication for Both Protocols:** Furthermore, token-based authentication in the hybrid model serves as a reliable solution that identifies the current clients who choose either REST or WebSocket. When a client enters the application, a token – usually in a JSON Web Token format – is generated and sent with each REST or WebSocket connection request. It makes the user sessions very scalable yet offers reduced vulnerability to hijacked user sessions. The token preserves and encodes the state of the user's access privileges to authorize every single REST operation and any subsequent continuous WebSocket connection.
- **End-to-end encryption (TLS):** It is crucial to ensure that the data being exchanged between the client and server sides is secure. TLS offers adequate protection for both the RESTful and WebSocket profiles, providing end-to-end encryption. TLS ensures data confidentiality when in transit by preventing interception and modification of sensitive data, including Tokens, personal user information, etc. This encryption mechanism is important in most hybrid architectures where two different types of protocols are in use, as it makes the security of both streams similar. Also, there is an opportunity to use certificate-based authentication to provide an additional identity check for both the servers and the clients, providing privacy during communication.
- **Rate-Limiting and IP Blacklisting:** However, since DoS can be a common abuse vector, the system uses the IP filter and rate limiting. It limits the number of requests a client can make within a given period to ensure that both REST endpoints and WebSocket connections are not congested. This is crucial to ensure the system remains agile and constant, especially when many people demand it at the same time. Also, IP blacklisting is used for safety in terms of blocking the traffic originating from certain sources whose behavior looks suspicious or which has been marked as dangerous. Collectively, these ensure the availability of the service and secure the backend from the brunt of attempts as well as other cyber vices.

4.Results and Discussion

4.1 Experimental Setup

To systematically evaluate the proposed hybrid communication architecture, the controlled experiment was set up with the help of software and hardware that meets industry standards. The fundamental part of the work was implemented using Node.js – a rapidly growing cross-stage, lightweight runtime environment with high concurrency ability. An establish a constant connection, WebSocket protocol was contributed through Socket.IO to foster a two-way, real-time connection with low response time between the server and the clients. The application built employing Node.js and Express.js provides RESTful API that allows performing typical operations such as authentication, data access, and configuration change within a request-response framework. The server-side application was installed on a mid-tier virtual private server, having features from 4 vCPUs capacity, 8 GB of RAM, and a reliable network connection, which are essential to mimic a real-world setting for the application testing. This setup confirmed that the server resource was robust enough to handle REST and WebSocket requests with and without imposed artificial limitations.

Therefore, Apache JMeter was adopted as the primary load-testing tool to emulate client activity and place pressure on the system. JMeter was used to create the concurrent users interacting with the environment, which is a natural usage pattern such as login, chat service, and background refresh. Thus, there are three testing scenarios: without using WebSocket only, without using REST only and using both WebSocket and REST. In particular, throughout each test run, average latency, throughputs in terms of requests per second, CPU load, and memory usage were logged in real time. These metrics were chosen to have as much perspective as possible to observe how the different communication models behave under stress. WHICH RESPONSE TIME-RESOURCE TRADEOFF COSTS WERE MADE. The objective of this configuration was to show that effective work on the model for an increase in load volume would be possible to carry out without overburdening the equipment resources.

4.2 Test Cases

Table 1: Test Cases (Percentage-Based Load Distribution)

Description	REST Usage	WebSocket Usage
REST-only API under simulated load	100%	0%
WebSocket-only real-time event streaming	0%	100%
Hybrid approach using REST + WebSocket	40%	60%

- **REST-only API under Simulated Load (100% REST, 0% WebSocket):** As in the previous sections, all communication with the client is conducted only through RESTful APIs in this test case. This is the typical case of the interactions in traditional web applications where each client process step is hyper-tuned to a new HTTP request/response cycle. Such actions, including data access, user logins, and modifying the system's data, are processed in a stateless style. This is to determine how well the system handles a large number of requests for data using REST protocol with regard to response time, speed, and the demands placed on the servers. Implementing REST will allow connections to be closed after each request, meaning this model could have issues when polling or have a high concurrency level.

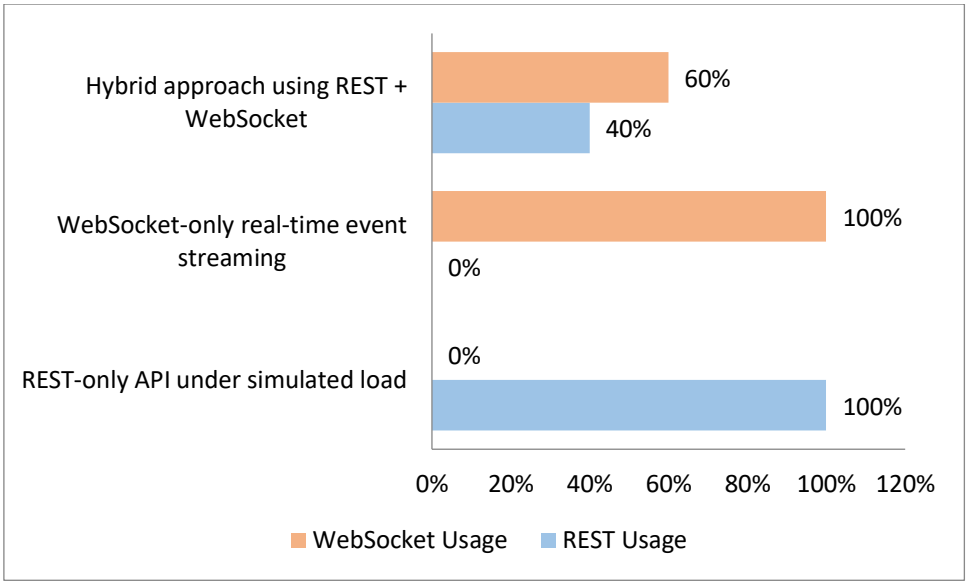

Figure 5: Graph representing Test Cases (Percentage-Based Load Distribution)

- **WebSocket-only Real-Time Event Streaming (0% REST, 100% WebSocket):** This case concerns a synchronous communication process using WebSocket connections that provide such data exchange. It's connected constantly with clients, thus using the latter's persistent, bi-directional connection to send messages to the server without requiring repeated HTTP requests. This model is well-suited for applications that work with live updates, such as chat platforms, teamwork tools, or real-time panels. From this aspect, WebSocket can waste less time and more flows per second compared to HTTP. However, in this case, also important things to bear in mind. Flickr pat also has disadvantages, such as high memory and the complexity of handling many long-lived connections.
- **Hybrid Approach Using REST + WebSocket (40% REST, 60% WebSocket):** It is, therefore, necessary to adapt the type of communication undertaken by the application to the current operation using the hybrid of REST and WebSocket. According to this test, REST still serves 40% of the traffic, including data loading, user authentication, and settings. WebSocket is employed for the remaining 60% of communication that is time-iterative or requires actual-time figures such as notifications or updates. This model should incorporate the elements of simplicity and stability usually related to REST with the feature of reactivity related With WebSockets.

4.3 Performance Metrics

The following table summarizes the results from all three test cases:

Table 2: Performance Metrics

Metric	REST	WebSocket	Hybrid
Avg Latency (ms)	320	50	85
Throughput (req/s)	600	1000	900
Memory Usage (MB)	220	450	310

- **Average Latency (ms):** Latency refers to the delay between a client's request and the system's response. This was due to the fact that there were many HTTP requests and responses sent back and forth, and there was no reuse of connections as in the other cases. Real-time communication implemented by WebSocket had the least latency, which, on average, was 50 milliseconds; messages are delivered over continuous tunnels. The integration of REST and WebSocket also proved useful as the setup was more reliable while running the live data in WebSocket, ensuring that the interactions have a low latency of about 85ms on average.
- **Throughput (requests per second):** Throughput is the ability to process the number of customers served in a one-second interval. The WebSocket-only milliseconds topped the other possibilities with a frequency of 1000 requests per second because of its simple three-part protocol and the absence of extra connection-establishment time. REST, however, while being more stable, indicated that it only allowed for 600 requests per second, demonstrating its limitation in servicing concurrent requests, especially when there was frequent polling. The lift of the hybrid model reached 900 requests per second by connecting requests with WebSocket. This outcome implies hybrid communication can manage highly fluid loads since it directs different traffic types to the most appropriate protocol.

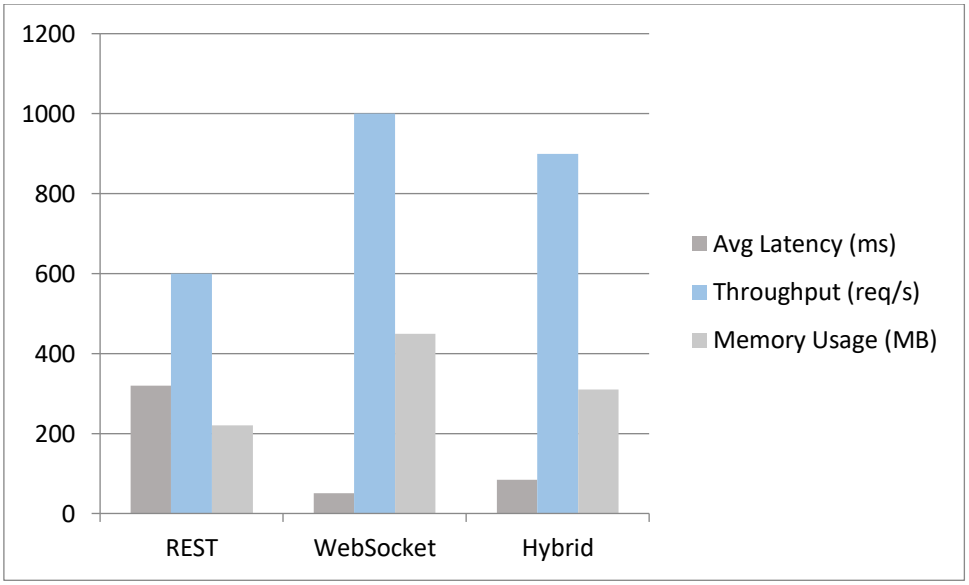

Figure 6: Graph representing Performance Metrics

- **Memory Usage (MB):** Regarding a concurrent system, the use of memory is one of the most essential factors to be considered. Here, the WebSocket-only model boasted the highest memory use of 450 MB, but this was expected due to the need to have a connection open and active for each client in a specific chat room, which has an additional state on the server. REST uses the least memory at 220MB because it is stateless; therefore, it frees up available resources after every call. The hybrid model involved a basic 310 MB and was important as it allowed a balance between the usage of resources and the functions offered. This moderate usage is because WebSocket is dynamically used to request memory for the connection while the rest of the application is kept as fast and simple as possible by using REST.

4.4 Discussion

The experimental findings testify that the hybrid communication model is workable and optimizes both responsiveness and scalability as well as a communication resource. Thus, the given system is proven effective in relying on REST and WebSocket to meet different operation demands. REST is best suited for request-response applications where there is not much complexity involved, and the action is not time-sensitive, particularly for operations such as authentication, configuration modifications, and initial data provision. These capabilities take advantage of the REST approach, which is stateless and compatible with HTTP caching, thereby greatly minimizing memory consumption on the server side. In turn, Websockets are an excellent choice for real-time communications such as chat, notifications, collaboration, etc. Due to their highly invariant connectivity with low latency, there can be fast data delivery, making interactions in fluctuating applications appealing to users. The choice is also well thought out: REST is resorted to when structure and reliability are required; WebSockets apply for continuous data streams.

This division minimizes endpoint load based on REST architecture and uses WebSocket to perform high-frequency, event-triggered tasks. Therefore, the system can achieve higher concurrency, and there is no overwhelming occurrence of polling in REST-only models. However, using a hybrid architecture is an added advantage; there is added intricacy regarding design and implementation. Certain issues are important when running two communication paradigms, including routing, authentication, and errors. These can be mitigated effectively through architectural patterns such as the API Gateway pattern, which deals with traffic management and protocol conversion. For WebSocket integration, there are tools such as Socket.IO; for hybrid API management, tools like Kong API Gateway offer quite a few features. Other frameworks like NestJS make development even more advanced by providing modularity, scalability, and code maintainability for REST and WebSocket services. However, in the case of proper design, a hybrid model also brings higher performance and creates a more reliable solution for modern web applications that may be easily adapted to future changes.

5. Conclusion

Thus, this research has indicated that integrating REST and WebSocket is an effective solution for web applications with RT to interact with clients and excellent data storage. For standard operations like authentication, the performance is optimized using REST's topology, which lacks data persistence, while high-realtime data is supported using WebSocket protocol with real-time persistence. This conclusion can be justified based on the empirical results derived from the above experimental setup that shows how the hybrid model performs better in terms of parameters such as average latency, throughput, and memory consumption compared to the system implemented solely based on REST and WebSocket.

WebSocket's fully connected and constant connection significantly enhances efficient response times for live update messages. At the same time, REST's normal connection and widespread usability are optimal for undertaking interactions that are not continuously changing but rather recurrent and definite. Still, several possibilities can improve this hybrid model in the future. For this case, incorporating GraphQL in the system can eliminate challenges relating to over-fetching and under-fetching data when using APIs, especially REST. GraphQL could be integrated with WebSocket subscriptions to add a more powerful real-time communication layer. Concerning the application's deployment and possible future growth, managing microservices with Docker and using Kubernetes to orchestrate the containerized application would provide automatic scaling capabilities, enhanced dependability, and straightforward application management.

Moreover, manual load balancing, particularly WebSocket-aware proxies like NGINX sticky state or Envoy, can also contribute to distributing the traffic more effectively, avoiding instability and reducing the effect of concurrency on the system. Finally, as real-time interaction becomes more or less a norm in applications such as messenger, teamworking platforms, IoT devices, and financial systems, WebSocket-REST hybrid will be a standard design pattern. It fulfills the required need for high reactivity and retains all the potential for structure and development that are components of the REST style. The architecture added some complexity, but this can be overcome with the help of a modern programming framework, API gateway, and established design patterns. Therefore, the hybrid future model is a perfect basis for developers to create flexible, scalable, maintainable, and adaptive applications required in the constantly growing, interconnected world.

References

[1] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures. University of California, Irvine.

[2] Pautasso, C., Zimmermann, O., & Leymann, F. (2008, April). Restful web services vs." big'" web services: making the right architectural decision in Proceedings of the 17th international conference on World Wide Web (pp. 805-814).

[3] Fette, I., & Melnikov, A. (2011). Rfc 6455: The WebSocket protocol.

[4] Lubbers, P., Albers, B., Salim, F., & Pye, T. (2011). Pro HTML5 programming (pp. 107-133). New York, NY, USA:: Apress.

[5] Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A., & Buyya, R. (2011). CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and Experience, 41(1), 23-50.

[6] Halder, D., Kumar, P., Bhushan, S., & Baswade, A. M. (2021, July). FybrrStream: A WebRTC-based, efficient, and scalable P2P live streaming platform. In 2021 International Conference on Computer Communications and Networks (ICCCN) (pp. 1-9). IEEE.

[7] Zhao, S. M., Xia, X. L., & Le, J. J. (2013). A real-time web application solution based on Node. Js and WebSocket. Advanced Materials Research, 816, 1111-1115.

[8] Liu, Q., Yang, G., Zhao, R., & Xia, Y. (2018, October). Design and implementation of a real-time monitoring system for wireless coverage data based on a web socket. In 2018 IEEE 3rd international conference on cloud computing and Internet of Things (CCIOT) (pp. 63-67). IEEE.

[9] Liu, Q., & Sun, X. (2012). Research of web real-time communication based on web socket. International Journal of Communications, Network and System Sciences, 5(12), 797-801.

[10] Rasool, S., & Mukhtar, H. (2013, December). Adaptive traffic switching between WebRTC and WebSocket based on the battery status of portable devices. In 2013 Taibah University International Conference on Advances in Information Technology for the Holy Quran and Its Sciences (pp. 345-351). IEEE.

[11] Costa, B., Pires, P. F., Delicato, F. C., & Merson, P. (2014, April). Evaluating a Representational State Transfer (REST) architecture: What is the impact of REST on my architecture? In 2014 IEEE/IFIP Conference on Software Architecture (pp. 105-114). IEEE.

[12] Zou, J., Mei, J., & Wang, Y. (2010, July). From representational state transfer to accountable state transfer architecture. In 2010 IEEE International Conference on Web Services (pp. 299-306). IEEE.

[13] Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J. C., Canali, L., & Percannella, G. (2016). REST APIs: A large-scale analysis of compliance with principles and best practices. In Web Engineering: 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings 16 (pp. 21-39). Springer International Publishing.

[14] De Backere, F., Hanssens, B., Heynssens, R., Houthooft, R., Zuliani, A., Verstichel, S., ... & De Turck, F. (2014, May). Design of a security mechanism for RESTful Web Service communication through mobile clients. In 2014 IEEE Network Operations and Management Symposium (NOMS) (pp. 1-6). IEEE.

[15] Masse, M. (2011). REST API design rulebook: designing consistent RESTful web service interfaces. " O'Reilly Media, Inc.".

[16] Biehl, M. (2016). RESTful API design (Vol. 3). API-University Press.

[17] Subramanian, H., & Raj, P. (2019). Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs. Packt Publishing Ltd.

[18] Skvorc, D., Horvat, M., & Srbljic, S. (2014, May). Performance evaluation of Websocket protocol for implementation of full-duplex web streams. In 2014, the 37th International Convention on information and communication technology, electronics, and microelectronics (MIPRO) (pp. 1003-1008). IEEE.

[19] Imre, G., Mezei, G., & Sarosi, R. (2016, June). Introduction to a WebSocket benchmarking infrastructure. In 2016, Zooming Innovation in Consumer Electronics International Conference (ZINC) (pp. 84-87). IEEE.

[20] Ogundeyi, K. E., & Yinka-Banjo, C. (2019). WebSocket in real-time application. Nigerian Journal of Technology, 38(4), 1010-1020.