Pitfalls of asynchronous messaging | by Yuriy Ivon | Oct, 2024 | Medium

https://medium.com/@yurexus/pitfalls-of-asynchronous-messaging-93cff49e98e6

Sign up

Sign in

Sign up

Sign in

Yuriy Ivon

Follow

--

Listen

Share

Asynchronous messaging offers an attractive solution for building scalable, decoupled systems, which can be particularly tempting for developers with little or no experience in this type of communication. However, without a thorough understanding of the challenges and potential pitfalls, the implementation of asynchronous messaging can lead to unintended consequences that may outweigh its benefits. In this article, we will explore the most significant pitfalls associated with asynchronous messaging and provide practical recommendations for addressing each one.

One of the typical scenarios with asynchronous messaging is sending a notification to message bus subscribers about changes in data.

On a high level, typical code sending a message looks like this:

Due to the order of operations, a message will be sent only if the main transaction commits successfully, which is good. But if the fourth step fails due to an issue with the message broker or a crash in your service, the message will never be sent to consumers, potentially leading to various inconsistencies.

If a message loss in this scenario is critical for your system, there are basically only two practical approaches you can take to mitigate the issue: the Transactional Outbox pattern and distributed transactions.

Although distributed transactions theoretically offer the highest level of consistency in scenarios like these, I have rarely seen them being used in practice. It can be explained by several drawbacks associated with them:

Thus, the most optimal way to ensure the reliability of message publishing is the Transactional Outbox pattern. I am not going to delve into the specifics of its implementation, but the core concept involves recording the message in an ?outbox? table within the same database as part of the original transaction. Once the transaction is committed, a separate process or service retrieves the message from the outbox and publishes it to the message broker. Variations exist, such as where this separate service only sends messages if the primary application fails to publish them initially, but the fundamental principle remains consistent.

It makes sense to look at how the outbox is usually processed:

If step 3 fails for any reason, the next attempt to process the outbox table will send the same message again. Depending on the implementation of message handlers, this may or may not pose an issue for consumers. Nevertheless, it is an important aspect to be aware of, and we will discuss how to deal with it later in the section.

We talked about potential issues during publishing, but a similar situation may occur on the message handler side:

1. Lock a message for processing.
2. Begin a database transaction.
3. Modify data.
4. Commit the transaction.
5. Acknowledge the message.

What if the fifth step fails due to an issue with the message broker or a crash in your service?

Unlike the first scenario described in this chapter, no data will be lost in this case. The unacknowledged message will be reprocessed, leading to a situation akin to message duplication seen with a transactional outbox. As previously mentioned, the impact of such duplication depends on how the message handlers are implemented.

Consider a ?create a new object? command represented by the message you are processing. If the handler has no means of checking if such an object has already been created, any failure at step 5 could result in duplication ? taking the same message for processing over and over again will create new objects with identical data.

Thus, the simplest way to avoid issues if the outbox produces duplicate messages or an acknowledgment fails is to make the handler?s logic idempotent, which points us to another pattern called Idempotent Consumer. How this pattern is implemented can differ based on the type of messages and the specific business logic involved. However, the core principle is always about having a means of verifying if the message being processed has already been acted upon. Most directly, this could involve tracking all processed message identifiers. Alternatively, it can be achieved by checking if the entity already exists for commands that create new entities and using a versioning mechanism for others. Some implementation details of versioning are discussed in the ?Message Order? section.

I would also like to highlight some naïve ?patterns? that must be avoided when dealing with messaging as a part of broader business transactions:

If you need to ensure consistency between message publishing/handling and related database changes, consider using the Transactional Outbox and Idempotent Consumer patterns, and avoid the rookie mistakes mentioned above.

When an error occurs during a synchronous WEB API call, the caller becomes aware of it and can react accordingly. However, a message publisher can only know whether the message was published successfully and is unaware of its further processing, which significantly complicates error handling in asynchronous communication.

Handling a queued message might fail for a variety of reasons. If a transient error occurs during processing, the message can be returned to the queue for a retry. But what to do if the code considers the error non-transient or the maximum number of retries has been reached?

Many developers are familiar with the concept of a Dead Letter Queue (DLQ) and might suggest that all messages that can?t be processed should be placed there. However, how should these dead messages be handled, and who is responsible for their analysis and reprocessing? A short answer is that the support team must be notified about new dead letter messages, and there must be a tool that allows support engineers to review and resend messages for processing.

The main issue here is that not all message brokers have convenient tooling for these purposes:

If convenient message management tools are unavailable, something must be developed for this purpose. Keep in mind that a queue is a queue, meaning it doesn?t allow random access to its contents. If you reviewed a hundred messages in a DLQ and figured out that the 20th can be safely resent for processing or deleted, you wouldn?t be able to do so without dequeuing the first 19. To overcome this limitation, some solutions employ a listener that reads messages from the DLQ and writes them to a database. This approach allows further analysis and reprocessing to be performed based on the database entries, offering much more flexibility.

There is no limit to perfection ? you can build a custom DLQ manager that does everything a support engineer may want, but since development resources are usually limited, it is better to define the bare minimum required from such tooling.

Thus, the minimum necessary feature set for a DLQ management tool is to view all messages, resend the first N or all messages to the original queue, and delete the first N or all messages. Ideally, all manual deletions should be logged in an audit trail, allowing the support team to reconstruct the message lifecycle when necessary.

While a DLQ management tool is crucial for error handling, it?s not sufficient on its own. For effective error handling in a messaging solution, ensure these pieces are in place:

I would like to emphasize that error handling is often the most challenging part of asynchronous messaging. I have seen several projects where this was overlooked, leading to significant difficulties in system support later on.

Quite often, developers expect to receive messages exactly in the order they were published. Unfortunately, this expectation is not always valid since there are at least two cases that inherently change the order of messages regardless of the guarantees a message broker can provide:

Therefore, you need to either prevent these two situations from happening or handle out-of-order messages gracefully. Let?s explore what can be done to avoid these issues.

It appears that only the second case of reordering can be efficiently avoided, so we need to know how to handle out-of-order messages gracefully.

It is worth noting that if the messages are independent and do not rely on the sequence of previous messages, there is no need to handle ordering issues. A typical example is telemetry collection ? even if an old metric was delayed and came to the handler after more recent telemetry data, nothing will be negatively affected, the consumer will simply fill the gap.

Unfortunately, the order of messages matters in most cases, and the system must be ready for that.

Your business logic may already handle some basic ordering issues. For example, if the service receives a command to create a new entity but an entity with the same identifier already exists, or if it receives a command to activate an entity that is already active.

However, there are many situations when a message received by the service is supposed to update some fields without relying on a sequence of entity statuses. A typical example is system synchronization, where a master system publishes data modifications, and receiving systems apply them to stay consistent with the master. A typical approach to enforce ordering in this case is entity versioning.

There are at least two widely used approaches to enforce update ordering based on versions:

The diagram below illustrates the general idea of both.

If the failed message from the ?Base Version Check? is reprocessed, it will be successfully applied to the target system. In the ?Monotonic Versioning? example, the failed message is not sent to the DLQ because it contains no new information and can be safely ignored.

Each of these mechanisms has its pros and cons, and to make an informed choice, it?s important to understand the key characteristics of each.

There might be other challenges with message ordering and more sophisticated ways to handle them. I want to emphasize that this issue is crucial for any solution using asynchronous messaging, and I would like to provide some common recipes on how it can be solved.

The considerations above can be summarized as follows:

In the world of HTTP REST, tools like Swagger and OpenAPI Specification have been instrumental in defining, documenting, and testing API contracts. These tools have enabled developers to standardize API definitions, ensuring consistency and clarity in communication between different services and systems.

Asynchronous communication, however, presents a different set of challenges. Unlike synchronous APIs, where the request-response model is straightforward, asynchronous messaging involves a more complex interaction pattern. Messages are sent without an immediate expectation of a response, and the communication can span multiple systems, queues, and brokers.

Due to this complexity, managing communication contracts for asynchronous APIs has often required significant manual effort. Developers have had to meticulously document message formats and interaction patterns and ensure compatibility across different system components. This manual process is prone to errors and inconsistencies, which can lead to integration issues and increased maintenance overhead.

To address these challenges and reduce the potential for errors, it is better to use specialized tools and frameworks.

The most comprehensive framework currently available for asynchronous contract management is AsyncAPI. It provides a common language and set of tools, allowing developers to define and document APIs in a standardized format. This further simplifies the creation of consistent documentation, enables client and server code generation, and improves the overall interoperability and maintenance of systems that rely on asynchronous messaging.

Even if your project is not ready to adopt AsyncAPI for any reason, it is essential to manage the message structure in a transparent and traceable way. CloudEvents is another high-level specification, which is focused on describing message data in common formats. It can help manage message schemas in a technology-agnostic way, which gives you the freedom to switch between serialization formats in the future.

Well-known binary formats such as Protobuf and Avro inherently provide capabilities for defining message schemas, which ensure that data can be easily validated, serialized, and deserialized across different systems. These formats also facilitate versioning and schema evolution, making them robust choices for long-term maintenance.

For projects using JSON-based messages, Pact.io may come in handy. It is designed for consumer-driven contract testing, enabling teams to define and test the contracts between services to ensure compatibility. This helps maintain clear and consistent message structures and ensures that any changes in the API do not break existing functionality.

One way or the other, contract management must be as effective as possible.

Testing interactions between components that rely on messaging is more challenging than testing components communicating synchronously. Automated integration tests are essential in this case, with all interacting services up and running.

One of the most important aspects in addition to regular tests is cases related to message delivery anomalies:

The simplest way to implement this kind of test is by publishing a sample series of problematic messages from a test script directly to the message broker and waiting for the expected results.

Connectivity issues are also important to test:

These conditions are usually more difficult to simulate, and overall, there are three levels where connectivity failures can be injected:

I would definitely recommend the latter two because the first approach pollutes the application code. Strictly speaking, testing system behavior in the case of unavailability of its individual components should be done regardless of whether or not it uses asynchronous communication. However, due to the higher complexity of asynchronous systems, I consider the importance of such testing to be greater.

Another challenge is the eventually consistent nature of asynchronous systems, where there is no definite point in time when you can expect your conditions to be satisfied. A publisher can send a command to consumers, but the timing of when the command is processed by each one may vary. To account for this in automated testing, retries and timeouts must be implemented in the assertion checks to handle potential delays.

Some projects do not use a dedicated, permanent environment for integration tests, instead running all solution components as containers. This allows the solution to be deployed anywhere, without depending on a specific cloud environment. While this approach has its advantages and disadvantages, which are beyond the scope of this article, it?s important to note that certain cloud-based message brokers, like Azure Service Bus, may not have a readily available containerized equivalent yet. Therefore, when adopting this approach for integration testing, ensure that the message broker you choose can be run as a container.

Finally, when messaging is used to synchronize data across multiple systems, it is often necessary to develop a data comparison tool that periodically verifies consistency. This can present significant challenges due to various factors, but without such a tool, ensuring the accuracy of your data synchronization process becomes tremendously difficult.

Every modern popular message broker has capabilities for implementing high availability and disaster recovery, though their setup complexity and limitations may differ.

The first issue with high availability and disaster recovery is that those who introduce a message broker into a solution may not fully consider the associated setup and maintenance effort. While it is usually quite easy to run Kafka or any other broker locally using a publicly available Docker image, setting up a production configuration for a non-PaaS message broker and maintaining it is a completely different challenge. Therefore, always consider the operational complexity of a technology when making a decision.

Cloud-based offerings typically have built-in redundancy, support failover to another region, and are generally much easier to set up. However, I strongly recommend carefully reading all documentation about supported features and limitations. A few years ago, I drafted an architecture for a system that used Azure Service Bus and was supposed to be deployed across multiple regions. Then I was surprised to discover that there was no built-in mechanism for message replication between regions. As a result, if a region went down, all unprocessed messages in that region would be lost. At that time, I had to revise the design, but it looks like the issue was finally solved, and geo-replication for

Azure Service Bus is already available, albeit in preview.

Another common mistake is postponing the setup of a highly available configuration for the message broker and limiting it to the production environment only. This is usually done to save costs, but it often ends up being more expensive due to the potential consequences. The solution must be thoroughly tested in the highly available configuration before going live, and the team should gain experience in both setting it up and maintaining it. Even the application code may require adjustments to support a highly available message broker setup.

It is crucial to understand the complexity involved in implementing high availability and disaster recovery for a message broker and ensure that proper planning and preparation are done well in advance.

When utilizing asynchronous messaging systems, such as message brokers, one critical tradeoff to consider is between latency and throughput. This balance is very important in determining the efficiency and performance of the messaging infrastructure.

Latency refers to the time it takes for a message to travel from the sender to the receiver. Low latency is essential in applications where real-time or near-real-time processing is crucial, such as financial trading platforms or live communication systems. However, achieving low latency often requires prioritizing quick message delivery, which can limit the system?s overall throughput.

Throughput measures the number of messages a system can process in a given time period. High throughput is vital for applications dealing with large volumes of data, like telemetry collection systems or online analytics solutions.

There is a high chance that the default settings will not meet your needs in terms of latency and throughput. To achieve the desired level of performance, you will need to tune either the broker?s configuration, your client code, or both. Every technology has its unique set of controls, so for specific recommendations on the subject, refer to the official documentation or other related articles.

Developers usually try to abstract the specific database engine out as much as possible, which is good. For some reason, this is not always the case with message brokers. I have seen a couple of projects where a minimal abstraction layer was added just to enable testability. It was supposed to allow the use of stub implementations for a sender or a receiver, but some broker-specific implementation details still leaked beyond the abstraction. At the same time, I know projects that migrated from one message broker to another; one of them has even undergone this migration twice.

Unless you are building an ultra-low-latency solution, it is always better to use libraries that introduce a level of abstraction to promote testability, reduce the amount of boilerplate code, and minimize the effort required to migrate from one messaging platform to another. The examples are:

I guess finding equivalents for other technology stacks should not be a problem, but bear in mind that any library like that supports only a limited set of message brokers.

If you are considering introducing asynchronous messaging into your solution, think carefully about whether its benefits truly outweigh the challenges discussed in this article. There are situations when it can be avoided, for example:

Some of these alternatives may still require addressing certain issues discussed in the article, but not all of them.

And if you need asynchronous messaging anyway, consider all the mentioned pitfalls to estimate and implement the solution properly.

--

--

Help

Status

About

Careers

Press

Blog

Privacy

Terms

Text to speech

Teams