A Simplified Approach to Distributed Message Handling in a CQRS Architecture

Munonye Kindson, Martinek Péter

Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics
H-1111 Budapest, XI. Egry Joszef u. 18, Budapest, Hungary
Email: kindson.munonye@edu.bme.hu*; martinek@ett.bme.hu

Abstract: Architecting distributed information system is not a trivial task. This is especially true when a relatively novel design pattern such as Command Query Responsibility Segregation and Event Sourcing is applied. This research aims to describe a simplified approach to the handling of three kinds of messages involved in a CQRS architecture: Commands, Events, and Queries. An evolutionary approach to microservices design was applied to study the information flow within the architecture and establish the reliability properties. Using this approach, a prototype of a Basic Order Fulfillment System was designed and software complexity analysis was applied. The results obtained showed a significant reduction in complexity. The metrics indicated that the approach proposed in this research not only simplifies the process of distributed message handling, but has lower overall complexity than conventional microservices design methods. Therefore, with further refinement, the model can be a standard for building Event Driven Systems.

Keywords: CQRS; Event Sourcing; Command Handling; Aggregates; Event Handling; Microservices; Query Handling

1 Introduction

CQRS is a relatively new architectural pattern for software design based on Command Query Separation (CQS) This concept was proposed by Bertrand Meyer in the late 90s in the context of Object-Oriented Software Construction [1]. The idea of CQS is the separation of operations on application objects into two categories: (1) methods that modify the state of the object are called commands and (2) methods that retrieve the current state of the object are called queries. Naturally, such partitioning adds additional complexity relative to a single partition architecture since both the commands and the queries would have to be in sync to ensure the underlying data store remains in a consistent state.

Command Query Responsibility Segregation (CQRS) which derives from CQS is therefore a design pattern that partitions the commands and queries into separate services based on microservices. It is an Event Driven Architecture (EDA), where communication between the services is achieved via messages [2] [3]. This makes it possible to communicate state changes in an application to other parts of the application anytime when changes in the state occur. These changes occur as a result of a command modifying the read store. Such modification needs to be updated in tandem in the write store. Hence, Event Sourcing (ES) is used with CQRS to enable synchronization between the read data store and the write data store each time a change occurs [4]. Therefore, ES and CQRS are usually used together, though this is not a requirement [5]. However, some challenges need to be overcome. This includes the complexity of the CQRS/ES architecture, performance optimization [6], and managing data conversion across data stores [7]. This research focuses on mitigating complexity-related challenges by simplified message handling (SMH).

The rest of this paper is arranged as follows: Chapter 2 covers the architectural details of the CQRS pattern including aggregates specification, commands, events, and queries. Related works are discussed in Chapter 2. The methodology applied for this research is explained in Chapter 3. In Chapter 4, the analysis, design, and implementation process of a prototype based on the CQRS is covered with a focus on commands, events, and query handling. Chapter 5 presents the results and discussion. Finally, the summary and conclusion sections provide a summary of achievements, limitations, and areas of further research.

2 Theoretical Overview and Related Works

This chapter provides a general detail of the CQRS pattern and establishes the need for efficient message handling.

2.1 Aggregate and Entities

An aggregate is the primary building block for implementing the command model (write-model) in a CQRS-based system. In the context of CQRS, aggregate has variously been defined as a group of entities or functional entities [8] grouped within a single ACID transaction [9]. But a more encompassing definition is:

"an aggregate is a functional entity or group of functional entities in a specific domain and processed together within a consistent boundary such that it is always kept in the consistent state".

The concept of aggregates is generally understood by considering the relational entities in a conventional three-tier architecture. These entities are @Entity annotated classes that map to tables in a relational database. In CQRS, we can consider aggregates as @Aggregate annotated classes that map to aggregate in an

event store and as such, the state of the aggregate is derived (sourced) from the collection of events. Table 1 provides some differences between an entity used in relational systems and an aggregate used in CQRS applications. A summary would be that an aggregate is always made up of an entity (or some entities) but not vice versa. A key point is that aggregates are not instantiated by a 'command-handling constructor'. This constructor publishes an event that is event-sourced by an event-sourcing handler and the aggregate identifier must be set in the event-sourcing handler of the very first event published by the aggregate.

Table 1
Aggregate in CQRS vs Entity in RDBMS

Aggregates in CQRS	Entities in Three-Tier Architecture
Annotated with @Aggregate annotation	Annotated with @Entity annotation
Persisted in an Event Store	Could be stored in a Relational DB
Contains Command Handlers	Contains normal methods
Part of the command model	Implemented as part of the query model
Contains only relevant attributes	Contains all attributes

2.2 Message Handling

The CQRS pattern is a message-driven architecture that leverages the use of message objects. A message is a unit of information or an object that can be passed from one component to another. Therefore, messages have to be defined just like other classes in the application. Messaging-based architecture offers the following benefits:

- maintainability is improved since explicit messaging focuses on message design,
- Message objects can be stored and used subsequently for processing
- Explicit messaging eases the transparent distribution of information to remote components

In this research, the handling of three specific types of messages is considered with each playing a different role. Command messages are those that express an intent to mutate the application state and therefore are routed to a specific destination. Query messages just like queries in relational databases are requests for some data. Event messages are messages that provide some notification that an operation has been performed. Normally, events are generated when a command is carried out. Types of messages in a CQRS architecture are provided in Table 2.

Table 2
Types of Messages in a CQRS Architecture

Message Type	Details	Method	Handler
Commands	Changes state of the application	CommandHandler	*handle()*

Events	Notification that some action occurred	EventHandler	*on()*
Queries	Request for data	QueryHandler	*on()*

2.3 Related Works

Some of the existing works in the area of CQRS/ES are provided in this section. In related research on Determining Critical Properties of the CQRS pattern, the authors of the thesis with the title '*Correctness for CQRS Systems'* worked on determining the proof that a system is correctly implemented [10]. They applied a verification model to determine the critical properties of the system whose presence could determine a correct implementation thereby bridging the gap between system specification and actual implementation. So, the focus was not on mitigating complexity but on determining functionality and correctness. Another research is in the area of using CQRS and Event Sourcing to enhance transaction performance for distributed systems. This research covers the application of CQRS and Event Sourcing on distributed systems as a way to mitigate issues related to monolithic application design such as Create, Read, Update and Delete (CRUD) approach. Performance improvements were also highlighted via the creation of read-optimized views from data denormalization [11]. Additionally, it was shown that persisting of prebuild aggregates could also improve performance.

Research on the application of CQRS to Point Trading System highlights the specific techniques and tools for CQRS pattern in a point trading system [12]. This was achieved in combination with the Actor model and event sourcing to achieve scalability and performance. In this research, aggregates are encapsulated as actors with state and behaviour and therefore can be manipulated in a message-driven way via messages. The aggregates can then be reconstructed by events replay from an audit log which serves as an event source.

2.3.1 Application to Medical Information System

The authors in this research proposed an approach for handling the challenge of fragmented data from several tables which is the case in many medical information systems. Therefore, the CQRS is recommended as a method for denormalizing the read database to reduce data access time for frequent queries [13]. Model-driven approach was used for data modelling, mapping, and code generation. Finally, they proved how a synchronization component can be used for the migration of data from the main database to a read-database using an existing data replicator.

2.3.2 Use of Distributed Command Bus

The book '*Practical Microservices Architectural Patterns*'[2] shows how a distributed command bus can be used for relaying a command generated in one service to another service. Clustering event bus and other components including Advanced Message Queuing Protocol (AMQP) was used to route command and

event between 5 services. This, however, introduced some complexity to the architecture.

Of the related works cited, a key limiting factor is the complexity of the CQRS architecture compared to the 3-tier architecture arising from the addition of more artifacts to ensure synchronization between the command and the query aspects of the system. This is what this research aims to solve.

3 Methodology

The methodology applied in this research follows the algorithm for the minimization of Finite State Machines (FSM) [14]. So, if the message handling within the CQRS architecture can be specified using an FSM, then a reduction algorithm can be applied to eliminate complexity. Therefore, an architecture with x components could be scaled down to y, where $y < x$ without losing any functional requirements.

3.1 Basic Message Handling Model

Figure 1 indicates a general message-handling architecture for three-layer architecture (TLA) and CQRS architecture. Two message types are handled for the layered pattern: request message and response message. For the CQRS, the three messages handled are commands, events, and queries. For the TLA, messages are routed through 4 components while for the CQRS, 8 components are indicated. Therefore, the complexity of the CQRS is obvious from the added interfaces.

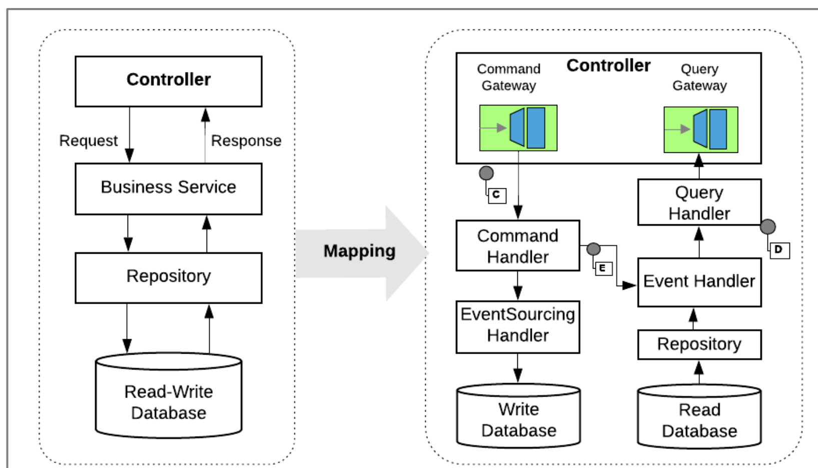

Figure 1

Simplified Message Mapping

To successfully model commands, events, and query handling in CQRS and ES-based systems, it is necessary to first identify the building block of the architecture. Then, information flow across the fabric of the architecture could be formally represented. The following three sections outline the core components playing a role in CQRS and ES. Since events are just notifications of an occurrence of some operation, event handling overlaps with both commands, queries, and common components.

3.2 Command/API Components Modelling

The command components are architectural pieces involved in the execution of a command message from the instantiation throughout the life cycle of the command. These are presented in Figure 2. Codes have been assigned to enable representation using property graphs.

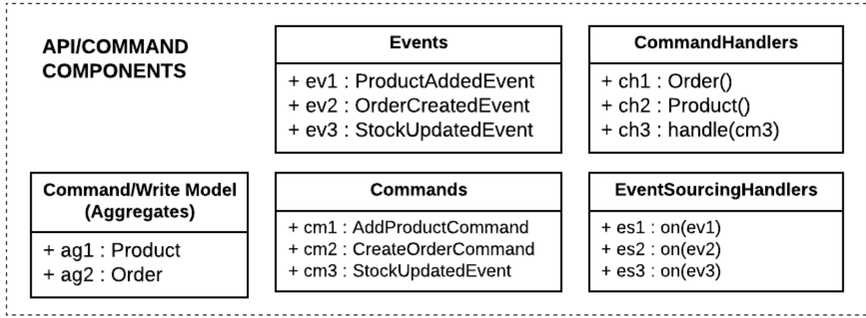

Figure 2
Command Handling Components

Based on Figure 2, any system consisting of *k* commands, *m* events, and *m* aggregates can be represented using equations 1 to 3:

$$Commands\ cm = \{cm_1, cm_2, \dots, cm_k\} \quad (1)$$

$$Events\ ev = \{ev_1, ev_2, \dots, ev_m\} \quad (2)$$

$$Aggregates\ ag = \{ag_1, ag_2, \dots, ag_n\} \quad (3)$$

If there are *x* commands and *y* command handlers, then $x = y$. Similarly, if there are *x'* event and *y'* event-sourcing handlers, then $x' = y'$. It is also necessary to note that the first command in an aggregate is the *'command handling constructor (CHC)'* since it instantiates the aggregate. Therefore, it has the name of the aggregates. Other command handlers have the name *handle ().* In Figure 2, the two CHCs include *order ()* and *product ()*. The set of event-sourcing handlers and command handlers is represented in Equations 4 and 5.

$$EventSourcing\ Handlers\ es = \{es_1, es_2, \dots, es_m\} \quad (4)$$

$$Command\ Handlers\ \ ch = \{ch_1, ch_2, \dots, ch_n\} \quad (5)$$

Both the event sourcing handlers and the command handlers are generally defined in the aggregates. The command handlers handle commands publishing events while the event sourcing handlers allow the aggregates to be sourced from their events [15]. Therefore, state changes must be set in the event sourcing handler.

3.3 Query components Modelling

These components are responsible for ensuring that queries originating from external interfaces are routed in an efficient manner and the resulting response is sent back to the user. These are shown in Figure 3

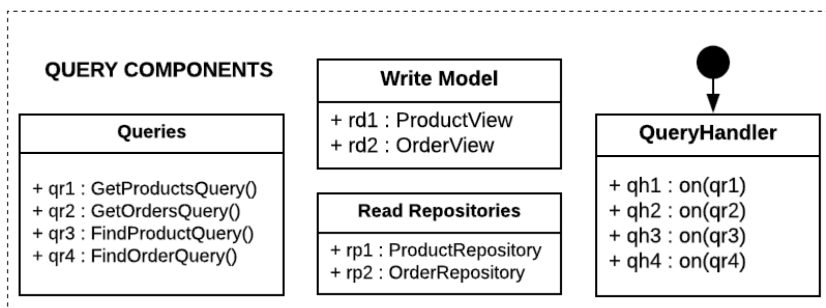

Figure 3
Query Handling Components

The number of query handlers (*qh*) as can be seen from Figure 3 equals the number of queries. Also, the number of read models equals the number of repositories. These can be represented using the four equations below:

$$Queries\ qr = \{qr_1, qr_2, \dots, qr_n\} \tag{6}$$

$$Query\ Handlers\ qh = \{qh_1, qh_2, \dots, qh_n\} \tag{7}$$

$$Read\ Models\ rd = \{rd_1, rd_2, \dots, rd_m\} \tag{8}$$

$$Repositories\ rp = \{rp_1, rp_2, \dots, rp_m\} \tag{9}$$

3.4 Events/Common Components Modelling

These are components involved in both commands and queries and general message routing. They include the gateways (*gw*), buses (bus), and controller endpoints (*ce*). Events have also been included in this section. They are simply a notification of some operations performed. These are given in Figure 4.

The controller endpoints are the methods that get executed when an *HTTP* request is received based on a *url* pattern. The number of methods would vary based on the system requirement. 6 have been provided here. The event handlers are the actual methods that get executed when an event is emitted: either dispatched from an aggregate as a result of command handling or published from some other object.

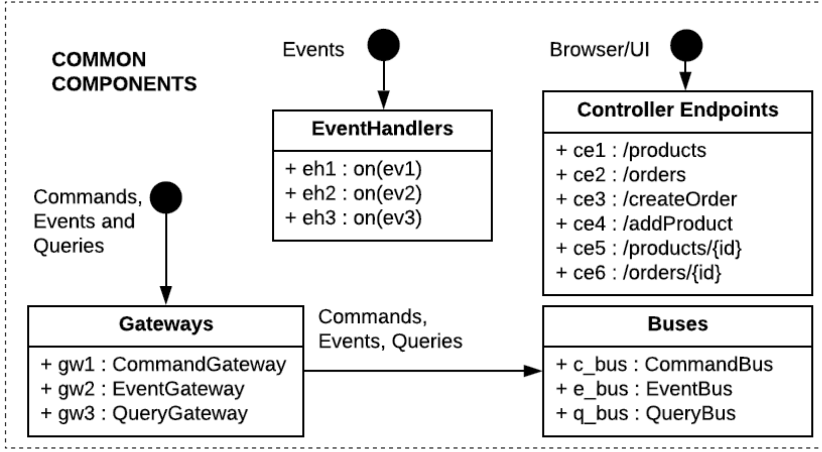

Figure 4
Common Components

The gateways are used to send messages to the buses while the buses relay the messages between components. These artifacts are represented as follows:

$$Gateways\ gw = \{e_{gw},\ q_{gw},\ c_{gw}\} \tag{10}$$

$$Buses\ bus = \{e_{bus},\ q_{bus},\ c_{bus} \tag{11}$$

$$Controller\ Endpoints\ ce = \{rd_1, rd_2, \dots, rd_l\} \tag{12}$$

$$Event\ Handlers\ eh = \{eh_1, eh_2, \dots, eh_m\} \tag{13}$$

3.5 Basic Query Message Handling

To be able to derive the state transitions for the message handling, a very basic message handling is defined: Query Messages for the CQRS which correspond to a normal GET request for TLA. The steps are represented as s_i where *i* is the number of the specific states as follows:

- A request is sent from a UI service to the controller endpoint (s_1)
- The controller interprets this request as a GET operation and routes it to the business service (s_2)
- The Service receives the request and executes a specific method (s_3)
- The request reaches the repository interface (s_4)
- The repository loads the database driver and connects to the database (s_5)
- Data is retrieved from the database by executing an SQL Select statement (s_6)
- The retrieved payload is sent back to the UI as an HTTP response (s_7)

The equivalent FSM is given in Figure 5. The five states are represented using the components of the architecture.

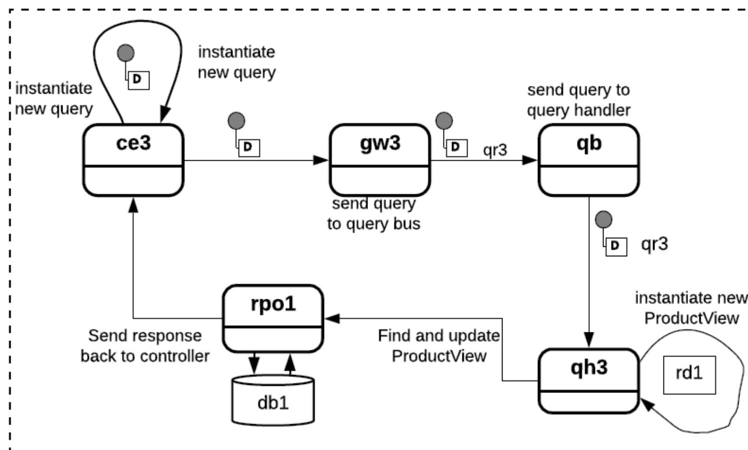

Figure 5
Query Handling in CQRS

3.6 Commands and Events Message Handling

Having considered query handling, this section now models the handling of the two other message types: commands and events. Figure 6 shows, the state machine for the order fulfilment process. The request for new order creation originates from the browser/UI and hits the controller endpoint (*cm1*). The command cm1 is sent through the command gateway (*gw1*) into the command bus (*c_bus*). Since the command handler (*ch1*) is defined in the order aggregate (ag1), the command bus sends the command to ch1. The command is executed and a new event *ev2* is published to the event bus. At the same time, the event-sourcing handler *es2* receives the event and updates the state of the aggregate. The event handler (*eh2*) for *ev2* is defined in *OrderProjector* and therefore gets fired. The *eh2* uses rpo1 to save the new order to the read datastore. Then a stock-updated event *ev3* is fired which is handled by *eh3*. The *eh3* uses *rpo2* to find the entity, update the state end save it back to the read data store. Figure 6 provides the message routing for this process. The detailed use case algorithm is provided in the next section.

Additional operations could be easily modelled using this simple approach. CRUD operations such as the addition of a new product and deleting and updating an order all follow the same approach. Requirements such as customized queries (QR), could simply be designed by defining more query classes in the projector files and adding the relevant rest endpoints (ce) to the controller file. This approach simplifies the CQRS/ES design and eliminates the associated complexity as would be shown in the implementation.

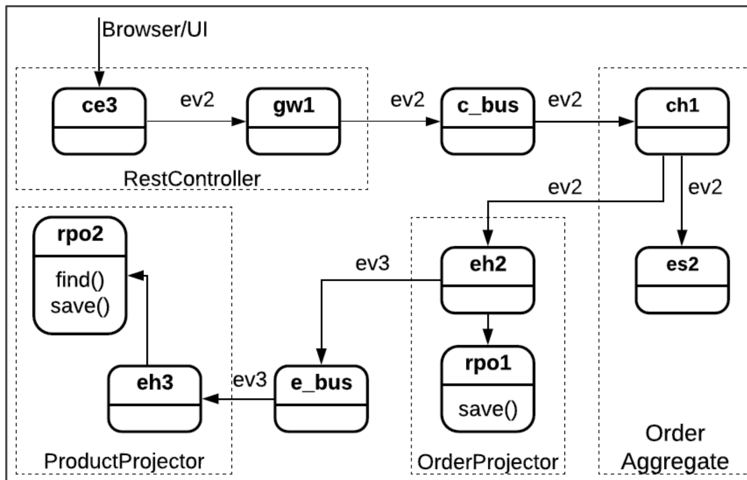

Figure 6
Command and Events Handling

3.7 Use Case Algorithm for Simple Order Processing

In this section, a scenario for message handling which combines all three types of messages is presented. The process is given for CQRS and then analysed and simplified using the FSM minimization algorithm.

The steps are given as:

A user places an order by clicking a link in the User Interface (UI). The request gets to the order controller (*OrderController.java*).

The order controller interprets this request as a post request and instantiates a *CreateOrderCommand* using the request *MultiValueMap* request parameter. This new command is sent through the command gateway to the command bus using the *send ()* async method of the command gateway.

The command is delivered via the command bus to the command handler in the aggregate (*Order.java*).

The command handler, which is a command-handling constructor, is called after that. It invokes the aggregate life,cycle *apply ()* method. This method is used when an event message needs to be published. Hence, a new *OrderCreatedEvent* is created using the details provided in the command and passed to the apply method. The event is published within the scope of the *Order* aggregate.

Then the event sourcing handler is called to 'event-source' the aggregate, i.e. to set the state of the new aggregate instance.

The event sourcing of the Order fires the event handler for the *OrderCreatedEvent* defined in the OrderProjector (*OrderProjector.java*). A new *OrderView*(read model) object is instantiated and saved to the database using the repository's *save ()* method.

With this, a *StockUpdatedEvent* object is instantiated using the details for the *OrderCreatedEvent*. This object is published through the event gateway to the event bus.

The event handler for *StockUpdatedEvent* defined in the ProductProjector (*ProductProjector.java*) receives the event from the event bus. Based on the events data, it finds the ProductView (read model) by calling the repository's *findById ()* method. It updates its stock and saves back the changes using the repositories *save ()* method.

Also, the event sourcing handler for the *StockUpdatedEvent* defined in the product aggregate (Product.java) gets fired to set the new state of the particular product aggregate

4 Analysis and Design

This chapter details the design and analysis process and techniques in the prototyping of an e-commerce order fulfilment process. The use case is an example of distributed command and event handling using Axon Framework [2]. We first design a basic CQRS architecture based on a single microservice. Next, it is extended by adding distributed command handling functionality. But first, here is a brief highlight of the key design and implementation tools used.

4.1 Axon Platform

Axon is a platform (Axon Framework + Axon Server) that enables the application of the CQRS and ES architectural style in developing modular, microservices-based solutions. Java 8+ was used for this research and the implementation was done using Maven. Axon abstracts all asynchronous behaviour via the use of executors, which decouple task submission from the internal process of how each task is run.

4.2 Scenario 1 – Basic Message Handling Design

A single CQRS-based microservices is described based on an e-commerce application. The service allows users to place an order. When this happens, a new order is created, and the product stock is depreciated based on the quantity in the order.

Based on the model outlined in Chapter 3, an evolutionary design approach is adopted which begins with the design of a single structured monolithic service. Then, it is partitioned based on the specification.

The message handling architecture in Figure 7 corresponds to the algorithm in section 3.7 of Chapter 3 and therefore would not be repeated. However, the actual implementation structure of the application is presented here.

The structured monolith consists of three packages: controllers, *coreapi*, read_model(command), write_model, and query. These packages are then later taken apart into separate microservices to form a distributed system.

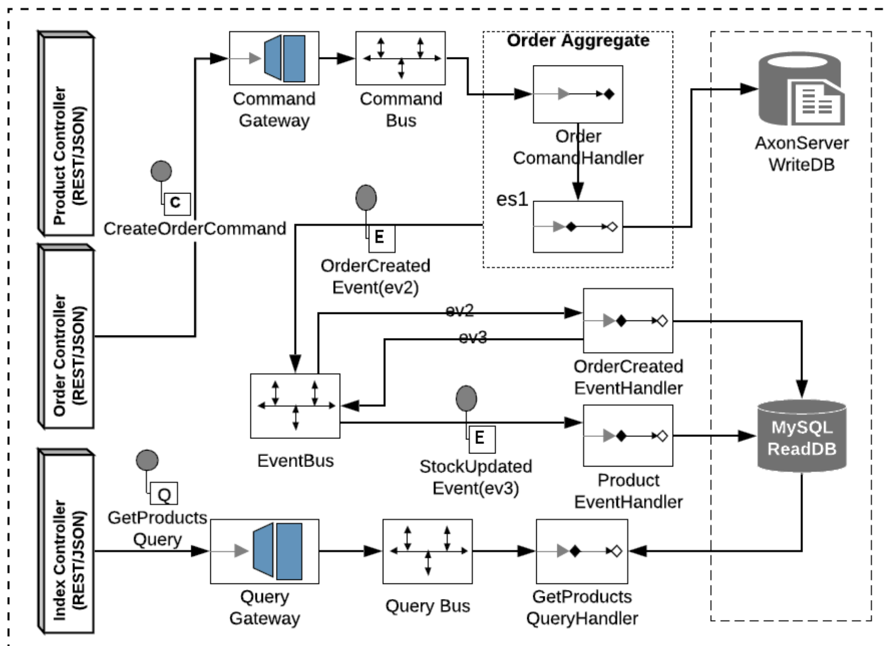

Figure 7

Message Handling Implementation with Axon 4.3

coreapi: The command and events are defined in this package under two respective Kotlin files, *commands.kt* and *events.kt*. These files contain data classes specifying the commands and the events.

write_model: The package contains the implementation of the aggregates. In this use case, two aggregates are identified: Product and Order defined in the *product.java* and *order.java* files respectively.

read_model: This is what is stored in the read database. In this case, it is a relational database.

query: The query package contains the *api.kt* file which has a definition of the queries and repositories. This package also contains the projectors. This is a component where the event handlers and query handlers are defined.

controller: This package contains all the rest controller classes annotated with @RestControllers. Methods available to HTTP endpoints are defined in the controller files.

4.3 Scenario 2 – Distributed Message Handling

In this scenario, the order processing is extended to distribute across JVMs and nodes. The processes are split into six different microservices as shown in Figure 8.

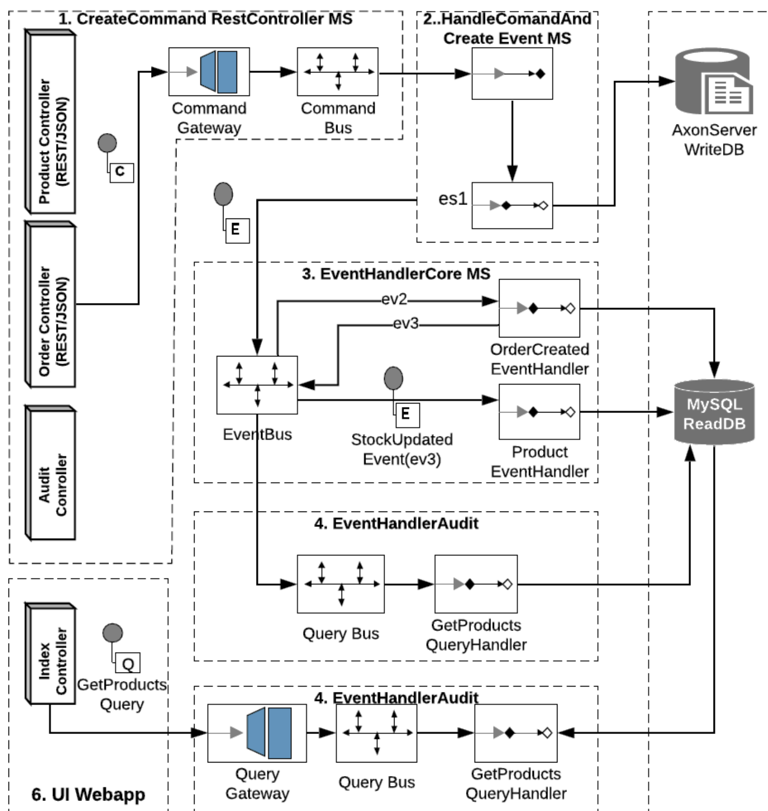

Figure 8

Distributed Command and Event Handling Implementation

The design of the distributed message handling follows from the structured monolithic design. For the distributed architecture, the contents of different packages are then built into an independent service.

UI-App µ-service: This is a UI service built with Angular.

Controller µ-service: This service is similar to the structured monolith except that this sends the *CreateOrder* command to a distributed command bus.

HandleCommand µ-service: This service receives commands coming from the Controller service. Also, it creates and sends out events to the distributed event bus.

HandleEvent µ-service: This service handles events published by the *CreateCommand* service

Audit µ-service: This service also gets notified when a *StockUpdated* event is published by the HandleCommand service

Common µ-service: Contains common configuration used by all the other services.

4.4 Measuring Complexity

As a proof of concept for the model used in this research, we adopt a formal strategy for the measurement of system complexity. Complexity in application development is defined as a measure of the resources expended by a system while interacting with a piece of software to execute a specified task [16]. There are also other variations of this definition, but the basic idea remains the same: a measure of the resources (time and efforts) expended to carry out some operation. The following two methods are used:

First, we use the weighted methods per class (WMC) which is part of the McCabe structure complexity metrics as defined in [17]. The second complexity measure used is known as software science [18] which provides a metric based on the program controls structure, program size, and the nature of the module interfaces.

4.4.1 McCabe Structure Complexity Metric

This approach to complexity measurement considers the program as a directed graph where the edges are lines of control flow while vertices are the linear segments of the program code. This is a computation of the WMC for each of the classes that make up the application. For each class *C*, there are M_1, M_2, … Mn methods defined, where each of the methods has complexity C1, C2, … Cn respectively. The WMC is given by

$$WMC = \sum C_i (i = 1, \dots n) \tag{14}$$

The complexity *C* of method *M* is calculated based on the flow graph derived from the business logic. With the flow graph, a complexity metric V(G) is calculated using the number of edges e, number of nodes n, and number of connected graphs p. The metric is given as:

$$V(G) = e - n + 2p \tag{15}$$

For our prototype services, the flowchart is presented in Figure 8 in Chapter 3. To be able to calculate the required metrics, we need to transform the flowchart into a data flow diagram (DFD). The corresponding data flow diagram is modelled according to the Gane and Sarson notations [19] which represents processes using rounded-corners squares.

4.4.2 Halstead Software Science Metric

This is a measurement of software complexity based on the code structure making up the modules. The Halstead approach computes three complexity metrics: the program Volume (V), the program difficulty (D), and the effort (E).

This measurement is based on the size of the program, the program module interfaces, and the control structure regardless of program comments on the stylistic components making up the code such as naming conventions and indentations.

The first step to evaluating the Software Science metric is to determine the functions making up the program. Then, the number of operators and operands is then determined. With these, we can then compute the volume of the program V and the program difficulty D. The SS metric are defined as follows. Let

- n_1 be the count of distinct operators
- n_2 be the count of distinct operands
- N_1 is the total number of operators
- N_2 is the total number of operands

Then the volume of the program, V is given by:

$$V = (N_1 + N_2)\log_2(n_1 + n_2) \tag{16}$$

And the program difficulty D is given by:

$$D = \frac{n_1 + N_2}{2n_2} \tag{17}$$

The value of D describes the complexity of the program and therefore a higher value of D indicates a higher complexity and vice versa. For this research, we first evaluate the complexity measures for each of the six object services. Then, we compute the parameters also for the combined microservices architecture. The detailed results are presented in Chapter 5.

5 Results and Discussion

This chapter presents the results in terms of complexity reduction achieved using the methodology proposed in this research. Complexity in applications is a measure of the resources consumed by a system while interacting with a piece of software in the execution of a specific task [18]. We focus on two:

- Structured complexity metrics
- Software Science metrics

The results of our methods relative to the existing approach for CQRS/ES are presented. The existing scenario for order fulfilment service can be found on the GitHub repository [20].

5.1 Structured Complexity (SC) Metrics

In this section, the values of the complexity metrics are presented based on the data flow diagram (*dfg*), G of each approach. The three values needed to determine the overall complexity are number of edges (e), number of nodes (n), and number of connected graphs (c). Based on these variables, the overall complexity V is calculated as $V(G) = n - e + c$. Table 3 shows the structured complexity metric values as calculated for the three approaches: structured monolith, existing microservice, and simple message handling. It could be observed that the structured monolith gave a value of 3 while the microservices architecture has a value of 12 which means a 4-fold increase in complexity. The approach in this research indicates a complexity value of 10, which is an improvement over the existing microservices design. However, thins the SC metrics are largely based on *dfg*. The next experiment provides a more complete result which includes program volume, difficulty, and effort as well.

Table 3
SC metrics for design approaches

Metrics for complete microservice	**Metrics**		
	Monolithic	**Existing**	**Our Approach**
Number of edges, e	12	21	**17**
Number of nodes, n	9	16	**13**
Number of connected graphs, c	6	17	**14**
Overall Complexity	3	12	**10**

5.2 Software Science (SS) Metrics

For the first part, the primary components of SS metrics for distributed messaging are tabulated and evaluated. Table 4 provides *N, n, D,* V, and E metrics for both approaches: the existing approach and the simplified message handling approach. Table 4 indicates that our simplified approach required more effort (540.9) than the monolithic approach (525.43) but less than the Microservices (MS). However, our approach provides the least difficulty (D) value of 1.03 against 1.2 and 1.14 for monolith and MS. The reduction in the program volume and difficulty is likely as a result of autoconfiguration provided by Spring and Axon which abstracts certain aspects of the application that would have required addition coding efforts.

An example is the *CommandGateway's send ()* method which uses the *CommandBus* underneath to perform the actual dispatching of commands.

Table 4
SS Metrics comparison

Metrics for complete microservice	Metrics		
	Monolithic	Microservices	SMH
Number of unique operators – n1	14.00	17.00	15.00
Number of unique operands – n2	25.00	29.00	33.00
Total number of operators – N1	37.00	38.00	41.00
Total number of operands – N2	46.00	49.00	53.00
Volume of program – V	438.69	480.55	524.99
Program difficulty – D	1.20	1.14	1.03
Effort – E	526.43	546.83	540.90

Table 4 indicates the SS metrics for individual services. In the case of the UI Service, the key difference is that it has been built with JavaScript based on AngularJS for the existing architecture. As for the SMH approach, Angular 9 has been used with the actual *HTTP* requests placed in a different *HttpClient service* file written in *TypeScript*. Table 5 shows the SS metrics computed by individual services. An important aspect of Table 5 is the *Common* microservices. This microservice becomes optional in our simplified approach due to autoconfiguration provided by the Spring Framework and therefore reduces the efforts and difficulty. Additionally, it could be noted that the metrics for the Controller MS remain the same in both Table 5 and Table 6 due to having to use the same endpoints and request methods in both approaches.

Table 5
Complexity metrics for Microservice

Microservices	Complexity Metrics						
	n1	n2	N1	N2	V	D	E
UI-App	8	12	14	14	121.01	0.92	110.93
Controller	4	7	11	8	65.73	0.86	56.34
HandleCommand	9	8	15	14	118.54	1.44	170.40
HandleEvent	10	8	13	13	108.42	1.44	155.85
Audit	6	3	6	9	47.55	2.50	118.87
Common	8	9	12	12	98.10	1.11	109.00

The same complexity metrics were calculated for the simplified message handling. Lower values of D and E have a slightly higher value of V indicated in Table 6. The UI-App with a volume of 116.76, a difficulty of 0.83, and an effort of 97.2 perform better than the MS and monolithic approaches since the UI was based on Angular 9 with Node Package Manager (*npm*) which used TypeScript instead of JavaScript. TypeScript is regarded as an extension of JavaScript and enables large-

scale development while addressing the failures of JavaScript [21]. Similar improvement was obtained for other services as well. It could also be noted that the highest program volume is recorded for the UI-App but has the least difficulty. The Audit microservice provides the highest difficulty (2.00) but the least volume (51.89) due to being based on three preceding services and required classes already defined.

Table 6
Complexity metrics for a simplified approach

Microservices	Complexity Metrics						
	n1	n2	N1	N2	V	D	E
UI-App	6	12	14	14	116.76	0.83	97.30
Controller	4	7	11	8	65.73	0.86	56.34
HandleCommand	9	8	14	13	110.36	1.38	151.75
HandleEvent	10	7	11	10	85.84	1.43	122.62
Audit	7	4	6	9	51.89	2.00	103.78
Common	0	0	0	0	0.00	0.00	0.00

Comparing the results of Table 5 and Table 6 (simplified approach), it could be observed that our approach not only simplifies the design process but abstracts some of the complexities via autoconfiguration.

Conclusion and future work

This research has provided a realistic approach to managing the complexity associated with the design of a CQRS and ES-based architecture. This was achieved, by first adopting a formal model to represent the components of a CQRS system. The data flow within the architecture was simplified by isolating the commands, events, and query components. The message handling was represented using a generic notation which can be extended to cover more complex systems. Implementation was achieved via an evolutionary approach from a structure monolithic system to a full-fledged microservices architecture. The novel approach of this research was also successfully tested with a use case of the order fulfilment application.

With respect to the CQRS and ES pattern, there are still more areas of further work. For instance, while this approach provided interesting results in a controlled lab-scale network environment, it would also be necessary to evaluate its performance in a cloud-based environment.

Additionally, a key area of interest for CQRS is the management of transactions distributed across services as in a Saga pattern. It is believed that the simplified approach of this research would also mitigate such challenges. However, this is currently being researched and the results would be provided subsequently as well.

References

[1] B. Meyer, “Object-Oriented Software Construction SECOND EDITION.” Accessed: Jul. 09, 2020. [Online]. Available: http://www.tools.com

[2] B. Christudas, *Practical Microservices Architectural Patterns*. 2019

[3] K. Machado, R. Kank, J. Sonawane, and S. Maitra, “A Comparative Study of ACID and BASE in Database Transaction Processing,” Vol. 8, No. 5, 2017, [Online]. Available: http://www.ijser.org

[4] “State Machine Design, Persistence and Code Generation using a Visual Workbench, Event Sourcing, and CQRS MSc Dissertation Author : Se ´ an Fitzgerald A thesis submitted in part fulfilment of the degree of MSc Advanced Software Engineering in Computer Sci,” 2012

[5] P. L. Meena, S. P. Sarmah, and A. Sarkar, “Sourcing decisions under risks of catastrophic event disruptions,” *Transp. Res. Part E Logist. Transp. Rev.*, Vol. 47, No. 6, pp. 1058-1074, 2011, doi: 10.1016/j.tre.2011.03.003

[6] Z. Long, “Improvement and Implementation of a High Performance CQRS Architecture,” *Proc. - 2017 Int. Conf. Robot. Intell. Syst. ICRIS 2017*, pp. 170-173, 2017, doi: 10.1109/ICRIS.2017.49

[7] M. Overeem, M. Spoor, and S. Jansen, “The dark side of event sourcing: Managing data conversion,” in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, Mar. 2017, pp. 193-204, doi: 10.1109/SANER.2017.7884621

[8] G. Maddodi, S. Jansen, and M. Overeem, “Aggregate architecture simulation in event-sourcing applications using layered queuing networks,” *ICPE 2020 - Proc. ACM/SPEC Int. Conf. Perform. Eng.*, No. 1, pp. 238-245, 2020, doi: 10.1145/3358960.3375797

[9] “Introduction - Axon Reference Guide.” https://docs.axoniq.io/reference-guide/ (accessed Jul. 03, 2020)

[10] H. Kamil, “Correctness for CQRS Systems: Elicitation and validation,” 2012, [Online]. Available: www.kth.se/csc

[11] S. O. Diakov, T. E. Zubrei, and A. S. Samoidiuk, “Application of Event Sourcing and CQRS in Distributed Systems,” No. 34, pp. 16-22, 2019

[12] Y. Zhong, W. Li, and J. Wang, “Using event sourcing and CQRS to build a high performance point trading system,” *ACM Int. Conf. Proceeding Ser.*, pp. 16-19, 2019, doi: 10.1145/3317614.3317632

[13] P. Rajković, D. Janković, and A. Milenković, “Using CQRS pattern for improving performances in medical information systems,” *CEUR Workshop Proc.*, Vol. 1036, pp. 86-91, 2013

[14] J. M. Pena and A. L. Oliveira, “Incompletely Specified Finite State Machines,” *Comput. Des.*, Vol. 18, No. 11, pp. 1619-1632, 1999

[15] "Aggregate - Axon Reference Guide." https://docs.axoniq.io/reference-guide/implementing-domain-logic/command-handling/aggregate (accessed Jul. 14, 2020)

[16] T. Mens, "Research trends in structural software complexity," *Semant. Sch.*, 2016, [Online]. Available: http://arxiv.org/abs/1608.01533

[17] H. Tu, W. Sun, and Y. Zhang, "The research on software metrics and software complexity metrics," *IFCSTA 2009 Proc. - 2009 Int. Forum Comput. Sci. Appl.*, Vol. 1, pp. 131-136, 2009, doi: 10.1109/IFCSTA.2009.39

[18] J. P. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler, "Software complexity measurement," *Commun. ACM*, Vol. 29, No. 11, pp. 1044-1050, 1986, doi: 10.1145/7538.7540

[19] D. Moody, "The physics of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Trans. Softw. Eng.*, Vol. 35, No. 6, pp. 756-779, 2009, doi: 10.1109/TSE.2009.67

[20] "practical-microservices-architectural-patterns/Christudas_Ch12_Source/ch12/ch12-02/Ax2-Commands-Multi-Event-Handler-Distributed at master Apress/practical-microservices-architectural-patterns GitHub." https://github.com/Apress/practical-microservices-architectural-patterns/tree/master/Christudas_Ch12_Source/ch12/ch12-02/Ax2-Commands-Multi-Event-Handler-Distributed (accessed Jul. 13, 2020)

[21] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014, Vol. 8586 LNCS, pp. 257-281, doi: 10.1007/978-3-662-44202-9_11