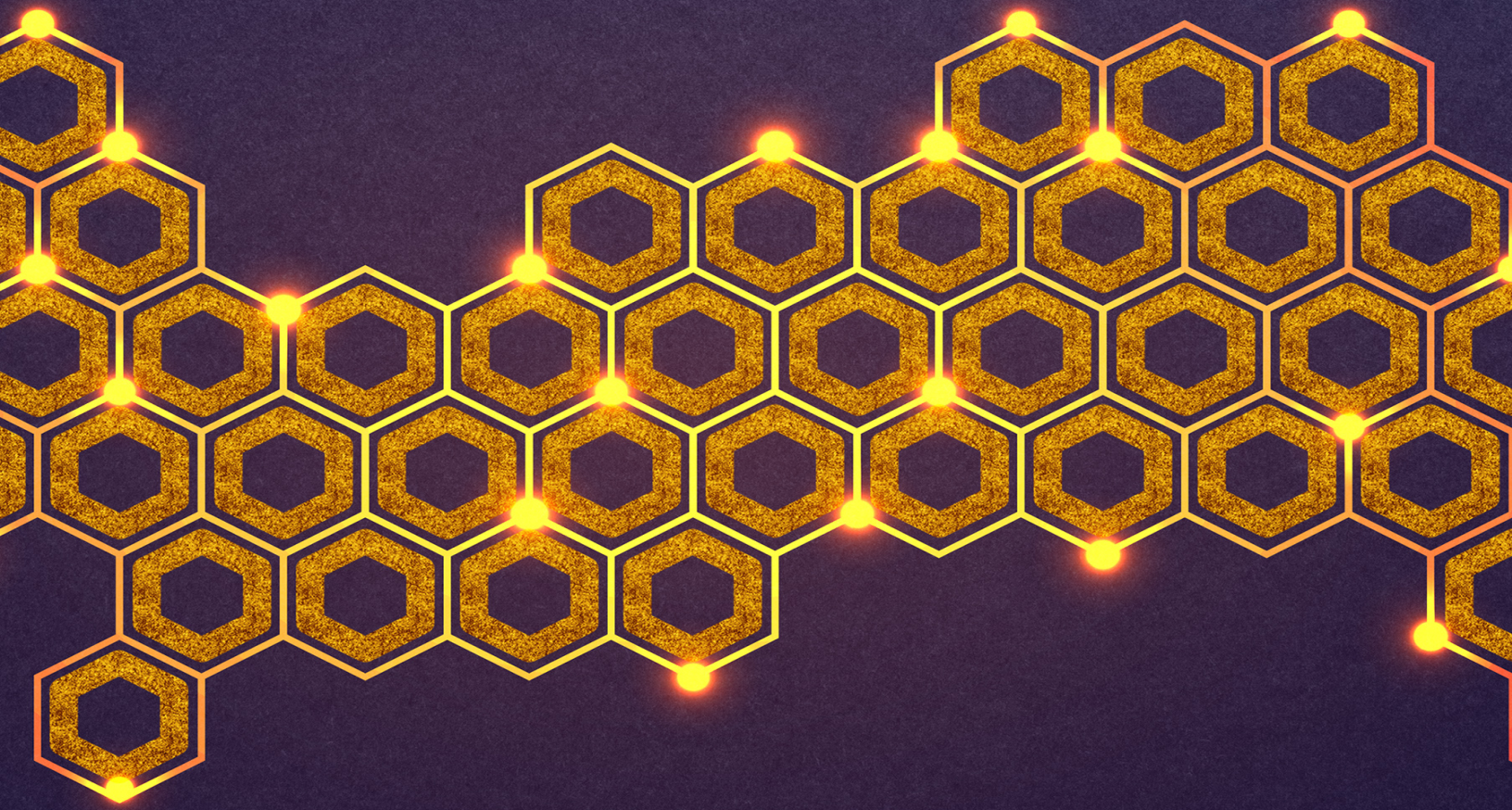
MICROSERVICE
TRANSACTION PATTERNS

FOR THE ENTERPRISE



GUY PARDON

Microservice Transaction Patterns

For The Enterprise

guypardon

This book is for sale at http://leanpub.com/microservice-transaction-patterns

This version was published on 2020-07-25

To my family.

Contents

Introduction

We all know microservices and cloud are shaking up the IT industry. It's happening at an ever faster pace, and for sure it's hard to stay on top of things. So we're all trying to find out how to do things in this new world.

What about transactions and data consistency? We keep hearing that things don't work the way they used to.

Are you wondering what to do instead? Most experts will recommend workaround patterns that are either code- or design-intensive, like:

- eventual consistency
- idempotent consumer
- event store
- saga

These are all interesting patterns in their own right, but often overkill. They demand a custom design that is in itself challenging to say the least. The resulting architectures are often easy to break by accidental code commits.

Moreover, most implementations suffer from risk of lost transactions or duplicate transactions. Developers often don't know, because they are under stress to deliver working code. They rarely get time to think about what could go wrong.

For many projects there is a simpler way, like I will show you in this book.

Like you, I was once puzzled by the mystery of reliable distributed systems, back in 1993 already. I started learning about it and even went on to get a PhD from ETH Zürich, Switzerland. I didn't stop there: I founded a company (Atomikos) and the only thing we do is transaction management. I still do some research today, and try to publish a paper every now and then.

At Atomikos, our story is one of ruthless simplification by keeping only the things that work. For instance, we pioneered JEE without application server in 2006. Later we adhered to SOA without ESB. I am proud to say that we got Gartner's "Cool Vendor" award for our work.

Today we help our customers move away from the application server. We help them transition to enterprise-grade microservices instead. With microservices, transactions become *distributed* transactions whether you like it or not. For most people that is a challenge. Not for us: we have decades of pioneering experience under our belt.

5 simple patterns are at the core of what we do today, battle-tested solutions that stood the test of time. They demand little or no coding and work even for (and especially with) *microservices*. They

are *cloud-ready*. They work for our customers, mainly in financial services. I wanted to share these with you by means of this book. I have tried to keep it short and practical: this book contains only what you need to know, nothing else. Busy readers will appreciate the brevity for sure.

Disclaimer: I am not saying that the solutions in this book will work for everyone. There will be situations where you have to make trade-offs. All I want to show you is that there are a few simple (but little-known) solutions that may work for you. For simplicity's sake: try these first and resort to more complex solutions only if you really have to. That way, you will minimise the accidental complexity of your microservice architecture.

If that sounds interesting to you then please read on to get started!

Some Terminology

I will use the words *request* and *message* interchangeably depending on the context. To me, either word means the same thing: *data flowing from one microservice to another to update state.*

We could distinguish between read requests and update requests. But: the patterns in this book concern updates, so the word *request* means *update* to me. Same thing for a *message*.

In a distributed system the following are always interesting questions:

- *Can we lose messages?* For instance, can a payment order get lost after a crash or a bug?
- *Can we receive duplicate messages?* For instance, can we accidentally process the same payment request twice?

Failures usually imply that you can lose requests. In practice, the common approach is to retry when there is uncertainty about the outcome of a request. This in turn means you can get duplicate requests.

We'll see that this is a common theme in this book.

The only way to avoid that is to use the patterns presented in the rest of this book (so if that is what you need then feel free to skip this chapter)…

Eventual Consistency

Eventual consistency is a basic and very important notion so let's start with this one. What does eventual consistency mean? It sounds complicated but it's really simple:

Rather than updating related data in each microservice, you let changes ripple through. Eventually everything gets updated.

For non-native English readers: eventually means "in the end" or "sooner or later", not "maybe". It's a common misunderstanding so I thought I'd point that out.

Let's say you make a payment by wire transfer. The money will leave your account right-away. But does the receiver get the money immediately? Usually not: for various reasons (technical and commercial ones) it takes time. The money gets there, eventually.

That is the meaning of eventual consistency. The desired end state is that money from your account appears at the receiver's end. But it takes a while for that to happen.

In the meantime, the money is gone from your account but not yet at its destination. So if you look at both accounts then you only see half of the desired end state. It's not consistent yet.

Messages that travel from one computer to another are typical for eventual consistency. This usually requires messaging middleware (like ActiveMQ, MQSeries or Kafka) as the backbone. This type of messaging middleware is also called a "broker".

Needless to say, if you lose requests (or messages) along the way then a consistent end state may never happen. Like debiting your account without the receiver ever getting anything.

If the receiver's bank is not reachable then the system may retry later. Without extra care, the receiver may see his account credited more than once. How? I won't go into the details here since I dedicate a whole chapter to this, so please be patient.

For now, just take note of the following:

Eventual consistency can yield inconsistent end states when there are failures. Our simple patterns can prevent this - this book will show you how.

Google's Take on Eventual Consistency

You don't have to take my word for it. Here is what Google says about the requirements for their core business (Adwords):

We store financial data and have hard requirements on data integrity and consistency. *We also have a lot of experience with eventual consistency systems at Google. In all such systems, we find developers spend a significant fraction of their time building extremely complex and error-prone mechanisms to cope with eventual consistency and handle data that may be out of date.*
Source: this Google paper[1].

Much of this complexity comes from the companion pattern discussed later in this book: *idempotent consumer*.

[1] http://datascienceassn.org/sites/default/files/F1%20A%20Distributed%20SQL%20Database%20That%20Scales.pdf%20

Synchronous versus Asynchronous Microservices

Synchronous Microservices

Synchronous microservices refers to one microservice invoking (delegating work to) another one and waiting for the result to come back. This is like a human making a phone call: you expect a reply and are waiting for it during the conversation (although some people may be exceptions).

Synchronous calls have been around for decades, with enabling technologies like remote procedure calls (RPC), the common object request broker architecture (CORBA), Java's remote method invocation (RMI), and more recently HTTP in any of its flavors (SOAP, JSON-RPC, ...).

The advantage of synchronous microservices is the simplicity in programming them, and also in diagnosing errors.

However, the devil is in the details...

Building highly available systems can get very expensive very fast because the whole system is only as available as the weakest microservice. You have to invest heavily into each microservice to make this work. If one microservice is down a lot, most of the other won't work either.

There are workarounds like the "circuit breaker" pattern or caching (for read requests), but you in general this is the price you pay for synchronous systems.

For this reason, I recommend that you use synchronous calls only when they are needed. In other words, if the result you get back is really useful for the business logic you need to support.

One common example is for checking the balance of an account in before making a payment. If the account is managed by one microservice and the payment by another, then the following should happen:

- The payment service checks the balance of the account
- Depending on the result, the payment service proceeds with the payment, or if there are insufficient funds on the account then the payment is rejected.

Of course this is not the whole story, we will see more about this in the part of this book dedicated to synchronous microservices.

This type of scenario is one of the only cases where I would recommend synchronous microservices.

Asynchronous Microservices

Asynchronous microservices don't wait for the result of interactions with other microservices. They issue a message that goes onto the network and immediately continue doing something else.

In asynchronous systems, it is left to other microservices to pick up the message and process it on their own pace. Hence the term 'asynchronous': both ends of the message operate on their own. It is like humans sending email to each other: you don't sit around wait for the reply do you? I do hope not! Just like humans send emails, asynchronous microservices send messages, too. They don't wait for any result either.

Usually this means that the sending microservice is unaware about the outcome of the request. Basically this can mean two things:

1. Best effort: the sending microservice doesn't care whether the remote receiver can process things, or
2. Reliable messaging: the sending microservice counts on the remote receive processing the message at some time

Option 1 is what platforms like social media do: they don't guarantee that updates / posts make it through. Sometimes you can lose updates.

Option 2 is what financial services prefer: if you send a payment to a different bank, you don't want it to get lost underway.

This book is concerned mostly about asynchronous in the sense of option 2. We can even make a distinction between processing the message exactly-once of processing it twice or more – more on that later.

Asynchronous Infrastructure: The Message Broker

In order to send messages, the message has to be stored somewhere until the remote is available and ready to process it. There are specialized products / platforms that do this: they are called "message brokers" (or brokers for short).

Brokers act like the "mailbox" for messages between microservices. Most of them support persistent messaging, meaning that the messages are kept on disk or stable storage and survive a crash or restart of the system. Persistent message support is often a requirement in financial services where message loss is not an option.

I am a big fan of asynchronous architectures because systems that use asynchronous messaging exhibit some very nice properties; let's go over some of these…

Zero downtime installations during normal hours

You can deploy a new installation of a microservice without impacting the rest of the system because all interactions are buffered in the broker.

As long as an intervention does not take too long, you can install a new release without anybody noticing.

This means you don't need to work weekends or nights to install a new release.

Polyglot programming

If messages are formatted in an open way (text, XML, JSON, …) then the sender and receiver can be implemented in different languages and run on different platforms. As long as the message broker has "client connector" libraries for the languages / platforms used, this works.

For clients written in Java, many brokers offer the JMS (Java Message Service) API. This is a standardized way of sending and receiving messages from Java programs. Many brokers implement it and are automatically compatible with Java microservices. Many brokers also offer other access options like C++, .Net or other. This supports polyglot architectures – which can be handy for integration with legacy systems that are not necessarily API-based.

Extensible Architectures

Messaging systems are easily extensible: the sender of a message does not need to know who processes it or how it is processed (although we may want to count on the fact that it IS processed sooner or later).

This enables extensibility: if you develop a microservice that sends a message, you can have other teams "extend" the scope of processing by adding one or more backend processing microservices. These can evolve at their own pace – as long as the message format stays more or less stable.

Automatic and Cheap Scalability

Scalability often means a load balancer, which is both expensive and complex error-prone infrastructure.

With messaging you get load balancing for free, via the "competing consumer" pattern: it is possible to configure brokers so that multiple recipients for the same type of message will divide the messages between them.

All you need to do is monitor how many messages are waiting in the broker, and if this number increases too much you add one or more new processors in parallel to the existing ones.

Cheap High Availability

As mentioned during the discussion of synchronous microservices: if one microservice is down a lot, most of the other won't work either.

Not so with message brokers. Assuming that the broker itself is highly available, each microservice itself can have lower availability – without impacting the other ones. This allows you to tune availability per microservice – based on the needs of its users.

Loose Coupling

Finally, message brokers enable loose coupling between different microservices and/or other legacy systems. This really follows from the other characteristics we discussed. It means that each part of the system can evolve more or less independently.

Synchronous - Over - Asynchronous Microservices

Besides synchronous and asynchronous, there is also a hybrid type that I would call "synchronous-over-asynchronous" microservices.

These are like synchronous, except that the communication happens via asynchronous store-and-forward infrastructure like the broker.

This style is more robust than the regular synchronous style because the ultimate destination of the call does not have to be available 100% of the time: while my service MS 1 is calling another service MS 2, it is entirely possible for MS 2 to be restarted during the call. If the restart happens quickly enough then MS 1 will still get the result because everything is being buffered in the broker:

- If the restart of MS 2 happens before it gets the invocation then the result will be produced once it has restarted (the request will just be waiting in the broker).
- If the restart of MS 2 happens after it gets the invocation then either it will already be underway in the broker – back towards MS 1.
- If the restart of MS 2 happens during the processing of the invocation then it depends on how well MS 2 was designed. If MS 2 follows the patterns in this book then it will still produce the result, exactly once.

However, keep in mind that this style is still inherently synchronous: MS 1 waits for a result from MS 2. This means that the same transaction problems apply as for a regular synchronous architecture – as outlined in the rest of this book.

Asynchronous Microservice Patterns

Pattern: Exactly-Once Sender

The problem

Asynchronous microservices send messages to each other. It turns out that a lot can go wrong and messages can be either lost or sent multiple times. In addition, we can have "phantom" messages. Let's go over some examples to show how this can happen…

Example: Order Processing Microservice

Let's take a simple microservice that processes orders with the following workflow:

1. A new order is inserted into the database
2. A message "OrderCreated" is sent out to other microservices.

Other microservices like a delivery service can then arrange their part of the overall business transaction, like planning delivery to the customer.

How messages can be lost

If there is a crash in between 1 and 2, then the database will contain the order but no message has been sent out. Delivery will not get anything, nor will the customer.

How messages can be phantoms

In order to prevent message loss, developers could change the logic to this one:

1. A message "OrderCreated" is sent out to other microservices.
2. A new order is inserted into the database

Now image the system's user enters an order via the user interface of the order processing microservice. While processing this order, there is a crash in between 1 and 2. Now the message has been sent (and delivery arranged) but there is no record of any order. This is probably not what the business wants.

How messages can be sent multiple times

Let's take the phantom case again, and now image that the user retries after things failed (and a phantom message was sent). Upon retry things now work and the order is inserted in the database. But the message was also sent out twice, which may lead to problems.

Solution: Exactly-Once Sender

We use a transaction manager and modify / enhance the workflow of order processing to this:

1. Start a new XA transaction
2. Insert the order in the database via an XA driver
3. Post the message on the broker via an XA driver
4. Commit the XA transaction

Why does this work? Because of the transaction manager doing all the hard work behind the scenes.

Explaining all the details is beyond the scope of this pattern but suffice it to say that the XA transaction makes steps 2 and 3 tentative until the commit in step 4 succeeds. Any failure will lead to XA rollback, in which case the database update will not be persisted and the broker message will not be made visible to any other microservice.

Concluding Thoughts

Many people claim that XA technology is not readily available for cloud or microservices. That is fake news: implementations such as Atomikos exist (https://www.atomikos.com[2]).

This pattern only works if you have XA transactions, which excludes popular brokers like Kafka or RabbitMQ.

Without broker support for XA you risk all the anomalies described in this chapter. The same is true for the database, but luckily most enterprise databases support XA.

Pattern: Exactly-Once Receiver

The problem

Receiving a message from a broker turns out to be more complex than you would expect. Losing a message, or processing it multiple times are more common than most people think.

Example: Delivery Service

Let's look at a simple microservice that processes incoming "OrderCreated" messages as follows:

1. The incoming message is taken off the broker
2. The delivery data is assembled and inserted into the database

[2] https://www.atomikos.com/

It is likely that other messages will get created in a real-world example, but for the sake of showing the problem we will stick with simplicity.

How messages can be lost

If there is a crash between 1 and 2 then the message is gone, but no delivery data was inserted. The message is effectively lost. That is because in step 1 we implicitly delete the message from the broker.

How messages can be processed multiple times

To avoid message loss, developers change the processing flow to the following:

1. Reading an incoming message from the broker
2. Assembling and inserting deliver data into the database
3. Removing the message from the broker

The difference here is that we split the actions of reading a message versus removing it from the broker. Most brokers allow this.

What happens if there is a crash?

If the crash happens before step 2 then the message is still on the broker and will be re-read later (assuming the system is still available or restarts). So that should be fine.

If the crash happens after step 3 then the results are already in the database, so that is fine too.

The problem is with a crash between 2 and 3: in that case the message stays on the broker, will be re-read later and inserted again in to the database. We have a message that is processed twice. For repeat failures, this can become 3 or more times.

Solution: Exactly-Once Receiver

We use a transaction manager and modify / enhance the workflow of order processing to this:

1. Start a new XA transaction
2. Take the message off the broker via an XA driver
3. Insert into the database via an XA driver
4. Commit the XA transaction

The trick is that the XA transaction and the transaction manager do the hard work for us: steps 2 and 3 now coincide with the commit in step 4. If there are any failures then the outcome will be rollback instead, and step 2 will leave the message on the broker whereas step 3 does not really save anything in the database.

Many people claim that XA transactions are slow, but actually it is quite the opposite. Our implementations with XA are usually quite faster than the non-XA alternatives discussed later in this book.

Pattern: Exactly-Once Messaging

The problem

Consider a payment from one bank to a different bank. Each bank has its own accounts database. How can we make sure money transfers are processed exactly once, without message loss or duplicates?

Solution: Exactly-Once Messaging

The solution consists of a composition of 3 patterns:

1. Exactly-once sender at the bank of origin
2. A persistent message broker in between the two banks
3. Exactly-once receiver at the destination bank

We won't say a lot more about this because it reuses concepts we've outlined in the earlier parts of this book.

Synchronous Microservice Patterns

Pattern: Transactional Call

The problems

Network timeouts

One microservice (MS 1) invokes another microservice (MS 2) via some synchronous mechanism like JSON-RPC, HTTP POST or any other remoting protocol. What happens if the call times out?

MS 2 could be in either of two states:

- It has received the call and processed it (possibly leaving persistent state changes in its database), or
- It may not have received anything.

The difference can be significant, and it introduces uncertainty at the level of MS 1. A common approach with REST purists is to start retrying (except for HTTP POST), but that introduces a lot of complexities that we want to avoid here. Besides that, retries don't work for the second problem addressed here:

Calling 2 microservices

Let's add another microservice (MS 3) in the mix. The overall flow now looks like this:

1. MS 1 calls MS 2
2. MS 1 then calls MS 3

Suppose step 1 updates database of MS 2. Now what happens if step 2 fails and we can't retry? This can happen if step 2 encounters a fundamental business exception - where retrying simply won't work.

What can we do? If we can't go forward with step 2 then a nice alternative would be to "rollback" step 1. But how can we do that?

Solution: Transactional Calls

If we can somehow create a transaction that spans microservices then there would be no problem:

1. MS 1 starts a new transaction
2. MS 1 calls MS 2 with the transaction context included in the call
3. MS 2 performs its updates but leaves commit pending until it hears back from MS 1
4. Steps 2 and 3 are repeated for MS 3
5. MS 1 commits the transaction, which includes committing the changes in MS 2 and MS 3 in the same transaction.

If step 4 fails then MS 1 can still rollback, and this will wipe out the changes of MS 2. This leaves the system in a globally consistent state – even if there are errors.

What happens in case of a timeout at step 2? MS 1 will perform rollback of its transaction. There are 2 possibilities in this case:

- MS 2 was never reached by the call. In that case, no data was updated so rollback does not have to do anything except wipe out the changes locally in MS 1.
- MS 2 was reached by the call and made some data changes, meaning the response / acknowledgement to MS 1 was lost. MS 2 is aware of the pending transaction of the call and after some internal timeout it performs autonomous rollback.

In any case, the overall result is that all changes undergo rollback. So timeouts leave the system in a globally consistent state.

Implementation

See this blog post[3] for how to implement / use this pattern.

Pattern: TCC

The problem

Consider Internet-wide service collaborations across multiple independent websites. How can they be made reliable so we're not stuck with a partial business transaction?

Example: Booking Connecting Flights

Let's take the booking of a long-distance flight involving connecting flights between two airlines. In order to book the whole flight, we need to book at two independent airline websites, say Swiss and British Airways. What happens if we end up with one booking, but the other one fails somehow?

[3] https://www.atomikos.com/Blog/TransactionalRESTMicroservicesWithAtomikos

This problem is similar to the "transactional call" pattern in the previous chapter. However, the participating services are not just microservices but completely independent stand-alone services offered via independent websites. This makes the simple solution of the "transactional call" less practical.

Solution: TCC

TCC stands for Try-Confirm/Cancel (or Try-Cancel/Confirm – which means the same thing). It works well for all scenarios where the business works with some kind of "reservation" model. It is a bit like the Saga pattern, but superior because it does not leave the participating services in-doubt about the possibility of later undo.

TCC was developed by me and my ex-colleague Cesare Pautasso (we both worked in the same research group at ETH Zürich while doing our PhD).

TCC is defined as a complete REST API contract along the following lines:

1. We "Try" to book at Swiss
2. We "Try" to book at British Airways
3. If everything went OK then we send "Confirm" to both airlines
4. If not, we "Cancel" where needed

We won't mention all the details here because the market is not yet ready for doing the kind of service interactions that TCC offers. Let's just say it's ideal for microservices: just like Sagas, it implies significant coding overhead because cancellation operations have to be defined and invoked. That is why our customers have been asking us to support the "transactional call" pattern instead. People like the simplicity.

If you're interested in finding out more about TCC, check https://www.atomikos.com/Blog/TransactionsForRestApiI for all the details.

[4] https://www.atomikos.com/Blog/TransactionsForRestApiDocs

Legacy Microservice Patterns

Most people take "legacy" with a negative meaning. I don't, to me legacy is what's working in production today.

That's why this chapter is about "legacy patterns" – because you are likely using some of these already.

The goal here is not to outline every detail of each, but just to do a review of existing "well-known" patterns for microservice transactions and how they differ from what I propose in this little book.

Pattern: Transactional Outbox

The problem

Let's revisit the problem of sending messages reliably. How could you go about it when you don't have XA transactions available?

Example: Order Processing Microservice

Let's take a simple microservice that processes orders with the following workflow:

1. A new order is inserted into the database
2. A message "OrderCreated" is sent out to other microservices.

We've discussed possible anomalies and problems in the section that introduced our "exactly-once sender" pattern – so I will not repeat the whole discussion here. Let's just introduce what is a common pattern that people advocated before this book.

Solution: Transactional Outbox

The transactional outbox works like this:

The application workflow

1. Start a new database transaction
2. Insert the order into the database
3. Insert a "message" entry into a separate "outbox" table (in the same database)
4. Commit the database transaction

Message publishing workflow

A separate process monitors the outbox table for new records:

1. Start a new database transaction
2. Read the next record
3. Publish a corresponding message on the broker
4. Delete the record
5. Commit the database transaction

Some variations of this pattern are possible, for instance brokers like Kafka can extract log records from the database instead of reading an outbox table.

This pattern works but only offers at-least-once sending [cc1] – making it less reliable than our exactly-once sender pattern. This may seem like no big deal, but the consequences are huge for the consumers: they now need to implement the "idempotent consumer" pattern, discussed next.

In addition, this pattern implies that your developers are now responsible for implementing and maintaining and testing their own "message broker" infrastructure code. That may not be what you want (nor what they want).

Pattern: Idempotent Consumer

The problem

Let's revisit the problem of receiving messages reliably. How could you go about it when you don't have XA transactions available?

Example: Delivery Service

Let's look at a simple microservice that processes incoming "OrderCreated" messages as follows:

1. The incoming message is taken off the broker
2. The delivery data is assembled and inserted into the database

In the discussion of our "exactly-once receiver" pattern we have shown some possible anomalies that can arise for this case. I will not repeat the details here, rather let me just present a common pattern that people used before this book.

Solution: Idempotent Consumer

The idempotent consumer works like this:

1. The incoming message is taken off the broker

2. The deliver data is assembled and inserted into the database
3. The database changes are committed
4. The message is removed from the broker

More refined variations are but the essence stays the same. Failures between 3 and 4 will trigger redelivery of the message at a later time, so care is needed to avoid duplicate DB inserts / updates at 3.

Idempotent processing is a way of tackling this problem. Idempotent means: the result of doing something twice (or more) is no different from doing it once. So if we write our program such that step 2 is idempotent, then we are good to go.

The problem with this is: writing idempotent software is harder than it looks. It is what Google's Adwords team means with "complexity" (see the intro of this book). You can find some more details in this blog post[5].

Somewhat surprisingly, implementations of this pattern tend to be slower than with XA – where you avoid the need for idempotent processing in the first place.

The idempotent consumer is an essential pattern: many microservices depend on it. This book has shown you a simpler solution. So if you read this book, you will be able to simplify many of your microservices.

Pattern: Event Store

The problem

Let's revisit the problem of receiving messages reliably. If you assume that the consumer can lose messages, then how can it still get them back afterwards?
Example: Your Bank Account

Did you forget the balance of your bank account? If you consult the statements then you will see the chronological history of withdrawals and deposits. The end balance at any moment is simply what you get when you replay the history of operations on the account. Each operation is an "event" and the event store is this history – stored somewhere on stable storage.

Solution: Event Store

Instead of sending messages out via a broker, this approach stores messages in a database "log". This log keeps messages around for later reprocessing and can consume a lot of disk space over time.

Applications can (re-) process messages at any time by querying the event store. This allows restoring the system state at any time, but at a cost.

Needless to say, the event store adds some complexity. It also requires idempotent consumers to avoid side effects. This brings back all complexity discussed for that pattern.

[5] https://www.atomikos.com/Blog/IdempotentJmsConsumerPitfalls

Pattern: Saga

The problem

Let's revisit the problem of the transactional call (or of TCC for that matter).
Calling 2 microservices

Let's reconsider 3 microservices MS 1, MS 2 and MS 3:

1. MS 1 calls MS 2
2. MS 1 then calls MS 3

Suppose step 1 updates database of MS 2. Now what happens if step 2 fails and we can't retry? This can happen if step 2 encounters a fundamental business exception - where retrying simply won't work.

What can we do? If we can't go forward with step 2 then a nice alternative would be to "rollback" step 1. But how can we do that?

Solution: Saga / Undo

We define a "Saga" process that tracks the calls to each microservice. When "rollback" is desired, the Saga triggers "undo" calls to each relevant microservice (a bit like the "Cancel" in TCC).

Issues

There are some issues with this approach:

- You have to code the undo logic in each service and this can be expensive and complex (or even impossible) to get right – particularly in case of concurrent requests on the same data.
- Failed requests (both undo and regular ones) can leave the system in a state of doubt – generally requiring idempotent consumers and retries. Each of these add some more complexity to the mix.
- A service never knows if it will get a "Undo" request for a prior invocation, making it hard to do correct reporting queries (among other things).
- As always, duplicate messages and/or message loss make it much more complex than needed.

After reading this book, you now know you can resort to the "transactional call" or TCC instead.

If you're still interested in Sagas then you can get a feel of the complexity here[6]. I know I sometimes exaggerate, but I feel like it almost takes a PhD to understand the high-level diagram.

[6] https://github.com/Azure-Samples/saga-orchestration-serverless

Conclusion

Here we are! 5 simple patterns are what I wanted you to know: exactly-once sender, exactly-once receiver, exactly-once messaging, transactional call and TCC. If you understand these then you will have a good solution ready for every data consistency challenge in your microservices. No need to bother with the complex and error-prone techniques that most people are advocating. Use these 5 simple patterns instead!

What's Next?

Ready to put these patterns into practice? This book has a more elaborate companion course - available online and intended to help you implement the patterns covered here. It also includes some one-on-one time with me.

BONUS OFFER: As a reader of this book I can offer you a time-limited welcome offer of 80% off the official course price. To claim this offer, please go here[7].

Help me improve this book

Get a better version of this book - for free! I have setup a 1-question survey here[8]. It only takes a few seconds to complete, so please fill it out because it will help me improve this book (and if you bought it on Leanpub you will get every new edition for free).

[7] https://atomikos.teachable.com/p/microservice-transaction-patterns/?product_id=1840400&coupon_code=WELCOME
[8] https://www.surveymonkey.com/r/GVYPK9P