A deep dive into asynchronous communication patterns in Cloud-based distributed systems

16 February 2024 - 8 min. read

Decoupled Architectures | Microservices

Introduction

Asynchrony is a widely used and complex concept in software development. It can be present in many contexts, from frontend applications to backend systems and even between services in distributed systems. Based on the context, the asynchronism mechanism, features, and impact on user experience greatly vary, and each has unique pros and cons.

For example, in a frontend application, implementing asynchronism ensures that users can immediately receive feedback on their actions while further processing occurs on the backend side, enhancing system efficiency and user experience.

The backend usually implements asynchronism to optimize resource utilization and facilitate concurrent request handling. It enables parallel processing of computationally intensive tasks and efficient input/output operations management.

Speaking about distributed systems, leverage asynchrony raises various challenges, including the unreliable nature of the network, the selection of appropriate messaging protocols and serialization formats, and the implementation of robust error handling and retry mechanisms. It requires careful design consideration; however, its scalability, resilience, and fault tolerance benefits outweigh the complexities involved.

This article explores various aspects of implementing asynchronous communication within distributed systems. We will also cover some nuances of asynchrony in both frontend and

backend contexts. By comprehensively examining these aspects, we aim to shed light on the complexities, best practices, and potential pitfalls inherent in distributed asynchrony.

Asynchrony in front-end development

Let's focus on web application development.

To build web applications with a good user experience, it is crucial to implement **asynchronous API calls and response handling**.

By default, modern web development frameworks and libraries offer robust mechanisms for asynchronous communication with server-side APIs. Through techniques such as AJAX and asynchronous JavaScript features like promises and async/await, frontend applications can initiate requests to backend services without blocking the browser's main thread.

With this approach, you can build a responsive user interface, creating a "native" and natural experience for the end user by reducing page reload and the overall time the user must wait for the output.

The user experience is usually way better, and the overall bandwidth consumption is generally reduced, lowering cost and infrastructure burden.

Additionally, asynchronous patterns empower front-end developers to implement features such as real-time updates and live data synchronization, further enriching the user experience. For example, leveraging WebSocket communication or long-polling, web applications can receive instantaneous updates from the backend, significantly increasing the range of features that can be implemented.

The main asynchronous programming techniques for front-end development are Callbacks, promises, and Async/await.

Callbacks are functions passed as arguments to other functions and executed later, usually when an asynchronous operation completes.

While this mechanism constituted the fundamental asynchronous technique in JavaScript and was formerly the only one available, it can lead to callback hell and make code difficult to read and maintain when nested deeply.

Promises represent an asynchronous operation's eventual completion or failure and allow the chaining of multiple asynchronous actions. They are widely used in modern JavaScript and provide better readability and error handling than callbacks, allowing you to chain operations using .then() and .catch() methods.

Async functions enable writing asynchronous code using a synchronous-like syntax by allowing the await keyword to pause execution until a promise is resolved or rejected. Async/await is especially beneficial when dealing with multiple asynchronous operations that depend on each other.

Asynchronism in the back-end

Adopting asynchronous and parallel components in back-end development has become increasingly valuable and has been pushed by the need for scalable and efficient systems. Back-end services can unlock many benefits by leveraging parallelism and async tasks; however, unique challenges are hidden in asynchronous API construction.

One of the main advantages of parallelism is the **reduction of response time**. For example, splitting a compute-intensive task in parallel threads can reduce the time needed for the response while simultaneously optimizing CPU utilization. Writing parallel code can also optimize I/O operations, using buffered streams and limiting the time the software is blocked for I/O.

Most application servers or web servers used as interfaces for the back-end system can use multi-threading or multitasking to scale more effectively and accommodate growing user traffic. This scalability is essential to exploit all the horsepower available in the computational node.

But there are also a lot of complexities and disadvantages.

Implementing asynchronous and parallel components introduces additional complexity, requiring careful design and accurate error handling. Addressing concurrency, synchronization, and data consistency becomes more challenging, potentially leading to subtle bugs.

Another aspect to consider is that the debugging of heavily parallelized code is usually complex and more error-prone. Race conditions, deadlocks, and timing-related issues may arise, necessitating comprehensive testing strategies and debugging tools.

The most common technique is leveraging asynchronous programming paradigms and libraries to manage non-blocking operations efficiently, such as callbacks, promises, or async/await in Node.js or Python with asyncio.

Another common way is to directly leverage threading and multitasking in languages that allow it, for example, Java, Rust, or Python, to execute multiple tasks concurrently. This involves spawning threads or processes to handle parallelizable tasks, thus leveraging multi-

core processors effectively. There are libraries, both language-provided and by thirty parties, to simplify low-level task handling, leveraging concepts as thread pools and implementing synchronization mechanisms like locks and semaphores to manage shared resources safely in a multi-threaded or multitasking environment.

For specific tasks, you can also leverage products employing distributed computing frameworks such as Apache Kafka, Apache Spark, or Apache Flink to process large volumes of data across distributed clusters.

Distributed systems

In distributed systems architecture, asynchronous communication patterns are key to achieving modularity, scalability, and resilience.

At the infrastructural level, asynchrony is introduced by decoupling services and enabling non-blocking communication between them; the whole solution gains robustness and usually allows for graceful degradation instead of service unavailability.

One of the primary motivations behind building complex systems using loosely coupled services lies in managing complexity. By breaking down monolithic applications into smaller components, we can streamline development, facilitate independent deployment, and promote agility in response to evolving requirements. Moreover, decoupling services can reduce the risk of compromising the integrity of the overall system.

It is also worth noticing that some features are naturally better developed using decoupling or asynchronous services, so a complex application may be composed of some decoupled services for asynchronous operation.

At the heart of decoupling lies the concept of asynchronous communication, which serves as the linchpin for orchestrating interactions between distributed services.

Many different asynchronous patterns exist, such as message queues, publish-subscribe systems, and event-driven architectures. Those provide the foundation for building resilient, loosely coupled systems.

Message queues, for example, allow services to communicate asynchronously by decoupling the production and consumption of messages. This enables services to operate independently, processing messages at their own pace without being tightly bound to the availability or responsiveness of other components.

Similarly, event buses facilitate loosely coupled communication by enabling services to publish and subscribe to events of interest. This event-driven paradigm promotes flexibility and extensibility, allowing services to react to system state changes or external events.

Many benefits are linked to decoupling services and can be achieved by building asynchronous distributed systems. The most important one is that asynchronous communication **isolates and mitigates cascading failures** so that a failure in one service does not necessarily propagate to others, minimizing the impact on the overall system's availability.

Usually, if a subset of microservices is not healthy, only a few functions of the whole system will malfunction, and if the application is designed for graceful degradation, there is a chance that the user never has to do anything to cope with the issue. For example, when using a queue to decouple a cart from the order service, a temporary problem in the latter is automatically managed, and the end user will experience some delay before receiving the confirmation for the order. In the meantime, he can still see his order in the queue. It is correctly taken into account but still needs to be confirmed.

Another significant benefit is that decoupling and asynchrony allow services to scale independently, ensuring optimal performance under variable traffic conditions.

All this leads to a better user experience. Users benefit from faster response times, real-time updates, and smoother interactions.

Of course, user interaction must be considered and carefully designed to ensure a seamless and enjoyable experience, mainly because requests are processed asynchronously, and their effects become visible later. Crafting a user experience that is both pleasant and non-frustrating requires thoughtful design choices and a deep understanding of user expectations and behaviors.

One effective strategy is to provide progressive feedback and support transparency about the status of ongoing operations. Instead of leaving users in the dark while their requests are being processed, applications should display clear and timely feedback, indicating that the system has acknowledged their input and is actively working on their behalf.

This can be achieved through visual indicators, such as loading spinners, progress bars, or status messages, informing users about the progress of their requests, and collecting statuses in an easy-to-read report so that the user is never stuck to an infinite spinner, and can safely proceed his navigation when his request has been correctly submitted and is being processed. Of course, it is not always possible to detach the response and free the

user, but when it is, it is way better to acknowledge the request and give the user a convenient way to check the status without being forced to wait.

Effective error handling is another critical aspect of designing user-friendly experiences in asynchronous systems. Since asynchronous operations may encounter errors or failures during execution, it's essential to anticipate potential issues and provide clear, actionable feedback to users when problems occur.

Rather than presenting users with generic error messages or technical jargon, applications should strive to communicate errors in a user-centric language that conveys the nature of the problem and suggests actionable steps for resolution. Additionally, providing contextual guidance or links to relevant help resources can empower users to troubleshoot issues independently and reduce frustration.

Conclusion

In this article, we dived into the field of asynchrony and decoupling, and we discussed the topic at different application layers and from the user perspective.

What are your considerations about the topic? Let us know in the comments.

For more details on the techniques mentioned in this essay, check these resources:

- A Deep Dive into Decoupled Cloud Architectures with Event Buses
- 3 ways to decouple your microservices: SQS queues, ELB load balancing, and SNS notification system
- Decoupling services using SQS as a Lambda Trigger

Stay tuned for other articles about complex systems design!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Alessio Gandini

Cloud-native Development Line Manager @ beSharp, DevOps Engineer and AWS expert.Since I was still in the Alfa version, I’m a computer geek, a computer science-addicted, and passionate about electronics all-around.At this moment, I’m hanging out in the IoT world, also exploring the voice user experience, trying to do amazing (Io)Things.Passionate about cinema and great TV series consumer, Sunday videogamer

---