# Ensuring Eventual Consistency in a Microservices Architecture

## 1 ABSTRACT

How can the architecture of transaction oriented business systems be transformed to a microservices architecture in a “transaction safe” way, where it can be ensured that the system ends up in consistent state?

The foundation of this work is the CAP theorem[1],[2]. The CAP theorem states that it is impossible for distributed systems to simultaneously provide more than two out of the following three guarantees: Consistency, Availability, Partition tolerance.

With respect to the CAP theorem, microservice architectures are in the AP category, leaving the system to deal with eventual consistency.

This is a major issue to handle in some business systems, where database consistency is an absolute must.

One solutions is the use of the SAGA pattern, where compensating transactions handles inconsistency and eventually ensures consistency.

However, there is a serious issue with compensating transactions – they require transactions to be reversible. Yet, many business transactions are irreversible by nature – for example an automated teller machine transaction, where reclaiming the out handed money cannot be guaranteed to be honored.

This paper will introduce a way to ensure eventual consistency in a microservices architecture with irreversible transactions. The paper will promote using a dynamic CAP approach, where the CAP positioning of the system is dynamic based on the concrete business event. Also we will formulate a microservice architectural pattern where the transactional business events are encapsulated in modular micro monoliths.

The dynamic CAP approach and the use of modular micro monoliths has been formulated and developed as a joint effort between a Bankdata[3] and UCL University College[4]. Bankdata has a long history of handling eventual consistency with a proven record of success. The dynamic CAP approach is based on these concrete experiences and a more theoretical approach is taken from a CAP perspective.

## 2 INTRODUCTION

Microservices is a promising architecture that enables businesses to gain an increased IT agility, and keeping IT up-to-speed with the businesses rapidly changing needs. The key feature "Independent deployability" gives the different software development teams the freedom to deploy independently. Hench they are set free of a central release cycle. If fully utilized, the need of hotfixes will be eliminated, and the support cycle from bug report to deployed bug fix can be shortened close to the pure development time. Also new functionality can be released as soon as the development is done, and thereby minimize time to marked, speed up market capitalization and support a marked leader position.

However all these wonderful benefits comes at a price. Building and running microservices is nontrivial and requires new capabilities in the IT department since both implementing and running microservices adds considerable complexity to the IT environment.

As stated by Sam Newman: "*developers should only consider using microservices when they have a "really good reason" for doing so.*"[5]

Why does Sam Newman make this statement? From our perspective, it is closely connected to the issues with ensuring eventual consistency.

Before diving into solving the problems with ensuring eventual consistency, we will take a brief look at the key features of microservices, followed by looking at microservices from a CAP perspective, which will set the ground for the consistency discussion.

## 2.1 KEY FEATURES OF A MICROSERVICE

The upside of microservices features enabling the wonderful benefits is also the downside of microservices.

Key features of a microservice:

- Highly distributed and dynamic scalability
- Independent deployability
- High availability

**One of the key feature of a microservice is independent deployability**[6]. This leads to every microservice owning their own database. Sam Newman states it very clearly:

*"Don't share databases, unless you really have to. And even then do everything you can to avoid it. In my opinion, it's one of the worst things you can do if you're trying to achieve independent deployability."*[6]

At the same time, every distributed system is constrained by the CAP theorem (Brewer's theorem). The CAP theorem state that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

- Consistency: Every read receives the most recent write or an error
- Availability: Every request receives a (non-error) response, without the guarantee that it contains the most recent write

- Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

Since microservices by nature is highly distributed, the CAP theorem implies that due to the presence of a network partition, one has to choose between consistency and availability.

**Another key feature of a microservice is the need for high availability**. By nature, networks are unstable, and leads to the need for handling errors gracefully in order to keep the system operational [7], [8].

**The feature "Highly distributed and dynamic scalability"** is related to the scale cube[9]

From a system perspective, the system is functional decomposed (y – axis) around business capabilities[7], and performance scaling (x – axis) is done by scaling the microservices individually – for example by using Kubernetes[10].

The z – axis scaling is about data partitioning an at a macro level, it is related to the first microservice feature "every microservice owns their own database". However, at a microservice level, scaling is pure x – axis scaling, where the multiple instances of the same microservice uses the same database. If the scaling problems ends up being related to database load, CQRS and event sourcing and be introduced, but from a conceptual perspective – all instances of a microservice uses the same database.

## 2.2 THE CAP TEOREM

The CAP theorem sets the ground rules for distributed systems. The theorem states that you cannot have consistency, availability and partition tolerance at the same time. You must choose which one you will sacrifice. And since the very nature of microservices systems is that they are network distributed, it leaves you to choose between consistency and availability [1].

Since a microservice architecture can have thousands of "moving parts" (microservices), that must interact to fulfill the systems requirements, you really cannot accept non-availability. This leaves consistency as the only realistic parameter to sacrifice.

To wrap it up: With respect to the CAP theorem, microservice architectures are in the AP category, leaving the system to deal with lack of consistency.

# 3 DEALING WITH EVENTUAL CONSISTENCY

## 3.1 EVENTUAL CONSISTENCY

The lack of consistency in an AP category system means lack of strict consistency, not lack of consistency in general. It is a shift from ACID to BASE [11]

The “E” in BASE stands for Eventually Consistent.

*“Eventually Consistent – The fact that BASE does not enforce immediate consistency does not mean that it never achieves it. However, until it does, data reads are still possible (even though they might not reflect the reality).”*[11]

This means that eventually you will read the correct version of data.

This works if you do not have a transactional coupling between the databases. However, in many business cases you do have a transactional coupling between the databases. You may think that using two-phase commit could be used, but if one thinks closely, you will realize that using two-phase commit will compromise the AP positioning in CAP, since the availability of the participating services will depend on each other thus compromising the “A” (availability), and leading to a CP CAP positioning of the system. As Graham Lea puts it

> *Distributed transactions are icebergs: they can be hard to see, and they can sink your ship. [12]*

> *The solution to distributed transactions in microservices is simply to avoid them like the plague. [12]*

Thus – you need another solution, an using the SAGA pattern may solve your problem [6].

## 3.2 SAGA

> *The saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes compensating transactions that counteract the preceding transactions. [13]*

The SAGA pattern uses compensating transactions to handle inconsistency and obtain eventually consistency. The compensating transactions is like an ACID “rollback”. However compensating transactions is not the same as ACID transactions. It works more like the book keeping principle of invoice / credit memo. This means that you do not delete data. You change data or inserts canceling data. For example. If you have taken 5 items out of stock, and need to “rollback”, you either increment stock with 5, or inserts “5 added to stock”. The business consequence is that the stock value is not valid in the timespan between the first transaction and the compensation transaction, but will eventually be consistent (hence eventual consistency).

## 3.3 COMPENSATING TRANSACTIONS

Compensating transactions could seem to be the silver bullet, However, there is a serious issue with compensating transactions – they require transactions to be reversible. That means that you must be able to counteract preceding transactions.

In many cases, it is possible to design the business process in a way where avoid getting into an irreversible state. In these cases, the way to go, is to **design for compensating transactions.**

In a number of cases you can get inspiration from *"Reversal Adjustment…"*[14]

The problem is that some business transactions are irreversible by nature – for example an ATM machine transaction, where reclaiming the out handed money cannot be guaranteed to be honored.

### 3.4 DEALING WITH IRREVERSIBLE STATE.

There are no easy way out of this and to the best of our knowledge: The only way to handle irreversible transaction is to design the event flow in a way where you do not get into an irreversible state.

However, there are cases – for example in the banking sector – where a redesign of the business process is not possible nor feasible. For example, an account balance must be "ensured consistent".

**This leads us to the following statement:**

*With respect to CAP. In an AP system (the most common approach for microservice systems). You can only assure eventual consistency if the microservices support compensating transactions.*

**This leads to the next statement:**

*AP systems cannot assure eventual consistency if one or more microservices have irreversible states.*

How do we get out of this deadlock – we want to harvest the benefits from microservices, but we cannot allow loss of consistency?

# 4 DYNAMIC CONSISTENCY

We now know that a microservice architecture containing irrepressible transactions is "mission impossible"

The CAP theorem clearly states that you cannot have all three guarantees:

- Consistency:
- Availability:

- Partition tolerance

The only way is to revisit the system design and rethink the way we think about microservices. We need to think of the CAP positioning of the system as a dynamic thing – not at static decision.

So, if we re-think the way we think of the system and take a business-event perspective, we can identify the business events that goes down an irrepressible transactions path.

Once these business events are identified, we can establish a handling environment for these events only, where we sacrifice availability or partition tolerance for these specific business events. In practice, you will properly end up sacrificing availability.

The advantage is that you maximize harvesting the benefits for microservices by having a fraction of your business events suffering from potential lack of availability but leaving the rest of the system fully operational based on eventual consistency.

**This is what we call: ensured eventual consistency, using dynamic consistency**

# 5 DYNAMIC CONSISTENCY IN THE BANKING INDUSTRY

## 5.1 HISTORICAL BACKGROUND AND CURRENT SITUATIONAL AWARENESS

Historically, the banking industry was one of the first movers of IT. The earliest forms of digital banking trace back to the early 1960's in the advent of ATMs and card transactions. Suddenly a banking customer no longer needed a physical accountant to withdraw and deposit his financials – a machine was now able to service this need.

As early IT adopters throughout the 60s, 70s and 80s, many banks had (and many still have) large mainframe installations for executing digital processes and large number- and data-crunching calculations. The reason of mainframe-choice as primary compute power was simply because it occurred before the IT era of personal computers (PCs), web servers (as we know them today), virtual machine-farms and so on. There were simply no hosting alternatives for automation of processes which human hands previously used to perform manually.

In the advent of the internet (and leaping forward into the IoT and cloud-native eras) many banks had already developed huge amounts of source code measuring in two and three digit mLoC (million Lines of Code), thousands of applications and IT services through decades of software development. The IT solutions were often designed in batch-oriented perspectives primarily targeting employees which was the perfect solution in a closed environment before the internet era but troublesome in open eco-system perspectives as open solution design and integration with external parties suddenly became the norm. Exposing large system functionality to other audiences and in other contexts than for what originally was designed has many challenges. Among others were principal changes of authentication, authorization, role and access-management, underlying system design principles / database relations and general functionality usage. Another seemingly simple challenge also emerged with high-cadence agile practices – in which frequent changes were a must but where the system landscape was designed monolithically and required massive planning to do monthly/quarterly release trains the Production environment.

One part of the solution was to introduce a “digital platform” able to address the high cadence need of services and openness of functionality without compromising the main IT systems carrying the responsibility of the main banking “license of operate”-processes. Designed for Digital [15] identifies the bimodal need of having an O*perational Backbone* serving the main processes of running an enterprise while having a *Digital Platform* addressing the need of frequent delivery models, end-user collaboration and adoption of new technology – which in turn reinforces new business and service models. These concepts go hand in hand with Gartner’s Pace Layered Application Strategy [16] where systems are categorized into different abstraction levels (Systems of Record, Systems of Differentiation and Systems of Innovation) – in this context primarily governing system cadence, decoupling principles and software delivery speed.

Bankdata has as a financial software company some of the same historical challenges as depicted above. During the last few years Bankdata has invested heavily in enabling agile and DevOps practices throughout the organization, installing and utilizing state-of-the-art technology for development/operations and modernized the entire IT estate with microservice architectures in mind. Examining event driven architectures and patterns such as Publish/Subscribe is part of Bankdata’s corporate IT strategy towards 2025.

It is not a coincidence that event driven architecture is a part of Bankdata’s 2025 strategy. Patterns such as Publish/Subscribe are extremely useful for not only decoupling technology components but can also induce positive organizational impacts minimizing lead time and allow for efficient service delivery and software construction processes. Let’s examine the current situation.

Bankdata has several strategic development platforms – the important ones here are the mainframe and the container platform. The change cadence of these two platforms are different as are the skillsets and competences needed for developing and operating each. The development methodologies for each platform often also differ. Mainframe applications are coded in an imperative style and designed for speed and execution throughput whereas the container applications are designed with an agile mindset and towards microservice architectures.

As the mainframe in most cases holds the master system applications and master data records there is an obvious need of integrating from a container application to the core banking systems residing on mainframe. The current method of integrating is via synchronous calls over HTTP and gives a consistent worldview across the two platforms as the latest database entry is fetched at every request but induce some challenges.

Whenever a new business feature is developed a lot of data- and application-integration need to be planned / executed and this impacts not only the primary development team responsible for the feature but often incur changes on other systems (a new database field in another system, a webservice delivering new content, etc.). This can in an enterprise perspective induce organizational chokepoints and bottlenecks as teams developing a business feature often relies on other teams and their priorities. Certain software areas in a banking context are very core to many business features and thus the actual teams responsible for these are very popular in terms of new services and change requests. This tend to result in longer lead-time and teams waiting for other teams to develop, test and promote their changes to development and production environments.

Bankdata proposal is to include an event driven architecture across the two strategic platforms which upholds to consistency requirements while enabling differentiating products and innovative teams to execute a high degree of change. The initial proposal of using a platform-wide event driven architecture pattern combining mainframe workloads with technology agnostic container applications is the scope of the next few sections.

## 5.2 Types of Event driven architectures

There are many types of event driven architectures with multiple utilization patterns and many esoteric details are involved. This whitepaper doesn't address every specific detail and characteristic of each technology component nor all available patterns. This section focuses on the concepts and specific implementation in the context of a dynamic eventual consistency model for an enterprise event platform.

Before digging into technicalities, first let's investigate the main integration patterns involved.

### 5.2.1 **Integration patterns**

#### *5.2.1.1 Synchronous integration*

The synchronous integration pattern maps one solution to another normally via of a directed system call or via a request-response over-the-wire communication. Often implemented as SOAP or REST endpoints to functionality, this pattern integrates one system with another. It however also places design- and run-time constraints on the actors involved in these processes (developer teams, architects, portfolio planners, etc.). The coupling on design-time often occurs as more than one team needs to be involved in the system design and development of a specific business feature. The coupling on run-time occurs as the services are often dependent on each other impacting resiliency, uptime and other operational measurements.

#### *5.2.1.2 Applicational event-driven architecture*

The asynchronous communication pattern allows asynchronous integration between one technical domain, module/program or microservice to another. Often implemented by a local or platform-native message-broker this pattern enables decoupling of both microservices, internal services within a functional area / organization and coherent small sets of functionality to other parties within reach of the event broker.

#### *5.2.1.3 Enterprise event-driven architecture*

The asynchronous communication pattern between large IT systems and execution platforms. Often implemented by an enterprise service bus and event platform this pattern enables decoupling of large-scale IT complexes and platforms on a generic level for the purpose of organizational reuse, cost-efficiency and decoupling. Often utilized when ownership of systems are organizationally divided, when IT system complexes rely on different cadences in development speed, technology foundation and/or IT system criticality. This pattern is the objective of the subsequent sections and is planned implemented in Bankdata for increased organizational agility, cost-effective integrations and for re-use of enterprise services.

## 5.3 Concepts

### 5.3.1 Generic concepts in Publish/Subscribe

Publish/Subscribe is an integration pattern in which there is one ***publisher*** (also called provider or producer) and one or more **subscribers** (also called consumers). The publisher publishes (or pushes) **business events** (also called events or messages) to a **topic broker** mechanism which is responsible for delivering the events to every subscriber associated with the topic. The integration pattern is used for application- and platform-decoupling when applications are highly dependent on reading domain models and states from other systems.

The concepts are shown and explained in detail in figure 1 and in the sections below.

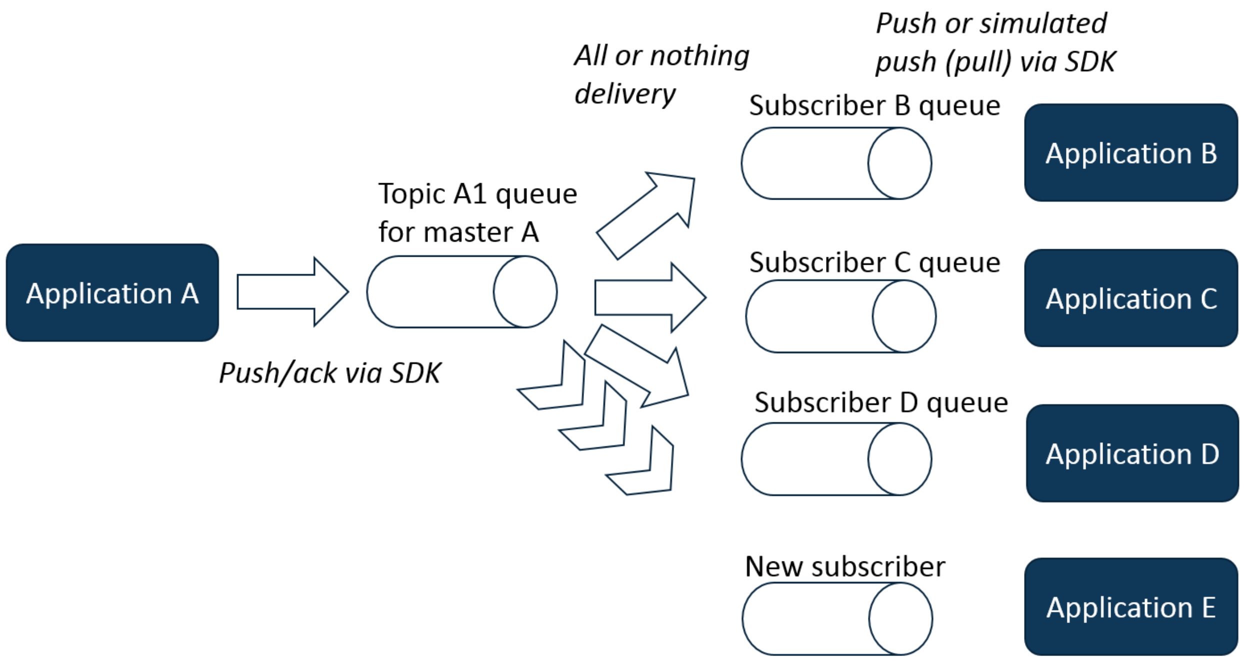


FIGURE 1 PUBLISH/SUBSCRIBE CONCEPTS

### 5.3.2 Publisher and master data ownership

The publisher (application A) is responsible for publishing events to subscribers when domain models change and for ensuring that application domain- and entity-models are covered by the topics offered. The publisher is also responsible for that the existing model is aligned with the topic events sent. In effect, this means that the event must adhere to the master tables schemas and the commit scopes of the applications in question. In Bankdata, the publisher owns the topic queue and must secure capacity management and monitoring of the same.

### 5.3.3 **Subscriber and slave data ownership**

The subscriber (application B, C, D and E) is responsible for offloading events sent by publishers and for making necessary changes in the subscriber domain before removing the event from the subscriber queue. In order of obtaining de-coupling, each subscriber owns its subscription queue and thus must secure capacity management and monitoring of the subscriber queue. It is not generally allowed for the subscriber to forward-delegate received events to other systems downstream as this can lead to cascading event-effects resembling distributed commits. The only entity which can publish *master* data-events are the master-data owner, but an application domain can undertake multiple roles in a complex integration, but may only publish their own master domain models/master data to others.

### 5.3.4 **Topics**

A topic contains a business event-representation and must be business-oriented and generally understandable from a business-perspective. The topic defines a structure for which events must adhere to (often represented in JSON format) and the structure and event granularity is owned by the publisher/master data-owner.

#### ***5.3.4.1*** *Types of delivery*

###### *5.3.4.1.1.1 At most once delivery [0-1 delivery]*

To secure availability some message brokers promise an “at most once delivery” meaning that there are no guarantees of delivery. This pattern is in use by certain social media platforms as it is not important in these contexts whether *all* your friends see your personal job-, birthday cake- or photo-update right away, in the right order or even at all. This is normally not applicable in a banking context as customers expect their financial statements to be relatively consistent and that the balance of your accounts is consistently maintained – a key principle in these scenarios. In some case e.g. streaming stock price-reads every microsecond for visualizing a price graph this delivery pattern can be applicable.

###### *5.3.4.1.1.2 At least once delivery [1+ delivery]*

Some message brokers promise an “at least once delivery” meaning that there is a guarantee that all events are delivered but the same event might be sent twice or more. This pattern is often used when multi-cast event propagation and horizontal scale-out is important but can introduce challenges at subscriber side. This is because the subscriber often needs to handle idempotency, identical events and event ranking which increases the complexity.

###### *5.3.4.1.1.3 Exactly once delivery [exactly 1 delivery]*

The message broker delivers events exactly once and guarantees an “all or nothing”-delivery scheme. This is the preferred delivery model in a banking context as it removes a lot of complexity and eases the effort needed to keep publishers and subscribers “in sync” while maintaining high degrees of decoupling.

#### ***5.3.4.2*** *A note on event granularity*

Getting the event granularity right is key to success when introducing event-based architectures for ensuring organizational reuse and keeping the enterprise information domain models intact. The event-granularity is a balance, often made erroneous, between fine- and coarse-grained information levels:

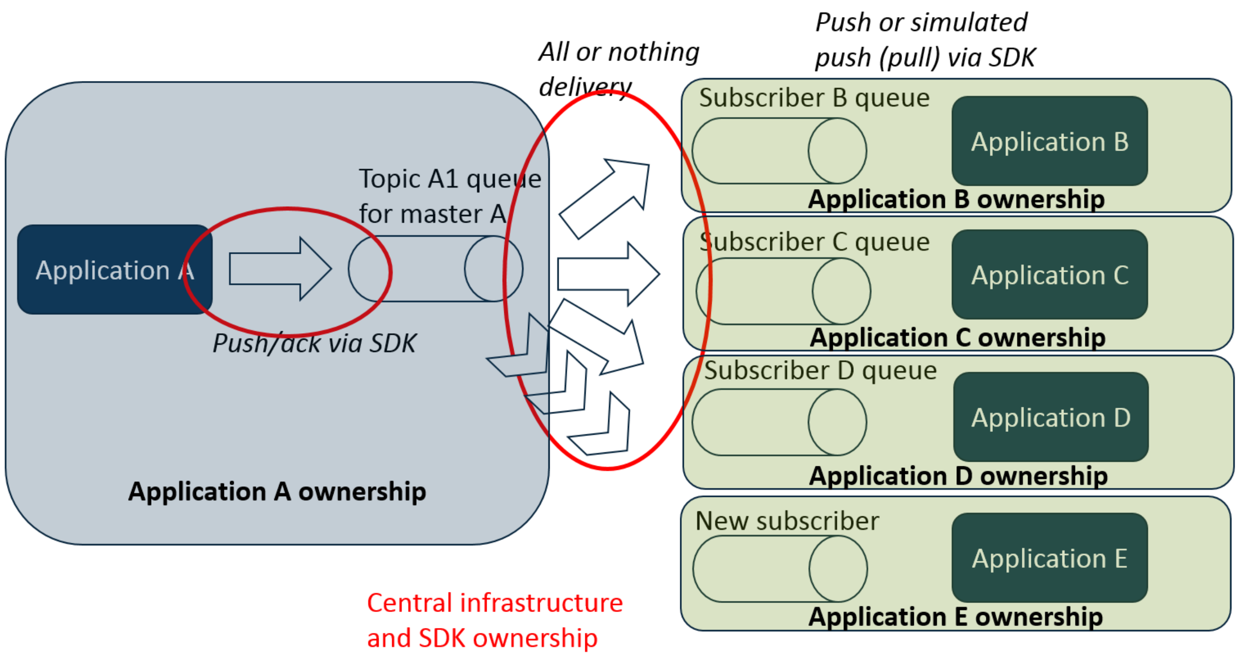
- Too fine-grained (or too specific) the event-reusability effect shrinks and subscribers need to subscribe to many topics to get the "full picture" of the domain model under change
- Too coarse-grained (or too generic) the subscriber needs to deduct and interpret the actual domain change (in effect this means that subscribers replicate both publisher-functionality and interpretation of data fields at master)

An example of a too fine-grained event could be *customerGenderChangedEvent* as this point to a very low level of reusability and low general applicability by subscribers (unless the domain of course involves gender changes of customers/users as its core functionality).
An example of a too coarse-grained event could be *somethingHappenedEvent* from the account domain as this would require interpretation to deduct what business event actually occurred (not to mention the number of attributes/fields needed to support this very general event notification).

Getting the balance of granularity just right is a very hard modelling task but as a rule of thumb if a) the event is a representation of a real-world business phenomena used in the application domain and b) if more than one subscriber is eligible for subscribing to it then there is a good chance that the event is an enterprise-wide business-event and “just-about-right” in granularity.

### 5.3.5 **Proposed reference architecture**

Figure 2 show the proposed reference architecture for the Publish/Subscribe integration pattern in Bankdata.



#### *5.3.5.1 Run-time step-by-steps*

1. Application A publishes events to a topic queue which contains the master-data events governing application domain A. On mainframe this occurs via IBM MQ and the publish itself is often implemented as a MQ PUT operation implemented via an SDK-enabled call to a library function to ensure a uniform and shareable format. The SDK is owned by a centralized department. The topic queue is owned by application A and A must secure that the database commit-scope follows the MQ commit scope as to align data on master to follow the publishing of the business event.

2. The topic queue then forwards the event to each subscriber in an "all-or-nothing" delivery-fashion. On mainframe via IBM MQ, the delivery can be implemented as a MQ ALIAS to the topic queue enforcing that the MQ PUT to all aliases either all succeed or all fail by use of standard IBM MQ return codes. In this context, IBM MQ acts as an *exactly once delivery* mechanism meaning that no event will be published twice and if an event is sent all listening subscribers will receive it. In this implementation, subscribers do not need to handle idempotency nor difference in ranking of events as IBM MQ is a FIFO-queue manager and events are ranked accordingly. Nevertheless, it is often a good approach to include a timestamp in the event to let subscribers know of new vs. old / obsolete events. The delivery association between topic and subscriber is owned centrally and documented in a software developer portal for increasing awareness and software/event reuse.

3. Subscriber B, C, D and E receives an event on their own local queue and functionality (such as onEventReceived) is triggered locally. This is often implemented once via an SDK listening to the local queue with a function/method call-back responsible for inspecting and offloading the event. Two methods are normally available.

    a. An onEventReceived event where the MQ client uses a MQ PEEK-like functionality allowing the event to be inspected without offloading it from the queue (in case of a subscriber-crash and/or unhandled exceptions in mid-transaction)
    b. An onEventHandled event where the MQ client uses an atomic MQ GET/COMMIT-like functionality where the event is offloaded the queue allowing the next event to be received (assuming a non-parallel synchronous offloading pattern).

## 5.4 KEY ADVANTAGES AND USE-CASES

### 5.4.1 **Decoupling**

In Publish/Subscribe, publishers should be unaware of subscribers which stands in opposition with point-to-point, request-response and bespoke integrations. This has multiple benefits, and a few are explained below.

#### *5.4.1.1 Organizational agility*

New subscribers should be able to be added to an existing topic without publisher development effort. This strengthen de-coupling of applications into isolated domains which in turn enhance and increase team autonomy.

#### *5.4.1.2 Re-use and portability*

As business events by default represent a business-oriented activity, it is also meant to be re-usable across contexts. Obviously, multiple versions of a business event should be able to co-exist allowing extended information models to be communicated without breaking former ones. Business events are also platform unaware (technological agnostic) and represented in text-form (often JSON) giving integration possibilities to all types of platforms which in turn support high portability possibilities for instance to cloud workloads.

#### *5.4.1.3 Operational availability and outages*

When outages occur, request-response integrations tend to be highly dependent which in turn means that down-stream systems also fail under the outage impact. In Publish/Subscribe, subscribers are not read-wise impacted during publisher outages so this integration pattern minimizes the blast-impacts outages – e.g. subscribers can still function with the domain-model held locally. Obviously, the subscribers are still impacted by the ability to update the system under outage in write-scenarios where a master system is down. If a specific subscriber has an outage this does not impact other subscribers nor the publisher as the subscriber has a local queue and if within queuing capacity the subscriber can offload the events with the outage has been resolved. If the queuing capacity is exceeded, the subscriber in question is solely responsible for getting back “in sync” via a full event bootstrap or re-requesting the original domain model from the publisher.

### 5.4.2 **Cost reduction**

In Bankdata most master applications currently reside on mainframe. The synchronous integration model dictates that de-central applications must access via SOAP/REST-realized webservices or CICS-endpoints. This model is very proven and secure but have additional drawbacks on the cost-effects. Mainframe MIPS are expensive and de-central applications need to enquire read-states at every request and enforce changes upon the master domain-model via direct integration. This model also scales poorly in terms of price because increase in utilizations (addition of new customers, products or channels) creates a more-than-proportional increase in MIPS as the price-model often is expressed in peak-load usages.

As a proposed asynchronous integration model, Publish/Subscribe will allow each subscriber to hold a persistent representation locally making repeating enquiries on read-states at master unnecessary as the subscriber holds enough information within his own domain context-boundary. Obviously, the representation needs to be bootstrapped once to allow initial data load but from that point and onwards synchronizations from master over MQ will happen in near real-time with each publish costing approximately O(1) as only one additional MQ operation is needed in the application commit-scope.

### 5.4.3 **Good use-case candidates**

#### *5.4.3.1 Low volatile data*

Applications with many reads and few writes are especially good candidates for allowing slave-representations and asynchronous integration over Publish/Subscribe. As data is persisted locally no read from master is necessary and thus data which change very rarely are prime candidates for this pattern usage as very few publish-operations are actually performed. Examples from the banking world are domain object such as accounts (an account rarely changes but financial statements associated with it though does) and customer information (a customer’s name and address also rarely change).

#### *5.4.3.2 COTS products and applicational eco-systems*

Commercial-of-the-shelf (COTS) and third-party products have their own data-representation by default. Thus, these products are highly eligible for integration via Publish/Subscribe as they require own data-models provided by the contractor/vendor. Specialized integrations often also have very low level of reusability (bespoke one-to-one integrations to this specific product) and are expensive to implement/operate in a total-cost-of-ownership perspective.

## 5.5 SOLUTION DESIGN ELEMENTS

### 5.5.1 **Where to implement event publishing**

Publishing of events can occur at many levels. In a SOA programming hierarchy, the options often include the database layer, business layer and presentation layer. There are different tradeoffs for each choice.

#### *5.5.1.1 Database layer*

Publishing of business events on the database layer is often very easy to implement and can be realized through simple database-replication mechanisms (such as database-triggers or even QREP on mainframe). Additionally, all SQL processes working directly on the database layer are handled automatically so that an update process (SQL tooling, stored procedures, SPUFIs, etc.) will end up sending events to subscribers. However, implementing the event-publishing in this layer quite often ends up with all subscribers needing to understand the publishing domain in order of interpreting the events which scales poorly and was the opposite of the main objectives (reuse and organizational agility)

#### *5.5.1.2 Data-access layer*

Publishing of business events on the data-access layer (that is: in code near the SQL statements). This has an upside that not all changes to master-data objects necessarily need to become a published event. However, a large interpretation of the actual business change is still needed at subscriber-sides. In this model, direct database changes (bypassing the code) is no longer permitted as it will in make the publisher and subscribers world-views inconsistent.

#### *5.5.1.3 Business layer*

Publishing of business events on the business layer is in the author's opinion the best fit choice. Here the collection of downstream functionality and shared commit-scopes are gathered and can easily express the event details – before returning upstream with an indication of a successful service/method invocation (http 200, return code 0, etc.).

### 5.5.2 **Data in events vs. notifications only**

In the former sections we have not touched on a small but important detail. Does the actual event hold specific domain/data information or does the event simply indicate that the subscribers should bootstrap a domain entity again? Both variants are equally valid but have different tradeoffs. The author's opinion is to include the relevant data in the event if the security, operational and governance processes and company policies allow it. It will make the solution code more clean and remove operational hurdles / runtime dependencies. It will also remove a lot of technical issues and architectural discussions on the concepts of "time" (e.g. what if a subscriber receives an event without data and the data-source for re-reading the object at master suddenly becomes unavailable right after the event have been received? How should the subscriber react in this case?).

## 5.6 RISKS, THREATS AND CHALLENGES

### 5.6.1 **Eventual consistency**

Publish/Subscribe lives on a foundation of eventual consistency as there is an implicit time-frame delta in which master is updated and subscribers are notified. This time-span is normally in the range of milliseconds (<10 milliseconds) but there is a need for a transparent business decision for whether asynchronous integration is an eligible pattern for the specific system. Most banking systems can actually live with this small time-span inconsistency but some, such as critical real-time trading systems, cannot. The systemic de-coupling, organizational agility and cost-reduction very often out-weights the disadvantages but in an enterprise the decision resides at the business side.

### 5.6.2 **Slave-representations of data**

As each subscriber often owns its own representation of master data, the subscriber is responsible for upholding to the data governance rules-of-conduct of the master system. It is very important to understand that Publish/Subscribe is not a database replication pattern but solely an integration mechanism of distributing a shared view on a centralized, de-coupled information-domain-model to peers. As such, business events do not normally reflect each database-field from master as the information needed in each context is naturally different. If one or more subscribers of some reason get out of sync or misinterpret business events from master, it is the subscriber's responsibility of getting back into a consistent world-view which of course enforce the master system to deliver robust service-handles in order of being able to do so.

### 5.6.3 **Delegated and cascading events**

Subscribers are permitted to have a local view of other's master data, but it is an anti-pattern to re-delegate master events to other peers. As mentioned earlier the reasons for not allowing this are many but most importantly are

- It creates an ownership model which is not consistent
- It requires that a specific subscriber is available for receiving master messages to send these further downstream
- if notified by both a data slave and a data master – which is then "most correct" – the newest, the master or a combination?

In short – don’t delegate, cascade or resend events for data and objects you do not have master ownership of.

# 6 THOUGHTS ON POTENTIAL PATTERNS FOR DYNAMIC CONSISTENCY

Inspired by:

- The way Bankdata has solved the problem with eventual consistency
- Microservices architecture
- Domain Driven Design
- Modular Monoliths

We purpose a pattern for implementing dynamic consistency.

## 6.1 ENSURING EVENTUAL CONSISTENCY USING A HYBRID ARCHITECTURE OF MICRO-SERVICES AND MICRO MONOLITHS.

We need to step back and think about why Sam Newman says:

As stated by Sam Newman: “*developers should only consider using microservices when they have a “really good reason” for doing so.*”[5]

One of the nice things about the monolith, is strict consistency (Aka ACID) is not a problem. But how could you keep strict consistency where is it needed, and still benefit from using a microservices architecture?

### 6.1.1 The modular micro monolith

We already talked about identifying the business events that requires strict consistency, but how do you proceed from that point. A possible process could be:

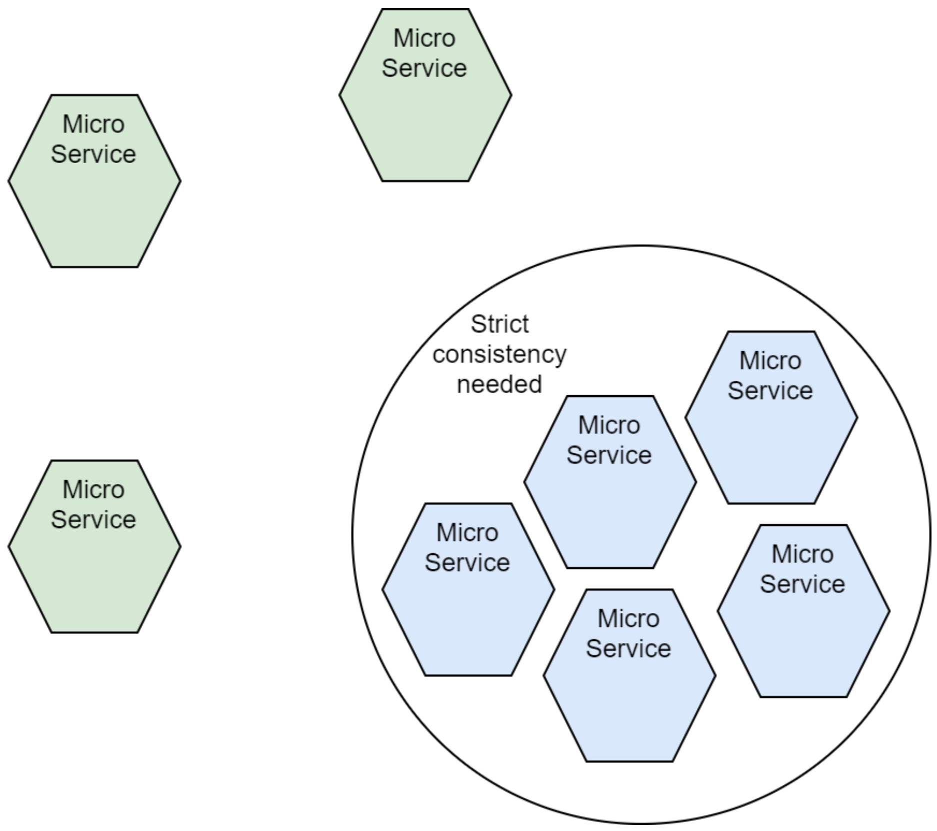
1) Start with an event storming session, where you have identified the business events.
2) Use the DDD concept of bounded context to organize the identified events into potential microservices.
3) Group events from a consistency perspective, hawing a pool of events that lives happily with eventual consistency and a number of groups where events that needs to participate in a strict consistency scheme.
   a) The eventual consistency group:
      i) Use the DDD concept of bounded context to organize the identified events into potential microservices.
   b) The strict consistency group:
      i) Use the DDD concept of bounded context to organize the identified events into potential microservices.
      ii) Use the consistency boundaries to bundle the potential microservices in such a way that all events of such an event group are in the same bounded context.
      iii) Assemble the potential microservices as modules in a modular micro monolith.

Step 1 and 2, is common practice when breaking a system up in microservices.

The ensuring of eventual consistency comes from step 3, where the architecture is transformed from a microservice architecture to a hybrid architecture of microservices and micro monoliths.

Lets look deeper into the process.

Figur 6-1 Grouping microservices by events shows the grouping of microservices by events that needs strict consistency.



FIGUR 6-1 GROUPING MICROSERVICES BY EVENTS

This way of splitting up your system will lead to a number of bounded contexts with a sharp business focus, that are candidates to be microservices. And groups of microservices candidates that has a transaction coupling (the ones holding consistency scoped events).

Each of these groups needs to be contained in a transaction scope, but using distributed transactions in a microservices architecture is bad practice.

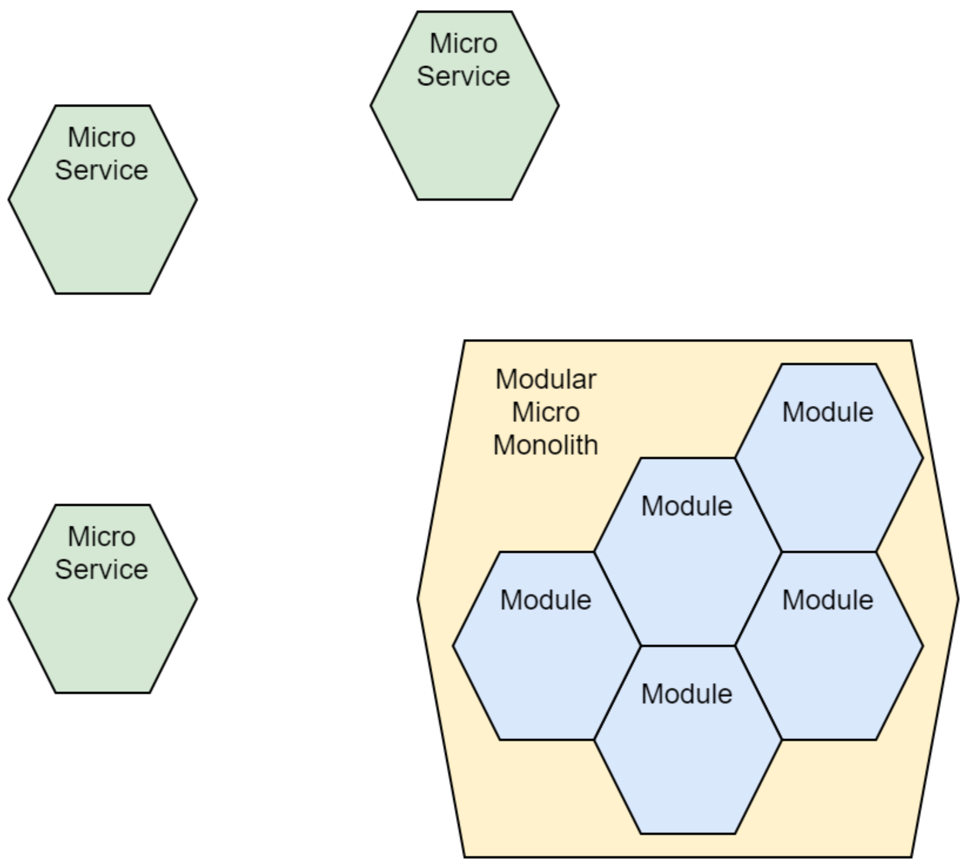
> *Distributed transactions are icebergs: they can be hard to see, and they can sink your ship. [12]*

*The solution to distributed transactions in microservices is simply to avoid them like the plague. [12]*

Our solution to this problem is to “merge” the microservices in a transaction scope into a monolith, but not just any monolith. We still want to gain maximal advantages from the microservice architecture. Therefore, we assemble the microservices in a transaction scope as modules in the monolith. Also we scope the monolith from the transaction scope, keeping the monolith as small as possible. We call these small transaction scoped monoliths *modular micro monoliths*.

Another way of saying it. You bundle up microservices that needs to share transaction scope into one big microservice.

Figur 6-2 The modular micro monoliths illustrates the concept.



FIGUR 6-2 THE MODULAR MICRO MONOLITHS

By using this pattern, we have encapsulated the problem of dealing with irreversible business transactions, in an architecture that comes very close to being a microservice architecture. The cost of this is several modular micro monoliths where your system is not perfectly separated in independent services. In that way, the price you must pay is reduced to shared deployment of the modules contained in the same modular micro monolith.

## 7 CONCLUSION

The architecture of transaction oriented business systems can be transformed to a microservices architecture in a "transaction safe" way, where it can be ensured that the system ends up in consistent state.

This can be achieved by using a dynamic CAP approach, where the CAP positioning of the system is dynamic based on the concrete business event, where the irreversible transactional business events are encapsulated in modular micro monoliths.

## 8 BIBLIOGRAPHY

[1] "CAP theorem", *Wikipedia*. dec. 25, 2019. Set: apr. 22, 2020. [Online]. Tilgængelig hos: https://en.wikipedia.org/w/index.php?title=CAP_theorem&oldid=932419887

[2] "Brewer's CAP Theorem <= :julianbrowne". https://www.julianbrowne.com/article/brewers-cap-theorem (set aug. 05, 2021).

[3] "Bankdata.en". https://www.bankdata.dk/en (set jun. 18, 2021).

[4] "Apply for one of our Full Degree Programmes at UCL". https://www.ucl.dk/international (set jun. 18, 2021).

[5] "When to Use Microservices: Sam Newman and Martin Fowler Share Their Knowledge", *DreamFactory Software- Blog*, maj 04, 2021. https://blog.dreamfactory.com/when-to-use-microservices-sam-newman-and-martin-fowler-share-their-knowledge/ (set jun. 18, 2021).

[6] S. Newman, *Monolith to microservices: evolutionary patterns to transform your monolith*. 2020. Set: apr. 22, 2020. [Online]. Tilgængelig hos: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=2316576

[7] "Microservices", *martinfowler.com*. https://martinfowler.com/articles/microservices.html (set dec. 01, 2020).

[8] "Making Your Microservices Resilient and Fault Tolerant - DZone Microservices", *dzone.com*. https://dzone.com/articles/making-your-microservices-resilient-and-fault-tole-1 (set dec. 01, 2020).

[9] "The Scale Cube". https://microservices.io/articles/scalecube.html (set dec. 01, 2020).

[10] "Running Multiple Instances of Your App | Kubernetes". https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/ (set dec. 01, 2020).

[11] "ACID vs. BASE: Comparison of Database Transaction Models", *Knowledge Base by phoenixNAP*, nov. 25, 2020. https://phoenixnap.com/kb/acid-vs-base (set jun. 22, 2021).

[12] "Distributed Transactions: The Icebergs of Microservices • Evolvable MeEvolvable Me". https://www.grahamlea.com/2016/08/distributed-transactions-microservices-icebergs/ (set dec. 01, 2020).

[13] fernandoBRS, "Saga distributed transactions - Azure Design Patterns". https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga (set dec. 01, 2020).

[14] "Reversal Adjustment", *martinfowler.com*. https://martinfowler.com/eaaDev/ReversalAdjustment.html (set maj 31, 2021).