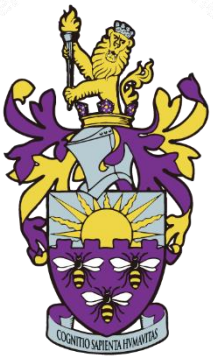
Maintaining Data Consistency in a Microservice Architecture

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER FOR THE DEGREE OF MASTER OF SCIENCE IN THE FACULTY OF SCIENCE AND ENGINEERING

Year of Submission

2020



Michael John Richards

Student Id Number: 10362882

School of Computer Science

University of Manchester

Declaration

I hereby declare that the contents of this dissertation is original except for references made to the work of others, and that no portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning. This dissertation contains the results of my own work and has not been used in collaboration.

Copyright

i. The author of this dissertation (including any appendices and/or schedules to this dissertation) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this dissertation, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has entered into. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the dissertation, for example graphs and tables ("Reproductions"), which may be described in this dissertation, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this dissertation, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/display.aspx?DocID=24420), in any relevant Dissertation restriction declarations deposited in the University Library, and The University Library's regulations (see https://www.library.manchester.ac.uk/about/regulations/_files/Library-regulations.pdf).

Acknowledgements

I would like to express my sincere gratitude towards my supervisor, Dr Liping Zhao, for her continuous support and motivation. Her passion and insightful knowledge of software design patterns was the perfect fit for this master's dissertation, and I could not have asked for a better supervisor.

I would also like to thank my beloved life partner, Eleanor, for motivating and supporting me during difficult times. She kept me focused when I needed it the most and it is hard to imagine what this experience would have been like without her.

Finally, a special thank you to my beloved family who supported me from afar, and to my boss and friend, Lee, who provided me with flexible hours during full-time employment and was there to talk to during stressful times.

Abstract

This dissertation investigates solutions for overcoming data consistency challenges when using multiple datastores across a distributed microservice architecture. It compares previous concurrency control protocols used to fulfil global transactions and discusses how such protocols can be used at the application level to improve scalability and performance. During evaluation of this project, data inconsistency anomalies were reduced by implementing a form of timestamp-based concurrency control using saga orchestration.

To improve the fault-tolerance of saga orchestration due to system failure, event sourcing was used with the Akka.Net actor framework to return the domain model back to a healthy state. This dissertation shows how the actor model can be used to implement message-driven inter-process communication (IPC) between microservices, where each actor reacts to incoming messages. It also discusses how complex stateful classes, such as saga orchestrators implemented using the finite-state machine model, can be persisted between restarts to preserve data consistency.

Finally, the implementation of the microservice architecture (MSA) created as part of this research attempts to maintain data consistency while also resulting in a reactive system as defined by the reactive manifesto. Therefore, each type of message identified in this dissertation is evaluated in terms of their responsiveness, with a short evaluation over user experience. Each implemented design pattern is qualitatively evaluated to discuss how they promote designing for change and software engineering best practices.

Table of Contents

Chapter 1

Introduction

As the microservice trend continues to grow in popularity, most new enterprise systems are developed with microservices as a key implementation detail while older monolithic systems are slowly migrating towards them. The number of academic research projects involving the use of microservices have also steadily increased with many aspects explored. From security, domain modelling, database and transaction management, distributed programming, and communication concerns, there are still many other aspects missing a well-defined solution to the technical challenges of maintaining such systems.

Microservice architecture (MSA) attempts to break up a large system into multiple sub-systems called microservices, which can then be individually developed, tested, and deployed in isolation. MSA attempts to improve horizontal scalability whilst providing high availability and reliability with the support of specialised infrastructure platforms and tools (e.g. load balancers with containerization). From a development perspective, this helps alleviate the technical debt of not being able to upgrade or replace old frameworks and database technologies without the fear of introducing system-wide breaking changes. It also allows separate development teams to work on individual microservices without clashing with other teams. This helps avoid merge conflicts when using version control, and less time is spent running system-wide unit tests because a microservice only needs to contain unit tests relevant to its behaviour.

Unfortunately, MSAs bring new risks and concerns that traditional monolithic applications lacked. With the rise of Domain-Driven Design (DDD), many microservices guard their own portion of the domain model by encapsulating the relevant data they need to work with inside their own datastore, and so microservices rely on specialized strategies to complete queries or transactions that span more than one service. Therefore, high latency is a potential bottleneck for system performance due to microservices experiencing downtime or high traffic. The ability for a microservice to recover from failure, known as fault tolerance, is vital to ensure that the state of a system remains consistent. If a transaction must span several services and one of service fails, then the system’s data integrity is at risk and must be accounted for.

1.1 Motivation

Many patterns and models for building MSAs have been proposed to tackle the generic challenges that have been identified. However, all systems based off MSA have their own unique requirements and use-cases that cannot make use of all proposed solutions, with many alternative approaches being developed as a result. Therefore, we are left with an open-ended list of possibilities that can discourage companies from adopting MSA. This is especially true when given the task of migrating away from monolithic architecture because the business risk can be huge, and so the task is often delayed further with the technical debt forever growing, making it increasingly difficult to migrate in the future.

Modern systems are discovering new requirements and patterns for dealing with these challenges, which led to the creation of the reactive manifesto; a set of guidelines agreed by the software engineering community for what characteristics a modern system should possess. These characteristics perfectly align with the goals of microservice architecture, but the challenge is designing such a system that complements them.

1.2 Research Aim

The primary aim for this research is to understand the potential data inconsistency anomalies caused by splitting up the domain model into multiple sub-domains with their own datastores, and how to reduce them. Many previous concurrency control protocols have been proposed and used by conventional database systems. Unfortunately, when databases directly communicate between themselves to synchronous their changes across multiple sub-domains, requests must be blocked resulting in a costly performance and scalability bottleneck. Instead, the aim is to implement message-driven, asynchronous, non-blocking IPC whilst favouring eventual consistency over strong consistency. Therefore, this research aims to implement global concurrency control across the domain with the help of modern design patterns that complement the reactive manifesto to build a highly responsive, non-blocking system.

1.3 Research Questions

This section outlines the primary research questions for this dissertation:

- Is it possible to improve the level of protection against data inconsistency anomalies when using saga orchestration? Typically, messages are processed when they arrive at a microservice without regards to when the message was created. A saga involves executing multiple sub-transactions as part of a globally distributed transaction by sending command messages. However, long running sagas may send commands to microservices targeting data that has already been changed by newer commands, causing unexpected user experience bugs. Is it possible to reduce the likelihood of such bugs caused by data inconsistency anomalies from occurring?
- Can we implement application-level concurrency control across a distributed system implementing the database per service pattern without blocking requests? The system should be responsive to client requests to be classed as a reactive system, as proposed by the reactive manifesto. Therefore, the system needs to handle the processing of requests asynchronously by returning the results to the client using a form of bidirectional communication. The client should never be blocked from performing additional requests while waiting for results.
- How can a system protect the domain model from entering an inconsistent state if a saga orchestrator restarts due to system failure? Usually, a saga orchestrator must rollback its global transaction using a series of compensating transactions to undo the changes made. However, if the saga orchestrator restarts then the saga's state data is lost. This means that it has lost any knowledge of what commands were sent before it stopped and therefore it cannot rollback.
- With all the added complexity caused by splitting up the domain and relying on messages to communicate between microservices, what patterns can we employ to reduce the amount of boilerplate code when creating new features whilst designing for change?

1.4 Objectives and Deliverables

This section outlines the objectives and deliverables for this dissertation:

- Implement a reactive MSA following the database per service pattern. Each microservice should isolate and guard its own datastore containing sub-domain data. Message-driven IPC should be used for each microservice to control how each sub-domain is accessed.
- Produce a qualitative evaluation of patterns used to reduce data consistency anomalies and how they promote designing for change.
- Create stress tests to execute the same request multiple times for each possible request type. Then, provide an evaluation of the minimum, maximum and average response times for each request, as well as providing a comparison of different message type characteristics, such as how saga orchestration response times differ from other message types.
- Deliver an implementation following the actor model to implement DDD while expanding upon previously defined ideas.
- Improve the fault-tolerance of saga orchestration when using the finite-state machine model to improve data consistency.

1.5 Dissertation Structure

This dissertation is structured into 7 chapters, including the introduction. This section gives a brief overview of each of the remaining 6 chapters as followed:

- **Chapter 2, Inter-Process Communication and Domain Modelling** – Provides a brief history of previously proposed IPC strategies within a distributed architecture, and how DDD can be used to implement a message-driven, microservice architecture.
- **Chapter 3, Data Consistency in Distributed Transactions** – Provides a brief history of transaction management in traditional database management systems and discusses previous research studies on improving data consistency through concurrency control protocols. The chapter concludes with mentions of snapshot isolation and distributed concurrency control.

- **Chapter 4, Implementing Distributed Transactions in Reactive Systems** – Aims to equip the reader with the necessary background knowledge surrounding distributed transactions and the challenges faced when implementing them in an MSA. This chapter includes an introduction to the saga pattern while comparing two methods for implementing sagas using either choreography or orchestration. The chapter concludes with an introduction of the actor model and the Akka.Net actor framework, and how to build fault-tolerance into a stateful application.
- **Chapter 5, Design and Implementation** – The chapter begins with an overview of the implemented web application from a usability perspective. Following this, technical implementation details of the reactive MSA and API gateway are covered while focusing on the design patterns chosen. Finally, the chapter ends by outlining the development and testing methodologies and tools used.
- **Chapter 6, Evaluation** – This chapter starts with a quantitative analysis of response times recorded for each message type, with additional data consistency use-case evaluations where the number of phantom writes were reduced using timestamp-based concurrency control at the application level. The chapter ends with a section containing a qualitative evaluation for each chosen design pattern.
- **Chapter 7, Conclusion** – This section summaries all technical achievements and reflects on the original aims and objectives of the dissertation, with a short discussion of possible future work efforts.

Chapter 2

Inter-Process Communication and Domain Modelling

There have been many strategies proposed for handling IPC within a distributed system, and many ways for modelling their domain. A controversial debate that is still often discussed, although arguably has died down over the years, has been the move to a network transparent, object-oriented programming model. Developers are initially intrigued with the paradigm of modelling a distributed system in such a way as to mask the underlining network communication involved with passing messages between remote systems. Many frameworks have been proposed over the years to mimic local method invocations, but under the hood are routed to remote systems to be called on remote objects. Thus, developers may fall into the misconception of believing that there is no difference between local and distributed programming. Unfortunately, there are several key factors where these two paradigms intrinsically differ and fail as a result.

2.1 A Brief History of Inter-Process Communication

Traditionally, all data interactions were conducted locally on the same machine. This is referred to as local computing when computing is restricted to a single address space as opposed to distributed computing where operations are computed across multiple address spaces [1]. Developers have more control over how to react to failure because the dataflow can be easily monitored and we are guaranteed a response, whether that response is successful or not. When we rely on remote services, the network is unpredictable and so we cannot guarantee that a request ever reached its destination. If a remote system carried out our request but failed to send an acknowledgement of any kind, then it becomes a far greater challenge to implement fault tolerance and reliability.

Several middleware strategies have been developed to address the unique requirements of distributed systems. Remote object invocation (also referred to as remote method invocation (RMI)), is an older approach that attempts to implement “the vision of unified objects” [1], where remote method calls as disguised as local ones and the underlining framework handles network specific details. This was seen by developers as the next step to remote procedure invocation (or remote procedure calls, RPC) that achieves the same

result except without the native support for object-oriented modelling. A unified object paradigm (also known as a distributed object paradigm) relies on well-defined proxy interfaces, as a way of communicating between objects, declared using an interface definition language. Implementation details are then hidden from the programmer, allowing the same interface to be used regardless of where the object is located (i.e. remotely or locally). The underlining system can then select the appropriate delivery strategy and accommodate the characteristics of the network. The Object Management Group's Common Object Request Broker Architecture (CORBA) is an early standard that proposes such a system. Other examples include SOAP and Enterprise JavaBeans (ELB) [2].

While there were some benefits to these designs, such as taking away the burden of low-level IPC concerns and allowing developers to focus on application logic, the characteristics of the network could never truly be ignored. An obvious characteristic of remote messaging is the latency increase as opposed to calling functions locally. This can be salvaged by upgrading hardware to increase processing power, using caching, and attempting to reduce the number of calls by keeping objects that often collaborate in the same address space. However, less obvious characteristics began to emerge, which created a difficult obstacle for achieving complete network transparency using this paradigm. Developers often fell victim to assumptions about the network and the illusion that they were working on a traditional object-oriented programming model.

Developers using an RPI-styled middleware would often make the misguided assumption that the invocation would respond within a timely manner. This made it difficult to guarantee responsiveness within their application. Another issue is assuming that all objects had access to shared memory, making pointers to object references difficult to implement. If the technology caters to shared object references in a distributed fashion, there is still the issue of handling concurrency. Because any object must potentially handle multiple concurrent method calls, the developer must use synchronous concurrency mechanisms to lock resources where appropriate, thus breaking the vision of a transparent network. Multiple threads interacting with the same object and its state must be carefully coordinated, which limits concurrency and can become very costly even for modern CPU architectures [3].

RPI-styled middleware designs, such as CORBA, were popular before focus shifted towards newer trends, such as service oriented architecture (SOA) and web service technologies [4]. Newer trends favoured explicit IPC strategies to avoid the locking and blocking of requests caused by transferring threads to maintain object references. Alternatively, message-driven architectures send serializable messages to remote services that are then picked up and executed on a single thread. There is no need to share object references as the response is sent back to the sender using a separate message rather than as a method's returned value. This also promotes asynchronous behaviour as the client is not blocked from processing further. The client instead makes use of callback functions to process the result only when the expected message is received.

2.2 Message-Driven Architecture

There are several ways to implement message-driven architecture. Enterprise systems often make use of a message-bus or messaging queue where messages are placed onto the queue via a message channel. A remote service then pulls the messages from the queue that have been assigned to them. This style of pulling messages from a queue differs from the traditional push architectures, such as REST or RPC, because they can be processed asynchronously without blocking threads and facilitate scalability using load balancers to increase the number of workers that can pull from the queue.

However, the use of a message queue can be difficult to implement predefined dataflows. For example, if a request must contact one or more remote services and expects a specific result back, it can feel overly complicated to retrieve the expected response. You can setup a temporary messaging channel for sending the response back to the client, create a separate REST API on the client to handle responses, or broadcast the message to all clients on a shared channel with only the appropriate client pulling the message from it [5].

An alternative solution is to use the publish-subscribe pattern. Instead of the client pushing messages onto a queue, or messaging remote services directly, the remote services decide what messages they are interested in by subscribing to events published by an observable. This simplifies the dependencies between services by giving more control over to the subscribers so that the publisher does not need to coordinate the entire

dataflow of the system. Published messages take the form of events, which represent that something has occurred, and the subscribers are free to ignore or react to events based on their own internal logic. Using this paradigm, it is easier to break apart the domain model by defining boundaries between sub-domains where each service owns a single sub-domain and subscribes to events that directly impact them.

2.3 Domain-Driven Design

The term Domain-driven design (DDD) was coined by Eric Evens in 2003 with the publication of his book "Domain-Driven Design: Tackling Complexity in the Heart of Software" [6]. Since then, the term gained huge attention and shaped the way modern architectures are designed. Eric Evans, along with Martin Fowler who helped pioneer the movement, offer guidance on the best practices for implementing DDD by highlighting several tactical and strategic patterns.

Tactical patterns are those that help identify system requirements, through knowledge crunching exercises, to create effective models for a complex domain. Strategic patterns are those that help shape the architecture to support the principles of DDD. Models identified using tactical patterns represent a section of the domain and are separated from one another using what Evans refers to as a bounded context. A bounded context is a strategic pattern that helps control the relationships of models by making them explicit [7].

Within a bounded context, a subdomain model takes the form of an aggregate, which can only be accessed through an aggregate root entity. The root defines the relationships between other aggregates so that any other object in the aggregate cannot be referenced directly. This improves the encapsulation between different subdomains and reduces the complexity that can emerge overtime as more relationships between objects are built [8]. If you were to delete a given entity within the domain without deleting or updating other entities that had relationships with that data, then the domain would be left in an inconsistent state. Evans provides an example where if you were to delete a customer record and the address of that customer, but other customers share that same deleted address then you will have objects referencing a deleted record [6]. Thus, weakening the integrity and consistency of the model. An aggregate root addresses this scenario by

limiting the number of object references allowed on the model to just one (i.e. the aggregate root entity itself).

The core philosophy of DDD stems from the need to have separate models to reflect different vocabularies used between different departments of a large organisation. The design should reflect the relationships between processes of the business so that developers and stakeholders can collaborate effectively. By using shared terminology in the form of a ubiquitous language, requirements are easier to capture between the domain experts and the development teams. Some domains may share terms but by using a bounded context, each with their own isolated models, terminology can remain consistent and avoid confusion [7]. For example, if your business model is to sell tickets for events (e.g. a music concert) but another department within the same business deals with IT tickets submitted by customers using a help centre platform, then both terms must exist within their own isolated models within the system without clashing. It would be impractical to ask the domain experts to change their terminology to benefit the software's architectural design. By using a shared ubiquitous language during the initial design phases, the barrier between technical terminology and domain terminology is lifted, thus improving collaboration.

2.3.1 Microservice Architecture using Domain-Driven Design

DDD fits very well with microservice architecture (MSA) because both philosophies share the goals of enforcing isolation. However, MSA takes it one step further by isolating subdomains using the database per service pattern. All communication must be performed using IPC mechanisms, such as a REST API. Thus, no object references can exist between entities belonging to separate subdomains and all subdomain data is persisted inside separate datastores that can only be accessed by the microservice that owns that store. This means that each microservice has full control over its own subdomain and datastore, which forms the basis of a bounded context.

RPI-styled middleware, as previously discussed, does not suit these goals of MSA and DDD because it attempts to make individual objects of an aggregate accessible over the network. Therefore, MSA tends to rely on a layered architecture where the outer layer uses interfaces or adapters to handle requests or incoming messages, which can then be

propagated to internal service-layer application logic. The service-layer is usually the only layer allowed to interact with the encapsulated subdomain model as this promotes many benefits, such as allowing the subdomain model to change when new requirements are introduced without affecting the infrastructure of the microservice. For example, the microservice API does not need to change to reflect the changes of the domain model, which means that the clients using that API can continue operating as normal. Some changes might directly impact the client, but version control can be used to maintain the old and the new version of an API, allowing the client to upgrade when they are ready. This requires careful design considerations to ensure that the system is backwards compatible where both versions can function simultaneously.

There are multiple ways to implement version control for an API. One such method is to prefix the request URL with the version number of the API to be used, or by specifying it inside a query string parameter. For more granular control, JSON/XML data contained in a POST request could represent a change to be committed to the datastore with an extra field representing the version of the API to use. This could signify to the system that the object should be handled in a different way, or that the data has been crafted to use a new feature offered by the new API version. This allows different parts of the data contained within the same request to use different API versions if required. API versioning is a good choice when migrating architectures or adding additional microservices to extend the functionality of the system.

Whatever strategy the architecture uses to separate the request data from the domain, the principle behind the idea is the same. The result is an adapter or façade, acting as an anti-corruption layer (ACL) to preserve the consistency of the encapsulated model. This is another strategic pattern described by Evans and can be used when gradually migrating a monolithic architecture into an MSA [9]. While breaking down a monolithic applications domain into subdomains, the ACL can help with this transition by ensuring that legacy systems are still accessible by the newer features introduced. Legacy systems tend to use obsolete IPC mechanisms or data schemas and so an ACL can be used to ensure that the microservice itself does not need to directly cater towards the legacy system's requirements. Therefore, when the monolithic architecture has been fully replaced and the ACL is no longer required, no further work is needed to remove obsolete code from the microservice codebase.

Chapter 3

Data Consistency in Distributed Transactions

A monolithic application using a centralized relational database management system (RDBMS) has a straightforward approach to preserving the consistency of the database. Typically, there are three types of errors that can occur when using a centralized RDBMS within a single address space; the transaction may experience deadlocks caused by concurrent transactions, the system restarts due to system failure resulting in the loss of in-memory data, or database failure. In a distributed system, you also must account for network communication failure such as a remote service being unavailable, loss of messages during transportation, or messages arriving out of order causing the transaction to abort.

In this chapter, we focus on online transaction processing (OLTP) databases for the purposes of building high-throughput, transaction-oriented applications. Online analytics processing (OLAP) database systems typically do not require the most up to date version of the data. They tend to be used for constructing complex queries for the purpose of analysing large historic data and do not require strong transaction consistency. We will also consider the differences between relational and NoSQL database designs and how they can be used in a distributed microservice architecture for implementing concurrent transactions.

3.1 A Brief History of Transactions

A transaction translates high-level queries (such as SQL queries) into a set of primitive read/write operations once an optimised execution plan has been chosen, with commands to signal the beginning of the transaction and its termination [10]. During termination, the transaction can be aborted, and rolled-back to undo the changes, or committed to persist the changes. During its execution, the database can be in a temporarily inconsistent state but must be consistent before and after execution. A database is said to be consistent if it obeys all the integrity constraints defined by its schema.

In 1975, the transaction model was first introduced by the IBM System R research project [11]. System R was an experimental database system that consisted of a locking

subsystem to ensure that conflicting data value writes caused by concurrent access could be detected and resolved. Later, the acronym ACID was coined by Theo Härden and Andreas Reuter in 1983 [12]. ACID principles define a set of database properties that attempt to provide safety guarantees when using transactions [13]:

1. **Atomicity** – all operations of a single transaction must be fully committed or aborted to preserve the consistency of the database.
2. **Consistency** – The database must be in a consistent state before and after the transaction has executed.
3. **Isolation** – The state of any given transaction is unknown to any other transaction. Transactions should be independent and able to run concurrently with any other transaction. Therefore, a transaction should only see one version of the data they are accessing.
4. **Durability** – Changes made to the database are guaranteed to be persisted even in the face of system failure. RDBMSs typically enforce this by logging changes into a log file or log table.

The implementation of transactions can vary between database vendors with each implementation consisting of a set of protocols and algorithms to enforce the ACID principles. A local transaction manager (TM) is given the responsibility of coordinating transactions by communicating with a scheduler. The scheduler uses a concurrency control (CC) method to ensure the correctness of data synchronization between concurrent transactions. Many research studies have been conducted on the use of such methods. They tend to focus on either a lock-based or timestamp-based CC to preserve the isolation of transactions. Both categories can either be implemented using a pessimistic or optimistic model. An optimistic model does not block transaction commits and allows the violation of isolation levels but resolves any conflicting transactions after execution. Pessimistic implementations enforce stronger consistency at the cost of performance by blocking transactions until data is freely available to use.

Isolation helps to avoid data inconsistency anomalies such as race conditions, dirty reads/writes, lost updates, and phantom reads/writes. Race conditions are the most common pitfalls caused by two or more concurrent transactions attempting to simultaneously modify to the same data value, or when one transaction tries to read the

same data value that another is in the process of modifying. The level of isolation between concurrent transactions can vary between database systems. Some systems implement weaker isolation as a trade-off for improved performance. The original transaction model only defines a standard for serializable isolation, which is considered the strongest level of isolation and is the most researched form of correctness criterion for concurrent transaction execution [14]. Serializable isolation describes the premise that two or more concurrent transactions should behave in the same way as if they were executed serially (as opposed to simultaneously) [15]. Unfortunately, it has some significant performance penalties when horizontally scaling out within distributed systems.

3.1.1 Lock-based Concurrency Control

To help preserve isolation, database systems employ the use of concurrency control (CC). Lock-based CC makes use of a lock manager to lock the data required by a transaction. The two-phase locking (2PL) protocol is a lock-based CC method which was the standard for implementing strong serializable isolation for many years [16]. During the first phase, a transaction attempts to obtain a lock on the data it needs to complete the transaction. This prevents no other transactions from accessing it (this includes both reads and writes). Transactions remain blocked until the transactions they depend on release their locks after termination, forcing them to run serially. An obvious downside to this is that if a transaction is particularly large and requires many locks, or is long running, the number of blocked transactions can grow exponentially, which incidentally slows down the entire system.

3.1.2 Timestamp-based Concurrency Control

An alternative to lock-based CC is to use timestamp-based CC protocols to preserves the serialization order of concurrent transactions. Each transaction is serialized with a timestamp to construct a dependency graph where newer transactions have dependency on older transactions if data access is to be shared [17]. However, maintaining accurate timestamps can be a challenge in a distributed system. If using the system's clock time, different sites must have their clock times synchronised using a protocol such as the Network Time Protocol (NTP). Another method is to use simple monotonically increasing time-stamp counter [18], but if one site is less active than another, the

differences between one site's local counter compared to a more active site could be exceptionally large. A transaction that originated from a less active site would therefore be interpreted by another site as an older transaction which could cause problems. Therefore, counters must be synchronised as well.

3.1.3 Single-Threaded Transaction Processing

Some database systems avoid the need for CC altogether by forcing transactions to run on a single thread, categorized as single-threaded databases. Because modern computers have a higher capacity of internal memory, serialized transactions can remain in-memory sufficiently. In addition, single-threaded transaction processing does not have the lock management overhead that is present in concurrent transaction systems [19]. However, transaction throughput is going to be limited to the use of a single CPU core, which means that transactions need to be processed fast to avoid blocking other transactions for too long. Fortunately, modern CPUs are increasing in processing speeds and single-threaded databases have seen adoption from popular database vendors; One example being PostgreSQL's query engine although it also uses thread scheduling to avoid blocks caused by accessing data across multiple partitions to continue processing if needed [20]. While they do not work for all types of database requirements, there are certainly some use-cases where they can be an efficient solution.

Parallelism can instead be achieved by partitioning data in a distributed setting where a single thread is in control of a separate partition. Transactions that span multiple partitions should be avoided, otherwise multiple partitions must manage partition-level locks and would suffer in performance costs [19]. If the transactions are short-lived, and transactions have been carefully designed to rarely need a combination of data from separate partitions, this can provide promising results. Stored procedures are a popular method for reducing the lifespan of transactions because the database does not need to communicate with the application to receive the next operation before continuing. Instead, the set of instructions for executing a transaction is contained with the database system rather than the application.

3.1.4 Snapshot Isolation

Snapshot isolation was introduced by Hal Berenson and Philip Berenson et al. in 1995 [21]. It is a weaker mechanism for guaranteeing isolation and follows an optimistic model. It is based off the works of multi-versioning concurrency control (MVCC) which is a method of storing different versions of the same data item for each currently execution transaction. MVCC was proposed back in 1981 by Phil Bernstein and Nathan Goodman to address the key differences between read-write and write-write synchronization [22]. The idea is that read operations should not block other operations and write operations should not block read operations. MVCC is used in popular modern database systems such as SQL Server, PostgreSQL, Oracle, IBM DB2, and more [23].

Instead of one transaction modifying a data value directly, another version of that data item is created with the new value assigned to it. Therefore, concurrent transactions can only use consistent values of data without the interference of other transactions, which preserves isolation as each transaction is unaware of the behaviour of others. Conflict resolution must instead be delayed until the end when transactions attempt to commit their changes. Database systems implementing MVCC can also support time-travel based queries to see changes of data overtime by comparing multiple versions of the same data.

The goal of snapshot isolation is not to enforce serialisability, but instead to enforce repeatable reads using a consistent snapshot of all data required by a transaction. Instead of creating multiple versions of data items, multiple versions of snapshots are created instead where a snapshot represents the portion of the database that a transaction relies on. This snapshot is taken directly before the transaction executes and can only be seen by that transaction. Read-only transactions avoid the synchronization overhead caused from serialization where even read-only transactions are blocked. During the commit phase, if the version of one snapshot is newer than another already committed snapshot targeting one or more shared data items, then the transaction using that newer snapshot is aborted and possibly retried. This is referred to as the “First-Committer-Wins” rule [24]. This prevents lost updates from occurring and can perform much better than pessimistic algorithms if the transactions infrequently make use of shared data.

3.1.5 Serializable Snapshot Isolation

A relatively new approach to snapshot isolation, referred to as serializable snapshot isolation (SSI), ensures that every execution is serializable whilst still maintaining the benefits of snapshot isolation (SI). First introduced by Michael J. Cahill et al. in 2009 [24], this method attempts to solve data consistency anomalies caused by traditional SI methods. Previously, explicit application-level locks were required to avoid some of the pitfalls of SI. Write-skew is a known anomaly of SI that occurs when one transaction T_i reads a value before a second transaction T_k updates that same value. Then, T_i uses the original value to decide on how to update another associated value. Unlike the lost-updates anomaly, this is much harder to detect because the isolation mechanism does not consider associated data. Because of this change, the original premise on how T_i reached that decision no longer holds true because T_i was unaware of the changes made by T_k. Because of this, SI is at risk of breaking the integrity constraints of the DBMS.

SSI supports the rules of SI by supporting non-blocking read and write operations whilst also preventing write-skew caused by concurrent transactions. In contrast to other lock-based methods such as 2PL, the performance penalty is relatively small. A major achievement of SSI is that it avoids the need for application developers to be educated on the shortcomings of SI and to compensate for them using explicit locking in application code. SSI detects conflicts caused by non-serializable concurrent transactions at runtime. If a transaction is non-serializable, it means that it cannot safely be serialized without moving the database into an inconsistent state. If SSI detects a pair of non-serializable transactions that conflict with one another, it will only abort the newer transaction rather than other optimistic CC methods that usually abort both. This is achieved all while maintaining a reasonably small overhead cost for storing metadata required for detecting conflicts [24].

3.2 Distributed Concurrency Control

A distributed computing system coordinates with other partitions of the database contained in individual “interconnected autonomous processing elements” [23], referred to as loosely coupled “sites” of a distributed database system [25]. These sites can be geographically distributed across a computer network and vary in their characteristics.

They may also be replicated or fragmented using sharding to improve the availability and scalability of the data.

A transaction manager (TM) is selected as the coordinating TM and is put in charge of coordinating the actions of other participating TMs running on other sites. The two-phase commit protocol (2PC) is a highly popular approach to coordinating transactions using a voting strategy between TMs. The simplest implementation of 2PC is known as centralized 2PC whereby only the coordinating TM messages the participating TMs. Then, the coordinating TM waits for the participants to approve the commit before sending out a global commit message during the second phase of the 2PC protocol. This finalises the decision and persists the changes at each site. Alternatively, if a single participant votes to abort the transaction, the coordinating TM receives this decision and sends out a global abort message to all participants. There is an exception to this rule where a participating TM can unilaterally abort the global transaction if a deadlock situation occurs [23].

Another approach is linear 2PC where participating TMs are ordered in a linear fashion with the coordinating TM at the front of the queue. Messages are then forwarded up and down the queue to collect the votes of all participants. This reduces the number of messages but at the cost of not supporting parallel processing. Alternatively, distributed 2PC gives the participants more control over the decision to commit or abort a transaction whilst also supporting parallel processing. This reduces the amount of network communication by making the second phase of 2PC unnecessary as participants can make their own decisions prematurely [26].

Managing locks in a distributed fashion has a similar set of alternative approaches. Two-phase locking (2PL) can be achieved using a centralized or distributed locking protocol. Centralized 2PL uses a single lock manager on a single site to grant locks on data as opposed to distributed 2PL where each site has a lock manager and does not need to wait for a centralised lock manager to grant them permission. However, distributed lock-based CC may cause deadlocks and must rely on a separate distributed deadlock management process [15]. Typically, a wait-for graph (WFG) is produced and maintained either in a distributed fashion where each site has their own local WFG and has the responsibility of detecting deadlocks, or a centralized site maintains a global WFG and takes the

responsibility of detecting deadlocks on behalf of all other sites. If using a centralized approach, each participating lock manager must send their local WFGs to the coordinating lock manager to be synchronized to form the global WFG [27]. The WFG represents a directed graph of dependencies between transactions. If a cycle occurs, then this means a deadlock has occurred and the appropriate measures can take place to abort the conflicting transaction/s.

SI largely remains unchanged when used in a distributed setting, except for the need to synchronize the monotonically increasing counter or timestamp (using NTP) as previously mentioned. Carsten Binnig et al. proposed an incremental approach to SI in 2014 [28]. They noticed that SI relies on upfront generation of snapshots which can increase overhead costs, but all known attempts to delaying this had occurred in high abort rates. Their solution was to generate the snapshots incrementally. First, the most recent local snapshot from the first originating site is used as the initial snapshot. Then, each time the global transaction is sent to other sites for further processing, the snapshot is extended using the local snapshots of the visited sites. From their research, incrementally extending the snapshot in this manner performed just as well for sharded or partitioned databases as for centralized, local databases.

Chapter 4

Implementing Distributed Transactions in Reactive Systems

Like previous methods of partitioning databases into isolated sites, the database per service pattern extends this idea further. Each service has its own database system with its own subdomain model and does not use any form of synchronization or concurrency control with other databases. Instead, each database remains completely isolated from other databases to achieve loose coupling between services. A large benefit of this pattern is that services are not constrained on the type of database software. One service might have a different set of hardware constraints or an entirely different data model that is better suited towards a different type of datastore such as a key-value or graph-based datastore. It also enforces DDD by keeping separate subdomain models entirely isolated from others, including how they are persisted, modified, and queried. Then, separate development teams are free to change any part of their subdomain or database without affecting other teams or services. A single datastore owns data relevant to a subdomain (within a bounded context) and can still be partitioned using the mechanisms and protocols of the chosen datastore technology. However, within an MSA the problems of distributed transactions previously discussed in chapter 2 extend further.

4.1 Problems of Distributed Transactions in Microservice Architecture

It is common for an MSA to require distributed transactions to span across multiple microservices and reliably modify each of the participating microservice's datastores in a consistent manner. Global transactions must still be rolled back if one microservice fails to send confirmation that it has successfully committed the requested change within a timely manner, else the domain may enter an inconsistent state.

To implement distributed transactions across an MSA, we must cater towards the polyglot nature of MSA database technology. For example, NoSQL databases cannot collaborate using 2PC and so they may be incompatible with other relational databases. The X/Open Distributed Transaction Processing Model (X/Open XA) uses 2PC as a way of achieving atomicity across distributed transactions and works across heterogeneous technologies

[29]. The problem is that not all database technologies support XA, which limits our flexibility to choose the most appropriate database technology for our microservices. It also means that the limitations of 2PC apply to all databases in the architecture.

We also want microservices to remain isolated from each other so that they can be scaled independently and introducing traditional distributed microservices may cause a bottleneck. For example, if one microservice becomes unresponsive (i.e. the transaction manager goes offline) and does not free the locks its distributed transaction has on other microservice datastores, then this impacts the performance of the entire system. In addition, the performance costs of coordinating locks does not scale well when more external systems are required for each transaction [30]. Therefore, having such a direct, shared dependency between each store defeats the goals of MSA.

According to the CAP theorem published by Eric Brewer in 1999 [31], within a distributed system you can only deliver two of the three properties: consistency, availability, and partition tolerance. Most modern systems prefer to achieve high availability and partition tolerance to support scalability as a trade-off from strong consistency. Instead, the term "eventual consistency" is used to refer to systems that may return out of date (i.e. stale) data from an out of date partition. Eventual consistency systems can maintain high availability and tolerate the loss of a partition without affecting its performance. When data is modified in one partition, the system takes time to synchronize those changes across multiple partitions. Distributed transactions can support strong consistency at the expensive of weaker availability caused by blocking transactions while locks are held on data. This is useful for some systems, such as online banking applications, but usually the user would prefer a reasonable level of stale data as opposed to long response times.

4.2 Reactive Systems

The reactive manifesto defines a standard for what a reactive system should be [32]:

1. **Responsive** – Systems must respond to user requests within a timely manner. This principle focuses on providing good user experience to build user confidence when using the system. By defining reliable methods for notifying the user after

a set time limit, and continue processing, if necessary, without the user waiting, the system is said to have an improved quality of service.

2. **Resilient** – The system should support strong fault-tolerance and recovery from system failure in a timely manner. If a system is not resilient, it will also be unresponsive after failure has occurred. Resilience can be achieved through replication and isolation of services and data. By delegating tasks using asynchronous IPC mechanisms, services can be safely restarted and tried later to improve resilience whilst also remaining responsive.
3. **Elastic** – The system should be able to dynamically scale-out based on varying network traffic loads and resource requirements to preserve responsiveness. Isolation and delegation can support the elasticity of a system by removing all central points of failure.
4. **Message Driven** – A core principle for reactive systems is to favour asynchronous message-passing IPC mechanisms with fault-tolerance in mind, over synchronous IPC. Message-driven architectures enforce non-blocking communication, which reduces the resource overhead caused by keeping connections open while waiting for results. Instead, subscribers or callback functions are used to process results only when required. It also favours location transparency as a method of supporting loose coupling through isolation and modularity.

MSA using message-driven IPC mechanisms are a natural fit for implementing DDD. By only relying on messages, such as serialized objects, and not sharing object references between microservices, the subdomain models of each microservice can be protected from corruption. When implementing message-driven architecture, developers must approach it with a vastly different mindset from traditional CRUD-like (create, read, update, delete) API designs typically implemented using RESTful principles. Rather than using a single blocking HTTP request to query the system for data, or to perform an action, and returning the result straight away, a message-driven system will continue processing the original request by sending messages to other microservices asynchronously. Each microservice must complete a relevant portion of the original request where each response forms the aggregated result to be sent back to the original client recipient. During this time, the client will either need to poll the system to check if the result is ready for them to read, or other bidirectional forms of communication are

needed for the server to send the request to the client directly. Web-sockets are the most common form of bidirectional communication with many implementations supporting fallback methods in case the client's software (e.g. a web browser) does not support them.

4.3 Aggregating Query Results

If a service requests data from multiple microservices and expects multiple results messages back, these will need to be aggregated in some way before sending the aggregated result to the client. One method is to incrementally build it by passing the previous microservice's result inside the message sent the next microservice, but this model cannot be parallelized, resulting in higher communication overhead and response times. Another method is to use a centralized aggregator component to send out all requests and wait until all results have been retrieved before aggregating the results. This avoids the need for each microservice to know about the other microservice participants and keeps each microservice simple in design. These benefits come at the expense of a single point of failure caused by the aggregator.

4.4 The Saga Pattern

By their nature, traditional globally distributed transactions are a form of synchronous IPC, which have a negative impact on availability. For MSA implementing eventual consistency, asynchronous IPC is preferable and so these transactions must be implemented differently. When using message-driven IPC, a message can represent a single transaction and multiple messages using a shared transaction ID can be used to represent a globally distributed transaction. For this, we turn to the saga pattern.

Queries are not the only concern of message-driven architecture. Performing distributed transactions through messaging means that we lose the benefits of CC protocols such as 2PC. Instead, it is up to the application to implement CC mechanisms, such as timestamp ordering or locking resources, and rolling back transactions in the face of failure. Message-driven architecture, unlike previous RPI strategies, make the pitfalls of distributed systems explicit. While it can be a complicated learning curve for developers, it does prevent the misconceptions that the vision of a unified object model had previously

led to. It also allows developers to follow the reactive principles by having greater control over communication and fault-tolerance to build responsive, resilient systems.

The saga pattern provides a method of simulating distributed transactions within an MSA. It caters towards isolated databases that are not coupled based on previously mentioned protocols and algorithms. Instead, the saga pattern allows us to coordinate our transactions within a message-driven system. A transaction request is sent to a microservice in the form of a command message. The command contains the necessary data to perform a sub-transaction on an individual microservice with optional metadata such as a given timestamp or a global transaction ID.

4.4.1 Compensating Transactions

If a microservice fails to return a message signalling that the sub-transaction was executed successfully on its datastore within a given time limit, then the saga pattern suggests using compensating transactions to undo the global transaction. Because each microservice executes transactions on their isolated datastores, we cannot simply abort and rollback those transactions because they have already been committed. Instead, rolling back a saga is achieved by executing a series of compensating transactions to reverse the changes made to each participating datastore. Compensating transactions typically make us of commands that implement the opposite behaviour of the commands executed as part of the failed saga.

A command reflects an application-specific unit of work. Rather than creating a command to undo a local transaction, the command should implement a real application use-case and should be used by compensating transactions to apply a second update to the state. Therefore, the state of the system does not rollback to the exact same state it was in before performing the saga, but instead it moves forward to a new consistent state to undo the transaction logic. For example, when a command reduces the stock quantity of an item a customer has bought but their payment method is declined, the compensating transaction could execute a command to add an item to the stock, which may be used in other areas of the system. This supports code-reuse by avoiding the need to create a separate process to handle saga failure, allows audit logging to track the history of commands, and reuses

existing application logic to maintain data consistency. Event stores work well in this scenario but are not required for implementing the saga pattern.

Compensating transactions using the saga pattern are executed in reverse order to maintain data consistency. If a saga executes three out of five transactions but the fourth fails, then the compensating transactions are executed in reverse order where the first one undoes the last successful transaction to have executed as part of the saga. This means that if a concurrent request is made during the rollback of a saga, they see a consistent state where integrity constraints are preserved. Otherwise, you risk other transactions coming to conclusions based upon a misleading premise, like the write-skew anomaly caused by snapshot isolation discussed in chapter 3. It is worth noting that some transactions do not require a compensating transaction, such as read-only transactions. Other transactions that are not critical for the performance and consistency of the system can be allowed to fail without aborting the whole global transaction if the application logic takes this into consideration without creating a confusing user experience.

4.4.2 Event Orchestration and Choreography

The two most popular choices for implementing the saga pattern is to either use orchestration or choreography where the results of commands are emitted as events to represent that something has occurred. This allows the system to listen out for events of interest and react to those events in an appropriate way to progress to the next step of the saga. The saga pattern does not enforce what type of IPC mechanism should be used. Events can be directly sent to participants, but a reference is required, which creates a dependency between the services. Other location transparent approaches can be used to reduce coupling and allow additional services to participate in the saga without changing any core logic. For example, events could be placed on a queue or emitted as part of a publish-subscribe model.

When using orchestration, a single orchestrator component receives all events and executes the next transaction, or set of parallelizable transactions, in the sequence and returns the result to the client recipient. The orchestrator contains all network communication logic, such as timeouts and retry logic possibly implemented using the circuit breaker pattern. If the orchestrator goes offline during execution, the orchestrator

must be able to recover and continue from where it left off while preserving consistency. Therefore, microservices must implement idempotent commands so that triggering the same command more than once with a given transaction ID or timestamp does not corrupt the domain's state. Ideally, the orchestrator should be able to horizontally scale to avoid becoming a bottleneck.

Implementing the saga pattern using choreography increases the complexity of the dataflow. Each microservice oversees the execution of the saga whilst also handling compensating transactions. There is no central orchestrator and so each microservice must implement their own fault-tolerant mechanisms previously described. Unlike an orchestrator, it can be difficult to see an overview of how a saga is carried out or what participants are involved without looking through the entire codebase. If separate development teams are working on separate microservices, more team collaboration is required. However, the message overhead is reduced because microservices do not need to send an event back to an orchestrator and can instead be picked up directly by the next participating microservice/s. When using the publish-subscribe pattern, each microservice can be observed by other microservices. An event published by one microservice can be received by all subscribed microservices, which triggers each subscriber to execute local transactions to progress the saga. If a message is to be sent back to the client, a separate saga-terminated event can be emitted to mark the end of the global transaction so that a separate process can subscribe to this event and notify the client.

One study conducted in 2018 found that choreography performed much faster in comparison to orchestration [33]. However, as more events increased, the complexity when using choreography became difficult to reason with. Thus, the study suggested that choreography is more suitable when used with fewer events where fast response time is critical. However, orchestration makes it easier to avoid cyclic dependencies between microservices. By keeping microservices simple and using the orchestrator as the only subscriber, it reduces the number of dependencies and avoids microservices accidentally depending on each other for shared data to complete their local transactions.

If using DDD, microservices should only need to perform basic CRUD-like operations and more complex application logic relating to its own subdomain. Sometimes, it is

necessary to hold foreign keys relating to another subdomain, which can be acceptable if following the aggregate root pattern, but this should be minimized. If one microservice requires data from another subdomain, causing a dependency, then this is easier to manage with orchestration while avoiding cyclic dependencies. The orchestrator can first request the necessary data from one microservice and then send it as part of a follow-up request to another microservice in sequential order.

4.5 The Actor Model

In 1973, Carl Hewitt, Peter Bishop, and Richard Steiger first proposed a new architectural model based on small units of work, called actors, for simplifying CC [34]. The behaviour of actors relies exclusively on the receiving and sending of messages to invoke the actor to perform a task. An actor model provides a high-level abstraction of CC within a distributed system. Actors can hold private state data and can stay alive for any arbitrary amount of time as required, but it can only perform a task for a given message serially. Multiple actors can be created to execute tasks in parallel but can only modify their own encapsulated private state data, which avoids concurrent access and data consistency anomalies. It also avoids the need for explicit lock-based synchronization of resources, allowing for a much simpler development experience.

The main motivation for the actor model is to facilitate highly parallel computing by scaling out the number of actors in an actor system. To support this, many framework implementations of the actor model support location transparency, clustering, and remote messaging. Although the theory of the actor model has been around for a long time, it never saw wide-spread adoption due to the limitations of technology. However, in recent years it has seen a resurgence in popularity due to the increase in performance provided by modern computer architecture with many frameworks proposed. From as early as 2020, several actor frameworks released new versions for modern languages such as Java, Python, C# and Rust, with new ones currently in development. They have been used in a variety of domains, most notably for games development, web applications, artificial intelligence, multi-agent systems, and for the internet of things (IoT) [35].

4.5.1 Fault-Tolerance in Stateful Applications

Object-oriented programming (OOP) models provide features for encapsulation, but all code is executed within the same thread. When multiple threads are executed within the same system, it is common for different threads to operate on shared code, whether that code is a method or stateful data. This is a potential risk to shared data and creates an illusion of encapsulation. Most high-level programming languages provide locking mechanisms to avoid more than one thread having access to the same data, but this can reduce performance and comes with its own technical risks such as deadlocks caused by cyclic dependencies. The CPU cost involved with suspending a thread and restoring it later once a new lock is obtained can be substantial, especially for high performance computing (HPC) systems. In addition, blocking threads reduces response times and wastes resources [3].

As we discussed with database-level locks, locks are a way of controlling concurrent access by forcing serialization, which does not scale well in an MSA. The actor model addresses this by forcing messages to be processed one at a time by individual actor instances. Actors are then free to maintain their own state, which works especially well for long running actor instances processing a group of related messages. However, if an actor instance restarts it will lose its private data. The event sourcing pattern addresses this issue by recording events that caused a change in the application's state. These events are appended to an event store and if the application restarts, its state can be recovered from the store. This works well for preserving and recovering the states of individual actors. Typically, the state of an actor represents some user interaction with the system and must be kept alive until that interaction terminates. In this sense, a stateful actor can represent a materialized view of ephemeral event data for a given user session. Developers are often forced to store short-lived data in persistent storage or as session data, but event sourcing allows stateful data to be used in-memory, avoiding the need to query a database each time the data is required. This reduces the resource overhead caused by establishing connections with the database. Finally, event sourcing incidentally maintains an audit log representing the series of events that occurred during a user's interactions, which is useful for debugging purposes.

An actor's state may influence how an actor reacts to incoming messages. Some actor frameworks approach this by representing actors as finite-state machines (FSM). If an actor is in one state it may refuse specific message types while allowing others. Other message types might trigger the actor to transition to a new state. For example, if the user is currently adding items to a shopping cart then the actor might be in a unique shopping state. Once the user goes to checkout, the actor may need to change to a new checking out state where no more items can be added to the actor's state data. If the actor restarts and recovers their state from the event store, then the user will not lose their items in the shopping cart. Actor state data should be kept private from other actors in the system and should be immutable to maintain data consistency when handling multiple messages accessing the same state data.

Fault-tolerance is one of many quality-of-life benefits ingrained within the actor model, which many actor frameworks typically support. The idea is to allow actors to create other actors, referred to as child actors, for delegating workloads, potentially in parallel, where the creator actor is said to be the parent who manages their lifetimes. If a child actor experiences a failure, the parent actor can act as a supervisor by deciding which recovery strategy to use on behalf of the child. For example, it could decide to restart the child or to retry the same message multiple times until it reaches some predefined maximum threshold, or possibly to log the error and continue onto the next message for situations where the task was not urgent.

By using a hierarchical supervision strategy with external fault-tolerance mechanisms such as event sources, the system supports the resilient and elastic principles of the reactive manifesto. Responsiveness is also achieved by allowing actors to divide large workloads between child actors to be executed in parallel, especially when using stateful actors to access state data instead of directly querying a database. Some actors may be deployed to different actor systems on different nodes within a cluster either explicitly or dynamically depending on the volume of incoming requests. In conclusion, the actor model is a perfect fit for implementing a reactive system and can address complex data consistency concerns and be used to improve the overall scalability of the distributed system.

4.5.2 A Short Introduction to Akka.Net

There are many implementations of the actor model for different programming languages and frameworks. Akka.Net is an implementation of the actor model for .NET developers. It consists of a set of open-source packages to provide many CC features for building distributed reactive systems, such as FSMs, hierarchical supervision, mailbox queues for processing messages in serial order, routing strategies to dynamically scale actors, and more. It is based off the original actor model theory proposed by Carl Hewitt but also offers a feature-rich ecosystem for addressing the challenges that modern computer architecture's face when designing distributed systems [36].

Akka.Net was ported from the popular Akka framework to be used by .NET developers. The original Akka framework, released by Jonas Bonér in 2010 [37], was designed to run on a Java virtual machine (JVM) and supported both the Java and Scala programming languages. Akka attempts to increase the level of abstraction by alleviating the need for developers to manually manage locks and synchronization required by previous Scala CC mechanisms, with the goal of providing a framework for building highly concurrent, event-driven applications. Akka.Net is based on Akka but has evolved over the years to improve the integration with newer .NET Core technologies. While there are many alternative actor model frameworks, this project makes use of Akka.Net with other .Net technologies to build a reactive system.

Chapter 5

Design and Implementation

As part of this dissertation, a reactive, distributed MSA was developed using Akka.Net to handle CC and scalability. Each microservice was implemented using the latest version of .NET, which at the time of writing is .NET 5 (renamed from .NET Core 3.0). The project includes a front-end, single-page web application (SPA) to send requests to the MSA, developed using a customised version of the React.js library, called Gatsby.js. Gatsby.js simplifies many details of front-end development, such as optimising page load speeds, and offers many templates to alleviate the developer from writing lots of boilerplate code. The SPA uses the Model-View-View-Controller pattern by dynamically updating separate components, using a view model, retrieved from the MSA. Communication between the two was implemented with SignalR; a WebSocket library developed by Microsoft with support for falling back to older transport technologies, such as long-polling, if the web browser does not support the use of WebSockets.

WebSockets are a bidirectional communication technology, allowing the server to message the client when data is available. Because the MSA relies on asynchronous, non-blocking, message-driven IPC to create a reactive system (based off the reactive manifesto), the traditional synchronous request-response model would not suffice. Messages sent to individual microservices to fulfil a client request take an arbitrary amount of time to complete with some requests taking much longer than others. If using a traditional request-response model, the system would waste resources while keeping the original TCP connection to the client open and would possibly timeout due to the nature of MSAs. Also, parts of the system would be blocked from processing further requests while waiting for microservices to return separate responses to fulfil the requirements of the original request. This would not scale well, and so the MSA must send back a separate second response bidirectionally, without the client needing to request it, to avoid timeouts and blocking synchronous IPC.

However, the user experience must be carefully designed with asynchronous IPC in mind by loading resources in the background and displaying a loading animation for long running requests while keeping the user informed. Gatsby.js is referred to as a static site

generator by making static content immediately available for the user. Some client-side views are entirely static except for HTML forms used to submit POST requests to the server. Other views make use of an application shell pattern by making the static content immediately available and using placeholders and loading animations while the dynamic content is being retrieved.

5.1 An Overview of the Web Application

The goal of the web-based application is to provide a platform for users to manage projects and discussion groups for the purpose of collaboration. A user can create a project page and invite other users to help maintain the project as part of a team. The project owner can manage invited users by assigning them to different teams with various permissions to restrict what actions they can perform on the project page. A project page may contain instructions on how to download and install an external application, such as an app on an app store. Users can browse different project pages and can submit issues they are having with this external application on an issue listings subpage as shown in figure 5.1. Team members can then manage these issues by responding to them and closing them once the issue has been resolved.

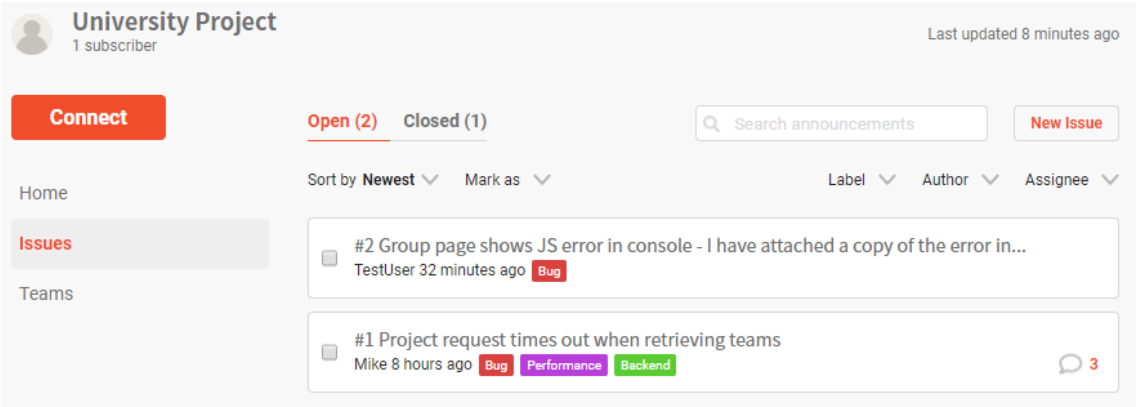


Fig. 5.1 User submitted issues on a project page.

Discussion groups can be created by any authenticated user, which can optionally be linked to one or more project pages. A project linked to a group will show a link to that group's page from the project page, allowing users to discuss the project on the group's discussion section. The purpose for the group page is to provide a community aspect to projects, but the discussion group can have no projects assigned and be used as a general-

purpose community based around shared discussion topics. Figure 5.2 shows an example of a user submitted post on a group page with a comments section. Users can submit markdown text for richer text content, but any special HTML or JavaScript characters are escaped to avoid cross-site scripting (XSS) or HTML injection attacks. Other users can comment, “like” (using a heart icon) and save posts and comments, like a typical social media platform.

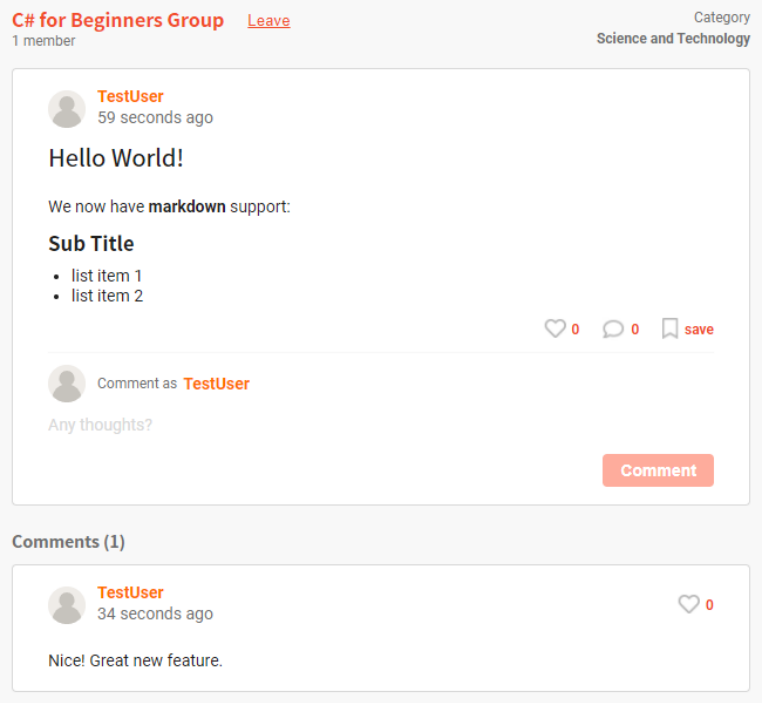


Fig. 5.2 A group discussion area with comments.

When creating a group, project, or post, the user will see a loading screen while the MSA processes the request asynchronously, as shown in figure 5.3. Once the user receives a second response from the server using SignalR, they are redirected to a page showing the contents of the newly created entity. A green notification appears on the screen to inform the user that their request was successful. If the server returns a user-friendly error message (e.g. a validation failure message or an internal server error), a red notification box appears instead as a method for keeping the user informed throughout the experience. The form contains client-side validation logic to highlight areas in red with text if the user fails to fill in the form correctly. The server contains the same validation logic to validate the POST request body once submitted to avoid attackers bypassing client-side validation, whilst also analysing user claims to check if they have authorization for specific

endpoints. Each microservice may contain additional business rules by comparing the submitted data with their datastore to see if certain actions are permittable.

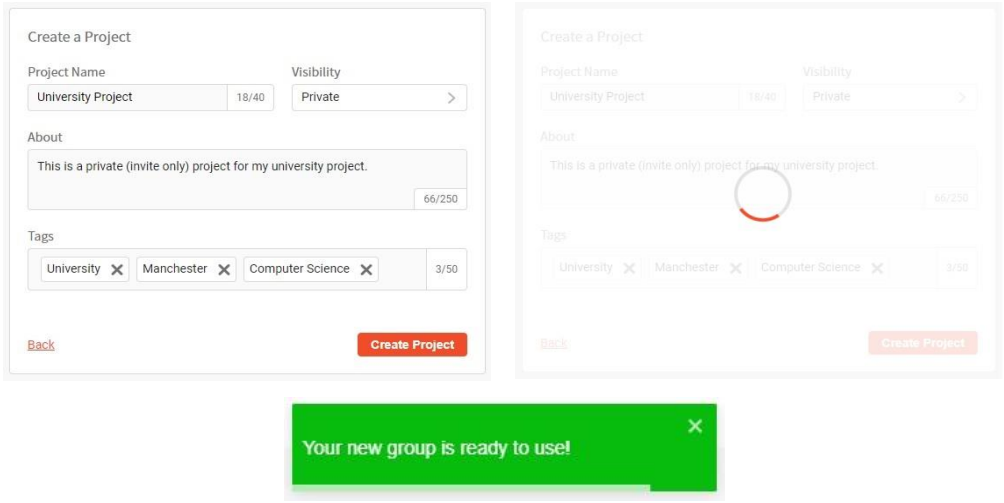


Fig. 5.1 User experience while waiting for a project or group to be created.

Other minor features are included in the application to create additional complex scenarios with potential data inconsistency risks to improve the research evaluation quality. For example, on the home page there is a recommended groups and projects area as shown in figure 5.4. If the user deletes a group or project, then this change must be synchronised across each microservice datastore with user claims updated. Otherwise, bugs may occur from an inconsistent domain model. Implicit bugs are harder to detect, such as a business rule failing due to foreign keys referencing a deleted entity, while others may be explicit runtime errors, such as an object null reference exception. If a user attempts to join a group in the process of being deleted, then this can cause undetected data consistency anomalies.

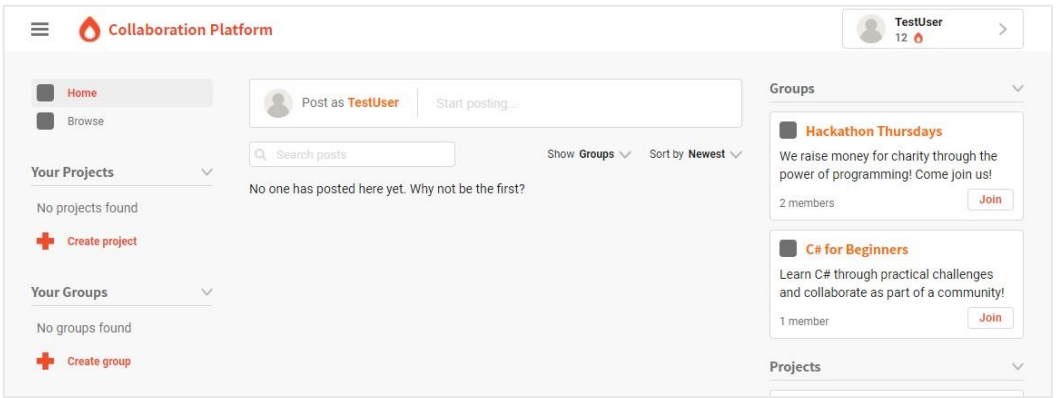


Fig. 5.2 Home page showing a recommendations section on the right.

5.2 Overview of Microservice Actor Systems

Each microservice maintains their own actor system, implemented using Akka.Net, and makes use of message-driven IPC using actor references to maintain location transparency. Each microservice shares a class library containing unique classes representing different message types and view models where instances of these classes are serialised and sent using an actor reference's *Send* method. Akka.Net handles actor references to coordinate messages to their destination. The destination actor could be deployed remotely on a separate network cluster as part of another microservice, or it could be local to the sending actor. Akka.Net hides these low-level details from the developer while making the message-driven IPC and nature of the MSA explicit. Unlike older RPC technologies, the developer always plans for remote messaging even if some references may be local, instead of assuming that everything is local, which often results in false assumptions about the network.

The design and implementation of each actor system was inspired by the aggregate root pattern from DDD. Each system includes a root manager actor as the single point of entry, which forwards messages to the appropriate child actor. The manager acts as a supervisor for all created child actors; if one experiences failure, the error is propagated back to the manager who then decides how to mitigate the problem. The manager is therefore responsible for returning the system back to a healthy state to continue the processing incoming messages with minimal downtime.

Rather than creating new instances of child actors directly, the manager uses a customised version of the factory-method pattern by specifying how a child actor should be created using a factory read-only property (a C# getter property). This is used by Akka.Net router actors, each using a round robin routing strategy, as shown in figure 5.5. Router actors create instances of child actors as part of a pool to handle specific message types. There are many pools of child actors for each type of actor. Actors have been designed to contain small chunks of relative logic to isolate system failure and avoid it affecting more functionality unnecessarily. For example, if there is a problem with an actor whose responsibility is to create a project, then the actor responsible for updating a project, or retrieving a list of recommended projects, can still remain active even though they both share the same microservice and actor system. Each actor pool has been configured to

create between 1 to 10 instances of the same actor to scale based on fluctuations of network traffic. The manager may have several different pools based on the number of actor types that need to scale.

```
// GroupManagerActor class field
// A pool of SearchGroupsActor instances controlled by a route actor
private readonly IActorRef _searchPool = Context.ActorOf(
    SearchGroupsActor.Props, "SearchQueryPool");

// SearchGroupsActor class property
// A factory property using round robin
public static Props Props { get; } = Props.Create<SearchGroupsActor>()
    .WithRouter(new RoundRobinPool(5,
        new DefaultResizer(1, 10)));
```

Fig. 3.5 A code sample to show a pool of child actors controlled by a router actor using a round robin routing strategy.

The round robin routing strategy places each created actor into an actor pool and distributes the messages in circular order to be processed by different instances of the same actor in parallel. Each actor processes incoming messages sequentially using a mailbox queue controlled by Akka.Net. New messages delivered to an actor are enqueued and are individually taken off the queue by the actor one at a time for processing. The actor can then optionally reply to the original sender, if required, by sending a follow-up message containing results, or to signal a success or failure status.

5.3 API Gateway

The project consists of 4 microservices, one for each context based on the DDD bounded context pattern: teams, discussions, projects, and groups. The back-end infrastructure also includes an API gateway to manage all communication between microservices. It also receives all incoming front-end client web requests and contacts the appropriate microservices on behalf of the client. This allows the gateway to potentially scale, using a load balancer, to react to changes in network traffic and to keep the underlining MSA implementation hidden from the client, as shown in figure 5.6. The client only requires the address of the gateway rather than requiring the address of each microservice and background knowledge of multiple IPC requirements. This simplifies the design as each subsystem has its own well-defined responsibility and compliments the separation of concern principle at an architectural level. The gateway has two primary responsibilities:

enforcing security requirements, such as authorization and validating incoming requests, and coordinating messages between different microservices to handle the original request.

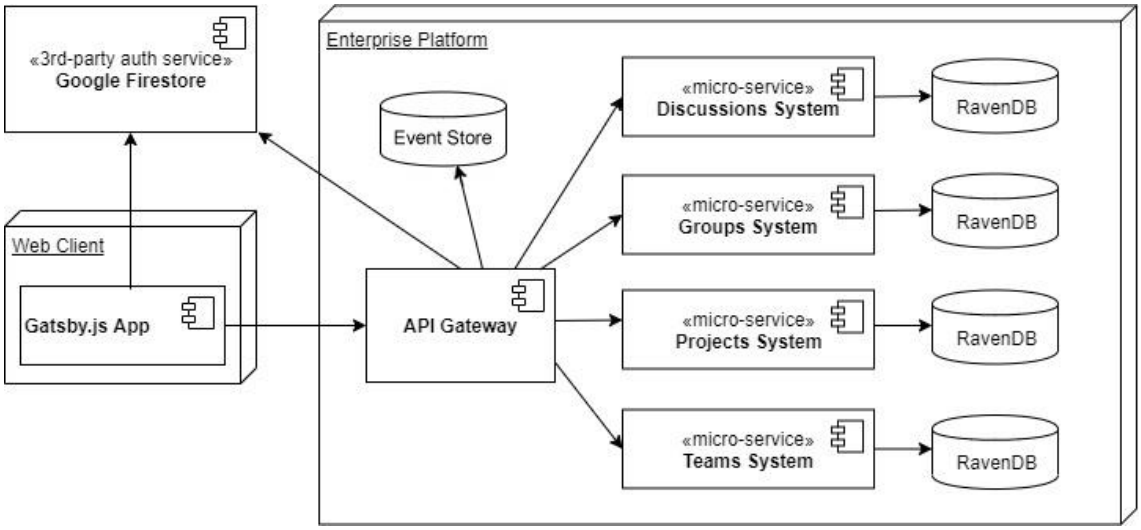


Fig. 5.4 Using an API gateway to separate the web client from the microservices.

A web request sent to the gateway from the client results in a message being created (after the request has been successfully validated) and sent to the appropriate microservice/s to fulfil the original web request. Five types of requests were identified as part of this research project with three types of messages used to fulfil those requests: commands, queries, and events. Below is a list of all five request types with descriptions for the types of messages they use and for what purpose:

1. **Commands** – Uses a command message to target a single microservice to change the state of its datastore.
2. **Queries** – Uses a query message to fetch data from a single microservice datastore without changing its state.
3. **Multi-Queries** – Executes a group of query messages where each query can be sent to a different microservice. The results can then be aggregated together once received. Some scenarios might allow for each query to be sent in parallel for reduced response times. However, some queries may need to wait for others to return the requested data (referred to as the payload) before they can be constructed and sent. This forms a dependency between queries, causing

sequential steps of multi-query processing. Multi-query messaging requires the use of an FSM to move to the next state when all payloads are received.

4. **Sagas** – A saga consists of multiple command messages, which tell one or more microservices to change the state of the subdomain persisted by each microservice datastore. Sagas represent a globally distributed transaction and are managed by an FSM following the saga orchestration pattern.
5. **Events** – Unlike the query and command message types, which are used to handle a request, events are only sent as a response from a microservice to inform the gateway or client that something has occurred. The most common event is the *PayloadEvent*, which is used to wrap the requested data (previously requested by a query message) or error message with meta-data to be sent back to the requestor. Other events may be sent to a saga orchestrator when a subdomain has changed state using a human-readable event class name, such as the *ProjectCreatedEvent*.

The gateway sends out messages to remote actor systems, where a message is either a command or query. The actor may then send a response, in the form of an event message, back to some type of callback actor located on the gateway's local actor system. Most events notify the receiver that something has changed, however the *PayloadEvent* is the odd exception because its purpose is to contain requested data after an arbitrary amount of time. The *PayloadEvent* also contains a boolean property to flag if the initial query message was carried out successfully. If the microservice's application logic notices that the data is not available, possibly due to a lack of user permissions or it does not exist, then this can be set to *false* and the payload data can contain a list of optional user-friendly errors to appear on the client's web page. For simple command and query message types, any event triggered as a result from this is sent back to a generic callback actor, which then passes it to SignalR to be received by the client.

5.3.1 Request Validation and Authorization

The gateway consists of a SignalR API hub containing multiple predefined endpoints for query-based requests to be sent to. The application has been designed to only use the API hub to handle query-based requests as they require less strict authorization and validation requirements. For command-based requests, ASP.Net web API controllers are used. All

controller endpoints are protected globally with an authorization filter, which requires the user to be authenticated by the external Google Cloud Firestore authentication provider.

Firestore offers many other features but for this project it is only used to control user sessions on the client-side (i.e. logging in and out), and authentication and authorization on the gateway. The gateway also contains a FirestoreService class used to update user claims using a Firestore client package. For example, if the user joins a new group then a claim is added and stored in Firestore to allow future requests sent by that user to be authorized to access that group's data. Of course, further microservice application logic may still restrict certain data access requests, but the gateway can filter out unauthorized requests early if a claim is missing, avoiding unnecessary messages being sent to microservices.

Web API controllers only support POST requests and validate the contents of the request body using ASP.Net's model binding capabilities. Controllers can then reject invalid data and immediately return a bad request synchronously without communicating with external microservices. If the data is valid, the controller sends the request to a mediator class which decides how to process the request asynchronously and the controller sends a 202 accepted response back to the client without waiting for the request to be completed, as shown in figure 5.7.

```
[ApiController]
[Authorize]
[FirestoreAuthorize]
public abstract class BaseController : ControllerBase
{
    protected readonly IMediator Mediator;

    protected BaseController(IMediator mediator)
    {
        Mediator = mediator;
    }

    protected async Task<ActionResult<string>> HandleRequest(IRequest<ValidationResult> request)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState.Values.SelectMany(v => v.Errors));

        var result = await Mediator.Send(request);

        if (result.IsValid)
            return Accepted(result);

        return BadRequest(result.Message);
    }
}
```

Fig. 5.5 The parent/base controller (for all web API controllers) handling command-based requests.

The mediator supports the single responsibility principle by keeping controllers simplistic; controllers are only responsible for returning HTTP responses with the appropriate status codes and validating incoming request bodies. The application logic involved with handling that request is forwarded to the mediator who decides what types of messages must be sent to what destination/s to fulfil that request. Authorization logic is self-contained in filter attribute classes which are applied to the controller class and are executed as part of the request pipeline before reaching controller actions.

Most SignalR API hub endpoints are not protected by authorization because visitors to the web application may not be logged in but should still be able to view the contents of web pages by requesting data. Some query-based requests may need protecting, such as an unauthenticated user attempting to access team-based private content such as a team's private chat history. Other claim-based validation checks may be performed by additional filters, assigned to API hub endpoints, by examining the user claims, similar to controller endpoints but for query-based GET requests. For example, if the intent of the request is to retrieve team-based data using a supplied team ID as part of the query-string parameter list, then a filter can apply pre-validation logic to analyse the user's claims previously provided to the gateway by Firestore during the request pipeline.

Additional checks relating to application logic may be required once the query reaches the appropriate microservice. For example, an authenticated user may be a member of a project team but lack the required application-specific team permissions to edit the

contents of project details contained on the project's page. The microservice, in this case, would need to use the data in its datastore to validate the request and send a failed message type back to the API gateway, which is then sent back via SignalR to the client using the previously established SignalR connection. If no such connection exists, then this means the user most likely disconnected by closing the web application and thus the response is lost until the user next reconnects and sends another request.

5.3.2 Saga Orchestration using Finite-State Machines

Sagas execute multiple commands to fulfil a globally distributed transaction spanning multiple microservices to update one or more datastore. Some microservices may be used to validate the global transaction by combining their subdomain application logic and datastore, whereas others may instead need to execute sub-transactions to modify their datastore. Once a sub-transaction has executed and committed successfully, it cannot be rolled back using conventional transaction principles. Therefore, if any pivotal sub-transaction fails to execute then all preceding sub-transactions must be rolled back using compensating transactions by the orchestrator to preserve the consistency of the domain.

Event orchestration was chosen over choreography for handling sagas due to Akka.Net's FSM support. By using FSMs, it is easy to support fault-tolerance using an event store to recover a saga's state data in case an application failure occurs before the saga has safely terminated. By centralising the saga logic, it is easier to build such recovery mechanisms as an extra level of protection across the entire process without code duplication, as an effort to protect data consistency. This allows an FSM to own recoverable shared state data, which can contain the results of events received from previously executed commands as a strategy for deciding what should happen next.

As previously discussed in chapter 4, section 4.4.2, previous research suggests that choreography-based saga management involves fewer network requests and results in lower response times at the cost of higher implementation complexity. However, an early theory as part of this research was that orchestration might reduce the number of data consistency bugs by providing full visibility to the state of the saga using FSMs. Also, reducing the level of coupling between microservices was desirable to promote adaptability and extensibility, which orchestration supports by acting as a bridge to

separate direct communication between them. Additional microservices could easily be added to the MSA to extends its functionality because the gateway would only need to know about the additional address for the manager actor, serving as the aggregate root for that bounded context. By using separate saga orchestrator FSM (SO-FSM) actors to handle each saga on the gateway, microservices can easily be modified without affecting the other microservices involved. Their only dependency becomes the SO-FSM actor itself, which reduces the number of messaging specifications, version control and development team collaboration, resulting in a cleaner architectural design and potentially faster releases to production.

The slight increase in response times caused by orchestration may be a desirable trade-off for a simplified design if the system can afford it. Some systems, such as HPC systems, will need to prioritise fast response times and will most likely benefit from choreography over orchestration. However, this project's web application does not require optimisation to the extent that an HPC system might need. It also attempts to make up for this by rendering the static content of the page almost instantly using Gatsby's static content generator and showing a loading symbol for dynamic content in the process of being queried for, to provide a reasonable user experience.

This project uses Akka.Net's FSM support to implement a unique style of FSM-based orchestration that aims to achieve well-defined states to avoid confusing data flows with the hope of avoiding bugs. Most importantly, it aims to fulfil a global transaction with a strong priority towards maintaining data consistency using carefully controlled state data that is both recoverable and reusable across states. Figure 5.8 shows the IPC dataflow across multiple remote actor systems when using saga orchestration.

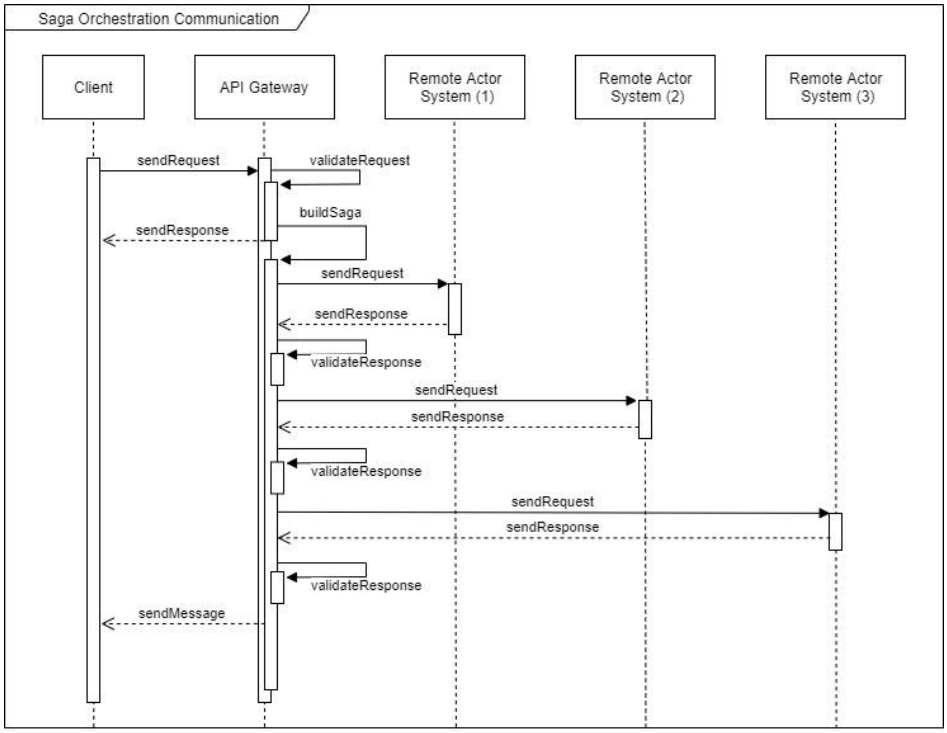


Fig. 5.6 Saga orchestration communication dataflow.

After the client request is validated, the saga context is built and a special *SagaExecutionCommand* is set to a SO-FSM to begin the saga. The saga sends multiple requests in the form of a command messages to a remote actor system during each state. The receiving actor in that remote system processes the command and responds with an event message containing event data. The data can be read and possibly stored in the saga's shared state data (SSD) for future commands to use. The SO-FSM decides if it needs to move to a different state based on the contents of the SSD. If no more commands are required, the saga can optionally send a message to the client with any results using SignalR, and then moves to a terminated state.

```
public CreateProjectSaga(IActorSystem actorSystem, IFirestoreService firestoreService)
{
    _actorSystem = actorSystem;
    _firestoreService = firestoreService;

    StartWith(SagaState.Idle, new CreateProjectStateData());

    When(SagaState.Idle, HandleIdleEvents);
    When(SagaState.CreatingProject, HandleCreatingProjectEvents);
    When(SagaState.CreatingTeams, HandleCreatingTeamsEvents);
    When(SagaState.RollingBack, HandleRollingBackEvents);
}
```

Fig. 5.7 The constructor for an FSM saga orchestration actor.

When a request triggers a saga, the SO-FSM is created by the gateway's local actor system. During its construction, it registers all the possible states it can move to using the inherited *When* method, and what method should handle the incoming events while in those states, as shown in figure 5.9. Each FSM starts off in an idle state with an empty SSD class.

The saga's SSD is persisted using an SQL Server event store using the *Akka.Persistence* and *Akka.Persistence.SqlServer* packages. It is basic in its implementation due to most of the configuration being handled by a HOCON (human optimized configuration object notation) file, so only a few lines of code are required to integrate event store persistence and recovery into the SO-FSM. Each time the saga's SSD changes, the change is persisted in the event store by serializing the SSD into a JSON object with other fields representing metadata for the event, such as a timestamp, persistence ID, sequence number, and more. If the SO-FSM restarts, the SSD can be rehydrated to return it to where it last left off. This is particularly useful for ensuring that a global transaction either fully commits or is rolled back using the saga's compensating transactions. If a global transaction only managed to execute a few of its sub-transactions (i.e. only a few commands to trigger them were sent out before the SO-FSM restarted), then the event store can replay the changes made to its SSD so it can decide what to do next and recover from system failure. This is highly desirable for maintaining data consistency while also supporting the resilient principle of the reactive manifesto.

5.3.3 Multi-Query Handlers and Aggregators

The multi-query handler FSM (MQH- FSM) actor has the responsibility of sending multiple query messages to request data from different microservice actor systems. It then moves to a waiting state where responses are gathered. Once all responses are received, it forwards the list of response messages to an aggregator actor. The aggregator must then convert each message into a single message to be sent back to the client using SignalR, as illustrated in figure 5.10.

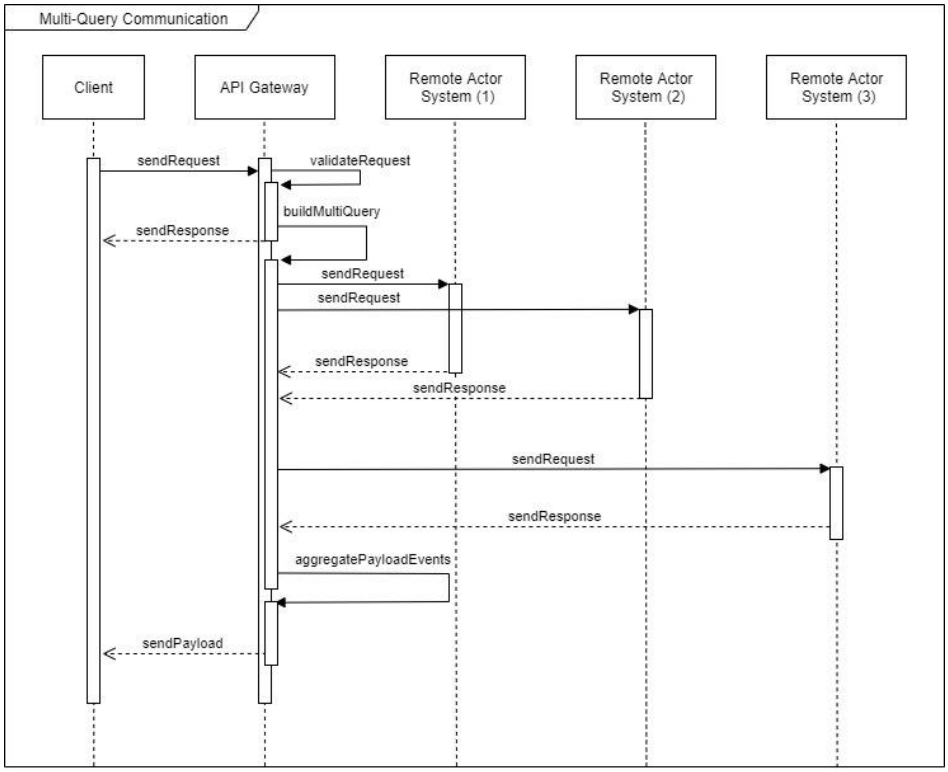


Fig. 5.8 Multi-Query communication dataflow.

The types of aggregator and MQH-FSM classes are specified by the mediator when building a context object to represent how the request should be handled. Each aggregator actor class extends from an abstract base aggregator class, which specifies an abstract *GetAggregatedPayload* method that must be implemented by the concrete aggregators. This follows the template method pattern by containing core aggregate logic within the abstract base classes *AggregatePayload* method, which calls the *GetAggregatedPayload* abstract method as part of its execution. This method constructs a single *PayloadEvent* object that wraps the aggregated payload data to be sent to the client. The *PayloadEvent* object contains additional metadata and a potential list of errors accumulated during the multi-query process.

The abstract class also registers a timeout configured by the mediator but has a default timeout value of 8 seconds if omitted, and contains all message receiving logic. If all queries have not been received in time, a special timeout message is set to itself and picked up from the mailbox queue. This triggers the aggregator to send a payload event to the client with an error to say that the request has timed out, as shown in figure 5.11.

To ensure that the expected payload event sent from each microservice is in response to one of the queries, a multi-query ID is used as part of the meta-data for each message type. Incoming payload events must contain the same multi-query ID, else the message is dropped, and an exception is raised to prevent corrupt data reaching the client. This method is also used for saga orchestration by using a global transaction ID.

```
protected BaseMultiQueryAggregatorActor(
    MultiQueryContext context,
    IActorRef callback)
{
    _context = context;

    SetReceiveTimeout(TimeSpan.FromSeconds(context.TimeoutInSeconds));

    Receive<MultiPayloadEvent>(@event =>
    {
        if (@event.MultiQueryId != _context.Id)
            throw new Exception(
        $"Invalid MQID: {@event.MultiQueryId}. Expected: {_context.Id}");

        // Call the abstract template method
        var payload = AggregatePayload(@event.Payloads);
        callback.Tell(payload);

        Context.Stop(Self);
    });

    Receive<ReceiveTimeout>(r =>
    {
        callback.Tell(new PayloadEvent
        {
            MetaData = new MetaData
            {
                ConnectionId = _context.ConnectionId,
                Callback = _context.Callback,
            },
            Errors = new[] { "Request timed out" }
        });

        Context.Stop(Self);
    });
}
```

Fig. 5.9 Base aggregator actor receiving events and registering a timeout.

Most multi-query requests can execute each query in parallel when no dependencies between queries exist. However, some queries cannot be pre-constructed by the mediator and must be constructed and sent after receiving other queries. For example, if one query needs to retrieve all groups from the groups microservice that a given user is a member of, and a second query must retrieve all posts from the discussions microservice for each of those groups (possibly with more filter logic), then a dependency exists; You cannot retrieve all posts without first acquiring the group data and so both queries cannot be sent in parallel.

To solve this, the parallelizable logic offered by the default MQH-FSM, named the *MultiQueryParallelHandler*, was extended by extending the class with a specialised MQH-FSM that hooks onto the change in state. The extended class registers additional intermediate states onto the multi-query process before starting, rather than only using the default idle, receiving, and terminating states. Each state change in an MQH-FSM can only move in a linear, directed way; it cannot go back to previous states once moved onto the next state. The extended class overrides the built-in Akka.Net *OnTransition* callback method to append additional queries into the list of pending queries between state changes, using the payload event data already retrieved as a response to previously sent queries. The MQH-FSM only moves to the terminated state once the list of pending queries is empty, and so this forces it to continue processing.

5.3.4 The Mediator and Builder Patterns

The API gateway uses a mediator to handle how a request is processed after it has been validated by the SignalR API hub and controllers. The mediator is implemented using the MediatR .Net package and is injected into the base controller and API hub by ASP.Net's default inversion of control (IoC) container to be used by each endpoint. MediatR then locates all classes that implement the *IRequestHandler* interfaces and executes the handlers *Handle* method based on the type of object sent to the mediator's *Send* method.

Each handler class is injected with an instance of the *IMessageContextBuilderFactor* interface, which is a factory used to create the appropriate builder to build a complex context object as shown in figure 5.12. This context object contains the necessary information to tell the system how the message should be processed and managed based on the requirements of the message type.

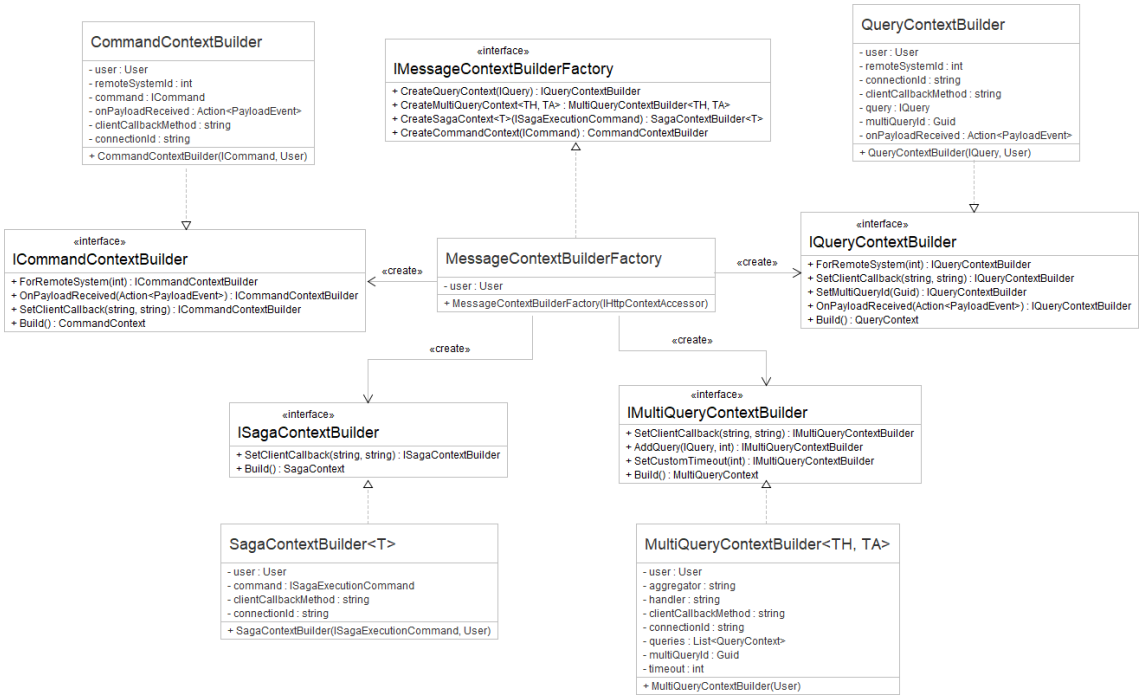


Fig. 5.10 Factory pattern used to create builders to build message type context objects.

Each builder consists of a *SetClientCallback* method that takes the client's SignalR connection ID and the name of a client JavaScript callback function as string parameter arguments. These are required to return a second response back to a client JavaScript callback function, using SignalR after an arbitrary amount of time, containing the results of the original request. If the client sends a query request type to the gateway's SignalR API hub, then they do not need to provide their connection ID as the hub already knows this. However, if they send a POST request to a Web API controller then they will need to supply their connection ID for SignalR to contact them with the result or payload at a later point in time.

The first response tells the client that the request has been accepted by the gateway (i.e. it passed the validation and authorization checks), whereas the second response is either the result of the executed command (e.g. success or failure) or the expected payload (e.g. the results of a query). Typically, the results of a query take the form of a serialized view model object to update the dynamic content of the web page. The second response cannot be immediately returned to the client, unlike the initial response, because it is asynchronously executed using message-driven IPC and so maintaining an open TCP connection would result in timeouts and poor performance.

The query and command context builders provide a *ForRemoteSystem* method that takes a remote system ID to specify which remote actor system should receive the message. Sagas and multi-query message types require the use of an SO-FSM or MQH-FSM to construct multiple command or query context objects to contact multiple remote systems, and so the saga and multi query builders do not have their own *ForRemoteSystem* method. The query and command context builders include an *OnPayloadReceived* method, which can be used to tell the system to call an event callback method on the server with the final payload. This is useful if any data contained in Firestore should be attached to the payload before sending back to the client. For example, when fetching all comments attached to a user-submitted post on a discussions group, the author ID of each comment is swapped out for the author's display name contained in Firestore. This prevents any user ID from being used directly on the client-side to improve security.

Below is a short description of each context builder:

1. **MultiQueryContextBuilder<TH, TA>** – A generically typed builder class where *TH* is the type of MQH-FSM actor class used to handle the execution of the multi-query request, and *TA* is a type of aggregator actor class used to aggregate the results of each query retrieved from an actor system.

```
var context = _builderFactory
    .CreateMultiQueryContext<MultiQueryParallelHandler, GroupConnectionsListAggregator>()
    .SetClientCallback(query.Callback, query.ConnectionId)
    .AddQuery(new UserGroupsQuery { OwnedGroups = true }, RemoteSystem.Groups)
    .AddQuery(new ProjectDetailsQuery { ProjectId = query.ProjectId }, RemoteSystem.Projects)
    .Build();
```

Fig. 5.11 Code sample showing the construction of a multi-query context.

2. **SagaContextBuilder<T>** - A generically typed builder class where *T* is an SO-FSM actor class used to handle the execution of compensating transactions during the rollback of a global transaction, and managing saga SSD recovery using an event store if the application experiences failure and must restart.

```
var context = _builderFactory.CreateSagaContext<CreateProjectSaga>(sagaExecutionCommand)
    .SetClientCallback(command.Model.ConnectionId, command.Model.Callback)
    .Build();
```

Fig. 5.14 Code sample showing the construction of a saga context.

3. **QueryContextBuilder** – Used by MQH-FSMs to construct non-parallelizable queries (i.e. ones that have a dependency on previous queries being executed to make use of the payload returned), or by mediator handler classes triggered by the API hub. Creates a context that targets one remote actor system to retrieve data from that microservice's datastore.

```
var context = _builderFactory.CreateQueryContext(remoteQuery)
    .SetClientCallback(query.Callback, query.ConnectionId)
    .OnPayloadReceived(OnPayloadReceived)
    .ForRemoteSystem(RemoteSystem.Discussions)
    .Build();
```

Fig. 12 Code sample showing the construction of a query context.

4. **CommandContextBuilder** – Used by SO-FSMs or mediator handler classes triggered by a Web API controller action. Creates a context that targets one remote actor system to perform a command to persist a change to that microservice's datastore.

```
var context = _builderFactory.CreateCommandContext(remoteCommand)
    .SetClientCallback(command.Model.ConnectionId, command.Model.Callback)
    .ForRemoteSystem(RemoteSystem.Teams)
    .Build();
```

Fig. 13 Code sample showing the construction of a command context.

5.3.5 Execution and Callback Handlers

After constructing the context object, the mediator passes it to an actor system service using a send or execute method, depending on the type of message contained within the context. The actor system service uses the information contained within the context object to decide how the request should proceed. The service is often used in an SO-FSM to send messages directly to the client using an event emitter class. These messages can be sent to the client during intermediate steps of processing the saga to keep the user informed on its progress for long running requests, or to mark the end of the saga. For actor system service methods that send messages using the event emitter, these have no

additional functionality and instead act as an adapter for the even emitter. In theory, these methods could be removed and allow SO-FSMs to use the event emitter directly, however this creates additional coupling. SO-FSMs already use the actor system service to send commands to other remote actor systems and so it is reused in this way to reduce the number of dependencies. Figure 5.17 shows a UML class diagram to illustrate this simplified dependency model.

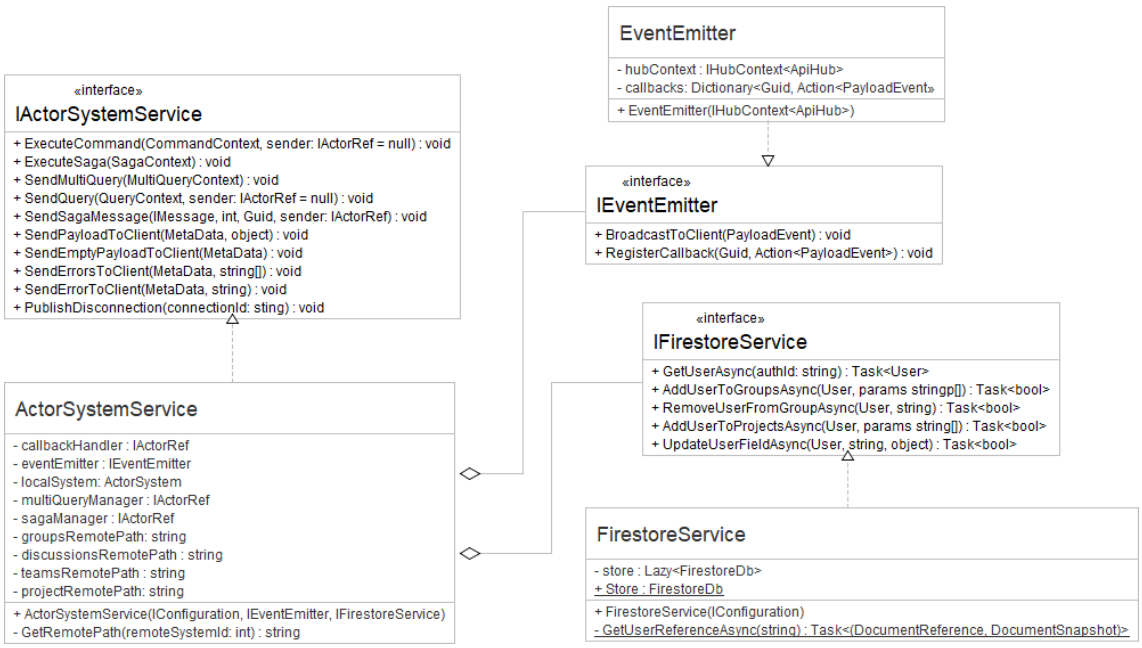


Fig. 5.14 The API gateway's actor system service with dependencies.

During construction of the actor system service, a saga and multi query manager is registered into the gateway's local actor system. They work in a similar way to a factor class except they do not return the appropriate SO-FSM or MQH-FSM actor instance. Instead, they create the actor and either inject the context object into its constructor or send it as part of an initialisation command, which then marks the beginning of the saga or multi-query process. They also both maintain an immutable dictionary of context IDs paired with references to the created FSM child actors. They also make use of Akka.Net's death watch, which is used for lifecycle monitoring by detecting when the child has terminated. This notifies the manager of child actor termination's so that the child reference can be safely removed from the dictionary to prevent further messages being sent to them after message processing has ended. Each manager has a factory method that uses the context object to create the correct FSM child actor instances (and aggregator

actor instances in the case of the *MultiQueryManagerActor*) if the context ID is not found within the dictionary, as shown in figure 5.18 with an example of the saga manager actor.

```
public IActorRef GetChildActorRef(SagaContext context)
{
    if (_children.ContainsKey(context.Id))
        return _children[context.Id];

    var sagaActorRef = Context.Watch(CreateSagaActor(context));

    _children = _children.Add(context.Id, sagaActorRef);

    return sagaActorRef;
}

private IActorRef CreateSagaActor(SagaContext command)
{
    var actorName = $"{command.SagaName}-{command.Id}";

    return command.SagaName switch
    {
        nameof(CreatePostSaga) =>
        Context.ActorOf(
            Props.Create(() => new CreatePostSaga(_actorSystemService)), actorName),

        nameof(CreateGroupSaga) =>
        Context.ActorOf(
            Props.Create(() => new CreateGroupSaga(_actorSystemService, _firestoreService)), actorName),

        nameof(CreateProjectSaga) =>
        Context.ActorOf(
            Props.Create(() => new CreateProjectSaga(_actorSystemService, _firestoreService)), actorName),

        nameof(ConnectProjectSaga) =>
        Context.ActorOf(
            Props.Create(() => new ConnectProjectSaga(_actorSystemService)), actorName),

        _ => throw new Exception($"Failed to find SagaActor: {command.SagaName}"),
    };
}
```

Fig. 5.15 SagaManagerActor using a factor method and controlling the lifecycle of saga orchestrators.

5.4 Microservice Design and Implementation

Each microservice manages its own local actor system and can be deployed in isolation from all other microservices and the API gateway. Microservices have been designed to not send messages between themselves and instead only the gateway sends messages from its own actor system. Only actors can communicate with other actors within the MSA using actor references controlled by Akka.Net. Each microservice only responds to the gateway by sending event messages back to the sending actor. They only maintain subdomain-specific application logic and persist and retrieve sub-domain data from their datastore to be used by the gateway. Domain entity classes are internal to each microservice and cannot be used outside the scope of the microservice's bounded context. Instead, data that needs to be displayed on the client's web page (using a Gatsby.js component) is added to a view model class. The view model is then wrapped inside a

payload event to be serialized and sent back to the gateway, which is eventually unpacked and returned to the client.

With the help of saga orchestration contained within the gateway, microservice can remain simplistic in nature and only become complex when dealing with their own sub-domain application rules kept. This supports the bounded context strategic pattern proposed by DDD by encapsulating all sub-domain logic in its own context with strict rules on how to access that logic. The aggregator root pattern of DDD inspired the use of root manager actors to encapsulate and control how sub-domain data is accessed, as well as to isolate the risk of failure.

Each microservice uses the RavenDB datastore but this is not a strict requirement. Originally, Elasticsearch was chosen for the discussions microservice datastore, but to keep things simple for the sake of this research project, RavenDB was used. However, the MSA has been designed with adaptability in mind and so a datastore can be easily swapped out with minimal code change. If one microservice changes its datastore, no other microservices, nor the gateway, is affected by this. The MSAs message-driven IPC strategy only requires shared references to the messaging and view model classes, allowing deserialization of JSON data to be converted back to the original object. Therefore, a separate class library containing only these classes is shared between each microservice. Each class is a plain-old C# object (POCO) and contains no shared functionality, which is an intentional design choice to prevent tight coupling of dependencies from occurring.

By reducing the level of IPC to simple JSON data, we avoid forming dependencies on the type of datastore being used. JSON or XML data formats can be used by any datastore but may require an internal transformation step, by the microservice, to fit the datastore model, such as transforming JSON into a series of SQL statements to be executed on a RDBMS. However, RavenDB is a NoSQL database with support for transactions, and fits the project research requirements while also using Lucene indexing, just like Elasticsearch, for fast processing of search queries, although admittedly with fewer search capabilities compared to Elasticsearch.

For each microservice, a simple console application is started with a *Program* class consisting of a *Main* method that is executed when the application starts, as shown in

figure 5.19. It reads a local HOCON configuration file containing all configuration settings for the actor system, such as the port and hostname to use, and other logging settings. The gateway needs to know this information to contact the remote actor system. It also needs to know what messages it can accept. If any actor within an actor system receives a message type that it cannot handle, it goes to a dead letters mailbox and logs this occurrence to the microservice's console window.

```
public class Program
{
    public static void Main(string[] args)
    {
        var configString = File.ReadAllText("discussions-system.conf");
        var config = ConfigurationFactory.ParseString(configString);

        // Create actor system
        using var actorSystem = ActorSystem.Create("DiscussionsSystem", config);

        // Create actors for system
        actorSystem.ActorOf(Props.Create<DiscussionManagerActor>(), "DiscussionManager");

        Console.WriteLine("Discussions actor system created.");
        Console.ReadKey();
    }
}
```

Fig. 5.16 Code example of the discussions microservice starting its own local actor system.

Figure 5.20 shows a common example of how an actor receives a message. There are a few base classes an actor class can extend from, but the most common one used is the *ReceiveActor* class. This provides the actor with a generic *Receive* method that can be called to attach a handler function to handle a specific message type. The *Receive* method's generic parameter type must match the specific message type to be handled. Akka.Net then executes the handler function by passing instances of the message class picked up from that actor's mailbox queue, allowing the actor to process the message in isolation while avoiding concurrent message processing against the same actor instance. However, multiple actors of the same instance can be created, each with their own private instance data, if parallel processing is required.

```
Receive<CreateCommentCommand>(command =>
{
    using var session = DocumentStoreSingleton.Store.OpenSession();

    // Code used to create the comment has been omitted in this example.

    session.Store(comment);
    session.SaveChanges();

    Sender.Tell(new PayloadEvent(command)
    {
        Payload = comment.Id.ConvertToClientId()
    });
});
```

Fig. 5.17 A *CommentsActor* class handling a received command and persisting a change to RavenDB.

In figure 5.20, a session to RavenDB is created, as well as a new comment entity object based off the contents of the command. The entity is then stored and persisted in RavenDB. Afterwards, a payload event is created by passing the original command to its constructor so that specific metadata properties can be copied across to the event. If the command was part of a saga, or in other scenarios where a query was part of a multi-query, then the original context ID (e.g. a transaction ID or multi-query ID) is passed to the event so that the FSM handling this request knows that the event is to be expected. Without this step, we cannot guarantee the consistency of data because we do not know if a query or command was successfully fulfilled, or if the contents of the payload was delivered to the correct address. In theory, this should never occur, but it adds an extra layer of security, as well as allowing us to track the activity of a saga or multi-query across the MSA by logging the contents of the metadata, which is especially useful for debugging purposes.

The *Payload* property of the *PayloadEvent* class is an *object* type, so any serializable instance of a C# class, or any primitive data type, can be assigned to this property. Therefore, it is sometimes required to check the type of payload on the gateway before it can make use of it. Sometimes the payload will be an instance of a view model containing data for a given Gatsby.js component. Other times it will be a simple primitive value, such as in figure 5.20 where the ID of the newly created comment is returned.

Finally, it is worth noting that each time a message is created, its metadata is given a *CreatedAt* timestamp value for when it was instantiated. This is needed for calculating how long a given request took to complete, as well as for sorting an actor's mailbox based

on timestamp ordering. Originally, the latter was not implemented and meant that an actor may receive some messages for newer requests before messages as part of an older request. This is was inspired by how previous timestamp-based CC algorithms work for database systems implementing ACID principles, as a way of increasing the protection of data consistency. If a saga executes multiple commands as part of a global transaction, and a single command is executed which indirectly affects the saga, then this can potentially cause the saga to unnecessarily abort and rollback using its compensating transactions. By ordering the actor's mailbox based on timestamps, this reduces the number of rollbacks but does not entirely solve the problem. Some rollbacks are inevitable, as demonstrated in the evaluation chapter.

5.5 Software Development Methodology

Agile software development best practices were followed throughout the project's development lifecycle based on a previously defined Gantt chart containing deadlines. The project was broken down into subsystems that needed designing, implementing, and testing. The front-end web client was created first using fake data that was later swapped out for concrete API calls to the gateway, with each bounded context developed individually and incrementally. Milestone deadlines were achieved after each bounded context was completed. A bounded context could be broken down into smaller steps, such as developing the actor system, the datastore, and how it would communicate with the gateway. Some saga and multi-query implementations were initially challenging and took most of the time for a given sprint. However, once a flexible design pattern was implemented, it could be easily reused for other microservice implementations, which improved the velocity of future sprints.

A Kanban board was created and maintained, using the Trello platform, to create a list of cards in the form of user-stories. Each card was given an estimate for how long it would take to complete in hours, as well as the actual time it took to complete the card. The example in figure 5.21 was estimated as taking 1 hour as shown in the square brackets on the card's title but ended up taking 2 hours as shown in the trailing round brackets. This was used to estimate how many work items (including design work, implementation, testing, bug fixes, and research tasks) could be achieved in a given time-boxed 2-week sprint, which helped meet small personal deadlines and milestones.

Fig. 5.18 A project context user story on Trello

5.6 Development Environment and Tools

The front-end web client was developed within the Visual Studio Code text editor, with node package manager (NPM) to install various JavaScript packages, including Microsoft's SignalR client, a Firebase client for handling authentication and providing access to the Google Cloud FireStore, as well as the Axios HTTP client for sending POST requests to the API gateway. It was built using Gatsby.js (a customised version of React.js) allowing the project to make use of all React.js NPM packages. TypeScript was used to provide strict type support for Javascript. This allowed interfaces to be defined that matched the structure of the payload view model data being retrieved from SignalR on the gateway. SASS was used to improve the development experience of writing CSS.

The gateway and each microservice was implemented in C# using the ASP.Net 5.0 runtime and SDK. Each of these makes use of their own project file but contained within the same solution folder and was developed in Visual Studio 2019 with Resharper for an improved development experience. Version 1.4.8 of the *Akka* NuGet package (a port of the popular Java/Scala Akka framework, but for C#), version 1.4.8 of the *Akka.Remote* NuGet package, each developed by the Akka.Net team, was used in each microservice and for the gateway project to build the actor system with support for remote messaging. Version 1.4.10 of the *Akka.Persistence* and *Akka.Persistence.SqlServer* was used on the gateway to provide access to the SQL Server event store for persisting the state of SO-FSM actors. The gateway also made use of the MediatR NuGet package to implement

parts of the mediator pattern used to separate the controller layer from the application layer. It also used the *Google.Cloud.Firestore* NuGet package to retrieve user claims to authenticate and authorize user requests.

NUnit was used to build the unit, integration, and system tests. The unit tests were created using the *FluentAssertions* and *Moq* NuGet packages to create mock objects to avoid calls to other microservices and bypass authentication. This allowed small units of code to be tested in isolation.

5.6 Testing Methodology

Test-Driven Development (TDD) was used throughout the project's lifecycle, based off the initial design choices and UML diagrams constructed at the start of each sprint. However, these diagrams were later updated iteratively in future sprints where needed. TDD worked very well for this project because it promoted strong consideration towards the data that needed to be retrieved from each endpoint or microservice to achieve a user-story, rather than returning incorrect data and fixing the problem later. TDD reduced the amount of unnecessary code changes and features implemented despite requiring additional research to learn how to correctly test concurrent code executed using Akka.Net.

The system tests were used to test specific scenarios as part of the evaluation portion for this research. However, for evaluating the response times of requests, a specialised Gatsby.js page was constructed to fire off many requests to simulate multiple user requests. Ideally, this could be improved upon by using a JavaScript testing framework, such as Jest.js. By sending requests from the front-end web client, rather than only capturing how long a particular message took to process on the server side, we gain a better understanding of how long a given user would typically have to wait until the data is presented in their browser. This better reflects the project goal of building a reactive system while focusing on the user experience of the web application.

Finally, user acceptance testing was performed by 4 volunteers to test the user experience of the front-end web application locally on one machine. This provided early feedback and allowed bugs to be discovered, such as being redirected too soon after creating a group or project causing queries for that entity to return empty results from RavenDB.

Chapter 6

Evaluation

This chapter is split into three main sections where the first two use quantitative research methods to record statistics for the system's performance. First, we analyse the response times recorded for each message type in the MSA running locally on a single machine. Then, we analyse complex scenarios that are expected to cause data inconsistencies within the domain and how solutions to these problems were implemented using timestamp-based ordering of incoming messages. Finally, we conclude with a qualitative assessment of the patterns used in the code and how they benefit the design of the system.

6.1 Analysing System Performance

Each command, query, saga, and multi-query was executed 50 times from Gatsby.js and was recorded once the contents of the payload event reached the browser. Outliers observed in the results were discarded from the calculation of each measurement as they reduced the accuracy of results. Each type of child actor, controlled by an actor manager, is placed inside an actor pool (i.e. one pool per each type of actor), which has been configured to scale between 1 to 10 actors to handle messages concurrently. The minimum, maximum and average response times in milliseconds were recorded for each message type, as shown below. The first 10 executions for each message type were omitted from the results due to showing unreliable measurements. This is because they often took up to 3 times as long to execute due to the initial start-up of actor's that are only created once they receive their first messages. Only successful messages were recorded as part of these results. In the next section, further evaluation is performed on conflicting messages involving saga rollbacks.

Commands

Request	Min. Response Time	Max. Response Time	Avg. Response Time (ms)
ChangePermission	23	45	26
ChangeVote	16	34	27
CloseIssue	23	60	39
CreateComment	34	53	48
CreateIssue	65	123	80
CreateTeam	230	300	274
JoinTeam	39	167	65
LeaveTeam	32	122	76
UpdateIssue	90	210	144

Sagas

Request	Min. Response Time	Max. Response Time	Avg. Response Time (ms)
ConnectProject	140	253	170
CreateGroup	190	657	349
CreatePost	189	269	205
CreateProject	275	910	346
RemoveGroup	303	559	494
RemoveProject	320	880	430

Queries

Request	Min. Response Time (ms)	Max. Response Time (ms)	Avg. Response Time (ms)
FetchCategories	36	133	78
FetchComments	38	90	75
FetchGroupDetails	78	240	120
FetchGroupProjects	32	49	34
FetchIssues	42	50	44
FetchProjectDetails	123	290	140
FetchTeamMembers	49	219	63
FetchTeamPermissions	30	87	55
FetchTeams	34	90	53
FetchUserGroups	171	817	410
FetchUserProjects	27	520	379
SearchGroups	34	75	52

Multi-Queries

Request	Min. Response Time	Max. Response Time	Avg. Response Time (ms)
FetchNewsFeed	102	430	223
FetchGroupConnections	43	156	71
FetchGroupPost	133	949	160
FetchGroupPosts	107	393	123

As expected, sagas took the longest to process due to their overhead in orchestration. The MQH-FSM and aggregator actor pairing performed much faster compared to requests using SO-FSMs. Other message types that required data from the external Firestore authentication provider, such as *FetchUserGroups* and *FetchUserProjects*, performed slower on average and varied the most between the minimum and maximum response times, possibly due to external caching.

6.2 Improving Data Consistency

Most data inconsistency anomalies were caused by saga's that had not fully completed, while other message types were concurrently reading or writing to the datastores. A local transaction avoids dirty reads and writes through transaction isolation using ACID principles. RavenDB also supports ACID transactions, so only committed data can be read by other transactions while also preventing uncommitted data from being overwritten. However, because sagas commit multiple local transactions before finishing, with potential rollbacks executed using compensating transactions, it is difficult to prevent dirty read and writes on a globally distributed scale.

To simulate a data inconsistency anomaly, three different conflicting messages were sent from the API gateway to different microservices:

1. **CreateProject** – Uses a SO-FSM to create a *Project* entity in the project microservice datastore, while also creating a two default "Admins" and "Moderators" *Team* entities in the teams microservice datastore. Each team contains a project ID field to reference the external project entity.
2. **JoinTeam** – Creates a Member entity inside the teams microservice datastore with a team ID field to reference the local *Teams* entity.
3. **RemoveProject** – Deletes the selected *Project* entity from the project microservice datastore and, if successful, deletes all *Team* and *Member* entities from the teams microservice datastore that references the deleted project.

Because the *RemoveProject* and *CreateProject* sagas must span two different microservices, the *JoinTeam* single command may execute while a saga is still in progress. In this scenario, a user may have retrieved the team before it was deleted and then add a member to that team after it was deleted. This would result in phantom writes

where a *Member* entity references a deleted *Team* entity. By running actors as part of an actor pool, messages are processed concurrently for improved performance, but initially without any form of concurrency control (CC) at an application-level.

By sending requests to trigger the *CreateProject*, *JoinTeam*, and *RemoveProject* message types from Gatsby.js, and then repeating this scenario 100 times, we should expect that no lingering *Member* entities are found within the teams microservice datastore. However, by running this experiment it was revealed that 32 *Member* entities were found. The same anomaly was noticed while trying to create a post while deleting the targeted discussions group except there were 14 lingering *Post* entities after 100 executions.

To reduce the number of distributed phantom writes, a form of timestamp-based CC was implemented, inspired by previous innovations with database technology. Lock-based CC was not chosen because a reactive system should be non-blocking and favour eventual consistency over strong consistency. Eventual consistency can result in rare frustrating user experiences (e.g. submitting a post to a deleted discussion group) but is still more desirable than locking access to domain entities and blocking all requests attempting to access them until a new lock becomes available. From previous research studies, lock-based CC tends to not scale well in an MSA and goes against the responsive principle of the reactive manifesto. There are methods to optimise the user experience in situations that are more common, such as informing the user that the page they are on no longer exists to avoid the frustration. For eventual consistency-based systems, this is preferable if it results in faster user experiences.

Each actor owns their own mailbox queue, but the mailbox only orders the messages based on when they were received. Previously, all messages had a *CreatedAt* timestamp property for logging purposes but each command as part of a single saga also had their own separate timestamps. This was changed so that when a saga is first executed, all commands share the same timestamp, so the last command executed as part of a potentially long running saga had the same timestamp as the first command. The default mailbox was replaced with a customised mailbox that ordered all message based on the timestamp property contained within the metadata of each received message. Therefore, an actor would always enqueue and process the oldest message.

By retrying the previous the *CreateProject*, *JoinTeam*, and *RemoveProject* experiment 100 times, the number of lingering *Member* entities was reduced from 32 to 4, and the other experiment of creating a post while deleting a group was reduced from 14 to 0. The other 4 lingering messages could potentially be fixed by changing the execution of commands within a saga to delete the *Team* and *Member* entities before deleting the *Project* entity, but this could potentially result in other unforeseen anomalies as the application scales. Although the timestamp ordering improved the results, it does not fix all potential issues caused by slow systems or lost messages in a distributed network.

6.3 Qualitative Analysis of Design Patterns

The MSA makes use of several conventional design patterns, as well as some new ones created as part of this research project. This section explains the motivation for using them, what problem they solve, and how they support designing for change and, where appropriate, the principles outlined in the reactive manifesto and those found in DDD.

6.3.1 API Gateway Pattern

Motivation: Separate the frontend web client from the backend microservices to promote adaptability and scalability.

The API gateway follows the separation of concern principle by grouping together all authentication and authorization logic. This provides a filtering mechanism to prevent unauthorized requests from sending messages to any microservice, allowing the design of all microservice actors to follow the single responsibility principle. Each actor only needed to execute work relating to the message type and persist changes if required, allowing additional microservices and actors to extend the functionality of the system and adapt application logic without affecting any other microservice or the gateway. The client only needed to know the URL of the API gateway and has no knowledge of the underlining MSA. Therefore, IPC between the MSA and API gateway, as well as microservice datastore technology, can be changed without needing to change parts of the client to meet new requirements.

6.3.2 Saga Pattern with Orchestration

Motivation: Centralise saga logic to promote extensibility and a simplified dataflow model with support for rollbacks of global transactions.

By centralizing all saga logic, the steps of a globally distributed transaction are defined in one place which allows sub-transactions in the form of command messages to be easily rearranged to improve data consistency and rollback performance. In certain situations, rearranging the execution of sub-transactions may prove beneficial by executing potentially risky operations before others, rather than performing them towards the end. If a saga needs to rollback, fewer compensating transactions are required to revert the changes to the domain's state with less risk of running into conflicting operations.

It was easy to build additional functionality on top of the saga orchestration process, such as adding retry logic using the circuit breaker pattern to retry the same message an arbitrary number of times with a given timeout. This project added an event store using the event sourcing pattern much later into development for improved fault-tolerance, which would have been difficult to do if using the saga choreography pattern.

6.3.3 Database per Service Pattern

Motivation: Assign a separate, isolated datastore to each microservice to separate each sub-domain to design for change and improve system resilience.

The database per service pattern compliments DDD best practices as it allows separate development teams to work in isolation and extend their sub-domain without affecting others. It also isolates the risk of system-wide downtime because if one sub-domain becomes unavailable, it will not stop the user from using features of the web application which does not require it. For example, users can update team permissions even if the group's datastore is unavailable. Different datastores can also be swapped out with other datastore technology or change their schemas with lower risks of breaking the rest of the system. Security concerns are also minimised because if one service contained unsecure application logic, such as allowing an attacker to delete or access unauthorized data, then only that datastore is at risk.

6.3.4 Microservice Architecture Pattern

Motivation: Create an architecture able to scale to meet the requirements of individual sub-domains, each with their own isolated application logic, to promote resilience, encapsulation, adaptability, and extensibility.

New microservices can be easily added to the MSA to extend functionality with minimal integration pain points. This project's MSA only required adding the URL of the new root manager actor, contained within a new microservice, to the actor system service located on the API gateway with no other changes required. Microservices can declare a specification for what types of messages they can accept and have the freedom to react to those messages without relying on any legacy code. Monolithic applications typically accumulate large amounts of technical debt over years of continuous development as old dependencies begin to age and show signs of weakness. Upgrading a microservice poses less of a risk to the system, which alleviates common fears expressed by developers and promotes the flexibility required to experiment with newer technologies. Less source code per project file also means that less unit and integration tests need to be executed when testing a change made to a single microservice, allowing for a better development experience, especially when using TDD. TDD was very convenient to setup in a new microservice during this project's development lifecycle because there were comparatively far fewer dependencies that required mocking and less scaffolding of test fixtures to setup.

6.3.5 Mediator, Builder, and Factory Patterns

Motivation: Each component in the API gateway should follow the single responsibility principle to design for change.

The mediator allowed all web API controllers and SignalR API hub endpoints to remain simplistic. Each controller only handles logic involving the HTTP request and responses, by returning the correct status codes depending on the input sent by the user. The request body was converted into a model object using ASP.Net's model binding feature, where each model property was decorated with a validation attribute. For example, some properties were marked as required or needed to be within a certain range for strings and numbers. The controllers would return a bad request status code if the model state were

not valid. If it were valid, the controller would create a specific wrapper object to package the model properties before sending them to the mediator for further processing of the message.

Depending on the type of wrapper object, the MediatR ASP.Net package would redirect the object to the correct mediator handler. Each handler then uses a context builder, using the builder pattern, to construct a complex context object, which contained the message/s needed to fulfil the request along with metadata. The context object represents a set of instructions for how the messages should be managed and is passed to the actor system service who makes these decisions. There are several different builders depending on the type of message that should be sent. Therefore, a builder factor was used to retrieve the correct builder and has additional logic to retrieve the *User* object if the user is authenticated, which is added to the builder instance behind the scenes. The *User* object then becomes part of the message type's metadata to be used on microservices if required.

The developer does not know about the internal implementation, allowing the factor and builder instances to change without updating each mediator handler. The build process remains completely isolated from the mediator and controller layer, while the controller logic has no influence on the other two layers. This promotes separation of concern between each layer, allowing them to change with minimal effort. It also encapsulates complex construction logic to avoid code duplication, which avoids bugs from occurring as code duplication often results in slight variances between duplicated code when updating one area while forgetting others.

6.3.6 State Pattern

Motivation: Protect the state of saga's during orchestration, and multi-query handlers, to improve data consistency by controlling their behaviour.

The state pattern was implemented using Akka.Net's finite-state machine (FSM) actor classes. Saga orchestrator (SO) and multi-query handler (MQH) classes extended these and were each assigned their own internal state data, which reflected what state they were in. Depending on the state, different types of messages would trigger different types of reactions. For example, an MQH-FSM only accepted new query messages during a transition between states, whereas once it was in a new state it could only accept event

messages representing the results of those queries. Once it had received all expected events, it would move to an aggregation state preventing it from receiving any other messages. Similarly, a SO-FSM could only accept certain expected messages depending on each state and what sub-transactions were currently being worked on. The state data would be continuously built upon throughout the process to transform the behaviour of the FSM.

This proved effective for detecting failure states, such as moving a saga into a rollback state, causing compensating transactions to execute to rollback changes made during the execution of the global transaction. Also, if the MQH-FSM did not receive all the responses within a given time-out then it could transition to a failed state to send the user a request timeout message before terminating. By encapsulating different state behaviour into a single FSM class, rather than separating out the logic amongst multiple classes, it allowed states to be controlled in one place without affecting other saga or multi-query messages. It also allowed other mechanisms to be built into individual FSMs depending on separate saga or multi-query requirements to improved performance, such as event sourcing and retry logic.

6.3.7 Multi-Query and Aggregator Patterns

Motivation: Separate the multi-query logic from the aggregator to design for change and modularity.

A single MQH-FSM class maintained the core logic for sending out queries in parallel during each processing state, which could either be used directly or by extending it with a child class. Child classes could register additional states and append more queries between each state transition as a method of reusing the core multi-query logic to avoid code duplication, while extending its use. A separate aggregator actor class was injected into the targeted MQH-FSM's constructor, with an *AggregatePayload* method contained in a base aggregator class. The base class uses the template method pattern where concrete aggregator classes would implement the template method. This has the same effect as the MQH-FSM child classes using the *OnTransition* method to extend the functionality of the parent class. Core aggregator logic could be reused with customised aggregator logic contained in the template method to improve code reuse.

The multi-query context builder, used in the mediator handlers, includes a generic method to assign the MQH-FSM and aggregator class types to be used during the execution of the multi-query message. This modular design choice allows different classes to be swapped out for others in a centralised location while preserving the single responsibility for each component.

6.3.8 Actor Model Pattern

Motivation: Allow individual actor classes to maintain small chunks of work to react to incoming messages, while allowing actors to scale to meet the demands of fluctuating network traffic. Also, it should support the reactive manifesto principles using message-driven, asynchronous, non-blocking IPC to establish fixed boundaries between microservices, loose coupling, isolation, and location transparency.

A separate class library was used to share message classes and payload view model classes for each microservice. This was needed to allow each actor within the MSA to serialize and deserialize known message types. Actors could then be deployed in isolation and could only communicate using the URL of the root manager actors included in as part of each remote actor system. Instead of using older RPC frameworks that attempt to simulate a shared runtime environment for remote objects, actors remained separated by only receiving serialized messages. This avoids potential bugs from bad assumptions made about the network from a developer perspective. For example, if an actor restarted due to failure in a remote system, the manager could create a new actor instance without the sending service knowing, and separate actors could be scaled in isolation. This is difficult to achieve if object references are to be maintained. A URL can remain static, but the underlining implementation attached to that URL does not have to be. Because no code is shared between services (apart from the shared class library), each service is free to change without affecting the chosen IPC strategy. The bounded context pattern was easy to implement when following the actor model as it allowed individual sub-domains to be kept in isolation rather than being used directly between services.

6.3.9 Event Sourcing Pattern

Motivation: Allow FSMs to recover from failure to promote system resilience, and to produce an audit log of events that had occurred.

Event sourcing was implemented using an SQL Server event store to record all events that took place during saga or multi-query execution. If an SO-FSM or MQH-FSM restarted due to failure, the SO-FSM could be moved to the roll back state to execute compensating transactions and the MQH-FSM could resend the pending queries that had not received their corresponding payload event. If a failure occurred, the SQL Server event store could be used to debug a given user scenario by viewing the history of events. This proved useful during development and could potentially be used for other message types other than sagas and multi-queries.

Chapter 7

Conclusion

This chapter concludes with a summary of achievements made as part of this research effort, with a reflection over the original aims and goals of the project. It ends with a short discussion over possible future work that could build on top of the ideas discussed in this dissertation.

7.1 Summary of Achievements

This dissertation has covered many design patterns to improve the level of data consistency protection using ideas taken from both DDD and the reactive manifesto. By analysing previous CC protocols from traditional database management systems, this research project was able to apply timestamp-based CC to message metadata to reduce the number of saga rollbacks within a modern, distributed architecture. As the actor model has rekindled the interest of software practitioners, this dissertation serves as an example of how to apply it to MSA and how FSMs using this model can be used to achieve improved data consistency protection when used with saga orchestration.

The implementation of the API gateway and each microservice demonstrates a good degree of separation of concern to design for change, whilst also promoting scalability using location transparency. By using actor references, the system can scale independently to meet different requirements for different microservices. The quantitative results of the evaluation showed that saga orchestration was slower compared to all other message types, including multi-query messages that performed faster than previously anticipated. Therefore, the user experience must be designed around this expected wait time and the number of sagas should be reduced through careful consideration over what data should reside in each sub-domain when using the database per service pattern, which this implementation achieved.

The chosen design patterns improved the development process by avoiding bugs caused by duplicate code and improved the level of encapsulation. Most importantly, the use of SignalR for bidirectional communication and message-driven IPC created a responsive system that allowed the user to continue using the front-end web client. The system could

send update messages in a non-blocking fashion once data had arrived at the API gateway, freeing the user from unnecessary wait times. Most messages could be retried on behalf of the client, with sagas retrying after recovering using the event store and rolling back to a healthy domain state. This level of self-aware fault-tolerance was hidden from the client and as such the number of error messages sent to the user was minimised to promote a good user experience.

7.2 Reflection

This project successfully implemented a reactive MSA as defined by the reactive manifesto, while implementing many mechanisms to improve the level of data consistency across the distributed domain model. Many software development companies are hesitant towards splitting up a monolithic application due to the high level of risks involved. The primary goal for this project was to demonstrate how to approach data consistency challenges when global transactions must span several polyglot datastores. Although it is still a challenge, using timestamp-based CC with message-driven IPC reduces the risks of data inconsistency anomalies. Eventual consistency should still be favoured when using MSA, but it is possible to improve the degree of consistency at the application level to reduce application-specific bugs.

Although there are some noticeable data consistency anomalies, the number of them was significantly reduced compared to earlier attempts without timestamp-based CC. This was achieved while avoiding lock-based blocking requests that would have severely limited the scalability and performance of the system. Many key patterns were identified to compliment reactive principles, such as the use of FSMs to improve both fault-tolerance and data consistency when used with saga orchestration. Another key concept to take away from this dissertation is the requirements needed to fulfil different message types. Overall, the results and progress made can be considered successful with a rewarding learning experience.

7.3 Future Work

Previous studies of CC have shaped how we view modern system design. This research project experimented with timestamp-based CC at the application level when used with modern patterns to handle distributed global transactions. Future work efforts are needed to improve the level of data consistency and response times. A potential alternative may be to use the publish-subscribe pattern rather than directly messaging individual root manager actors. This project may have also benefited from message queues and the solutions demonstrated in this dissertation may not work for different technology requirements. Therefore, a unified model could be constructed using the lessons learned from this research.

Event sourcing was not used to its fullest potential and adds a lot of overhead during rollbacks, causing slow response times for the user while state is being recovered. This worked well for saga orchestration, but saga choreography has been recommended for improved performance as actors could send messages directly to the next actor rather than using an orchestrator. If choreography can be implemented in a way that achieves the same level of protection for data consistency and fault tolerance, then this may improve the responsiveness of the system.

References

[1] J. Waldo, G. Wyant, A. Wollrath and S. Kendall, *A Note on Distributed Computing,* 1994.

[2] R.-T. Innovations, *Is DDS for You?,* 2018.

[3] "What problems does the actor model solve?," Akka.NET project, [Online]. Available: https://getakka.net/articles/intro/what-problems-does-actor-model-solve.html. [Accessed 1st August 2020].

[4] K. Indrasiri, "The Evolution of Integration: A Comprehensive Platform for a Connected Business," WSO2, [Online]. Available: https://wso2.com/whitepapers/the-evolution-of-Integration-a-comprehensive-platform-for-a-connected-business/. [Accessed 29 July 2020].

[5] J. Poole, "Thoughts on Push vs Pull Architectures," 1st April 2018. [Online]. Available: https://medium.com/@_JeffPoole/thoughts-on-push-vs-pull-architectures-666f1eab20c2. [Accessed 29 July 2020].

[6] E. Evans, Domain-Driven Design: Taskling Complexity in the Heart of Software, Addison-Wesley Professional, 2003.

[7] M. Fowler, "BoundedContext," 15 January 2014. [Online]. Available: https://www.martinfowler.com/bliki/BoundedContext.html. [Accessed 2 August 2020].

[8] M. Fowler, "DDD_Aggregate," [Online]. Available: https://www.martinfowler.com/bliki/DDD_Aggregate.html. [Accessed 27 July 2020].

[9] "Anti-Corruption Layer pattern," Microsoft, 23 June 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer. [Accessed 29 July 2020].

[10] C. Traina, A. J. M. Traina, M. R. Vieira, A. S. Arantes and C. Faloutsos, *Efficient processing of complex similarity queries in RDBMS through query rewriting,* 2006.

[11] D. D. Chamberlin, M. M. Astrahan and M. W. Blasgen, "A History and Evaluation of System R," *Communications of the ACM,* vol. 24, no. 10, pp. 632-646, 1981.

[12] T. Härden and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys,* vol. 15, no. 4, pp. 287-317, 1983.

[13] IBM, "ACID properties of transactions," 16th July 2020. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html. [Accessed 29th July 2020].

[14] D. Agrawal, J. L. Bruno, A. E. Abbadi and V. Krishnaswamy, “Relative Serializability: An Approach for Relaxing the Atomicity of Transactions,” *ACM SIGACT-SIGMOD-SIGART symposium,* pp. 139-149, 1994.

[15] M. Kleppmann, Designing Data-Intensive Application, O'Reilly, 2016.

[16] V. C. Lee, K.-W. Lam, S. H. Son and E. Y. Chan, “On Transaction Processing with Partial Validation and Timestamp Ordering in Mobile Broadcast Environments,” *IEEE Transactions on Computers,* vol. 51, no. 10, pp. 1196-1211, 2002.

[17] W. Xiaoqin, Z. Min and X. Yang, “Time-stamp based mutual authentication protocol for mobile RFID system,” in *Wireless and Optical Communication Conference (WOCC),* 2013.

[18] C. Yao, D. Agrawal, G. Chen and Q. Lin, “Exploiting Single-Threaded Model in Multi-Core In-Memory Systems,” vol. 28, no. 10, pp. 2635-2650, 2016.

[19] R. Lee and M. Zhou, “Extending PostgreSQL to Support Distributed/Heterogeneous Query Processing,” *Lecture Notes in Computer Science,* vol. 4443, pp. 1086-1097, 2007.

[20] H. Berenson, P. Bernstein and e. al., “A critique of ANSI SQL isolation levels,” *ACM SIGMOD international Conference on Management of Data,* vol. 2, no. 24, pp. 1-10, 1995.

[21] P. A. Bernstein and N. Goodman, “Concurrency Control in Distributed Database Systems,” *ACM Computing Surveys,* vol. 13, no. 2, pp. 185-221, 1981.

[22] M. T. Özsu and P. Valduriez, Principles of Distributed Database Systems, Springer, 2019.

[23] M. J. Cahill, U. Röhm and A. D. Fekete, “Serializable isolation for snapshot databases,” *ACM Transactions on Database Systems,* vol. 5, no. 20, pp. 1850-1861, 2009 .

[24] S. I. Khan and D. A. S. M. L. Hoque, “A New Technique for Database Fragmentation in,” vol. 5, 2010.

[25] G. Alkhatib and R. S. Labban, “Transaction Management in Distributed Database Systems: The Case of Oracle’s Two-Phase Commit,” *Journal of Information Systems Education,* vol. 12, no. 2, pp. 95-104, 2002.

[26] D. P. Mitchell and M. J. Merritt, “A Distributed Algorithm for Deadlock Detection and Resolution,” *In Proceedings of the third annual ACM symposium on Principles of distributed computing (PODC ’84),* p. 282–284, 1984.

[27] C. Binnig, F. Färber, S. Hildenbrand and D. Kossmann, “Distributed snapshot isolation: global transactions pay globally, local transactions pay locally,” *The VLDB Journal,* vol. 23, no. 6, pp. 987-1011, 2014.

[28] “X/Open XA,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/X/Open_XA. [Accessed 6th August 2020].

[29] T. Janssen, “Distributed Transactions – Don’t use them for Microservices,” [Online]. Available: https://thorben-janssen.com/distributed-transactions-microservices/. [Accessed 6th August 2020].

[30] A. Fox and E. A. Brewer, “Harvest, Yield, and Scalable Tolerant Systems,” *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems,* p. 174–178, 30th March 1999.

[31] “The Reactive Manifesto,” 16th September 2014. [Online]. Available: https://www.reactivemanifesto.org/. [Accessed 6th August 2020].

[32] C. K. Rudrabhatla, “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture,” *International Journal of Advanced Computer Science and Applications,* vol. 9, no. 8, pp. 18-22, 2018.

[33] C. Hewitt, P. Bishop and R. Steiger, “A Universal Modular Actor Formalism for Artificial Intelligence,” *Proceedings of the 3rd International Joint Conference on Artificial Intelligence,* pp. 235-245, 1973.

[34] “What is Akka.NET?,” Akka.NET project, [Online]. Available: https://getakka.net/articles/intro/what-is-akka.html. [Accessed 14th August 2020].

[35] J. Bonér, “introducing akka - simpler scalability, fault-tolerance, concurrency & remoting through actors,” 4th January 2010. [Online]. Available: http://jonasboner.com/introducing-akka/. [Accessed 14th August 2020].