Real-time messaging in Enterprise context

This paper serves to provide substantiation on how to make near real-time messaging possible within the context of an enterprise application.

Introduction

Effective communication stands as a cornerstone of enterprise success. The advent of real-time messaging has revolutionized the way organizations interact both internally and with their clients. However, implementing a robust and scalable real-time messaging system poses significant challenges, particularly for large enterprises with potentially millions of users. This paper aims to explore and provide solutions for implementing near real-time messaging within such a vast enterprise environment.

The relevance of this study is underscored by the evolving demands of the modern business landscape, where speed and efficiency in communication are not just advantages but necessities. As enterprises continue to expand, the capability to manage high-volume, instantaneous communication becomes critical. However, achieving this, while maintaining system performance, data integrity, and security, presents a complex array of challenges.

Research question

In order to solve the problem stated above, the following research question was formulated: How can ICAP achieve near real-time messaging when it has millions of users in the future? The following sub-questions were formulated to help answer the research question:

1. How do equivalent platforms achieve near real-time messaging?
2. What kind of additional technologies are needed to achieve near real-time messaging?
3. How can system scalability and performance be maintained with the increase in user load?

Methods

In order to answer the above-mentioned research question, the following methods were used:

- Literature Study: Review existing academic and industry literature on real-time messaging systems.
- Available Product Analysis: Analyze current products in the market that successfully implement near real-time messaging at scale.
- Design Pattern Analysis: Examine and evaluate various design patterns and architectures used in scalable messaging systems.
- Prototyping: Develop a basic prototype or model to test theories and solutions derived from the research.

Results

In the next sections each sub-question will be answered in a section dedicated to that question.

How do equivalent platforms achieve near real-time messaging?

A couple of social/messaging platforms come to mind that facilitate real-time messaging or have real-time messaging as a core functionality of their platform. Below here, a few of these platforms will be chosen and analyzed on how they achieve real-time messaging.

WhatsApp

WhatsApp has been right at the forefront of innovating and facilitating real-time messaging for billions of users from all over the world [1]. They manage to achieve such an impressive feat by using a number of solutions that are made to fit WhatsApp's specific needs for processing all data related to messaging.

CometChat has done an analysis out of which technologies the entire WhatsApp ecosystem is made [2]. The back-end of WhatsApp specifically is very interesting to look at, as this is where the bulk of the heavy lifting for the messaging system is done. A few key highlights of the platform's system design are:

1. **Erlang for Backend Processing**: Erlang, a functional programming language, is pivotal in WhatsApp's architecture. Its capability to handle high concurrency and support distributed, fault-tolerant, soft real-time systems is essential for managing the vast volume of simultaneous messages.

2. **Ejabberd for Message Routing**: WhatsApp utilizes Ejabberd, an open-source XMPP server written in Erlang, for message routing and deliverability. It's specifically tailored for instant messaging, handling tasks like one-on-one messaging, group chat, and contact list management.

3. **Mnesia Database for Data Storage**: For data storage, WhatsApp employs Mnesia, an Erlang-based distributed database management system. It excels in real-time key/value lookups and supports high fault tolerance, crucial for a platform with billions of messages exchanged daily.

The combination of these technologies, along with a few other technology and infrastructure choices (like using SQLite to store messages locally on the user's device), make it possible for WhatsApp to maintain a high level of performance and reliability, which is crucial for delivering messages globally with minimal latency. The platform's remarkable performance underscores the successful decision to base its backend technologies on Erlang, demonstrating how this strategic choice effectively bolstered its overall functionality and scalability.

Telegram

Telegram achieves real-time messaging through a mix of cloud-based storage, open-source technology, and an efficient user interface design. Key aspects include:

1. **MTProto Protocol**: Telegram uses its custom MTProto protocol for encryption and security, ensuring fast and secure message delivery.

2. **Cloud-Based Storage**: The app's cloud-based storage system plays a crucial role in real-time messaging. It stores user data on servers in encrypted form, allowing synchronization across devices for immediate data access.

3. **Distributed File System**: Telegram utilizes a distributed file system, the Telegram File System (TFS), for efficient and reliable storage and retrieval of user data.

4. **Data Centers in Multiple Geographic Regions**: The placement of servers in various regions helps reduce latency and enhances the speed of message delivery.

The exact details of Telegrams technology stack are not readily available for the public, but the described technologies, as retrieved from intuji [3], already cover a fair amount of the stack. The technologies work together to ensure that Telegram can offer secure, efficient, and near real-time messaging to its users.

Discord

Discord is one of the fastest growing messaging and social media platforms. By 2023, Discord's user count had grown to 560 million, a significant increase from 300 million in 2020 and just 45 million in 2017 [4]. In Discord's backend technology stack, several key components are specifically relevant to enabling its real-time messaging capabilities:

1. **Elixir and Erlang**: Discord uses Elixir, a functional programming language that runs on the Erlang virtual machine (BEAM). Erlang and Elixir are renowned for their performance in building scalable, distributed, and fault-tolerant applications. This is particularly important for real-time messaging applications that need to handle numerous concurrent connections and messages.

2. **Phoenix Web Framework**: Built with Elixir, the Phoenix framework is utilized by Discord for web development. Phoenix is designed for high-availability applications and includes features that make it ideal for real-time messaging, such as channels and presence features.

3. **WebSockets**: Discord leverages WebSockets, which provide a full-duplex communication channel over a single TCP connection. This technology is crucial for real-time interactions, allowing the server and client to exchange messages quickly and efficiently.

4. **PostgreSQL Database**: As the main database, PostgreSQL, a powerful open-source RDBMS, is used. While not directly related to the real-time aspect, it plays a vital role in storing and managing user data efficiently, which is essential for maintaining the overall performance of the messaging service.

5. **Rust for Performance Optimization**: Discord has been transitioning some of its services from Go to Rust, especially for parts of the system where garbage collection latency spikes were a problem in Go. Rust’s efficient memory management and lack of garbage collection directly contribute to more consistent performance, crucial for real-time messaging systems.

6. **Microservices Architecture**: Discord’s backend is structured as a microservices architecture, which allows different functionalities to be managed by separate, smaller services. This structure is key for scalable real-time applications, as it enables easier management and scaling of specific components.

These technologies collectively form the backbone of Discord's ability to provide a seamless real-time messaging experience. Elixir and Erlang’s concurrency and fault tolerance, combined with the real-time capabilities of WebSockets and the Phoenix framework, and the performance optimizations from Rust, all contribute to Discord's robust real-time messaging system.

How can system scalability and performance be maintained with the increase in user load?

Maintaining system scalability and performance with an increase in user load is a critical aspect of real-time messaging systems, especially in an enterprise context where the user base can grow rapidly. To address this challenge, a combination of strategies and technologies and design paradigms should be implemented to achieve the scalability and necessary. The following section highlights some of the design choices that are important for ICAP.

- **Adopt Microservices Architecture**: Adopting a microservices architecture will enhance scalability and maintainability if done properly. In the case of microservices, the application is divided into smaller, independent services that communicate over a network. This allows for scaling individual components as needed without affecting the entire system, making it easier to manage increased loads [5] [6].
- **Load Balancing and Elastic Scalability**: Using microservices on their own won't make the application able to scale with an ever increasing amount of users. Load balancing and elastic scalability are needed to achieve the goal. Load balancing distributes workloads across multiple computational nodes, ensuring that services do not get overloaded and are able to keep up with demand. Elastic scalability, which involves dynamically adding or removing computational nodes based on demand, will make sure that the appropriate amount of computational resources are dedicated to handle the load created by the end users [7].
- **Asynchronous Processing and Queue Systems**: Using asynchronous processing and message queues (like RabbitMQ or Kafka) would decouple the processes that produce and consume data, allowing for more efficient handling of requests and tasks [8] [9].

In conclusion, ensuring scalability and performance in real-time messaging for an enterprise application like ICAP requires a strategic blend of microservices architecture, load balancing with elastic scalability, and asynchronous processing with queue systems. Adopting a microservices based architecture provides the flexibility needed for scaling individual components in response to user load. Load balancing and elastic scalability ensure efficient distribution and dynamic adjustment of resources, while asynchronous processing and message queues like RabbitMQ or Kafka optimize the handling of data flow. These integrated approaches are key to maintaining robust, scalable, and efficient real-time messaging capabilities in a rapidly evolving enterprise environment.

What kind of additional technologies are needed to achieve near real-time messaging?

Given ICAP's developer's choice of C# as the primary programming language, it's crucial to integrate technologies and frameworks that are compatible and can leverage the strengths of C#. The goal is to achieve near real-time messaging while ensuring scalability, performance, and reliability. The following technologies and approaches are recommended:

1. **Azure Service Bus for Messaging and Queue Management**: Azure Service Bus, a fully managed enterprise integration message broker, can be particularly useful. It can handle messaging patterns such as queues, topics, and subscriptions, and is well-suited for complex messaging scenarios. For C# applications, integration with Azure Service Bus is straightforward and it offers reliable message delivery [10].

2. **Kubernetes for Container Orchestration**: To ensure scalability and efficient deployment, Kubernetes can be used for container orchestration. It facilitates automatic scaling, load balancing, and self-healing. Kubernetes can be integrated with Azure, which is beneficial if Azure Service Bus is used [11].

3. **WebSocket Protocol**: For real-time bidirectional communication between the client and server, WebSockets should be utilized. This protocol is supported in .NET and provides a persistent connection between client and server, allowing for immediate data transfer.

In conclusion, a combination of these technologies, aligned with C#'s capabilities and the enterprise’s infrastructure, can help ICAP achieve its goal of implementing near real-time messaging. It's essential to ensure that these technologies are integrated in a way that complements each other to create a robust, scalable, and efficient real-time messaging system.

Conclusion

For ICAP to achieve and maintain the ability to facilitate real-time messaging for possibly millions of users it will be important to use the previously found and mentioned system design paradigms and technologies to achieve the goal. This combination of paradigms and choices will be further expanded on and tested throughout the development and integration of real-time messaging for ICAP.

Future work

Moving forward, the progression of ICAP and this research has several avenues for future research. It is very important to keep exploring and developing the various features that have not yet been finished for ICAP.

Sources

1. WhatsApp Tech Stack Explored — The tech behind series. (n.d.). Intuji.
2. Understanding WhatsApp's architecture & system design. (n.d.). cometchat.
3. How does Telegram work? A look into the Telegram tech stack. (n.d.). Intuji.
4. Sukhanova, K. (2023, July 24). The latest Discord Statistics & trends for 2023. The Tech Report.
5. What are microservices? (n.d.). microservices.io.
6. Atlassian. (n.d.). Microservices Architecture
7. The Couchbase Blog. (2023, October 27). Your guide to scaling microservices
8. Tyagi, A. (2022, January 1). Introduction to asynchronous processing and message queues. DEV Community.
9. Banerjee, D., & Banerjee, D. (2023, September 13). Introduction to message queues. Baeldung,
10. Spelluru. (2023, April 3). Introduction to Azure Service Bus, an enterprise message broker - Azure Service Bus. Microsoft Learn.
11. Kubernetes overview (n.d.)
12. The websocket protocol. (n.d.). WebSocket.org.