ACIDBRAIN: Maintaining data consistency in microservices

Nana Pang*, Wode "Nimo" Ni †, and Xin Chen ‡
Columbia University
Email: *np2630@columbia.edu, †wn2155@columbia.edu, ‡xc2409@columbia.edu

Abstract

As software systems become more distributed and larger in scale nowadays, there is a growing need for novel architectures that are modularized, flexible, and scalable. Microservice architecture emerged as a popular software architecture where developers divide a large application into small, self-contained components (services). Although the separation significantly increases flexibility and reduces difficulty in development, maintaining data consistency across these isolated databases can still be be challenging. Our project targets maintaining data consistency among distributed microservices. To maintain data consistency, an event driven model called **Saga**[1] is often used to keep track of the data sources of each transaction. In this project, we investigate two predominant event sourcing frameworks: *Eventuate* and *Axon*, from an **end-users perspective**. We experiment with both of them by building data consistency guarantees on a dummy microservice system, and analyze the effectiveness of the two frameworks both quantitatively and qualitatively.

I. INTRODUCTION

Large software systems have been growing rapidly and becoming more distributed. As a result, the traditional monolithic software architecture, where modules of different functionalities are closely coupled and developed by the same group of developers, is now challenged by a new architectural pattern: **Microservices**. A software system that employs microservice architecture comprises of a suite of clearly defined small services, each running in its own process [2]. As a result, each service can be developed by completely separated teams and using distinct technology stack, thereby decoupling the modules and parallelizing development.

In a monolithic system, all modules typically share the same backend database and enjoy the ACID (Atomicity, Consistency, Isolation, Durability) properties provided by the database [3]. Under microservice architecture, however, each module would use a separate database to ensure good isolation among service modules, and each database might be using an entirely different technology. Therefore, large transactions that involves multiple modules, which used to run in the same database, now span across multiple databases with drastically different implementations. In this case, maintaining data consistency becomes a more complex task. **Two-phase-commit (2PC)** protocol [4] is a popular solution, where a coordinator process ensures all databases have the requested resource available before commiting the transaction. While 2PC ensures the correctness of distributed transactions, it requires all resources to be exclusively locked until the transaction finishes.

Unfortunately, in real world applications, many distributed transactions are long-lived, meaning they take much longer (in terms of hours and/or network gaps) to finish. For example, an e-commerce application may have ordering, billing, and shipping modules, and a complete transaction of purchasing an item would involve all components and would not complete until the item is shipped. In the case of these long-lived transactions, 2PC does not scale well because it locks all databases involved in the transaction and there can be many other transactions occurring in the system simultaneously.

Among the existing solutions to maintain data consistency in distributed databases of microservice systems, we chose to look at the **Saga pattern** [1]. There are a few arguments for Sagas over traditional consensus protocols such as 2PC. Firstly, the Saga pattern does not require synchronization of all databases in a transaction, making it more suitable for long-lived transactions. Also, Saga tends to be easier to implement than 2PC, whose logic is relatively complex.

II. Background

A saga is a sequence of local transactions. Each local transaction updates the database and publishes an event to trigger the next local transaction in the saga [5]. Using this pattern, a large distributed transaction is broken into multiple local transactions that update their local databases and publish events globally to notify others, which are then coordinated by a saga. To maintain data consistency, each local transaction theoretically must be accompanied by compensatory transaction. As a result, the saga can rollback the large transaction by executing these compensations in a reverse order.

In general, there are two types of sagas: *choreography-based* and *orchestration-based*. An orchestration-based saga is a standalone object that coordinates multiple service in a centralized manner, whereas choreography-based saga is implemented by each local transaction publishing domain events that trigger local transactions in other services [5].

The use of microservices and sagas also motivates the overall architectural pattern to deviate from the traditional layered architecture. **Event sourcing** and **Command Query Responsibility Segregation (CQRS)** are often used to fully implement sagas. Since sagas coordinate local transactions by publishing events after updating local datastores, it is vital for the publication and update to be atomic - otherwise the system ends up in an inconsistent state. Event sourcing is used to enforce atomicity of events publishing and datastore updates. Event sourcing persists the state of entity as a sequence of state changing events. Whenever an entity changes, a new event is appended. The final state of each entity must be consistent since adding a single event is atomic. Therefore, we dont need compensation rollback to maintain data consistency but persist events in an event store, adding or retrieving event when an entity is changed.

Unfortunately, with multiple local databases involved in one event store, querying these databases becomes much more difficult to implement than in a monolithic system. Since the current state of all entities are stored as a sequence of changing events rather than a static record, it is difficult to obtain the most up-to-date state of a certain entity. To mitigate the complexity, the CQRS architecture is often used in implementations of sagas. CQRS separates queries from commands/modifications to the datastore

III. Existing Implementation of Sagas

During our early research on Microservices, we found a number of online documents about Microservices, but most of them only explain the architecture style of Microservices which more accessible but remain too high-level to guide implementation. For example, searching Microservices on Google gives around 1,700,000 results, Microservices example only gives about 706,000 results. Furthermore, Microservices data consistency gives only 62,300 results. Our project aims to find out more about what technology can be used in implementing Microservices with data consistency guarantees, which is a relatively less discussed topic with fewer alternatives in terms of concrete implementations.

Which systems we have chosen and why: Having decided to investigate Sagas, we looked into the existing implementations of Saga pattern for microservices. Among them, we picked **Eventuate** and **Axon**, which are the most popular and well maintained. We summerize the two frameworks briefly here:

- **Axon framework** [6] is a Java based framework for building scalable and highly performant applications. The main notion is the event processing which includes the separated Command bus for updates and the Event bus for queries. According to an original Axon authors response on StackOverflow [7], Axon has been around for about 8 years and is being used by many systems in production since then. Axon has extensive support for Spring. Yet Spring is not required for Axon, it made configuration easy with Spring annotations.
- **Eventuate** [8] is a platform that provides an event-driven programming model that focus on solving distributed data management in microservices architectures. The framework stores events in the MySQL database and it distributes them through the Apache Kafka platform. Eventuate is a framework that has integration with Gradle and Maven project. Therefore, we have implemented our prototype using Spring and Maven.

There are already comparisons online of the two frameworks [7], [9]. All of we could gathered, however, were written by implementers of transaction frameworks. For example, [9] was written by the a member of the Narayana team within Red Hat, which also has a similar implementation. Therefore, in this project, we would like to evaluate the end-user experience of using Saga frameworks to implement data consistency in microservice systems.

How we obtained the systems: Since both Eventuate and Axon frameworks themselves are freely available on the internet (it seems that some of other related services are not, we obtained them through their official websites [6], [8].

IV. Method

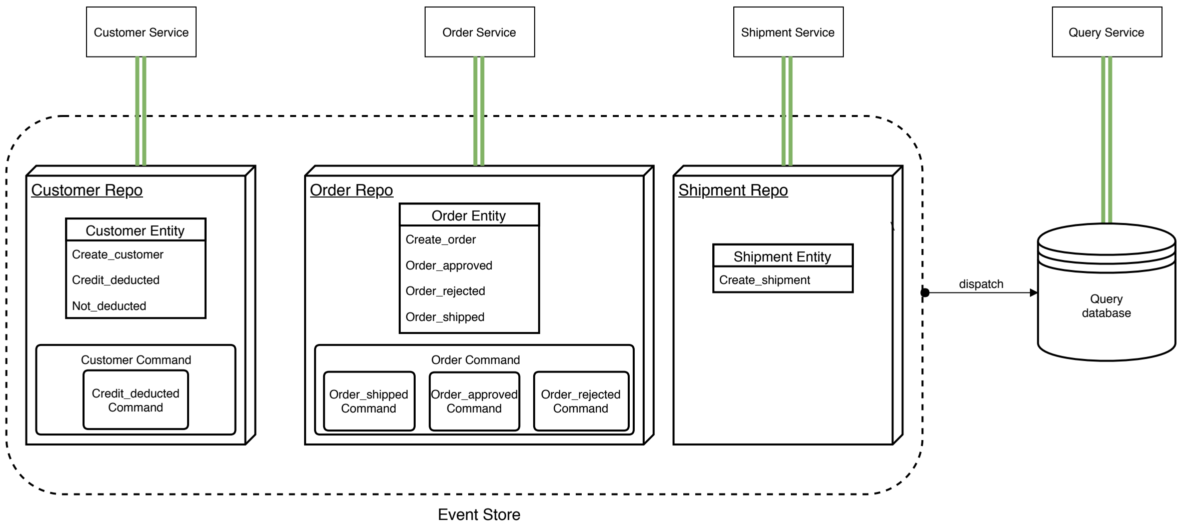

Figure 1: Overall structure of the e-commerce prototype

To evaluate the usability of Axon and Eventuate frameworks, we take on both quantitative and qualitative approaches. We implemented a dummy e-commerce system with a business logic that requires multi-database transaction in a rational manner and is very common in the real world. The general workflow of the system is the following:

1) Create a customer with name and credit limit
2) Create order
   - If order amount does not exceed customer credit limit, approve order and subtract credit.
   - If order amount exceeds customer credit limit, reject order and maintain credit unchanged
3) If order is approved, create shipment, order status change as shipped.

The system starts off with no data consistency guarantees. As the experiment, we implement data consistency using the two frameworks separately by two team members (Nana and Xin). To evaluate the usability quantitatively, we plan to record the following data:

- Total lines of Java source code (SLOC)
- Size of the systems on the disk
- Estimated time taken to implement the improvement
- Total time spent on the web (also separately on GitHub, StackOverflow, documentation website and other related reference sites)
- Documentation: total number of words, estimated coverage of the framework APIs
- Performance and stability: we use Artillery to load test the two implementations using Axon and Eventuate. If significant difference detected, we can also report system resource usages (CPU usage and network respond time).

Qualitatively, we will report our experience using the two frameworks, provide some more anecdotal cases of tasks such as reading the documentation, and showcase some snippets of code to discuss their API designs.

V. Results

A. User comments

This section includes first-person accounts from two members of our team who implemented data consistency for the same e-commerce prototype, respectively.

a) Eventuate (Nana):

- *Pros #1*: Eventuate framework maintains high modularity by dependency injection, using tags such as `@even-tentity`, `@event`, `@aggregate`, `@eventsubscriber` to define unit field. Each of these fields has a generic abstract class to inherit, which makes user easier to organize system structure. Decoupling is one of the benefits in modularity. Due to the high modularity in Eventuate, it also has low coupling as well.
- *Pros #2*: Eventuate persists transaction atomicity by using sequence of state-changing events in entity. Whenever the state of a business entity changes, a new event is appended to the list of events. Eventuate workflow includes command processing and event publishing while updating entity status. This design reconstructs an entitys current state by replaying the events. Since saving an event is a single operation, it is inherently atomic.
- *Pros #3*: Eventuate provides clean and concise samples for users to start up their application. Those samples have great code legacy in modularity, which divides a large system into different layers and helps user to find useful information efficiently.
- *Cons #1*: Eventuate is difficult to configure or saying, difficult to configure without Docker. Unlike Axon, which could download the released package and run locally, Eventuate provides services through Docker and Docker Compose. Therefore, it requires user to learn how to use Docker first. Eventuate also provides free AWS platform to run their services without local configuration. But it takes them 2 weeks to respond my request for AWS access key, which is quite slow.
- *Cons #2*: One key drawback of Eventuate is that it uses too many extra products such as Zookeeper and Kafka to implement functionality. In our system, we have implemented 4 services but using Eventuate requires 5 extra services to run first. The resource consumption such as CPU utilization is higher than Axon. Also, during our experiment, we found that some services in Eventuate is not stable and often stop running in Docker automatically. It is possible that Eventuate has bug not found, which makes it unstable or not compatible with Docker.
- *Cons #3*: Even though the sample code is well organized in Eventuate, its tutorial is worse than Axon or any other framework tutorial I have read before. The documentation merely defines the syntax of the API, but **not** the underlying semantics the intents of the functions and their effects on the whole system. Also, the tutorial covers only a small section of implementation in Eventuate. For example, EventUtil is frequently used in sample code but never explained in tutorial.

b) Axon (Xin):

- *Pros #1*: Building with Axon is highly modular, each entity or a type of entities is represented by an aggregate. Inside a aggregate class, we declare its attributes and aggregate id. `Commands` and `Events` are also classes that capture what actions to take. Axon extensively provide annotation support, such as `@aggregate`, `@commandHandler`, `@eventSourcing`, which made it easy to the developers.
- *Pros #2*: Axon official website [6] has documentations that explain terminologies and provide code samples. It is a well-documented framework, and updates regularly.
- *Pros #3*: Configuration with Spring is also easy. Spring is not a prerequisite for using Axon, so developers have the flexibility to make their own decision based on personal preference.
- *Pros #4*: Axon is open-source and freely available, there are big organizers such as banks using Axon in their core system. This also indicates that Axon is a fairly mature framework.
- *Cons #1*: Hard to handle mistakes in event handlers locally: if an error appears in one of the event handlers, there is no other way than reconstructing past states. The replaying process may take a long time depending on data size.
- *Cons #2*: Since commands and events are immutable, they have to be defined by Kotlin (Kotlin is a statically typed programming language that runs on the JVM and also can be compiled to JavaScript source code or use the LLVM compiler infrastructure.) and put into core api. Although Kotlin allows users to concisely define each event and command on a single line, it may add extra work for people who has no experience with Kotlin.

B. Code examples

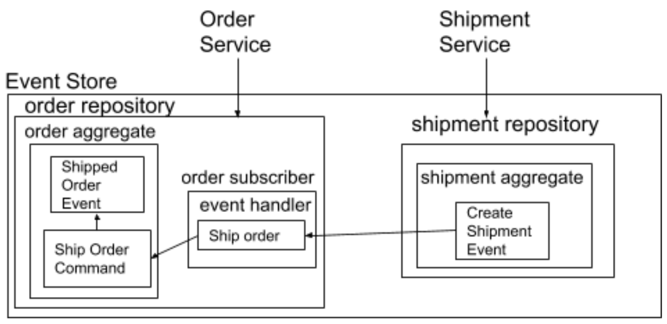

Figure 2: Content in event store for shipping and ordering service

1) Eventuate:

i Create basic components of the service: `ShipOrderCommand`, `OrderShippedEvent`, and `ShipmentCreated-Event`. (Listing 8 in Appendix)

ii Set up event handler method in order service to monitor `ShipmentCreatedEvent` and invoke `ShipOrder-Command` if `ShipmentCreatedEvent` is generated. (Listing 7)

```
// Register EventSubscriber for order service
@EventSubscriber(id="orderWorkflow")
public class OrderWorkflow {
  // Register EventHandlerMethod for listening ShipmentCreatedEvent from shipment service
  @EventHandlerMethod
  public CompletableFuture<EntityWithIdAndVersion<Order>> shipOrder(
          EventHandlerContext<ShipmentCreatedEvent> ctx) {
    // Get orderId from ShipmentCreatedEvent
    String orderId = ctx.getEvent().getOrderId();
    // Get order state from ShipmentCreatedEvent
    OrderState state = ctx.getEvent().getState();
    // Update order aggregate with same orderId and process ShipOrderCommand
    return ctx.update(Order.class, orderId, new ShipOrderCommand(state));
  }
}
```

Listing 1: Event handler example in Eventuate

iii Overload `process` function in `ShipOrderCommand` class and `apply` function in `OrderShippedEvent` class in `Order` aggregate to generate `ShipmentCreatedEvent` and change order status. (Listing 2)

```
// Order class extends ReflectiveMutableCommandProcessingAggregate class in Eventuate
// ReflectiveMutableCommandProcessingAggregate has two type parameters: the aggregate type
    and the aggregate s command superinterface.
// ReflectiveMutableCommandProcessingAggregate uses reflection to invoke the appropriate
    handler method for each command and event.
public class Order extends
    ReflectiveMutableCommandProcessingAggregate<Order, OrderCommand> {
  // Each orrder aggregate contains order state variable and customerId
  private OrderState state;
  private String customerId;
  // Overload process method for ShipOrderCommand
  // ShipOrderCommand creates new OrderShippedEvent
  //    in order aggregate repository
  public List<Event> process(ShipOrderCommand cmd) {
    return EventUtil.events(new OrderShippedEvent(customerId));
  }
  // Overload apply method for OrderShippedEvent
  // OrderShippedEvent change order state to SHIPPED
  public void apply(OrderShippedEvent event) {
    this.state = OrderState.SHIPPED;
    }
}
```

Listing 2: Overloading `apply` and `process` in Eventuate

2) Axon:

i Basic unit is aggregate. An aggregate is an entity or a collection of entities. It needs to have an aggregate id.

```
@Aggregate
public class OrderAggregate {
    @AggregateIdentifier
    private String orderId;
    private boolean completed;
    // ...more members in this class
]
```

Listing 3: `Aggregate` example in Axon

ii Command classes are identified by the aggregate id it relates to.

```
public class OrderCompletedCommand {
    @TargetAggregateIdentifier
    private String orderId;
    private ProductInfo productInfo;
    public OrderCompletedCommand() {
    }
    public OrderCompletedCommand(String orderId, ProductInfo productInfo) {
        this.orderId = orderId;
        this.productInfo = productInfo;
    }
    public String getOrderId() {
        return orderId;
    }
    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }
    public ProductInfo getProductInfo() {
        return productInfo;
    }
    public void setProductInfo(ProductInfo productInfo) {
        this.productInfo = productInfo;
    }
}
```

Listing 4: `Command` example in Axon

iii Each aggregate defines their own commandHandlers, each eventSourcingHandler corresponds to one type of commandHandler. For the example below, we assign orderId to this order object only if the command is of type `FileOrderCommand`.

```
@CommandHandler
public OrderAggregate(FileOrderCommand command) {
    apply(new OrderFiledEvent(command.getOrderId(), command.getProductInfo()));
}
@CommandHandler
public void handle(OrderCompletedCommand command) {
    apply(new OrderCompletedEvent(command.getOrderId(), command.getProductInfo()));
}
@CommandHandler
public void handle(CancelOrderCommand command) {
    apply(new OrderCancelledEvent(command.getOrderId()));
}
@EventSourcingHandler
public void on(OrderFiledEvent event) {
    orderId = event.getOrderId();
}
```

Listing 5: Handlers defined in Axon

iv After the order goes through, Shipment aggregate will receive `prepareShipmentCommand`, and apply `ShipmentPreparedEvent` for this order.

```
@CommandHandler
public Shipment(PrepareShipmentCommand command) throws InterruptedException {
    log.info("received PrepareShipmentCommand command for order: " + command.getOrderId());
    String id = Util.generateId();
    int shipment = computeShipment(command.getProductInfo());
    apply(new ShipmentPreparedEvent(id, command.getOrderId(), shipment));
}
```

Listing 6: Shipment aggregate in Axon

v The eventSourcingHandler for `ShipmentPreparedEvent` is called after applying the event successfully in the commandHandler.

```
@EventSourcingHandler
public void on(ShipmentPreparedEvent event) {
    this.id = event.getShipmentId();
    this.orderId = event.getOrderId();
    this.price = event.getPrice();
}
@EventSourcingHandler
public void on(ShipmentPreparationFailedEvent event) {
    this.id = event.getShipmentId();
    this.orderId = event.getOrderId();
}
```

Listing 7: Shipement event handler in Axon

C. Load test and Resource Consumption

	Axon	Eventuate
CPU Utilization	66.4%	19.6%
Median Request Latency	58.4 ms	345.6 ms
Service Break-down	No	Yes

Table I: Comparison between Axon and Eventuate

To test for the **stability** of Axon and Eventuate, we used Artillery to load-test the prototypes. Since data consistency is guaranteed by the framework and thus is not the central concern for this particular test, we only test for a part of the whole purchase transaction: *assuming an existing user, place an order with the ID of that user*. In this test, we repeatedly create new orders under the same customer and test whether the system can handle the traffic load. In the load-test for Eventuate, **12000** requests are sent within 120 seconds, and all request succeeded. However, the same load completely broke the Evantuate version of the same prototype. Note that although the requests were never successful, no partial transactions were performed by Eventuate, and the system is indeed consistent. Therefore, Evantuate was tested using fewer number of requests. The results are summarized in Table I.

D. Development tatistics

The following table summarizes statistics about the two prototypes and development of them. Fully LOC information can be found in Appendix (Listings 9, 10).

	Axon	Eventuate
Total Time Spent	41.3 hours	63.5 hours
LOC (Java)	1189 lines	2304 lines
LOC (Total)	2300 lines	3227 lines

Table II: Development statistics of two prototypes

During the development, we also attempted to search on `http://stackoverflow.com/` for either Axon or Evantuate related questions, As of May 4, 2018, searching or "Axon" gives **336** results[1], whereas searching for "Eventuate" gives **51** results[2].

VI. Discussion

A. Comparing Axon and Eventuate

	Axon	Eventuate
Aggregate	Use `@Aggregate` annotation in the beginning of class. Place `@AggregateIdentifier` on top of unique identifier for this aggregate. Aggregate contains `@commandHandler` annotation and `@eventSourcingHandler` and overload apply method to update entity state.	Extend `ReflectiveMutableCommand-ProcessingAggregate<aggregate, command>` class to define aggregate. Overload process and apply method in aggregate update entity state.
Event	No class to define events. Prototypes in core api by Kotlin	Use `@EventEntity(event=event_name)` annotation in Event class
Command	User defined command class which uses `@TargetAggregateIdentifier` to connect with Aggregate	Extend `Command class` to define command
Saga	Is a special type of Event Listener that manages a business transaction. Sagas are classes that define one or more `@SagaEventHandler` methods. All Sagas must implement the Saga interface.	Use `@EventSubscriber` annotation to define event listener in each service. Use `@eventhandler` method annotation to define transaction message.

Table III: Comparison between Axon and Eventuate

As shown in Table III, we can find some syntactic difference between Axon and Eventuate. Axon requires users to define their own command using `@TargetAggregateIdentifier` annotation corresponding to `Aggregate` identifier. Therefore, command and aggregate are combined using annotation in Axon. Eventuate has much clearer structural architecture. For instance, event, entity and command are units defined as generic abstract class in Eventuate. User could implement them without knowing `Aggregate` definition. Therefore, command is coupling with `Aggregate` in Axon, while decoupling in Eventuate. So, Axon has more complicated syntax and lower decoupling ability. Also, since Axon and Eventuate are primarily CQRS based frameworks, they require some additional handling for the saga execution. In Axon, saga is still the aggregate which means that it consumes commands and produces events. This may be an unwanted overhead when the application does not follow the CQRS domain pattern. The same applies to Eventuate but as the communication is shadowed through the messaging channels it does not force the programmer to follow CQRS. Instead, event handler methods in command service and query service are defined separately in Eventuate, which provides more flexibility of system architecture. So, we can say that Axon is more fixed while Eventuate is more flexible.

B. Prospective User Community

Target user community and value added

The user community of Microservices is large and keeps growing. As we presented, Microservices has lots of advantages. Even though Microservices architecture shares ideas with other models such as Service-Oriented Architecture, it has attracted much attention from big companies such as Amazon, Uber, and Netflix. Thus,

[1]https://stackoverflow.com/search?q=axon
[2]https://stackoverflow.com/search?q=eventuate

for software developers, learning and implementing Microservices architecture could be a increasingly popular requirement in the future.

To apply Microservices in real-world products, we need to deal with tradeoffs of such a decentralized system. Importantly, data consistency across distributed databases is vital for many business systems. In certain systems, it is conceptually and realistically difficult to perfectly divide your software into completely independent and isolated services. As a result, data transactions that span two or more services are inevitable in such cases. As the amount of Microservices architecture increases, the importance of maintaining data consistency in Microservices will be increasing as well.

What do your results show that was not already known

How members of the user community will find out about your study

Accessing our results

Since the concept of microservice is relatively young (the term was first used in 2011 [10], the topic of maintaining data consistency is under-researched in the academia and implemented mostly in an ad-hoc fashion. Our study investigated two of the few existing software frameworks that aim to achieve data consistency and work well with systems of microservice architecture. More importantly, instead of evaluating from an implementers point of view such as [7], [9], we experiment with these frameworks as end-users and attempt to evaluate the usability of them both quantitatively and qualitatively. Therefore, we believe that the results from this study can be shared to the user community of Axon, Eventuate, or any frameworks alike, and provide a reference point for choosing one of them. Our GitHub repo[3] contains original documents and prototypes that we built, and it is freely accessible to everyone.

C. ACIDBRAIN *Team Comments*

1) Nana:

- **Responsibilities**:Implement original prototypes using Java and Spring. Implement prototype with customer, order, shipment and query services by using Eventuate framework. Test Api through Swagger UI and deploy application on AWS.
- **Comments**: At the beginning of development, I found some methodology such as Saga pattern, CQRS view and event sourcing are difficult to define or distinguish. Online resource of these concept is incomplete and vague. During the implementation process, I stuck at the configuration and building part for a long time, since I am not familiar with Spring cloud and Docker which is used by Eventuate. Therefore, I chose to study from an existed sample Eventuate provided, analyzing its code structure and labeling functionality within each modules. Since I have experienced Java and Spring development before, I started to make some progress. After implement the shipment service, I realize that Eventuate doesnt need compensation function as I assumed because it maintains entity status by logical checking and only apply the eventually consistent event in event store. All the event states are remained unchanged after initialization and compensation is handled by adding new event such as order-approve-event and order-reject-event after order-created-event.This feature has both pros and cons. It simplifies the complexity of understanding system workflow but adds extra nodes in code implementation. But it is always difficult to find a perfect solution. Every method has its pros and cons. Therefore, in project report, I have discussed pros and cons of Eventuate and compared usability of Eventuate and Axon. This topic is related to my midterm paper. I didnt realize the data consistency issue in Microservices architecture until presentation. It is my pleasure to learn more about Microservices and implement a system in Microservices. I hope our project could be helpful to other end-users who are interested in software architecture as well.

2) Nimo:

- **Responsibilities**: Maintaining AWS server and docker, load-test scripting, (part of) implementation of original prototype, development statistics gathering, report drafting, and presentations.
- **Comments**: My software engineering experience has primarily been in academic settings. Therefore, reading about and experimenting with newer technologies from the industry was eye-opening to me. Interestingly, patterns such as microservices become popular because of not just its technical benefits, but also faster deployment and more distributed development model, both being more business and human related concerns. One of the first problem that we encountered was the lack of strict definitions of many terms such as event

[3]https://github.com/nanahpang/Microservices project deliverable

sourcing, sagas, or microservices, which is expected given that these terms are not used rigorously. Later on, we experienced roadblocks when setting up the development environment using AWS, docker, Maven, and other tools. The technical overhead significantly impaired the complexity of our prototype system and the later improvement on them using the two frameworks. Since both Axon and Eventuate are designed to solve issues in large, complex systems, it might be more reasonable to experiment them on a larger system with more services and significantly more complex and coupled business logic, which we unfortunately did not have. Finally, some comments on usability of frameworks: Frameworks exists to ease software development, among many other reasons. As we demonstrated in the project, there are still highly visible flaws in the design of the APIs of both Eventuate and Axon. There is always a human-factor in the design of frameworks. As API usability [11] gets more attention in the field of software engineering and Human-Computer Interaction, more principles and guidelines should be developed for better design of them.

3) Xin:

- **Responsibilities**: Learning and implementing original prototypes using Axon framework. Modify original prototypes to adapt axon environment. Test order, invoice, shipment services by posting and getting requests on Swagger. Axon development time statistics tracking and documenting obstacles.
- **Comments**: I have little preliminary experience in Spring, so I spend fairly large amount of time in learning Spring structure. There were a lot of axon-specified annotations to add in my code. Although axon is a more mature framework than eventuate, there is little small-scale complete code samples. Building with axon requires a throughout understanding of CQRS architecture, and this is where I harvested the most from this project. With reading other axon project codes, along with axon official guideline, I finally got the rationale of axon logics and got the code running. I can say that axon is not a very user friendly framework as users need to implement axon interfaces. However, a developer with system level development experience may have a different perspective. Before this course, I always heard things about microservices but never had a change to work on a project. Knowing the technique of how to keep data consistency between microservices, I can transfer this skill set further into distributed systems.

VII. Conclusion

In this project, we experimented with and evaluated two existing frameworks, Axon and Eventuate, that implements data consistency for distributed transactions using Saga pattern. The two frameworks are similar in principle. Both of them implement Saga pattern and CQRS view. From a end-users point of view, however, there are identifiable tradeoffs in choosing either of the frameworks. In short, Eventuate, being the newer design of the two, is more user-friendly and provides better decoupling of components in the system. However, comparing to Axon, it is more unstable and performs worse due to its shorter development time. Axon, on the other hand, has a less-friendly API with tighter coupling of components such as handlers for commands and events. Moreover, between the two frameworks, Axon is relatively more light-weight because it only provides a skeleton structure, whereas Eventuate provides functional implementation for most parts of the system and therefore contains more scaffolding code.

References

[1] Hector Garcia-Molina and Kenneth Salem. *Sagas*, volume 16. ACM, 1987.

[2] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. *MartinFowler. com*, 25, 2014.

[3] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154. Citeseer, 1981.

[4] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. 1987.

[5] Chris Richardson. Pattern: Saga. *Microservices.io http://microservices.io/patterns/data/saga.html [last accessed on May 4, 2018]*, 2014.

[6] Axon. The axon framework. *http://www.axonframework.org/ [last accessed on May 4, 2018].*

[7] stackoverflow. Axon framework vs eventuate comparison. *https://stackoverflow.com/questions/43136291/axon-framework-vs-eventuate-comparison [last accessed on May 4, 2018].*

[8] Chris Richardson. Eventuate - official website. *Eventuate.io http://eventuate.io/ [last accessed on May 4, 2018].*

[9] Saga implementations comparison. *http://jbossts.blogspot.com/2017/12/saga-implementations-comparison.html [last accessed on May 4, 2018].*

[10] Wikipedia contributors. Microservice, 2018. [last accessed on May 3, 2018].

[11] John M Daughtry, Umer Farooq, Jeffrey Stylos, and Brad A Myers. Api usability: Chi'2009 special interest group meeting. In *CHI'09 Extended Abstracts on Human Factors in Computing Systems*, pages 2771–2774. ACM, 2009.

Appendix

A. *Code snippets of Eventuate prototype*

```java
public class ShipmentCreatedEvent implements ShipmentEvent {
  // ShipmentEvent extends from Event class in Eventuate
  // ShipmentCreatedEvent implements ShipmentEvent
  // ShipmentCreatedEvent takes String orderId as argument
  //     for changing order status
  // ShipmentCreatedEvent defines OrderState as SHIPPED in getState()
  private String orderId;
  @Override
  public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj);
  }
  @Override
  public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this);
  }
  private ShipmentCreatedEvent() { }
  public ShipmentCreatedEvent(String orderId) {
    this.orderId = orderId;
  }
  public String getOrderId() {
    return orderId;
  }
  public OrderState getState() {
    return OrderState.SHIPPED;
  }
}
public class OrderShippedEvent implements OrderEvent {
  // OrderEvent extends Event class in Eventuate
  // OrderShipped Event implements OrderEvent
  // Constructor takes String customerId as argument for
  //     customer history order query
  private String customerId;
  private OrderShippedEvent() { }
  public OrderShippedEvent(String customerId) {
    this.customerId = customerId;
  }
  @Override
  public boolean equals(Object obj) {
    return EqualsBuilder.reflectionEquals(this, obj);
  }
  @Override
  public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this);
  }
  public String getCustomerId() {
    return customerId;
  }
}
public class ShipOrderCommand implements OrderCommand {
  // OrderCommand extends from Command class in Eventuate
  // ShipOrderCommand implements OrderCommand class
  // ShipOrderCommand takes OrderState as arugument for
  //     changing order status in Order Aggregate
  private final OrderState state;
  public ShipOrderCommand(OrderState state) {
    this.state = state;
  }
  public OrderState getState() {
    return state;
  }
}
```

Listing 8: Basic building blocks in Eventuate

B. LOCs of two prototypes

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                           104            711              5           2304
Groovy                          15             64              4            285
YAML                             6             12              0            172
Bourne Shell                     1             30             65            138
Bourne Again Shell               1             20             21            123
Maven                            1              8              5             97
DOS Batch                        1             24              2             64
XML                              2             13              2             32
Dockerfile                       4              0              0             12
-------------------------------------------------------------------------------
SUM:                           135            882            104           3227
-------------------------------------------------------------------------------
```

Listing 9: LOC statistics of the prototype built using Eventuate

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                            29            316             15           1189
Maven                            7             82              0            664
YAML                             8             37              7            145
Bourne Shell                     1             29             51            145
DOS Batch                        1             31              0            112
Dockerfile                       5             22              0             16
Kotlin                           1             13              2             16
XML                              1              4              0             13
-------------------------------------------------------------------------------
SUM:                            53            534             75           2300
-------------------------------------------------------------------------------
```

Listing 10: LOC statistics of the prototype built using Axon