Microservices: Migration of a Mission Critical System

Manuel Mazzara*, Nicola Dragoni†, Antonio Bucchiarone ‡, Alberto Giaretta §,
Stephan T. Larsen ¶, and Schahram Dustdar‖
* Innopolis University, Russia, m.mazzara@innopolis.ru
† Technical University of Denmark, Denmark, and Örebro University, Sweden, ndra@dtu.dk
‡ Fondazione Bruno Kessler, Italy, bucchiarone@fbk.eu
§ Örebro University, Sweden, alberto.giaretta@oru.se
¶ Danske Bank, Denmark , stephantl@gmail.com
‖ TU Wien, dustdar@dsg.tuwien.ac.at

Abstract—An increasing interest is growing around the idea of microservices and the promise of improving scalability when compared to monolithic systems. Several companies are evaluating pros and cons of a complex migration. In particular, financial institutions are positioned in a difficult situation due to the economic climate and the appearance of agile competitors that can navigate in a more flexible legal framework and started their business since day one with more agile architectures and without being bounded to outdated technological standard. In this paper, we present a real world case study in order to demonstrate how scalability is positively affected by re-implementing a monolithic architecture (MA) into a microservices architecture (MSA). The case study is based on the *FX Core* system, a mission critical system of Danske Bank, the largest bank in Denmark and one of the leading financial institutions in Northern Europe. The technical problem that has been addressed and solved in this paper is the identification of a repeatable migration process that can be used to convert a real world Monolithic architecture into a Microservices architecture in the specific setting of financial domain, typically characterized by legacy systems and batch-based processing on heterogeneous data sources.

Index Terms—Service Computing, Software Architecture, Scalability, Microservices

1 Introduction

The history of software architectures has been characterized in the last few decades by a progressive shift towards distribution, modularization, and loose coupling. The main purpose is increasing code reuse and robustness [1], a necessity dictated by the need of increasing software quality, not only in safety and financial-critical applications, but also in more common off-the-shelf software packages.

In service-oriented architectures [2], the emphasis was on cross-boundaries inter-organization technology-agnostic communication, and on orchestration of business processes [3]. The research community dedicated significant attention to foundational aspects, such as correctness and verifiability of service composition [4]. Nonetheless, little effort was spent on defining the nature of the internal logic of services, on scalability and maintainability issues, concerns of major importance for modern organizations.

The latest step in this process is the *microservice architecture (MSA)*. Inspired by service-oriented computing, MSA aims to change the way in which software is perceived, conceived, and designed [5]. A number of programming languages based on this new paradigm are emerging. As an example, Jolie [6] allows describing computation from a data-driven perspective, instead of a process-driven one [7], and introduces as *first-class entities* concepts that are fundamental to microservices [8].

The shift towards MSA is a sensitive topic these days, as several companies are deeply refactoring their back-end systems, which is the case of the institution considered in this paper (i.e., the *FX Core* of Danske Bank). MSA paradigm relies upon simple principles [5]:

- *Bounded Context:* first introduced in [9], this concept captures one of the key properties of MSA: focus on business capabilities. Related functionalities are combined into a single business capability which is then implemented as a service.
- *Size:* this represents a crucial concept for microservices and brings major benefits in terms of service maintainability and extendability. Idiomatic use of a MSA suggests that if a service is too large, it should be refined into two or more services, thus preserving granularity and maintaining focus on providing only a single business capability.
- *Independency:* this concept encourages loose coupling and high cohesion by stating that each service in MSA is operationally independent from others, and the only form of communication between services is through their published interfaces.

The MSA style enables to handle scalability almost out of the box, since that many of the techniques and principles used are inherently beneficial to scalability. Such characteristics have firstly been introduced in [10], although no practical case study has been considered in that contribution.

In this paper, we consider a real world case study concerning the migration of a mission critical system from an existing monolithic architecture (MA) to MSA, i.e., the *FX Core* system of Danske Bank, the largest bank in Denmark and one of the leading financial institutions in northern Europe. The contribution of the paper is threefold. First, we highlight the key technical aspects that need to be considered for full system scalability in the microservice context. Second, we show how a real world MA can be converted into a MSA, and we highlight the resulting benefits of this migration. Finally, we study in detail how scalability has positively been affected by this paradigm transition. To the best of our knowledge, this is the first fully and publicly documented real world migration of a mission-critical MA to MSA.

Outline of the Paper. The paper is structured as follows: Section 2 summarizes the related work in the monoliths migration topic, Section 3 discusses the technical aspects to consider, in order to exploit the scalability potential of microservices. In Section 4 the Danske Bank *FX Core* system functionalities are described, as well as its general structure. The legacy MA is then presented in Section 5, while the proposed MSA appears in Section 6. The comparison between the two architectures is detailed in Section 7. Section 8 concludes the paper with the lessons learned and open research challenges.

2 Related Work

Since the 2014, as shown by Balalaie et al. [11], MSA has steadily grown as a concept, and plenty of businesses decided to migrate their MA and service-oriented architectures (SOA) to MSA. Taibi et al. [12] conducted an empirical investigation through interviews to experienced practitioners, and outlined three migration processes adopted. Among the motivations for migration, both maintainability and scalability are the common ones. Unsurprisingly, the main issue was the monetary expenditure.

In another study, Knoche and Hasselbring [13] report that discussions with practitioners highlighted that industry looks at MSA as a promising way to solve maintainability issues, even in those cases where scalability is not a critical priority. The authors provide a decomposition process to achieve an incremental migration, which is the most common approach, and argue that for critical deployments it is better to implement new functionalities within the MA, and then incrementally migrate all the services, starting with the clients applications.

Di Francesco et al. [14] conducted another empirical study, similar to the one conducted by Taibi et al. [12] putting greater focus on the details of each migration phase, as well as analyzing migrations both from MA and SOA. In particular, the work shows that more than half of the migrations did not migrate the existing data together with the architecture. The authors argue that this does not align well with two microservices typical principles: hiding internal implementations details, and managing data in a decentralized fashion.

In 2015, Levcovitz et al. [15] proposed a technique to identify, within monoliths, service candidates for migrating to microservices based on mapping the dependencies between databases, business functions, and facades. To the best of our knowledge, apart from our work, this is the only publication that discusses migration techniques applied to a specific banking case study. Moreover, while their approach aims to automatize the migration from the legacy monolith, our case study was primarily business-driven. Therefore, our approach had to be necessarily manual and iterative.

Again, Balalaie et al. [11], [16] reported their experience of performing an incremental migration of a mobile backend as a service (MBaaS) to microservices, coupling with DevOps methodologies. The authors caution *a posteriori* about two important lessons learned. First, migration can introduce small errors in service contracts that can potentially break down a substantial part of the architecture. Second, a MSA is not a silver bullet, as it can bring scalability to services, but it can introduce higher complexity as well.

Following the previous works, the authors collected and reported some empirical migration patterns derived from medium to large-scale industrial projects, aiming to help others to perform a smooth migration [17]. They evaluate such patterns through qualitative empirical research, and cite as future work the development of a pattern language that would allow to automatically compose the patterns.

In a recent work, Furda et al. [18] agree on defining the migration to MSA a promising way to modernize MA and to full-scale utilize cloud computing. Moreover, they identify three major challenges in migrating a MA to MSA, namely: multitenancy, statefulness, and data consistency.

3 MSA and Scalability

Proponents of MSA claim that *per se* this style increases system scalability. However, it is necessary to pay particular attention to certain technical features, to fully enable its potential. This section covers all the aspects that need to be taken care of, in order to achieve full scalability: *automation, orchestration, service discovery, load balancing, and clustering*.

Automation. In a MA, at times it is possible to manually manage the system and the hosts on which it is running. However, as soon as the system scales, the number of hosts may increase leading to a hard-to-maintain system. This applies to MSA too, as services are scattered across multiple hosts, with each one running multiple services. Manually managing a MSA would result in an enormous overhead, since deployment, configuration, and maintenance extends to each and every service instance and host. Every time a new service or host is introduced, the system requires an increasing amount of time for manual management. When standard management activities (i.e., builds, tests, deployment, configuration, host provisioning, and relocation of services) are automated, the introduction of new services does not imply a management overhead. Only maintenance of scripts is required, and developers are expected to manage all the system via automation. The bottom line is automation of growth-sensitive tasks, in order to contain the time overhead.

Orchestration. In MSA, orchestration is necessary for managing service containers and infrastructure. Without an orchestration system, engineers would have to develop and maintain themselves a number of necessary features for a large scale system. Open source orchestration systems such

as Google *Kubernetes* [19], Mesosphere *Marathon* [20] and Docker built-in *Swarm Mode* [21] all provide a number of features which are necessary to achieve scalability, such as *service discovery*, *load balancing* and *cluster management*. Orchestration systems also handle replication of services and distribution of replicas across the nodes.

Service Discovery. Widespread diffusion of the traditional SOA failed due to fundamental shortcomings related to service discovery and, in particular, dynamic binding and invocation [22]. Microservices and orchestration tools are trying to overcome these issues.

MSA consists of many services, and a mechanism has to be deployed to keep track which instances are running and how to reach them. This is typically done with a *service discovery* tool, either a separate service such as *Consul* [23], or as part of the aforementioned orchestration tools. Service discovery provides more than simple DNS lookups, it also includes health-checking mechanisms that ensure that the services it resolves names to are actually alive and available.

Service discovery is a must in MSA, since services do not have static IP addresses and require a mapping from a hostname. Service discovery can make use of *locality*, resolving hostnames to the service instance that is closest to the requester, hereby achieving *geographical scalability*. Service discovery also creates the illusion of interacting with a single service, although a sequence of requests actually might be handled by multiple service replicas.

Load Balancing. *Load balancing* is critical for service discovery, necessary to ensure that load is equally balanced across service replicas. This can be done in a number of ways, such as by using DNS mechanisms to resolve hostnames, or looking up for a different replica IP each time. The latter option can be achieved through a *dispatcher* server that hands out replicas IPs (based on some scheduling algorithm), or by appointing clients themselves to decide which replica should be used. Typically implemented as part of service discovery and orchestration, if services integrate via a messaging system, distributing messages and events to different replicas can help to distribute load.

Clustering. MSAs can be deployed on a single host, but this would not contribute to scalability. To enable full scalability, deployment has to happen on a cluster of hosts. Clustering enables a system to utilize multiple hosts resources as a single system. It also enables elasticity in the form of expansion with additional hosts when needed and decrease of hosts when not. This may be achieved without clustering, but it would require the entire system to run on each host, like vertically scaled MAs.

Clusters can be configured and run with a variety of tools but, if containerization is used, it is typically part of the orchestration. Orchestration tools allow services and replicas to spread across the cluster (while ensuring that they can reach each other), enabling higher availability, increased resilience and better load scalability.

Running services in a cluster, requires them to either run actively in parallel or, in the case of infrastructure and data storage, to use clustering mechanisms in order to collaborate. These clustering mechanisms differ depending on their requirements to performance, consistency, availability etc., but are typically included in scalable messaging systems, such as *RabbitMQ* [24], and databases such as *Redis* [25].

4 Danske Bank *FX Core* system

The Danske Bank *FX Core* system is a paradigmatic case study to demonstrate how to effectively migrate from a MA to a MSA, and how this affects scalability. The documentation of the original system architecture was sparse and the vast majority of technical details have been obtained by direct conversations, interviews and discussions with the *FX Core* team, and by manually inspecting the source code. This was a lengthy process given the complexity of the MA. The outcome of this process is reported in this paper, where we describe the system in terms of responsibilities and organization. All confidential information, such as concrete names of protocols, external providers and specific services has been withheld in order for the results to be published.

4.1 Foreign Exchange

Foreign Exchange, often abbreviated as *forex* or *FX*, is the conversion from one currency to another. Exchange of currencies is of interest to private individuals, corporations, financial institutions, and governments. *FX* encompasses everything, from private transactions performed in foreign countries (e.g., use of credit cards while traveling), to corporations working with foreign markets. *FX* has become the largest financial market in the world, averaging a daily transaction volume of roughly 5 trillion dollars, with single transactions reaching the 100 millions of dollars. Unlike the centralized stock exchange, *FX* is decentralized and pen 24 hours a day, five days a week [26]. Transactions happen *over-the-counter (OTC)*, which means that traders (typically large multinational banks) negotiate directly with each other.

4.2 FX IT

The *FX IT* (Figure 1) system is part of the banks *Corporates and Institutions (C&I)* department and handles price streaming, trades, line-checks, and associated tasks, such as analytics and post-trade management. *FX IT* acts as a gateway between the international markets and Danske Bank clients, mainly large financial institutions and multi-national corporations. Such institutions continuously process currency prices and calculate margins to reduce important risks on *swaps* and *forwards*, before streaming final prices to their clients. Then, clients can act on a price by registering a trade, or check if they have the required collateral with line-checks.

4.3 FX Core

The *FX Core* system is part of *FX IT* and it handles trades and line-checks. This includes registration, validation and post-trade management. Below there is a brief description of the two main responsibilities of *FX Core*.

LineChecks are used to check whether a client has the financial collateral to perform a trade and how a trade will affect said collateral, also called their *Line*. This collateral can be a multitude of financial assets, e.g. stocks, bonds or cash. Line-checks are always executed as part of a trade, but is also run separately, so Danske Bank traders can ensure that their customers are capable of requested trades.

Trades are received from both Danske Bank clients and *external providers*, i.e., external clients and markets. The trade is then validated and line-checked, before being registered.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2889087, IEEE Transactions on Services Computing

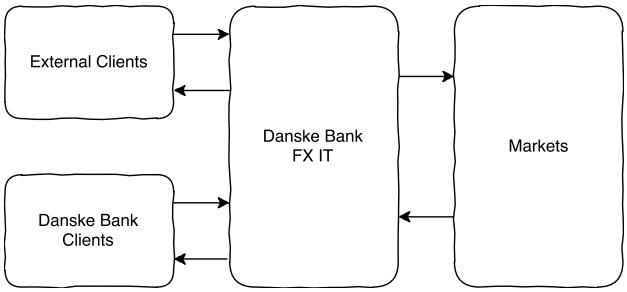

Fig. 1. FX IT handles both price streaming and requests for trades and line-checks from global markets. Prices of currency pairs are streamed to FX IT, which then calculates prices of specific trades, before streaming them to external and internal clients. Clients can request FX IT for trades or line-checks on the prices they have received. Clients are usually Danske Bank internal traders and external customers, but trade and line-check requests can also be received from the markets, when banks wish to exchange currencies directly. *FX Core* is part of *FX IT*, but handles tasks associated with trades and line-checks, thus not handling any of the price streaming and stream processing.

Depending on the type of trade, the trade is either performed immediately, (i.e., a *spot trade*), or registered in the system as a contract for future execution (i.e., *swaps* and *forwards*). When the trade is executed, financial assets are moved between banking books. After a trade has been registered, a number of actions can be executed: trades can be joined to ease administration or split into smaller ones to reduce margins, *forward* and *swap* contracts can be extended or pre-settled, and trades can be corrected or deleted by internal clients. Additionally, the system can also run batch jobs in order to balance books between departments, or to analyze trades.

5 FX Core Monolith

In this section, in order to evaluate the benefits of a MSA *FX Core* implementation, we cover the old MA. This includes an overview of the architecture, how it copes with scaling, the scalability techniques applied, and the related achievements. We also depict other problems, non-related to scaling, that motivated the system redesign.

5.1 Architecture

Danske Bank monolithic system was in part already service-based, as it can be seen in Figure 2. The system copes with scale in a variety of different ways. The services are deployable individually, and are actually already replicated and deployed across a cluster. The system also utilizes APIs as interfaces for clients to interact with the services of the system, and a messaging system to delegate received requests from external providers. At a first sight, it looks like an ideal and scalable solution. However, Danske Bank has experienced severe challenges when trying to rapidly develop the system and deploying consistent changes, and in general in handling system complexity. We will describe here systems components, how they integrate and how they are deployed.

5.1.1 System Components

The MA, shown in Figure 2, is componentized in a variety of ways. The system utilizes both services, shared software

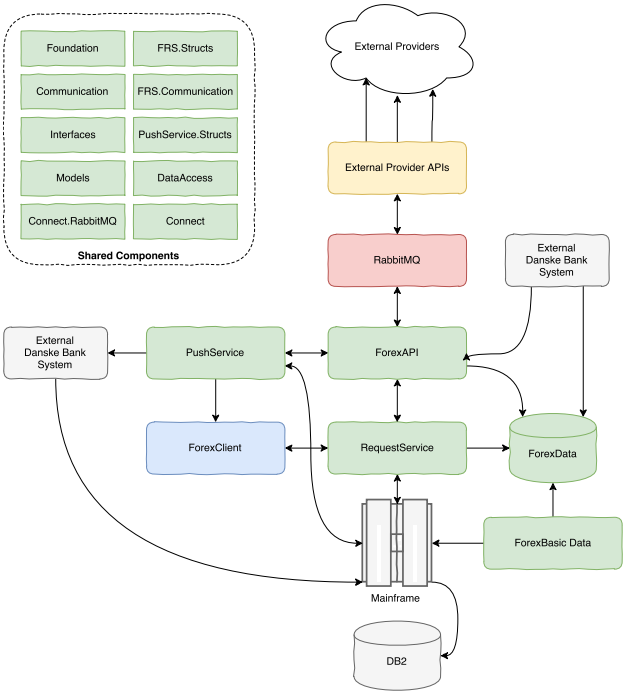

Fig. 2. Danske Bank MA. Red services are infrastructure services, green are part of the monolith, blue is the client, yellow are external provider APIs and grey are external Danske Bank systems. The external provider APIs are part of the monolith and consist of multiple services, with each one connecting to a different provider. Their names have been excluded due to confidentiality. The ForexData database is one big monolithic MS SQL database, shared amongst many of the monolithic components and also accessed by external systems

libraries and thick desktop clients. This section will briefly cover each of these components, their type and their responsibilities, in order to give an idea of how functionalities and data is distributed across the system. The thick clients will not be covered, as they are not going to be replaced by the MSA, but simply be updated to interface with it.

External APIs. The external APIs integrate with external providers of trades and line-checks, not mentioned explicitly due to confidentiality reasons. APIs have open TCP sockets to communicate with external providers. The APIs receive requests for trade and line-checks from different providers, each with their own trading protocol, and feed them into the system via the messaging queues in *RabbitMQ* [24]. The protocols are not translated by the APIs, so requests are simply wrapped in messages and fed to the system *as is*. The APIs are two-way, to notify the external providers with status of their line-checks and trades, and receive responses via *RabbitMQ* as well.

ForexAPI. The *ForexAPI* receives all requests for trades and line-checks from the external APIs through *RabbitMQ*. It translates the proprietary protocols from the APIs to a uniform local format, and the other way around when responding. Some integration with *RequestService* is done via RPC and some through *ForexData*, which is shared between the two. *ForexAPI* also provides interfaces to external clients and users for several system functionalities, which it either handles itself or mediates to *RequestService*. This also means that the *ForexAPI* knows of most functionality in the system, resulting in unintended functionalities having been imple-

mented directly in the service over time.

RequestService. The *RequestService* receives requests for trades and line-checks from *ForexAPI* and feeds them to the mainframe. Beyond this, the service provides data and information from the mainframe and *ForexData* to the *ForexClient*. Most of the business logic lies within this service as well, including authentication of clients and requests, trade responsibility assignment, trade validation, trade registration, and line-check processing with data from the mainframe.

The *RequestService* shares part of its business logic with the *ForexAPI*, for example trade registration logic, and the knowledge of all protocols from the external APIs. Records of all received messages and trades processed are stored in the database *ForexData*.

ForexData and ForexBasicData. The states of both *ForexAPI* and *RequestService* are persisted in the relational SQL database *ForexData*, which includes records of trades alongside with the original raw messages from the external APIs. The data is also used to integrate some of the trade-registration logic, spread across *ForexAPI* and *RequestService*. The *ForexBasicData* service synchronizes some of the often accessed static data from the mainframe database in order to speed up access, acting therefore like a cache. However, differently from a cache, some data is sometimes also synchronized down to the mainframe from *ForexData*.

PushService. It listens to updates on trades and line-checks in the mainframe and fetches additional information from the *ForexAPI*. It pushes updates to the *ForexClient*.

Shared Libraries. Following the *Do not Repeat Yourself (DRY)* principle [27], a number of shared libraries and components have been created, which are used across the system. These can be seen in the upper left corner, in Figure 2. They are simple *.NET DLLs*, which are maintained and used across almost all components of the system and include a unified model in *Models* and access to the database through *DataAccess*. The libraries have dependencies between each other, resulting in difficulties when it comes to updates.

Mainframe and DB2. Although the mainframe and its associated database *DB2* are not official parts of the *FX Core*, the system relies on its functionalities, such as fetching of account balances used for line-checks, and the final registration of trades, i.e. requests to move assets from one account to another. It also contains organization information, such as users and their access rights, which is used for authorization purposes.

5.1.2 Integration

A wide variety of integration mechanisms and technologies are used between components and to external clients. In the following we provide a brief description.

- *Proprietary external protocols* from external clients and providers of trade and line-check requests, to the *external APIs*. Protocol messages are sent to the system through a TCP socket established between the providers and the *external APIs*. One of these proprietary protocols is the *FIX* protocol, which is used by many financial institutions.
- *.NET RPC* over TCP is used to integrate some of the internal components, *RequestService*, *ForexAPI* and *PushService*.
- *Messages via RabbitMQ* is used to integrate the *external APIs* with the *ForexAPI*.
- *Web-service interface* in the form of *Windows Communication Foundation (WCF)* and *SOAP*, provided by the *ForexAPI* to some clients, including traders wishing to manually fetch information.
- *Mainframe calls* are done over a proprietary RPC protocol on TCP sockets, and is used by most of the services in the system, to integrate with functionality and data in the mainframe.
- *Database integration* is used between *ForexAPI* and *RequestService* for some functionalities, such as trade-registration, meaning that instead of communicating trade registration data directly, they do it indirectly through the database. It is also used by some traders and other external systems to fetch data directly from the database.

5.1.3 Deployment

The system is deployed on three Windows Server hosts, in three different Danske Bank data centers, as shown in Figure 3. Each component can potentially be deployed individually, as they are independent processes, but they are always co-located both for availability reasons and for the high coupling between them.

All the components hold local references to all instances (replicas) of the services, on which they depend. This means that, in case a co-located dependency should fail, components can fail-over and establish a connection to another instance, on another host. The external provider APIs all run in *active/passive* fail-over, since only one socket per external provider can be opened. If an external provider API terminates, its connections will be taken over by one of the *passive* replicated instances, hereby becoming *active*. The problem with both types of fail-overs is that, once fail-over has occurred and the failed component is alive again, the dependants will not fall back. This can result in only a single instance actually being active and serving the system, should two previous nodes have failed, whether or not they are alive again. Manual intervention is required to fall back services to their co-located dependency-replica.

The *RabbitMQ* messaging system runs clustered across all nodes, since it is responsible for routing messages. This effectively functions as a load balancer of trade and line-check requests from external providers. The clustering ensures that no messages from external providers are missed, should a *RabbitMQ* node terminate, maintaining the *at least once* delivery guarantee provided by *RabbitMQ*. All servers are manually maintained, hereby becoming snowflakes, i.e, manually configured hosts that increase the risk of heterogeneous and non-replicable environments [28]. Deployment of components is automated with the continuous integration system *GoCD* [29].

5.2 Scalability

The MA paradigm itself applies some scalability techniques.First, the system is *scaled horizontally* by usage of multiple hosts, as seen in Figure 3, in a non-elastic way. Hosts are manually configured, and services need to be configured upon introduction of additional replicas since

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2889087, IEEE Transactions on Services Computing

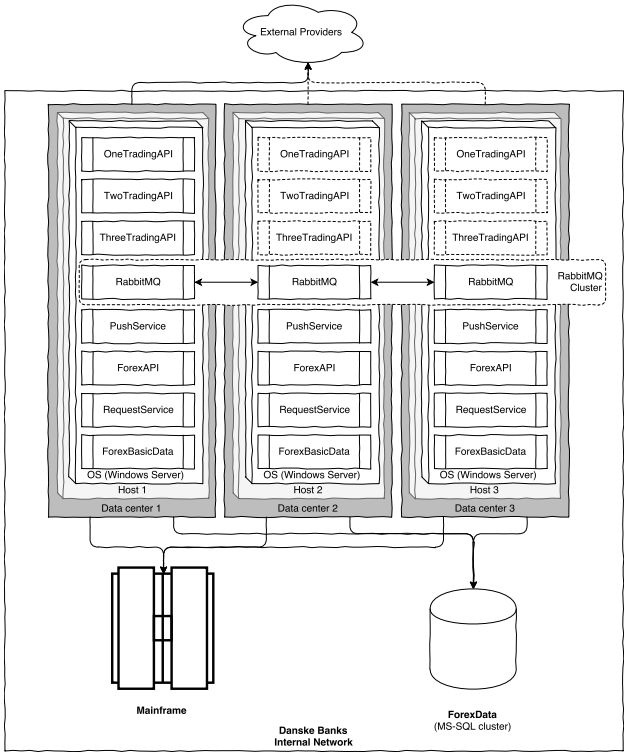

Fig. 3. Illustration of how the older MA is deployed on Danske Bank internal datacenters, which are connected via a *VPN* on a *WAN*. Three servers are provisioned to run the system, and the whole systems is replicated across the three servers. The mainframe and ForexData database are not deployed together with the system, but are managed by the IT department.

they hold internal references to all dependency replicas. Therefore, although possible, introducing additional hosts is too cumbersome to face temporary needs.

Distribution of functionality into services has been implemented, but such distribution does not result in *high cohesion* and *low coupling*. This results in small degrees of load distribution, not significant enough to improve load scalability. The usage of functionality distribution into thick clients is utilized, meaning that clients handle a lot of logic for data inspection, e.g. filtering, sorting and visualization, which relieves the monolithic system from some load. In theory, thick clients should enable better geographical scalability and lower latency, as it reduces the amount of networked communication necessary between the *ForexClient* and *FXCore*. Instead, the high coupling between *ForexClient* and *RequestService* results in extensive communication.

The whole system is *replicated and runs concurrently* in *active/active* mode across the three servers. The external APIs are the only exception. Replicated in *active/passive* mode, only a single replica is active at a time, as shown in Figure 3. This replication allows load to be split amongst replicas, resulting in better load scalability. The replicas also act as redundant instances hereby improving availability.

Since all trades and line-checks are independent they can be *executed concurrently*. The trades are spread amongst the replicated systems by *RabbitMQ*, so multiple trades can be executed in parallel. Services such as *RequestService*, also use multi-threading, in order to concurrently processes requests.

In order to achieve higher availability and throughput, *system clustering* is applied to *RabbitMQ* messaging. Should a *RabbitMQ* node terminate or become unavailable due to a network partition, the cluster automatically handles partitioning based on its consistency configuration. The *ForexData* database is stored on a database cluster, externally managed by the IT department, thus out of our scope.

The *ForexBasicData* component mirrors some data to *ForexData* from the mainframes DB2. This is somewhat a *cache*, as it speeds up access to some mainframe static data, but not significantly fast as the database is not optimized for fast reads. Additionally, many services use simple internal memory caches to reduce latency on serving requests.

The *Brewer's CAP theorem* [30], states that at any given time it is impossible for a distributed system to simultaneously provide *consistency*, *availability*, and *partition tolerance*. In network partition scenarios, only the clustered components are required to choose between consistency and availability (i.e., only *RabbitMQ* since *ForexData* is an external dependency). Since availability is desired, *RabbitMQ* is configured to handle partitions with its *auto-heal* feature, optimized for availability [31].

Fault tolerance is mainly implemented as part of the failover mechanisms in replication, load-balancing and routing. This ensures that if a component of the system fails, a replica is ready to take over its load.

Load balancing is mainly handled by *RabbitMQ*, which distributes messages from the external APIs in a *round-robin* manner among *ForexAPI* replicas. For other clients, e.g. the *ForexClient*, load is determined by which host they have a reference to, configurable on each client from a central management tool. Routing between the system components is manually configured, so each instance of a component has a list of all available replicas.

The MA has no centralized logs nor monitoring, but instead relies on *manual inspection and reports of erroneous behavior* from the users. All system components create local logs, which are manually aggregated, searched, and investigated by the developers as a *post-mortem* analysis (i.e., after errors have occurred). Only one preventive monitoring is done, by inspecting the size of *RabbitMQ* error queue. This is where messages are located, when not handled successfully.

5.3 Achieved Characteristics and Goals of the Monolith

Here we provide a summary of the (fully or partially) achieved scalability characteristics.

Load scalability has been achieved, since it can handle load up to the statically allocated resources limits. The system can be expanded manually, in case more load occurs.

Geographical scalability has been partially achieved. Since the system is accessed by external providers, and is not latency critical, geographical scalability is mostly a question of keeping the system available via the Internet. However, components are highly coupled, resulting in extensive message exchanges and reduced geographical scalability

Elasticity has not been achieved, since the infrastructure cannot expand and contract based on load. Additionally, the architecture is not suited for dynamic additions of resources and replicas, as load balancing is mainly done by the requester having references to all replicated dependencies.

Fault-tolerance is achieved to a limited degree, since the system can handle faults with fail-over, but fall-back after a component is alive again does not function optimally.

High Availability is achieved to a limited degree, since the system has implemented fault-tolerance mechanisms in the form of redundant replicas, fail-over, and configuration of *RabbitMQ* to prefer availability during network partitions. That being said, since the system does not provide any centralized aggregation of health-checking, monitoring, or logging, the system cannot act in a preventive manner, and this could lead to reduced availability.

Weak Consistency is achieved by choosing availability over strong consistency within *RabbitMQ*. *ForexData* is an external transactional MS SQL database, which ensures that writes are *strongly consistent*. The asynchronous *at-least-once* delivery guarantee, together with the *strong consistency* guarantee of MS SQL, results in an *eventually consistent* system.

Three main techniques have contributed to the scalability goals. **Throughput** has been improved by implementing horizontal scaling, replication, concurrency, and load-balancing. Distribution of functionality has not contributed much to throughput, as the components are highly coupled.

Availability has been improved through horizontal scaling, replication, load-balancing, clustering, and fault-tolerance. Accordingly to the CAP theorem, we configured the clustering to prefer availability over consistency. Fixing the fail-over mechanisms, as well as allowing the system to act preventively on aggregated health-checking, monitoring and logging, could lead to further improvements.

Latency has been improved by caching, load-balancing, replication, and concurrency. Services tight coupling between, and the use of many communication paradigms makes it difficult to optimize latency even further. Distribution might also have introduced some extra latency, since the system can no longer rely on IPC for communication. Again, tight coupling increases the messages exchange, simultaneously introducing even more latency.

5.4 Problems

Beyond scalability, the system has some other problems which motivated the team to design and implement a new architecture from scratch. Below are some of the major problems with the old MA.

5.4.1 Large Components

As many organizations experience, functionality after functionality, at some point the components grow too big. In particular, the analyzed system suffers from monolithic services that contain too many functionalities. This results in unnecessary complexity, confusion on where to locate new functionality and consequent hindered development.

As an example, *RequestService* suffers from size and contains too many functionalities, some of which are even shared with *ForexAPI*. As visible in Figure 2, it interacts with nearly all the system components, making it both a critical and a too complex component to handle. Over time, this resulted in low cohesion and high coupling, especially between *RequestService* and *ForexAPI*.

5.4.2 Shared Components

Although the system is split into separate services, a lot of functionalities are shared in the form of shared components, as it can be seen in Figure 2. Since the components are shared across the services, updating a shared component can result in forced updates across all services, as well as comprehensive testing of such changes across all dependent components. In turn, the more the shared components, the tighter the coupling and the lower the cohesion, since the functionalities are shared amongst dependent services. Simply put, shared components are tempting, but lead to unclear boundaries and unnecessary coupling.

5.4.3 The Mainframe

Due to the system age, a lot of the business logic and data are located within the organization mainframe. The developers estimate that around 90% of the business logic is still located in the mainframe. Clearly, this results in some difficulties. First, most of the mainframe code is developed with old legacy technologies, such as *Cobol* and *DB2*, and follows the imperative paradigm. Therefore, its structure is complex and nearly incomprehensible by any developer. Calls and dependencies criss-cross the system, with no kind of management or overview, making extremely difficult to optimize the system. The mainframe is not an easy component to replace, as it contains many core functionalities. However, the developers have started to pull out some functionalities and migrate them in external services, hereby slowly abstracting the mainframe away and minimizing its necessity over time. The *FX Core* MA was the first attempt to decouple the mainframe.

5.4.4 Complex Deployment

Deployment is somewhat risky and a very intricate process. Although the system has automated pipelines, the high coupling between components and the usage of shared components, makes deployment pipelines coupled and complex. Updating a single component can result in the whole system requiring a rebuild and redeployment, meaning that the whole system now needs to be tested.

5.4.5 Organizational Culture and Unknown Dependants

Danske Bank is a huge organization and the system has a large number of users outside the IT department, some of which have the capabilities to develop their own solutions, dependent on internal system components. An example is business people which rely on the data found in *ForexData*. Since these people are not educated in software architecture, they have made the mistake of writing quick scripts that read data directly from the database. This makes difficult for the team to modify the database structure, since it might break unknown important business processes. The team has already developed APIs for stakeholders and clients, but the transition is a slow process. At some point the system needs clear boundaries, hindering similar practices that slow down development and might cause errors.

5.4.6 Multiple Communication and Integration Paradigms

The system utilizes various integration and communication paradigms. This makes communication and integration between components unnecessarily complex, often resulting in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2018.2889087, IEEE Transactions on Services Computing

violation and bad definition of interfaces. The usage of *RPC* and *database integration* also results in high coupling between components. Often, services also communicate directly and two-way, resulting in even higher coupling.

5.4.7 Technology Dependence

Monoliths also strongly limit the use of different technologies. If a developer needs to develop a new feature within the monolith, said developer is limited to the technologies the monolith is already implemented in, although another technology might be a better fit, either for the feature or the developer's expertise. Although the system is not one big monolith, the choice of databases, integration paradigms, reliance on shared components and choice of deployment platform, limits the choice within Microsoft *.NET* platform. In turn, heavy reliance on Microsoft technologies also limits the deployment platform to *Windows Server*, which in general provides less flexibility, compared to running on *Linux* servers.

5.4.8 Missing System Status Overview

Since the system does not have a central location and aggregation of monitoring data, health-checks and logs, there is no way to get an overview of the system status. This shortage forces developers to manually investigate logs or the messaging system, minimizing any opportunity to apply preventive measures based on system warnings.

6 FX Core MSA

The Danske Bank new *FXCore* architecture is based on the MSA style and is intended to completely replace the old MA. This section will cover how Danske Bank *FX Core* team has chosen to implement a MSA, thus giving an idea of how such an architecture can be implemented in an enterprise setting. This includes a description of the infrastructure as depicted in Figure 4, a brief description of the implemented services, some of the additional architectural principles used, what scalability techniques have been applied and what scalability characteristics and goals have been achieved.

Danske Bank *FX Core* MSA is hosted on private data-centers, i.e. not in the cloud. This means that new hosts can not be provisioned and de-provisioned as rapidly and automated as in a cloud. It is in their interest to provide a private cloud for systems to run in, but due to regulations on banking data, this is still work in progress. There are three data-center locations in Denmark, which can be utilized to achieve better availability and increased resilience to the internal systems.

On the IT department roadmap there is the adoption of the *Red Hat OpenShift* [32] Iaas/PaaS platform, on the internal data-centers. However, at the moment, the infrastructure consists of VMs ordered through a web-portal, manually setup by the *FX Core* team.

6.1 Containerization

All services in Danske Bank *FXCore* architecture are hosted in *Linux Containers* on the *Docker Swarm* cluster [21]. Containerization enables a whole suite of tools, provided by

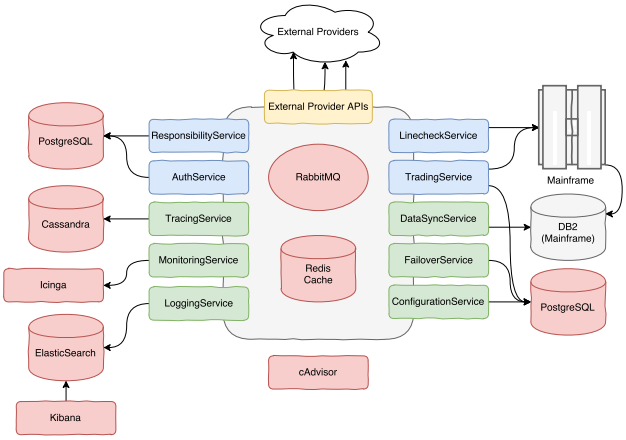

Fig. 4. The new *FX Core* MSA. Red services are infrastructure services, green are foundation services, blue are business services and the yellow is external provider APIs. Databases in the diagram should be seen as database management systems (DBMS), meaning that although four services use *PostgreSQL* they all have their own standalone database within the DBMS

Docker platform. *Docker Compose*, for example, allows the deploy the entire architecture with a single command, so that developers define all service dependencies and deploy them for local testing during development. Services are deployed locally, since they are running in containers, but their environments are exactly as if deployed to production.

All container images are hosted on an internal *Docker Registry*, a central repository for container images, where the official registry is `hub.docker.com` [33]. New images are deployed to the internal registry when a new version of a service is successfully built and tested by the *continuous integration* system. Furthermore, the services images inherit from infrastructure and base images hosted on the same hub. A list of all FX Core images can be retrieved with a search within the local registry.

6.2 Automation

All services in the architecture, including infrastructural clusters, has an automated *continuous integration and continuous deployment (CICD)* pipeline on their internally hosted *GoCD* server [29]. The *GoCD* platform offers a simple interface, which gives an overview and interaction with building, testing and deployment. The tooling which comes with the orchestration system *Docker Swarm*, has APIs which enables automation of many infrastructural tasks, such as rolling updates. These are utilized by the CICD system, combined with checks on correct functioning.

6.3 Orchestration

All deployment and execution of services, in containers, is managed by *Docker Swarms* orchestration on the *swarm* cluster. *Swarm* uses the notion of a `service` which is an aggregation of containers, meaning that multiple replicas of the service containers are treated as a single swarm `service`, also called *managed containers*. An example of this can be seen in Figure 6, where the service `trading-service` has multiple replicated containers, i.e. `trading-service.1`

and `trading-service.2`, but service discovery and persistence in the same database, allows them to act as a single service. The swarm cluster is also managed by *Swarm* and hosts all services on the cluster. The orchestration tooling also handles service discovery and load balancing, and has web and command line interfaces which can be used for automation of *rolling updates*, *scaling* etc.

6.4 Clustering

Clustering is one of the primary techniques used in the *FX Core*. The architecture runs across five virtual hosts located in the three data-centers. On the hosts a *Docker Swarm* cluster has been setup, with each host acting as a *Swarm Node*. This allows the three container engines on the *Swarm Nodes*, i.e. *Docker Engines*, to act as a single engine, allowing containers to run spread across the cluster. This is illustrated in Figure 6 Since *Swarm* is also a container orchestration system, it provides the features mentioned in Section 3.

Docker Swarm [21] allows for overlay networking, which enables the developers to define internal networks which is used to communicate between service containers, which all expose their ports to the internal network. *Docker Swarm* also allows for management of storage volumes, which are spread across the cluster nodes and are used to persist data from databases and *RabbitMQ*. The cluster allows for dynamic joining and leaving of *Swarm nodes*, and automatically rebalances location of services to efficiently use resources.

Beyond clustering the container engines, some of the services also utilize clustering. This includes the messaging system *RabbitMQ*, monitoring system *Icinga* and all databases, i.e. *Redis*, *Cassandra* and *PostgreSQL*. These services use clustering mostly due to requirements to their availability, since they are critical components of the infrastructure. Therefore all infrastructure service clusters are deployed with a service cluster node on at least one Swarm cluster node in each datacenter. Ensuring that the system can keep running as long as a single datacenter is available and has an active Swarm cluster node.

6.5 Load Balancing and Service Discovery

Service discovery is implemented as part of *Swarm*, which ensures that service hostname lookups from containers are translated into IPs of concrete containers. Since *RabbitMQ* is used to communicate between services, Swarms service discovery is only used by services requesting infrastructure services. *RabbitMQ* knows of services which have actively subscribed to one of its queues, hereby not needing service discovery. Load balancing is therefore required to be implemented by both *RabbitMQ* and *Swarm* service discovery.

RabbitMQ implements load balancing by distributing messages between subscribers to a queue, hereby spreading load between them. Usually all replicas of a service subscribe to the same queue, and they can hereby share the load. This is usually done in a *round-robin*, distributing messages to replicas in sequential order. *RabbitMQ* queues rely on acknowledgements upon successful processing of a message. Should a replica not acknowledge a message, it will simply be handed to the next replica, ensuring the message is processed *at-least-once*. In the case no replica can handle the message, it will be sent to an error queue, hereby notifying the developers. *RabbitMQ* can be configured to distribute messages in other ways if load balancing is not needed, which is done upon creation of a queue.

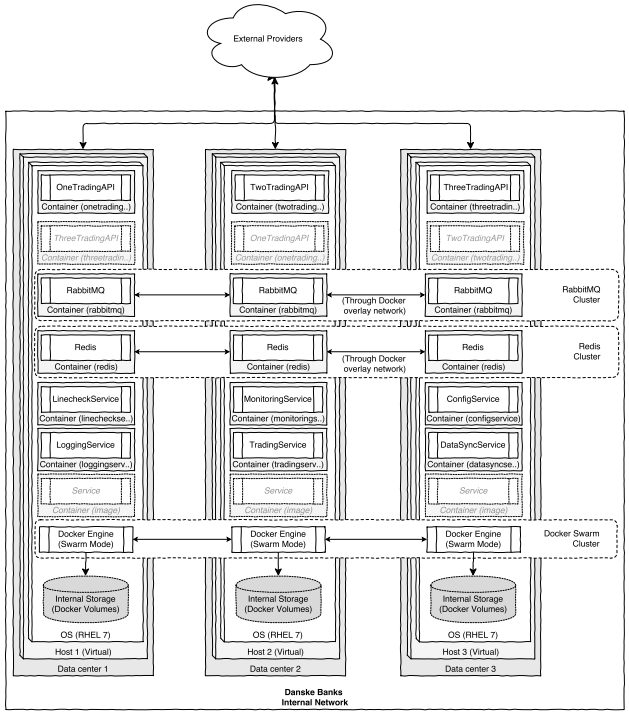

Fig. 5. Danske Bank MSA is deployed in three datacenters. The operating system on all hosts is *Red Hat Enterprise Linux (RHEL) 7*, which has been configured to open network ports for the installed *Docker Engines* to run and communicate. The *Docker Engines* are configured to run as a cluster, i.e. each in *Swarm Mode* as part of the *Swarm Cluster*. The infrastructure services, such as databases and messaging software, here *Redis* and *RabbitMQ*, are configured to run on at least one host in each datacenter, and do so in clusters. Services running in *active/passive* failover, such as the *TradingAPI* services here, also run replicated across the cluster.

Swarm utilizes the built-in service discovery to balance load between replicated service containers. When a service hostname is requested, *Swarm* translates to an IP of a replica container. For now, this is done in a *round-robin* fashion, but one might consider translating based on *proximity*, i.e., to co-located replicas to reduce latency, or based on *load*, i.e., to least busy replicas to improve throughput.

An illustration of *RabbitMQ* and *Swarm* load balancing, can be found in Figure 6.

6.6 Services

Here will give a short overview of the services within the system, how they are implemented, how they integrate and their different fail-over modes, which are important to how they are scaled. The services' responsibilities and functionality will not be covered in depth, but it should be apparent from their naming.

6.6.1 Integration

All services written by Danske Bank integrate via *message-based choreography*, as it is asynchronous and decouples services entirely. The chosen messaging system is *RabbitMQ* [24], which provides configurable *publish/subscribe*

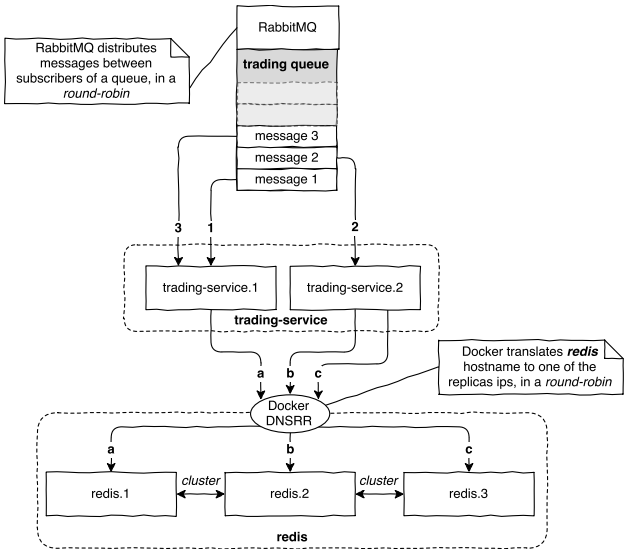

Fig. 6. Load balancing is implemented in two places in the infrastructure, as part of *RabbitMQ* message distribution between replicas and as part of *Docker Swarm* built-in service discovery. In this diagram, when a `trading-service` replica sends a request to `redis`, *Swarm* will translate the message to one of the replicas, e.g. `redis.1`, `redis.2` or `redis.3`. Since all three replicas run in a cluster, they ensure that state is shared and consistent amongst each other.

mechanisms in the form of messaging exchanges, queues, and bindings between these two. Typically, a producer service pushes messages to an exchange component, whereas consumer services pull messages from a queue. Between exchanges and queues occur bindings, which define how messages are distributed from the exchange component to one (or more) queues, based on message metadata. This decouples services from each other and makes all communication between them asynchronous.

Queues function as a *load balancer* between consuming replicas of specific services. When a queue is shared among services replicas, each replica gets messages in a *round-robin* manner, as shown in Figure 6. *RabbitMQ* supports acknowledgements from consumers, so if a message is not acknowledged after a predefined timeout it is redistributed to another replica. If the redistribution happens too many times, the message is forwarded to an error queue. Acknowledgements are put in place, in order to ensure that all messages are handled eventually and if not the developers will be notified from the error queue.

6.6.2 Fail-over Modes

All the services are categorised into two failover modes, which not only describes how the services handle failure, but also helps define how they are run in production.

Active/Active failover means that multiple service replicas can run alongside each other, providing better scalability through load sharing. Besides, if one replica fails others can take over its intended load, while the failed one recovers. This can also be used for rolling updates, where replicas are updated one at a time, resulting in zero-downtime updates.

Active/Passive failover means that only a single instance of a service can be running at a time. Therefore it cannot be scaled by replication, but only by increasing resources, i.e. vertical scaling. During runtime a passive service will be idling until the active service fails, and the passive service will become active and take over the workload from the failed service. The same approach applies to updates, where a new version of the service will be deployed and take over the old versions workload when ready, hereby letting the old service terminate.

6.6.3 Foundation Services

These services function as the foundation of the architecture, meaning that they implement supportive functions and not business related functionalities. They implement centralized logging and monitoring, centralized service configuration and handling of *active/passive* failover. All of these services run in *active/active* failover, meaning they can be replicated and run concurrently:LoggingService, MonitoringService, ConfigurationService, FailoverService, DataSyncService, TracingService.

6.6.4 Business Services

These are the services that are actually implementing business logic. They process trades, line-checks and authorization of actions in the system. This is mainly the group of services which will be expanded before deployment to production. All of these services also run in *active/active* failover, meaning they can also be replicated and run concurrently: LinecheckService, TradingService, ResponsibilityService, AuthService.

6.6.5 Infrastructure Services

All of these services make up the infrastructure of the architecture, which includes messaging, monitoring, logging and databases. All of these infrastructure services run in clusters, to provide high availability and better performance, i.e. load scalability.

- *Elasticsearch* stores logs and health check data from services.
- *Icinga* aggregates, visualizes and inspects monitoring data.
- *Kibana* aggregates, searches and inspects logs from all services.
- *PostgreSQL* is a database used by most of the services which require persistence.
- *RabbitMQ* is the main messaging system, used by all services.
- *Cassandra* is a database used by the *TracingService*.
- *Redis* is used as a cache for static data from the mainframe database.
- *cAdvisor* is used to retrieve performance metrics about containers and hosts, from the Swarm cluster nodes.

6.6.6 External API Services

These services provide interfaces to the external providers of *trades* and *line-checks*. Their main task is to have an open socket to the providers, translate the messages they receive on proprietary protocols to a standard format that can be fed to the system, via *RabbitMQ*. They are all *active/passive* as a provider typically only provides a single socket. The names of the concrete services will not be mentioned, as they are confidential.

7 MA vs. MSA

After having presented both architectures, we will now discuss the migration process how they differ in handling the effects of scale. Beyond comparing their scalability, this section will also explain how the new MSA copes with the problems we have presented for the MA.

7.1 Migration Process and Principles

In Section 8 we will emphasize the key aspects that are relevant to the migration, and that could help to replicate the process within another organization. Here we focus on the process of migration itself, showing how the MA was converted into MSA and what principles have been followed to divide a complex business service into multiple microservices. First, as we will repeat in the lesson learned, the scenario has to be *business-driven* and *outside-in*, meaning led by the necessity of the stakeholders, and in a precise order elicited through conversations with them. In our work, the business functionalities were defined mostly by communicating with forex traders and were iteratively added according to the level of priority for the business itself. The complexity of the whole system is therefore addressed outside-in, therefore not moved by internal needs of the hosting organization. The migration was performed manually, considering each specific functionality and identifying whether it should have resulted in a new service or not.

The *divide et impera* principle that emerged in our work can be synthesized as follows, as result of empirical experience: *when a business functionality is isolated and sufficiently big, or shared among many other business functionalities, it should result in a new service*. As we will discuss later, often the iterative approach was necessary and some functionalities were included in the same service, before splitting them on a successive iteration. This approach has at least one advantage and one disadvantage: on the positive side it progressively distances the team from the legacy system, avoiding the implementation of a distributed monolith; on the negative side, "sufficiently big" is a fuzzy and highly subjective definition. In this case, we have to keep in mind that the process is business-driven and, considering business priorities, it is generally clear how to identify the functionalities which need to be often updated and redeployed, therefore worth to become a new standalone service.

7.2 Effects of Scale

Both architectures apply techniques to achieve a certain degree of scalability, which ensures they can cope with the effects of scale. In the following, we will discuss how they differ in handling these effects.

Availability is handled better by the MSA, since the MA has problems with fall-back after a fail-over. They have both applied techniques to improve availability, but the MSA loose coupling and reliance on replication and load-balancing of individual services has ensured availability will not be affected by scale.

Reliability may become an issue at scale since both architectures integrate components with unreliable networked communication. In the MSA, all integration between services rely on *RabbitMQ* which can be configured to ensure reliable transfer of messages [31]. This may apply to the APIs used to integrate with infrastructure services as well. The simpler integration in the MSA, combined with its principle of *designing for failure*, could result in better tackling of *reliability* at large scale. Additionally the use of containerized and independent environments of the individual microservices, should also provide the same reliability between local testing and deployment. This is not the case with the monolithic components, which are run directly on the developers machines for testing, and server OS for deployment.

System Load is handled in both architectures by horizontally scaling the hosts and load-balancing between replicas, thus spreading overall system load between hosts. One might also argue that the MSA, although distributed, ensures through loose coupling that messaging does not create too much network traffic. Elasticity also ensures that the MSA can make use of extra allocated resources, which could be used to reduce system load on individual hosts, when needed.

Complexity is handled better by the MSA, although more distributed and thus with more moving parts. This is mainly due to its high cohesion, low coupling, extensive monitoring and logging, and reliance on automation. All of this contribute to reduced complexity in structure, separation of responsibilities, and deployment. This is likely to endure over time, as the architecture is optimized to evolve by services addition. Conversely, the MA exhibits high coupling between large components, making structure and services integration complex, without a clear separation of responsibilities. The variety of integration patterns also contributes to such complexity and, although automated, deployment is still a complex process due to shared components. Furthermore, the missing centralized logging and monitoring makes complex to keep the whole system running.

Administrative costs are greatly reduced in the MSA as it relies on orchestration tooling, automation and extensive centralized monitoring and logging. The MA requires more manual work to keep running at large scale, as deployments are more complex, there is no centralized monitoring, and the server environments are manually maintained.

Consistency in both systems is simply kept *weak*, or more precisely *eventual*. This also ensures that the system can be kept highly available at large scale, although network partitions might occur.

Heterogeneity is effectively handled by the MSA due to the use of containerized environments, which results in highly portable services. This is also substantiated by its ability to be deployed to heterogeneous infrastructure, such as differently sized hosts. On the other hand, MA is less portable as it requires a specifically configured and maintained environment, in this particular case a *Windows Server*. Besides, it requires homogeneous infrastructure (i.e., same amount of resources), since the whole architecture is replicated on each and every host.

From the above analysis, it results evident that, in general, *FX Core* MSA scales better than the MA version.

7.3 Solving Monolithic Problems

Let us now see how the MSA has improved or solved some of the problems identified in the MA.

Large Components. The large components of the MA which were highly coupled, had overlapping responsibilities and integrated in a multitude of ways, have been substituted with several independent microservices. Just the name of the services reveal their responsibility and they are generally way smaller compared to the large monolithic services. They do not integrate directly, resulting in looser coupling and less chance of feature overlapping in the future. As an example, trade-registration and line-checks were handled both by *ForexAPI* and *RequestService* amongst almost all other functionality in the MA. In the MSA a *TradingService* and a *LineCheckService* are handling these tasks individually instead. This is the case with all other functionalities in the MSA, resulting in low coupling, high cohesion and small services.

Shared Components. The shared component were many in the MA, but in the MSA, this has been reduced to only one shared component, the *Lambda* framework. *Lambda* is very minimal and is only meant to be a framework to connect to the infrastructure and provide standard formatting methods for e.g. messages, logs and health-checks.

The Mainframe. The mainframe will still be attached for a while in the MSA, but over time the functionalities from the mainframe will be implemented as new services. In turn, this will result in all *Forex* functionalities being extracted, totally decoupling the mainframe from the system. For now, the impact of the mainframe has been reduced by caching.

Complex Deployment. Since the microservices are independent, loosely coupled and isolated components, they can be deployed individually, without affecting the other components. There is no *dependency hell* and the only shared component is *Lambda*. Even when *Lambda* is updated, all the services are not necessarily required to update, since they run in their own containerized environments and do not directly share any dependencies, i.e. libraries. This makes deployment very simple and the usage of Docker and Linux containers ensures that services run in the same environment during local testing, on test servers and in production.

Organisational Culture and Unknown Dependants. The whole re-implementation brings other benefits with it than a new MSA. It also allows the team to kill all paths into the system, which they do not control. Since the team controls the whole infrastructure with Docker, including databases and ports open to outside clients, the team can eliminate all unwanted access. This allows the team to develop open APIs for clients and traders in the bank to use, thus eliminating direct database queries and the like. This gives the team full ownership and control of internal implementation details.

Multiple Communication and Integration Paradigms. Internally the microservices integrate only via messaging on *RabbitMQ*. Due to using *message-based choreography* the services do not call each other directly, thus resulting in very low coupling and no interfaces to violate. The system does communicate to external systems via other paradigms, such as the proprietary protocols to external providers and future *REST* APIs, but this does not compromise internal system complexity. The integration between services and their infrastructure dependencies, does not result in internally complexity either, as it is not used for any integration between business or foundation services.

Technology Dependence. The team aimed for a polyglot architecture, meaning that it is not technology dependent. The team is no longer dependent on the *.NET* platform or MS SQL databases, but can implement the services in whatever language they like. One might argue that they are just becoming dependent on other technologies, such as *Docker*, but Linux containers are becoming a standard through the *Open Container Initiative* [34].

Missing System Status Overview. The MSA has centralized logging in the form of *LoggingService*, *ElasticSearch* and *Kibana*, allowing for aggregation of logs from all services. The same applies to monitoring implemented with the *MonitoringService*, *Icinga* and *cAdvisor*, allowing for aggregated monitoring of metrics. Centralizing and aggregating both logs and monitoring, gives the team a complete system status overview, allowing them to act proactively on suspicious and faulty behaviour.

8 Conclusion and Lessons Learned

An increasing interest is growing around the idea of microservices, and companies are evaluating pros and cons of a complex migration. Not every business domain is affected the same way by the necessity of migrating legacy systems. In particular, financial institutions are positioned in a difficult situation due to the economic climate, but even more by the appearance of small players that grew big fast in recent times, such as alternative payment systems that can also navigate in a more flexible (and less regulated) legal framework. Evolution is necessary to stay competitive. When compared with companies (such as Paypal) that started their activities using innovative technologies as a business foundation, in order to scale and deliver value, old banking institutions appear outdated with regards to technology standards. The Danske Bank system itself was largely monolithic and mostly batch-based, generally dealing with an heterogeneity of data sources. For example, batch jobs were in place to balance books between departments, and to analyze trades aiming to detect fraudulent behaviors. Nowadays, customers expect to conduct online most of their operations but, even if mobile banking applications can be built from scratch according to agile and DevOps principles, data dependencies still apply. It is easy to see that, if the core banking legacy system is based on batch jobs, huge performance bottlenecks can be expected. A MSA solution, like the one developed for Danske Bank, is capable of increasing the agility of the whole company.

The re-engineering of the system discussed in this paper led to reduced complexity, lower coupling, higher cohesion, and a simplified integration. Comparing the two architectural designs, we have seen how microservices led to better scalability and solved the major problems caused by the MA solution. Although the comparison did not include quantitative metrics, the implementation of specific techniques has been used as an argument in support of increased scalability.

There are a few points worth emphasizing about the migration process, which may be useful to engineers and specialist that have to cope with similar problems. First, our approach was incremental and we would generally suggest to follow this line starting from a proof-of-concept and then generalizing into a replicable process. Approaching a legacy

system that served successfully its purposes for literally decades does not seem to be advisable, independently from how much support is received by top management.

Instead of starting with several services, developing functionalities in a single one allows the team and the organization to uniform the vision, but also the understanding of the specific approach and of the coding standard. A split into two or more services will then appear natural. Second, use of agile methodologies, in any declination of the concept, suits particularly well the migration process. However, for this to be effective it is necessary that the company has already an established agile culture, otherwise risks and threat to success may sum up. In this sense the incremental approach and the agile culture go together, operating the development sprint by sprint.

Third, DevOps and MSA appear to be an indivisible pair for organizations aiming at delivering applications and services at high velocity. The philosophy may be introduced in the company with adequate training, but only if certain technological, organizational and cultural prerequisites are present [35]. If not, the prerequisites should be developed. Investing in DevOps is a good idea in general, and after a migration of this kind is even more crucial.

Fourth, the order in which the migration is performed is important. Even if the whole system will be migrated, it is advisable to focus first on those parts that are important for relevant stakeholders and their specific business. In this case, we designed and implemented one business functionality at a time, following an order defined together with forex traders, and iteratively integrated them accordingly to the level of priority for the business itself.

Fifth and last, although somehow obvious, things should follow appropriate communication patterns. This is a prerogative of any business and any process, so it does not add much to the discussion, but it is important to organize the channel through which the information is passed, in order to avoid work duplication or lack of synchronization.

The future will see a growing attention regarding the matters discussed in this paper, and the development of new programming languages intended to address the microservice paradigm [8]. Object-Oriented programming brought fresh ideas in the last decades, and the expectation is that a comparable shift may be just ahead of us. Innovative engineering is always looking for adequate tools to model and verify software systems, as well as support developers in deploying correct software. As we have demonstrated in this paper, MSA is an effective paradigm to cope with scalability. However, the paradigm still misses a *conceptual model* able to support engineers since the early phases of development. In the following, we describe a set of research challenges that a complete software-engineering approach (within the microservices field) must cover in the next years.

To make the engineering process of a microservices-based application efficient, we need a *uniform way to model autonomous and heterogeneous microservices*, at a level of abstraction that allows for easy interconnection through dynamic relations. Each microservice must have a partial view on the surrounding operational environment (i.e., system knowledge) and at the same time must be able to be specialized/refined and adapted to face different requirements, user needs, context-changes, and missing functionalities.

An important feature of dynamic and context-aware service-based systems is the possibility of handling at run-time extraordinary/improbable situations (e.g., context changes, availability of functionalities, trust negotiation), instead of analyzing such situations at design-time and pre-embedding the corresponding recovery activities. The intrinsic characteristics of microservice architectures make possible to nicely model run-time dependability concepts, such as "self-protecting" and "self-healing" systems [36]. To make this feasible, we should enable microservices to monitor their operational environment and trigger adaptation needs each time a specific system property is violated. To cover the aforementioned research challenges, we already started to define a roadmap [37] that includes an initial investigation on how *Domain Objects* [38] could be an adequate formalism both to capture the peculiarity of MSA, and to support the software development since the early stages.

REFERENCES

[1] E. S. de Almeida, A. Alvaro, D. Lucrédio, V. C. Garcia, and S. R. de Lemos Meira, "Rise project: Towards a robust framework for software reuse," in *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, IRI - 2004, November 8-10, 2004, Las Vegas Hilton, Las Vegas, NV, USA*, 2004, pp. 48–53.

[2] M. C. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton, "Reference model for service oriented architecture 1.0," *OASIS Standard*, vol. 12, 2006.

[3] Z. Yan, M. Mazzara, E. Cimpian, and A. Urbanec, "Business process modeling: Classifications and perspectives," in *Business Process and Services Computing: 1st International Working Conference on Business Process and Services Computing, BPSC 2007, September 25-26, 2007, Leipzig, Germany.*, 2007, p. 222.

[4] M. Mazzara, "Towards abstractions for web services composition," Ph.D. Thesis, University of Bologna, 2006.

[5] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, B. Meyer and M. Mazzara, Eds. Springer, 2017.

[6] F. Montesi, C. Guidi, and G. Zavattaro, "Service-Oriented Programming with Jolie," in *Web Services Foundations*. Springer, 2014, pp. 81–107.

[7] L. Safina, M. Mazzara, F. Montesi, and V. Rivera, "Data-driven workflows for microservices (genericity in jolie)," in *Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016*.

[8] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi, "Microservices: a language-based approach," in *Present and Ulterior Software Engineering*. Springer, 2017.

[9] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[10] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How to make your application scale," in *A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)*. Springer, 2017.

[11] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, May 2016.

[12] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, September 2017.

[13] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," *IEEE Software*, vol. 35, no. 3, pp. 44–49, May 2018.

[14] P. D. Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, April 2018, pp. 29–2909.

[15] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," in *III Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, 2015, pp. 97–104.

[16] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures usingmicroservices: An experience report," in *Advances in Service-Oriented and Cloud Computing*, A. Celesti and P. Leitner, Eds. Cham: Springer International Publishing, 2016, pp. 201–215.
[17] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," *Software: Practice and Experience*, July 2018.
[18] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency," *IEEE Software*, vol. 35, no. 3, pp. 63–72, May 2018.
[19] Kubernetes, "Kubernetes - production-grade container orchestration," http://kubernetes.io.
[20] I. Mesosphere, "Marathon: A container orchestration platform for mesos and dc/os," https://mesosphere.github.io/marathon/.
[21] I. Docker, "Swarm mode overview - docker," https://docs.docker.com/engine/swarm/.
[22] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar, "Towards recovering the broken soa triangle: A software engineering perspective," in *2Nd International Workshop on Service Oriented Software Engineering: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. IW-SOSWE '07, 2007, pp. 22–28.
[23] HashiCorp, "Consul by hashicorp," https://www.consul.io/.
[24] I. Pivotal Software, "Rabbitmq - messaging that just works," https://www.rabbitmq.com.
[25] redis.io, "redis," http://redis.io.
[26] A. MacEachern, "How are international exchange rates set?" http://www.investopedia.com/ask/answers/forex/how-forex-exchange-rates-set.asp.
[27] S. Smith, "Don't repeat yourself," http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Repeat_Yourself.
[28] M. Fowler, "Snowflakeserver," http://martinfowler.com/bliki/SnowflakeServer.html.
[29] T. Inc., "Gocd, open source continuous delivery service," https://www.gocd.io.
[30] E. Brewer, "Pushing the cap: Strategies for consistency and availability," *Computer*, vol. 45, no. 2, pp. 23–29, Feb. 2012.
[31] I. Pivotal Software, "Rabbitmq - clustering and network partitions," https://www.rabbitmq.com/partitions.html.
[32] R. Hat, "Openshift: Paas by red hat, built on docker and kubernetes," https://www.openshift.com/.
[33] I. Docker, "Overview of docker hub - docker," https://docs.docker.com/docker-hub/.
[34] O. C. Initiative, "About, open container initiative," https://www.opencontainers.org.
[35] M. Mazzara, A. Naumchev, L. Safina, A. Sillitti, and K. Urysov, "Teaching devops in corporate environments: An experience report," *CoRR*, vol. abs/1807.01632, 2018. [Online]. Available: http://arxiv.org/abs/1807.01632
[36] N. Dragoni, F. Massacci, and A. Saidane, "A self-protecting and self-healing framework for negotiating services and trust in autonomic communication systems," *Computer Networks*, vol. 53, no. 10, pp. 1628 – 1648, 2009.
[37] K. Mikhail, A. Bucchiarone, M. Mazzara, L. Safina, and V. Rivera, "Domain objects and microservices for systems development: a roadmap," in *Proceedings of 5th International Conference in Software Engineering for Defence Applications*, 2017.
[38] A. Bucchiarone, M. D. Sanctis, A. Marconi, M. Pistore, and P. Traverso, "Incremental composition for adaptive by-design service based systems," in *IEEE ICWS 2016, San Francisco, CA, USA, June 27 - July 2*, 2016, pp. 236–243.

AUTHORS' BIOGRAPHIES

Manuel Mazzara is Professor of Computer Science at Innopolis University (Russia) with a research background in software engineering, service-oriented architectures, concurrency theory, formal methods and software verification. He cooperated with European and US industry, plus governmental and inter governmental organizations.

Nicola Dragoni is Associate Professor in Distributed Systems and Security at DTU Compute, Technical University of Denmark, and Professor in Computer Engineering at Centre for Applied Autonomous Sensor Systems, Örebro University, Sweden. His main research interests lie in the areas of pervasive computing and cyber-security, with focus on Internet-of-Things, fog computing, and mobile systems.

Antonio Bucchiarone is a senior researcher of FBK in Trento, Italy. His main research interests are: self-adaptive systems, applied formal methods, run-time service composition and adaptation, specification and verification of component-based systems, dynamic software architectures. He has been actively involved in various research projects in the context of service-based adaptive systems.

Alberto Giaretta received his M.Sc. degree in Computer Science from the University of Padova, Padova, Italy, in 2016. He is currently a PhD Student at the Örebro University, Örebro, Sweden, under the supervision of Prof. Nicola Dragoni and Prof. Amy Loutfi. His main interests include Security, Internet of Things, and Bio-inspired Networks.

Stephan Thordal Larsen is software engineer at Danske Bank, Copenhagen, Denmark. In 2017, he got a MSc in Computer Science and Engineering at Technical University of Denmark. He has been the key actor in the Danske Banks transition of their FX Core system to a microservice architecture, showing that this transition significantly yields increased scalability over their legacy monolithic architecture.

Schahram Dustdar is Full Professor of Computer Science and head of The Distributed Systems Group at the TU Wien, Austria. He is an Associate Editor of IEEE Transactions on Services Computing, ACM Transactions on the Web, and ACM Transactions on Internet Technology and on the editorial board of IEEE Internet Computing and IEEE Computer. Dustdar is recipient of the ACM Distinguished Scientist award (2009), the IBM Faculty Award (2012), an elected member of the Academia Europaea: The Academy of Europe, and an IEEE Fellow (2016).