Microservices 101: Transactional Outbox and Inbox

Setting up proper and reliable communication channels between microservices is not a piece of cake! We're having a look at how it's done with transactional outbox & inbox patterns.

By Krzysztof Atłasik

11 мин. на чтение · Посмотреть оригинал

One of the fundamental aspects of microservice architecture is data ownership. Encapsulation of the data and logic prevents the tight coupling of services. Since they only expose information via public interfaces (like stable REST API) and hide inner implementation details of data storage, they can evolve their schema independently of one another.

A microservice should be an autonomous unit that can fulfill most of its assignments with its own data. It can also ask other microservices for missing pieces of information required to complete its tasks and, optionally, store them as a denormalized copy in its storage.

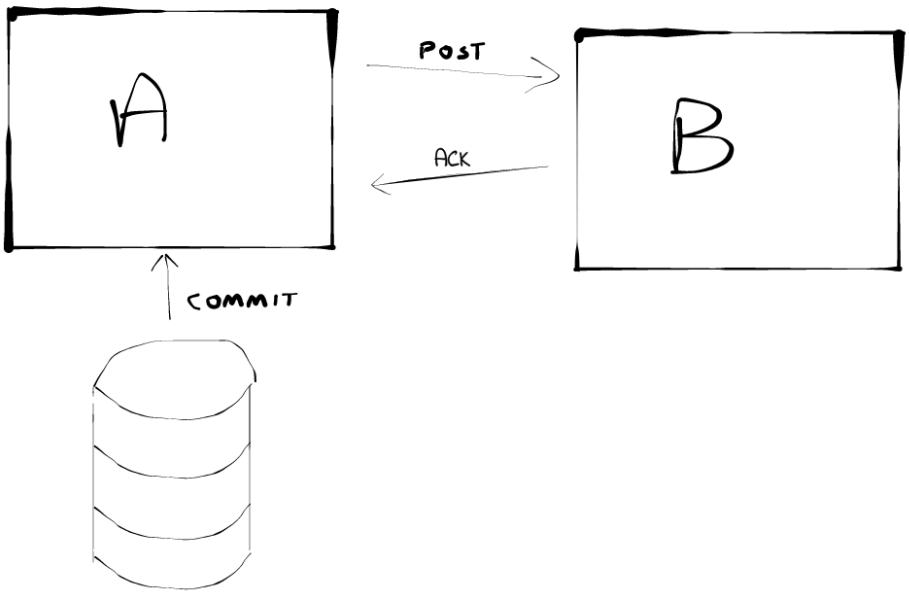
Inevitably, services also have to exchange messages. Usually, it’s essential to ensure that the sent message reaches its destination and losing it could yield serious business implications. Proper implementation of communication patterns between services might be one

of the most critical aspects when applying microservices architecture. It's quite easy to drop the ball by introducing unwanted coupling or unreliable message delivery.

What can go wrong?

Let's consider a simple scenario of service **A** having just finished processing some data. It has committed the transaction that saved a couple of rows in a relational database. Now it needs to notify service **B** that it has finished its task and new information is available for fetching.

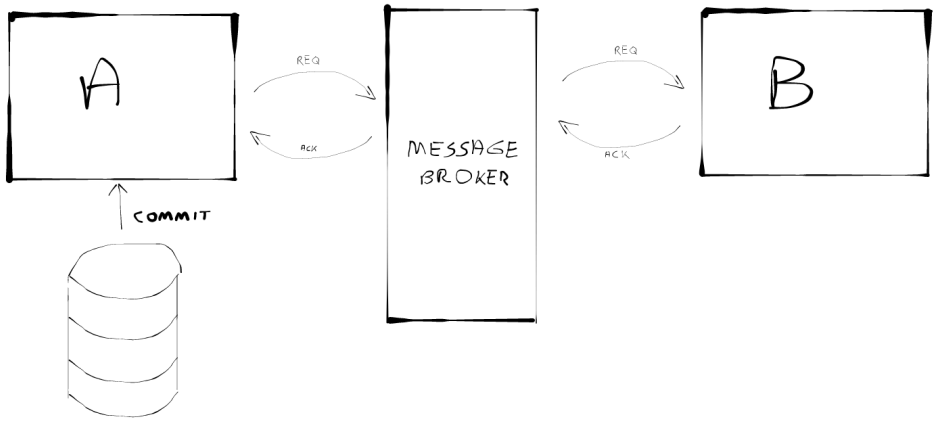
The simplest solution would be just to send a synchronous REST request (most probably POST or PUT) to service **B** directly after a transaction is committed.



This approach has some drawbacks. Arguably, the most important one is a tight coupling between services caused by the synchronous nature of the REST protocol. If any of the services is down because of maintenance or failure, the message will not be delivered. This kind of relationship is called **temporal coupling** because both

nodes of the system have to be available throughout the whole duration of the request.

Introducing an additional layer - message broker - decouples both services. Now service **A** doesn't need to know the exact network location of **B** to send the request, just the location of the message broker. The broker is responsible for delivering a message to the recipient. If **B** is down, then it's the broker's job to keep the message as long as necessary to successfully pass it.

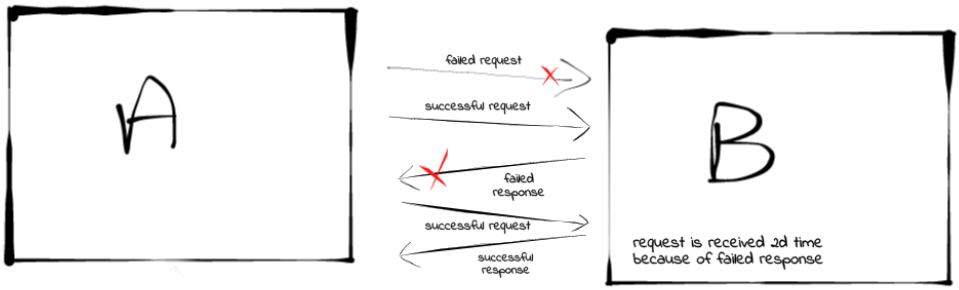


If we take a closer look, we might notice that the problem of temporal coupling persists even with the messaging layer. This time, though, it's the broker and service **A** that are using synchronous communication and are coupled together. The service sending the message can assume it was correctly received by the broker only if it gets back an ACK response. If the message broker is unavailable, it can't obtain the message and won't respond with an acknowledgement. Messaging systems are very often sophisticated and durable distributed systems, but downtime will still happen from time to time.

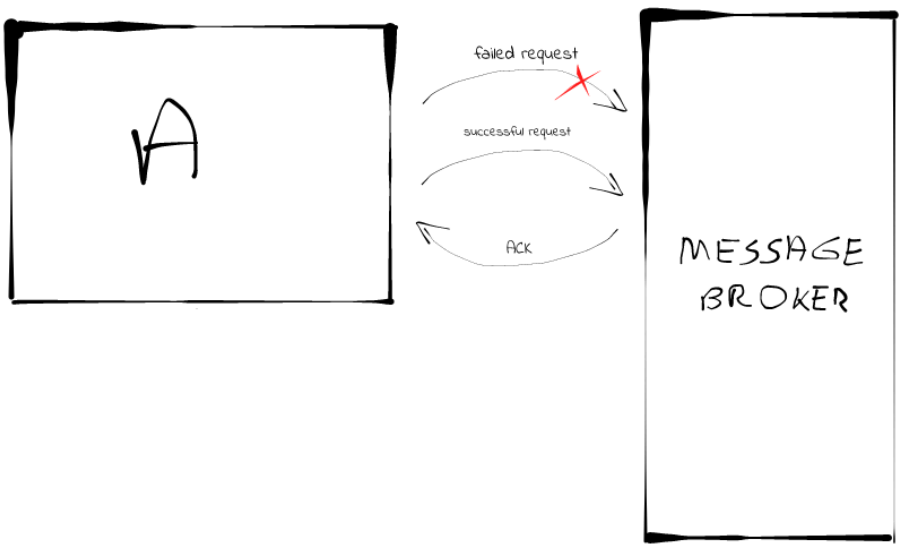
Failures very often are short-lasting. For example, an unstable node can be restarted and become operational again after a short period of downtime. Accordingly, the

most straightforward way to increase the chance for the message to get through is just retrying the request. Many HTTP clients can be configured to retry failed requests.

But there's a catch. Since we're never sure whether our message reached its destination (maybe the request got through, but just the response was lost?), retrying the request can cause that message to be delivered more than once. Thus, it's crucial to deduplicate messages on the recipient side.



The same principle can be applied to asynchronous communication.
For example, Kafka producers can retry the delivery of the message to the broker in case of retriable errors like `NotEnoughReplicasException`. We can also configure the producer as idempotent and Kafka will automatically deduplicate repeated messages.

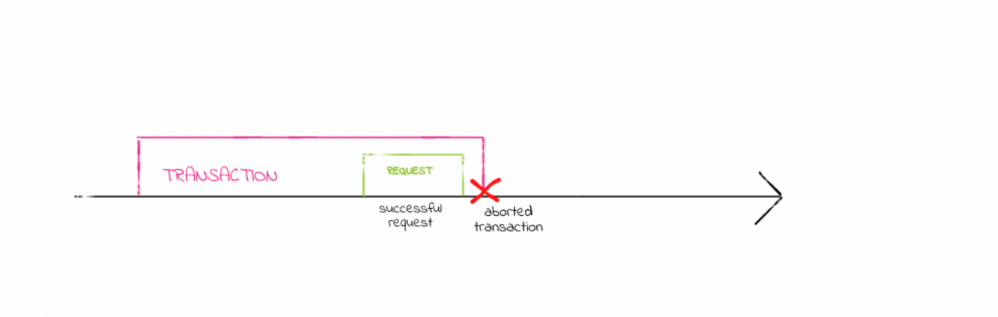


Unfortunately, there's bad news: even retrying events doesn't guarantee that the message will reach its target service or the message broker. Since the message is stored only in memory, then if service **A** crashes before it's able to successfully transfer the message, it will be irretrievably lost.

A situation like this can leave our system in an inconsistent state. On the one hand, the transaction on service **A** has been successfully committed, but on the other hand, service **B** will never be notified about that event.

A good example of the consequences of such failure might be communication between 2 services when the first one has deducted some loyalty points from a user account and now it needs to let the other service know it must send a certain prize to the customer. If the message never reaches the other service, the user will never get their gift.

So maybe the solution for this problem will be sending a message first, waiting for ACK, and only then committing the transaction? Unfortunately, it wouldn't help much. The system can still fail after sending the message, but just before the commit. The database will detect that the connection to the service is lost and abort the transaction. Nevertheless, the destination service will still receive a notification that the data was modified.



That's not all. The commit can be blocked for some time if there are other concurrent transactions holding locks on database objects the transaction is trying to modify. In most relational databases, data altered within a transaction with an isolation level equal to or stronger than the *read committed* (default in Postgres) will not be visible until the completion of a transaction. If the target service receives a message before the transaction is committed, it may try to fetch new information but will only get the stale data.

Additionally, by making a request before the commit, service **A** extends the duration of its transaction, which may potentially block other transactions. This sometimes may pose a problem too, for example, if the system is under high load.

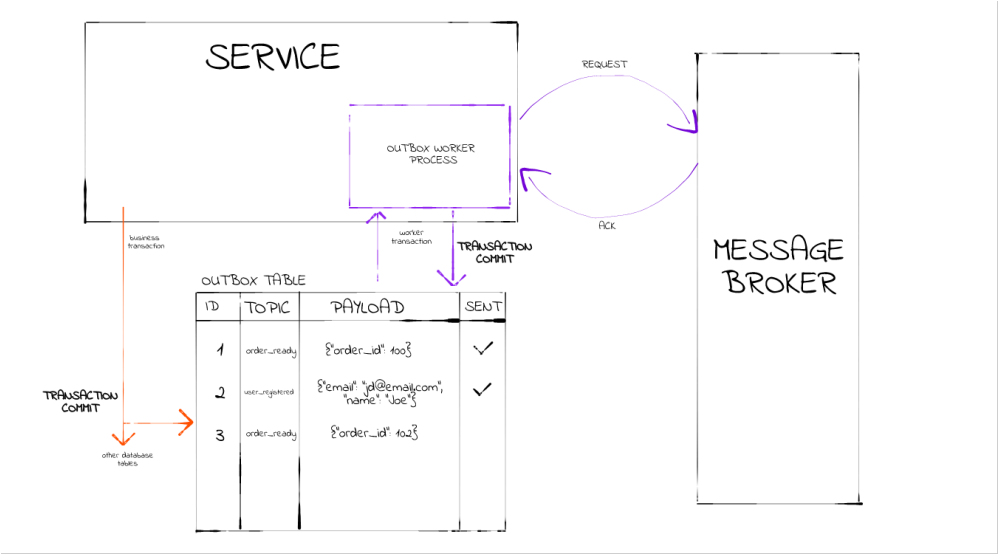
So are we doomed to live with the fact that our system will get into an inconsistent state occasionally? The old-school approach for ensuring consistency across services would use a pattern like distributed transaction (for example 2PC), but there's also another neat trick we could utilize.

Transactional outbox

The problem we're facing is related to an issue that we can't atomically both perform an external call (to the message broker, another service, etc.) and commit the ACID transaction. In the happy path scenario, both tasks will succeed, but problems start when one of them fails for any reason. I will try to explain how **we can overcome these issues by introducing a transactional outbox pattern**.

As the first step, we need to introduce a table that stores all messages that are intended for delivery - that's our **message outbox**. Then instead of directly doing requests, we just save the message as a row to the new table. Doing an *INSERT* into the message outbox table is an operation that can be a part of a regular database transaction. If the transaction fails or is rolled back, no message will be persisted in the outbox.

In the second step, we must create a background worker process that, in scheduled intervals, will be polling data from the outbox table. If the process finds a row containing an unsent message, it now needs to publish it (send it to an external service or broker) and mark it as sent. If delivery fails for any reason, the worker can retry the delivery in the next round.



Marking the message as delivered involves executing the request and then a database transaction (to update the row). That means we are still dealing with the same problems as before. After a successful request, the transaction can fail and the row in the outbox table won't be modified. Since the message status is still pending (it wasn't marked), it will be re-sent and the target will get a message twice. That means that the outbox pattern doesn't prevent duplicate requests - these still have to be handled on the recipient side (or message broker).

The main improvement of the transactional outbox is that the intent to send the message is now persisted in durable storage. If the service dies before it's able to make a successful delivery, the message will stay in the outbox. After restart, the background process will fetch the message and send the request again. Eventually, the message will reach its destination.

Ensured message delivery with possible duplicated requests means we've got an **at-least-once processing guarantee** and recipients won't lose any notifications (unless in case of some catastrophic failures causing data loss in the database). Neat!

Not surprisingly, though, this pattern comes with some weak points.

First of all, **implementing the pattern requires writing some boilerplate code**. The code for storing the message in the outbox should be hidden under a layer of abstraction, so it won't interfere with the whole codebase. Additionally, we'd need to implement a scheduled process that we'll be getting messages from the outbox.

Secondly, polling the outbox table can sometimes put significant stress on your database. The query to fetch messages is usually as plain as a simple *SELECT* statement. Nevertheless, it needs to be executed at a high interval (usually below 1s, very often way below). To reduce the load, the check frequency can be decreased, but if the polling happens too rarely, it will impact the latency of message delivery. You can also decrease the number of database calls by simply increasing the batch size. Nonetheless, with a big number of messages selected, if the request fails, none of them will be marked delivered.

The throughput of the outbox can be boosted by increasing the parallelism. Multiple threads or instances of the service can be each picking up a bunch of rows from the outbox and sending them concurrently. To prevent different readers from taking up the same message and publishing it more than once, you need to block rows that are just being handled. A neat solution is locking a row with the *SELECT ... FOR UPDATE SKIP LOCKED* statement (*SKIP LOCKED* is available in some relational databases, for example in Postgres in Mysql). Other readers can then fetch other unblocked rows.

Last, but not least, if you're sending massive amounts of messages, the outbox table will very quickly bloat. To keep its size under control, you can create another background process that will delete old and already sent messages. Alternatively, you can simply remove the message from the table just after the request is acknowledged.

A more sophisticated approach for getting data from the outbox table is called database log tailing. In relational databases, every operation is recorded in WAL (write-

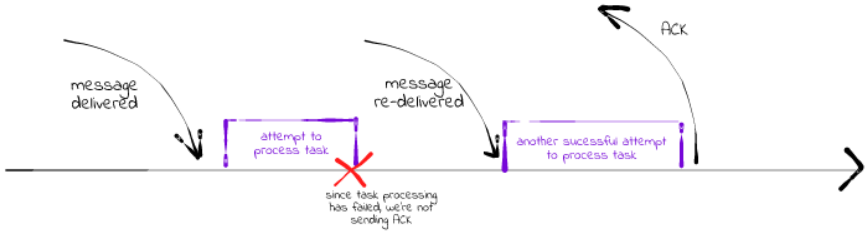
ahead-log). It can be later queried for new entries concerning rows inserted into the message outbox. This kind of processing is called CDC (capture data change). To use this technique, your database has to offer CDC capabilities or you'd need to use some kind of framework (like Debezium).

I hope I have convinced you that the outbox pattern can be a great asset to increase the robustness and reliability of your system. If you need more information, a remarkable source to learn more about transaction outbox and its various applications is the book titled *Microservices Patterns* by Chris Richardson.

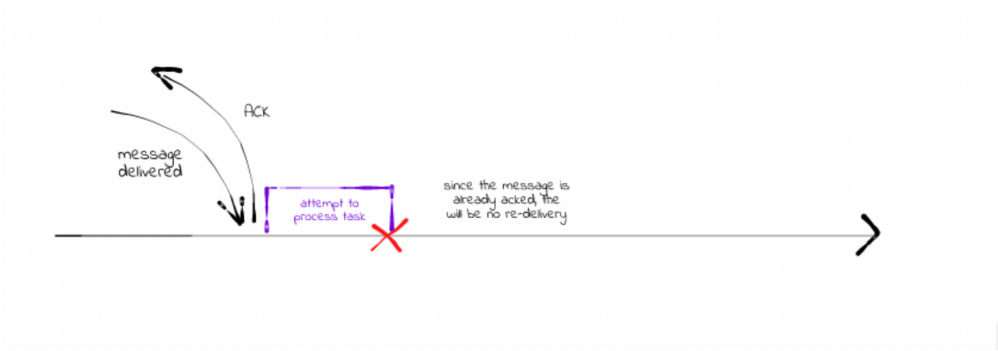
Inbox pattern

A properly implemented outbox pattern ensures that the message will eventually reach its target. To get this guarantee end to end with messaging system, the broker also needs to assure at-least-once delivery to the consumer (like Kafka or RabbitMQ do).

In most circumstances, we do not only want to simply deliver the message. It's also important to ensure that the task that was triggered by the message is completed. Hence, it's essential to acknowledge the message receipt only after the task's completion! If the task fails (or the whole service crashes), the message will not be acked and will get re-delivered. After receiving the message again, the service can retry processing (and then ack the message if the task is finished).



If we do things the other way around: ack the message first and only then start processing, we'll lose that at-least-once guarantee. If the service crashes when performing the task, the message will be effectively lost. Since it is already acknowledged, there will be no retries of delivery.

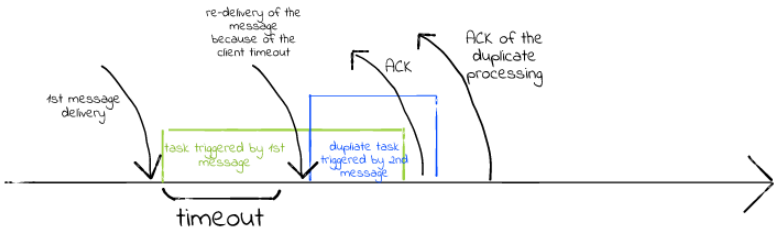


The at-least-once processing guarantee means that the recipient will from time to time get repeated messages. For that reason, it's crucial to actively eliminate duplicate messages. Alternatively, we can make our process idempotent, which means that no matter how many times it is re-run, it should not further modify the system's state.

Message delivery can be retried in case of explicit failure of processing. In this case, the recipient could return NACK (negative acknowledgement) to the broker or respond with an error code when synchronous communication is used. This is a clear sign that the task was not successful and the message has to be sent again.

The more interesting scenario happens when the work takes more time to finish than expected. In such a situation, the message, after some time, can be considered as lost by the sender (for example, an HTTP request can hit the timeout, visibility timeout of SQS can pass, etc.) and scheduled for redelivery. The recipient will get the message again even if the first task is still alive. Since the message is not yet acked, even a proper deduplication mechanism won't prevent triggering multiple concurrent processes caused by these repeated messages.

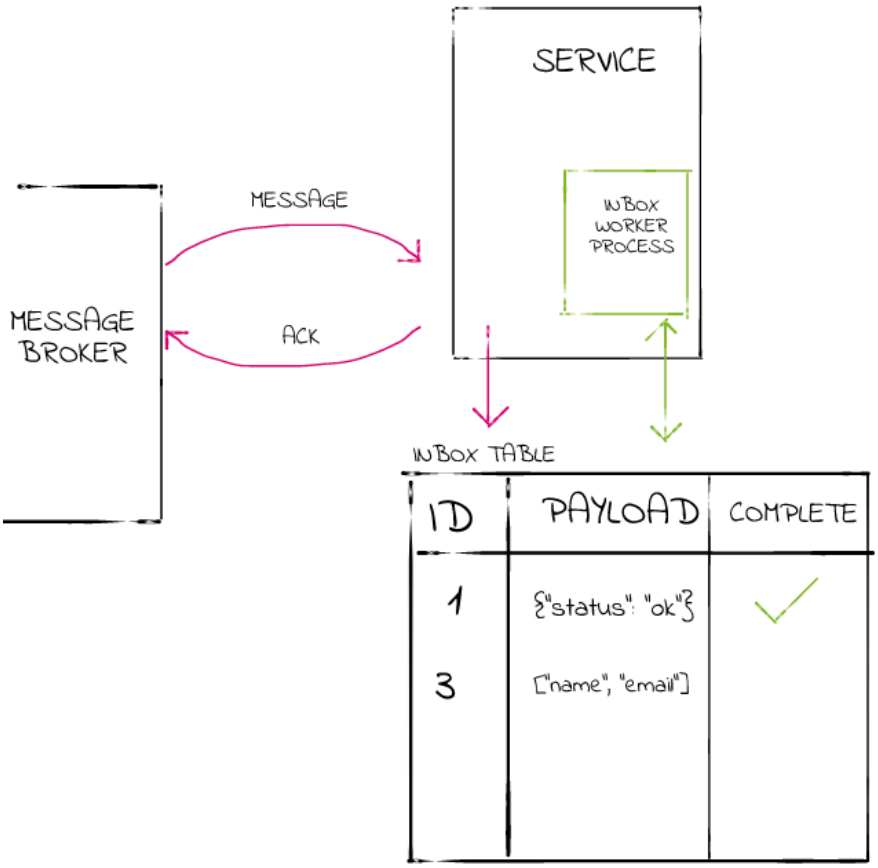
Timeouts can be fine-tuned so they take into consideration the prolonged time when the task is handled. On the other hand, finding out the appropriate value is sometimes difficult, especially when completing an assignment by a client takes an unpredictable amount of time.



Repeated work is very often not a big deal with correctly working duplicate elimination. We can simply get the message and reject the result after task completion (or upsert if it's idempotent). But what if the processing involves some costly action? For example, after receiving the request, the client could spin up an additional VM to handle the assignment. To avoid

unnecessary waste of resources, we could adopt another communication pattern: the **transaction inbox**.

The inbox pattern is quite similar to the **outbox pattern** (but let's say it works backwards). As the first step, we create a table that works as an inbox for our messages. Then after receiving a new message, we don't start processing right away, but only insert the message to the table and ACK. Finally, the background process picks up the rows from the inbox at a convenient pace and spins up processing. After the work is complete, the corresponding row in the table can be updated to mark the assignment as complete (or just removed from the inbox).



If received messages have any kind of unique key, they can be deduplicated before being saved to the inbox. Repeated messages can be caused by the crash of the recipient just after saving the row to the table, but

before sending a successful ack. Nevertheless, that is not the only potential source of duplicates since, after all, we're dealing with an at-least-once guarantee.

Again, the throughput of processing can be improved by increasing parallelism. With multiple workers concurrently scanning the inbox table for new tasks, you need to remember to lock rows that were already picked up by other readers.

Another possible optimization: instead of waiting for the worker process to select the task from the inbox, we can start it asynchronously right after persisting it in the table. If the process crashes, the message will still be available in the inbox.

The inbox pattern can be very helpful in case the ordering of messages is important.
Sometimes, the order is guaranteed by a messaging system (like Kafka with *idempotence* configuration turned on), but that is not the case for every broker. Similarly, HTTP requests can interleave if the client doesn't make them in a sequence in a single thread. Fortunately, if messages contain a monotonically increasing identifier, the order can be restored by the worker process while reading from the inbox. It can detect missing messages, hold on until they arrive, and then handle them in sequence.

What are the disadvantages? Similar to the outbox pattern: increased latency, additional boilerplate, and more load on the database.

Very often, the recipient service can cope without the inbox. If the task doesn't take long to finish or completes a predictable amount of time, it can just ack

the message after processing. Otherwise, it might be worthwhile to spend some effort to implement the pattern.

Wrapping up

As I stated at the beginning of this article: setting up proper and reliable communication channels between microservices is not a piece of cake! Thankfully, we can accomplish a lot by using correct patterns. It's always important to consider what guarantees your system needs and then apply appropriate solutions.

And always remember to deduplicate deduplicate your messages 😉