***Annexure A***

**Eventual Consistency in real time financial systems with Microservices architecture**

### BITS ZG628T: Dissertation

by

**Tushar Dilip Phadke**

**2016HT12516**

# Dissertation work carried out at

## Amdocs Development Centre India LLP, Pune

****

**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE**

**PILANI (RAJASTHAN)**

April 2018

***Annexure B***

**Eventual Consistency in real time financial systems with Microservices architecture**

### BITS ZG628T: Dissertation

by

Tushar Dilip Phadke

2016HT12516

# Dissertation work carried out at

## Amdocs Development Centre India LLP, PUNE

Submitted in partial fulfillment of M.Tech. Software Systems degree programme

Under the Supervision of

Anurag Srivastava, IBM India Pvt Ltd, PUNE

****

**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE**

**PILANI (RAJASTHAN)**

April, 2018

**Annexure C:**

#### CERTIFICATE

This is to certify that the Dissertation entitled “Eventual Consistency in real time financial systems with Microservices architecture” and submitted by **Tushar Dilip Phadke** having ID-No. **2016HT12516** for the partial fulfillment of the requirements of M.Tech. Software Systems degree of BITS, embodies the bonafide work done by him under my supervision.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Signature of the Supervisor

Place : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Name, Designation & Organization &Location

**Abstract:**

One challenge in highly distributes systems is transactional consistency. In the case of financial transactions this has a direct revenue impact. Therefore, it is necessary to employ resilience strategies which address the financial risk. For the purpose of this dissertation, we'll look at a communication service provider (CSP). The CSP is a service provider that transports information electronically—for example, a telecommunications service provider. The term encompasses public and private companies in the telecom (landline and wireless), Internet, cable, satellite. The CSP industry is gradually employing micro services to replace large monolithic IT systems, as a strategy for a more agile and more robust infrastructure.

The CSP uses an internal payment ('micro-') service, which encapsulates the external acquirer interaction. The other internal ('micro-') systems like billing, ordering uses the payment service. The eventual consistency is the strategy of choice to sync up internal systems.

The dissertation will examine API strategies for eventual consistency between internal services and their respective requirements on the micro service architecture in order to minimize financial impact on CSP.

**Acknowledgements**

The satisfaction and euphoria that accompanies the successful completion of any task would be incomplete without mentioning the people who made it possible, because success is the epitome of hard work, perseverance, undeterred missionary zeal, steadfast determination and most of all “ENCOURAGING GUIDANCE”.

I express my gratitude to my supervisor Mr. Anurag Srivastava for providing me a means of attaining my most cherished goals. I record my heart full of thanks and gratitude to my additional examiner Mr. Tejas Jog and BITS WILP for providing me an opportunity to carry this project, along with purposeful guidance and moral support extended to me throughout the duration of the project work.

Also I expresses my gratitude to Mr. Jens Emmerich, Chief Architect, Amdocs Development Centre from Mobile Financial Solutions division for providing me a valuable input and feedback in understanding financial domain and Amdocs business model which caters to Communication service providers.

I would like to express my love and gratitude to my beloved family, my friends, my Amdocs Development Centre India LLP, team members, for their understanding & motivation throughout the duration of this project.

**Mr. Tushar Dilip Phadke**

**List of Abbreviations and Acronyms**

|  |  |
| --- | --- |
| **CSP** | **Communication Service Provider** |
| **MFS** | **Mobile Financial Solutions** |
| **PoC** | **Proof-Of-Concept** |
| **ACH** | **Automated Clearing House** |
| **TSP** | **Telecom service provider** |
| **XA** | **Distributed Transaction based on X/Open group** |
| **AP** | **Application Program** |
| **RM** | **Resource manager involved in XA** |
| **TM** | **Transaction Manager involved in XA** |
| **2 PC** | **Two Phase Commit** |
| **CI** | **Continuous Integration** |
| **CD** | **Continuous Delivery** |
| **PCI DSS** | **Payment Card Industry Data Security Standard** |

**TABLE OF CONTENT**

[Chapter 1: Introduction 1](#_Toc509004309)

[1.1 Background 1](#_Toc509004310)

[1.2 Objective 1](#_Toc509004311)

[1.3 Scope of work 1](#_Toc509004312)

[1.4 Plan of work 2](#_Toc509004313)

[1.5 Estimated Date & Progress 2](#_Toc509004314)

[Chapter 2: Overview 3](#_Toc509004315)

[2.1 Electronic Payment Systems 3](#_Toc509004316)

[2.2 Order and payment systems in a CSP environment 5](#_Toc509004317)

[2.3 Microservices Architecture 7](#_Toc509004318)

[2.3.1 Principal of Microservices 8](#_Toc509004319)

[2.3.1 Decomposing Microservices in TSP 9](#_Toc509004320)

[2.4 Eventual consistency problem with Microservices 10](#_Toc509004321)

[Chapter 3: Approaches for Eventual Consistency in Microservices Architecture 12](#_Toc509004322)

[3.1 Using Two Phase Distributed Transactions (XA) 12](#_Toc509004323)

[3.2 Using Service Orchestrator Pattern 14](#_Toc509004324)

[3.3 Using Service choreography Pattern 15](#_Toc509004325)

[3.4 Using Mix strategy – Hybrid Pattern 16](#_Toc509004326)

[Chapter 4: Proof-Of-Concept documentation 19](#_Toc509004327)

[4.1 Objective 19](#_Toc509004328)

[4.2 System design details 19](#_Toc509004329)

[4.3 Test Cases & Result 24](#_Toc509004330)

[4.3.1 Test Cases 24](#_Toc509004331)

[4.3.2 Architectural Observations and Actions Taken 24](#_Toc509004332)

[Summary 26](#_Toc509004333)

[Conclusions and Recommendations 27](#_Toc509004334)

[Directions for Future Work 28](#_Toc509004335)

[Bibliography 29](#_Toc509004336)

**LIST OF FIGURES**

[Figure 1: Payment Authorization through Acquirer (Merchant’s bank) 3](#_Toc509004284)

[Figure 2: Payment clearing through network clearing system 4](#_Toc509004285)

[Figure 3: Payment Settlement 4](#_Toc509004286)

[Figure 4: Use case I - Checkout and Payment 6](#_Toc509004287)

[Figure 5: Use Case II - Payment Settlement 7](#_Toc509004288)

[Figure 6: Principles of Microservices Architecture 9](#_Toc509004289)

[Figure 7: Domain Model 9](#_Toc509004290)

[Figure 8: Microservices at Telecommunication Service Provider 10](#_Toc509004291)

[Figure 9: Distributed Transactions at TSP 12](#_Toc509004292)

[Figure 10: Two Phase Commit in XA 13](#_Toc509004293)

[Figure 11: Order Checkout through Orchestrator 14](#_Toc509004294)

[Figure 12: Service Choreography for order checkout 15](#_Toc509004295)

[Figure 13: Hybrid Architecture 17](#_Toc509004296)

[Figure 14: Hybrid Architecture – Process flow 18](#_Toc509004297)

[Figure 15: Class diagram - Payment microservice 20](#_Toc509004298)

[Figure 16: Business Object for PoC 21](#_Toc509004299)

[Figure 17: Deployment Diagram 21](#_Toc509004300)

[Figure 18: Order Checkout sequence diagram 22](#_Toc509004301)

[Figure 19: Event handling sequence diagram 23](#_Toc509004302)

**LIST OF TABLES**

[Table 1: Plan of Work 2](#_Toc509004275)

[Table 2: Dissertation Phases and Progress 2](#_Toc509004276)

[Table 3: Test Cases 24](#_Toc509004277)

# Chapter 1: Introduction

## Background

One challenge in highly distributes systems is transactional consistency. In the case of financial transactions this has a direct revenue impact. Therefore, it is necessary to employ resilience strategies which address the financial risk. For the purpose of this dissertation, we'll look at a communication service provider (CSP). The CSP is a service provider that transports information electronically—for example, a telecommunications service provider. The term encompasses public and private companies in the telecom (landline and wireless), Internet, cable, satellite. The CSP industry is gradually employing micro services to replace large monolithic IT systems, as a strategy for a more agile and more robust infrastructure.

Let’s look at problem statement for dissertation. The CSP uses an internal payment ('micro-') service, which encapsulates the external acquirer interaction. The other internal ('micro-') systems like billing, ordering uses the payment service. The eventual consistency is the strategy of choice to sync up internal systems. The dissertation will examine API strategies for eventual consistency between internal services and their respective requirements on the micro service architecture in order to minimize financial impact on CSP.

## Objective

The objective of this dissertation will examine API strategies for eventual consistency between internal services and their respective requirements on the micro service architecture in order to minimize financial impact on CSP and demonstrate the approach with Proof-Of-Concept.

## Scope of work

Scope of work comprises of developing an API strategy for eventual consistency between internal Microservices in real time financial systems developed using Microservices architecture to reduce the financial impact on CSP.

Proof-Of-Concept for this study will be developed according to plan of work. The details of Microservices built as part this PoC will be

* **Order Microservice**
* **Payment Microservice**
* **Acquirer System** - This service will mock a system which deals with card network and payment settlement. Payment Microservice will interact with this system

## 1.4 Plan of work

|  |  |
| --- | --- |
| Analysis | Study the Microservices architecture and Eventual consistency in micro-services |
| Design | Design an architecture for handling eventual consistency in Microservices |
| Coding | Build a Proof-Of-Concept (PoC) for demonstrate approach designed for handling eventual consistency in Microservices |
| Testing | Eventual Consistency will be tested in developed PoC |
| Summarize | Summarize and document results of PoC |

Table 1: Plan of Work

## 1.5 Estimated Date & Progress

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SR No.** | **Phase** | **Description** | **Estimated Finish Date** | **Progress** |
| 1 | Analysis | Study the Microservices architecture and Eventual consistency in micro-services | 14/02/2018 | Completed |
| 2 | Design | Design an architecture for handling eventual consistency in Microservices | 24/02/2018 | Completed |
| 3 | Coding | Build a Proof-Of-Concept (PoC) for demonstrate approach designed for handling eventual consistency in Microservices | 10/03/2018 | Completed |
| 4 | Testing | Eventual Consistency will be tested in developed PoC | 12/03/2018 | Completed |
| 5 | Summarize | Summarize and document results of PoC | 16/03/2018 | In Progress |
| 6 | Documentation | Document dissertation report | 20/03/2018 | In progress |

Table 2: Dissertation Phases and Progress

# Chapter 2: Overview

## 2.1 Electronic Payment Systems

The following section describes the payment domain and associated payment processing steps in detailed. [1]

1. ***Authorization*** basically means permitting a transaction to go through. In most cases it is the cardholder’s bank (Issuer) that is responsible for authorization (for example, to verify that its customer has sufficient funds in their debit card account), though the network may perform some pre-validation (checking for fraud, or that correct information has been entered) before passing it through. This is the first message that is “switched” by the network to the cardholder’s bank. In following diagram, M-Bank indicates the Acquirer which routes the authorization call to Issuer through card network.

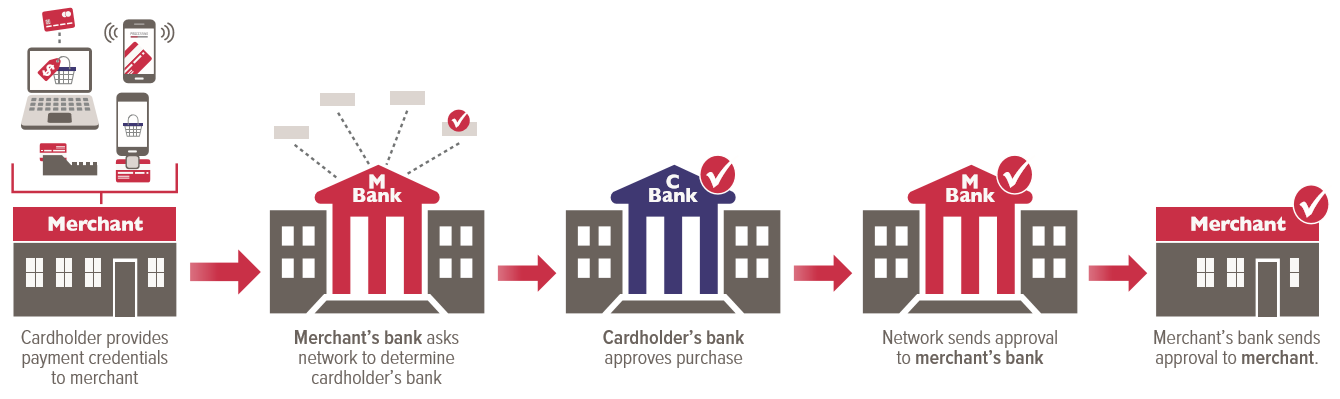


Figure 1: Payment Authorization through Acquirer (Merchant’s bank)

1. Once the transaction is authorized, the network is responsible to help ***clear*** the transaction: in other words, reconciling the amounts, in addition to any fees, sending information to both parties detailing the amount to be credited or debited, and referencing the consumer or merchant account in question

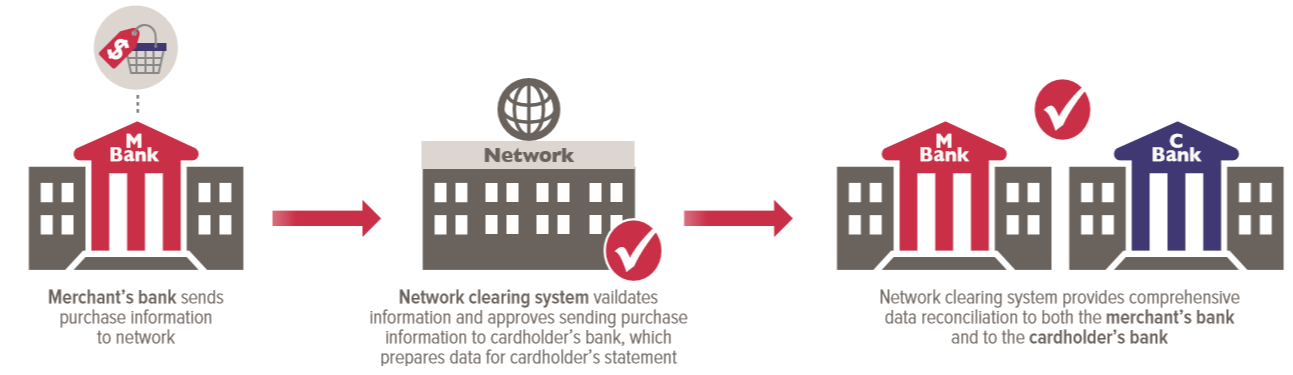


Figure 2: Payment clearing through network clearing system

1. ***Settlement*** occurs when funds are transferred from one bank to another. In card systems this process usually occurs at the end of every day (not in real-time) and represents the net position of all participating banks for all transactions that occurred during that period. It is important to highlight that payment networks do not handle the actual fund transfer themselves, but provide the instructions to do so; payment networks usually designate a separate “settlement” bank for members to move funds in and out of.

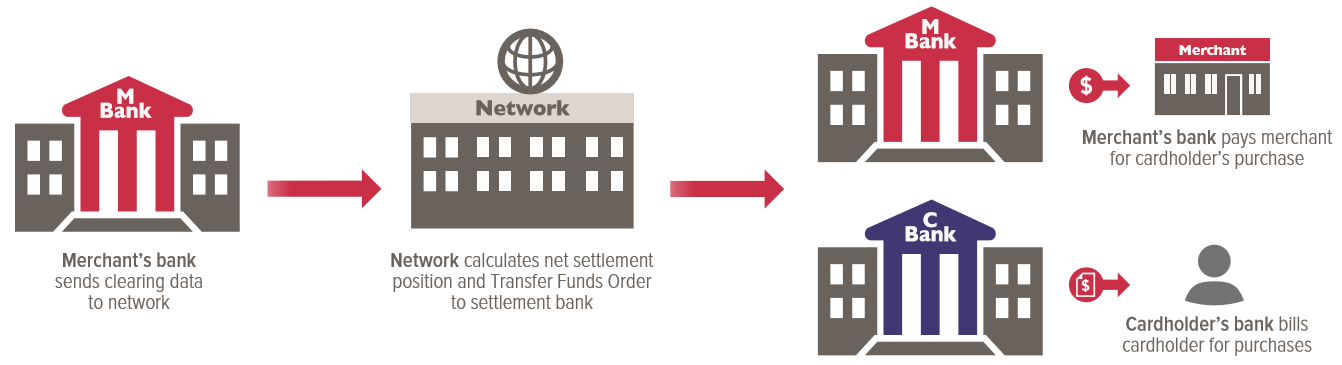


Figure 3: Payment Settlement

## 2.2 Order and payment systems in a CSP environment

The Communication Service Provider is a service provider that transports information electronically—for example, a telecommunications service provider. The term encompasses public and private companies in the telecom (landline and wireless), Internet, cable, satellite. Let’s take an example of a telecom service provider (TSP) E.g. T-Mobile or Vodafone. A typical TSP’s have an online store (Mobile or web application) called channel which is used by consumers to purchase new mobile handsets, top up mobile plans or data plans, pay monthly bills. Let’s say consumer uses web application for ordering a new mobile handset. The web application records this order in order management system before interacting with a payment system for handling payment against ordered goods. The payment system internally interact with an acquirer (Outside TSP system boundary) for authorizing payments. If authorization is successful then same is communicated to channel. Then channel sends request for capture of payment. At this point system records a payment without interacting with Acquirer and sends confirmation to channel. Then order management system is also notified about state of payment and it confirms the order. Also billing system is notified about successful payment. At end of day payment system collects all payments captured during a day and sends a file containing all payments to acquirer for a settlement. The acquirer work with payment networks and associated parties for settlement of payment and sends back a file containing status of each payment. The CSP uses this file to update internal system about status of payment. If payment is failed due to some reason, the billing and order system has to be notified about failure so that they can take corrective actions. These flow is explained in following process flow diagram. The “Use case I” indicates the authorization flow and “User case II” indicates the settlement flow.

Use Case I:

The use case depicts the real time flow in the typical real time financial system. The use case starts when user checkouts the shopping cart by entering credit card details. The customer care application is a web application which initiates the transaction. The use case I is depicted in following diagram.

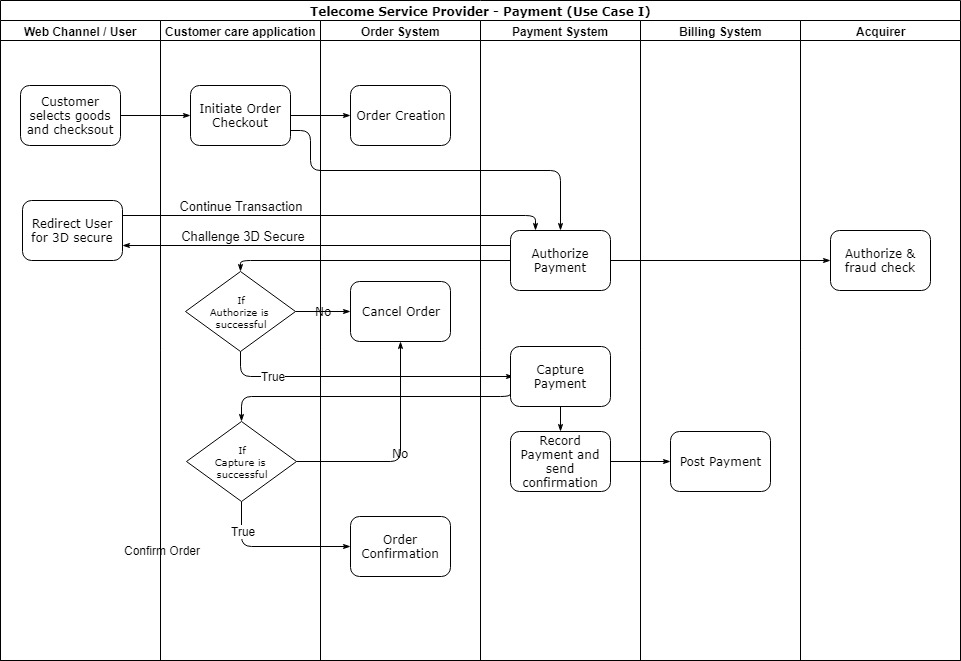


Figure 4: Use case I - Checkout and Payment

Use Case II:

The use case II depicts the settlement flow in which TSP sends a file containing all payments captured during a day to the acquirer for settlement. Once acquirer finishes his process it sends back a file containing status of payment. The payment system acts on failed transactions as depicted in following diagram.

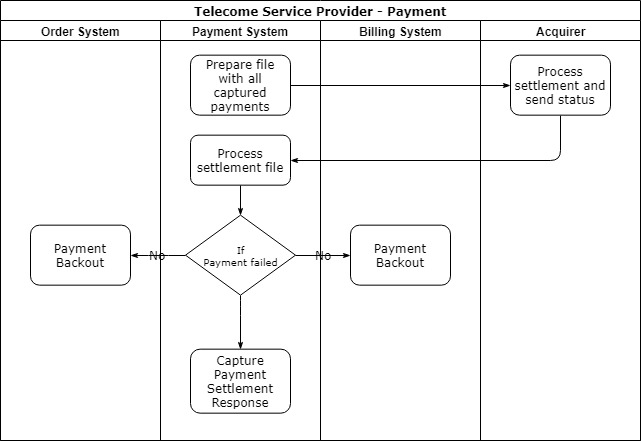


Figure 5: Use Case II - Payment Settlement

## 2.3 Microservices Architecture

Microservices is a major architectural shift and is fundamentally different way of building software. The Microservice architecture is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment mechanism. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. The Microservices architecture should adhere to following principles.

### 2.3.1 Principal of Microservices

**Hide Implementation details (encapsulation):** Services should hide how they work and do a single thing and do it well. The encapsulation is based on the business function of the service that it encapsulates; the functional requirements.

**Culture of Automation**: Service should use automation which keeps management and operational support efficient. Employing large number of small services can become mess if you do not embrace automation.

**Modelled around business domain**: Similar to encapsulation, each service should align to your bounded context within a domain model for this to make sense in the domain and in the large service architecture.

**Decentralize all things**: Every service should be decentralize and autonomous so that each service can operate on its own schedule and priorities. Each service, if independent, can update and deployed as required. Service versioning and backward compatibility is crucial because system will be dependent on other systems; we can implement features and prioritize services separately.

**Deploy independently**: Services should be independent of each other but to enable this similar to decentralization, we must implement versioning and backward compatibility. Being able to deploy a service separately is a main goal.

**Isolate Failure (fail-safe)**: Enterprise systems typically needs to be smart, in that they fail in way that doesn’t break things in unpredictable manner. Anticipate failures in everything, along with validation of input and validation of data send downstream. Also if one service fails then it should not impact the behavior of other services.

**Highly Observable**: While having many services enable scalability and simplicity of interaction, to get a bigger picture, each service should be observable through logging and correlation of messages. A common pattern here is to aggregate data for all services and collect and analyze for operational support.

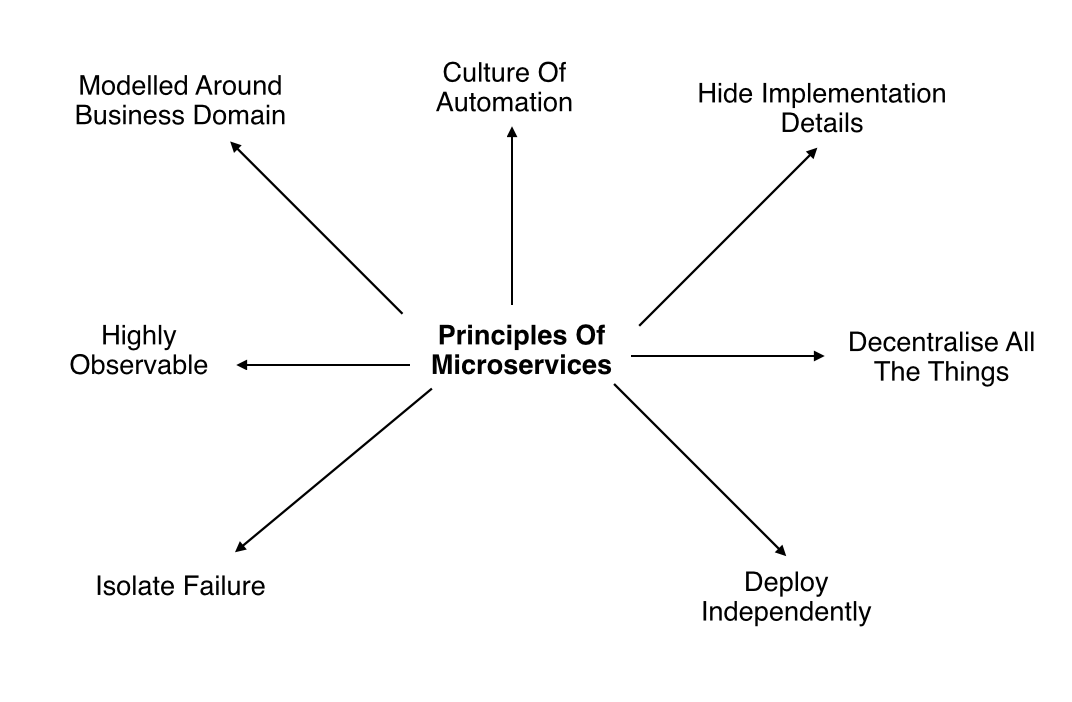


Figure 6: Principles of Microservices Architecture

### 2.3.1 Decomposing Microservices in TSP

The Microservices uses services as unit of modularity. Each service corresponds to a business capability. A Microservices based system in TSP (our uses cases) consists of services like Order, Payment, Billing. Each service has impermeable boundary that is difficult to violate. As a result the modularity of the application is much easier to preserve over time. The domain model for our problem statement is depicted in following diagram.

The domain model can be decomposed as following.

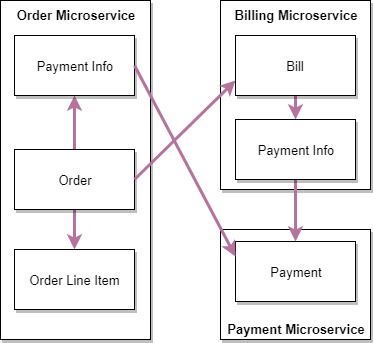


Figure 7: Domain Model

Then typical architecture depicting the Microservices in our problem statement is depicted in following diagram with its own data store system.

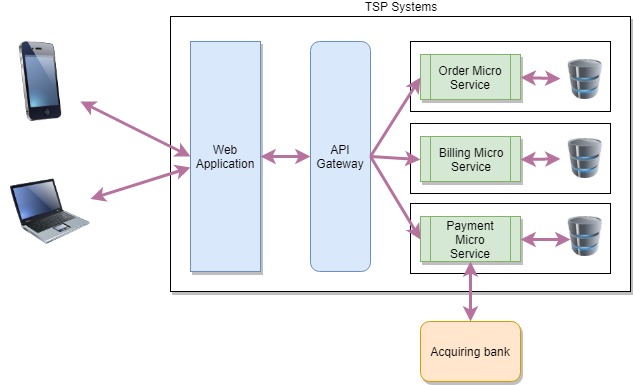


Figure 8: Microservices at Telecommunication Service Provider

## 2.4 Eventual consistency problem with Microservices

In a monolithic application, it is easy to deal with transactions because all of the components are part of the monolith. When you move to a Microservices architecture that is distributed, you now must potentially deal with transactions that are spread across multiple services. In a Microservices architecture, the preference is for CAP (Consistency, Availability, Partition tolerance) over ACID (Atomicity, Consistency, Isolation, Durability). Any distributed system is not safe from network failures, thus network partitioning generally has to be tolerated. In the presence of a partition, one is then left with two options: consistency or availability. If system prefers Availability over Consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning. Microservices are designed the BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency) philosophy (choosing availability over consistency). Also sharing a database does violate some of the principles of a Microservices-based architecture.

In our problem statement by any chance a payment is failed after authorize call, then Order, Billing system has to be notified about state of payment. So each system can take corrective action (eventual consistency). Also in “Use Case II” while processing payment settlement file if payment system encounters any payment failure then other systems has to be notified about failures in order they can take corrective action (eventual consistency). If eventual consistency is not maintained between these systems then CSP may suffer from financial impact like shipping goods without payment, inconsistent billing/revenue’.

# Chapter 3: Approaches for Eventual Consistency in Microservices Architecture

This section will be updated once design is complete. The following sub sections describes the strategies which can be applied to handle eventual consistency. The design for the handling eventual consistency in the problem statement is in progress and following sections will be completed once design is completed.

## 3.1 Using Two Phase Distributed Transactions (XA)

The Distributed Transaction (XA) is a protocol developed by X/Open (Open Systems) organization which is supported by most of world’s largest information system suppliers, user organizations and software companies. The X/Open distributed transaction processing encompasses three components namely, An **A**pplication Program (AP), Resource Managers (RM), and Transaction Manager (TM). An application program defines the transaction boundaries and specifies actions that constitutes transaction. The RM provides access to shared resources. The TM is intended to assign unique identifier for each transaction, monitor’s transaction’s progress and takes responsibility for transaction completion and for failure recovery.

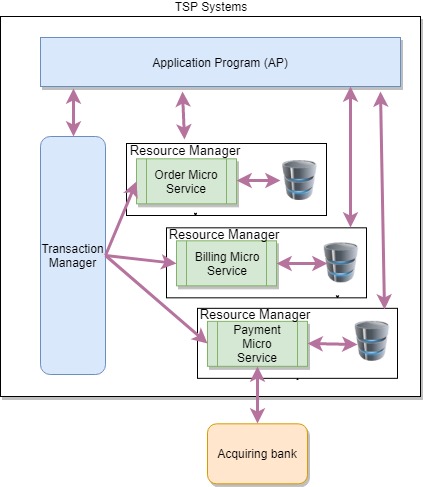


Figure 9: Distributed Transactions at TSP

In context of our problem statement, the web application can become a Application Program (AP) which controls the boundaries of transaction and initiates commit/rollback depending on actions. The TM will identifies and coordinate the transactions with each Microservice. Then each Microservice acts as Resource Manager which provides access to individual databases. The 2 phase commit (2 PC) protocol defines the contract between RM and TM to either commit transaction or rollback it. The working of 2 PC is illustrated in following diagram.

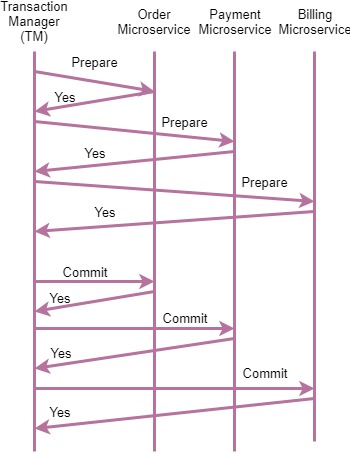


Figure 10: Two Phase Commit in XA

In above case, when AP initiates a transaction commit in coordination with TM, the TM starts sending “prepare” message to each RM (Microservices) the each RM prepare itself for transaction commit and sends success back to TM. Once TM receives success from all RM, it sends “commit” message to each RM to commit the transaction. Once all RM notifies the TM about success, the transaction is considered as successfully committed.

**Benefits**

* XA with 2 PC always give highest confidence and protection against failures where multiple diverse resources are being used
* Most major software vendor’s provides support for XA with 2 PC

**Drawbacks/Tradeoffs**

* Expensive because of additional I/O prescribed by the protocol and requires special-purpose platforms.
* Very complex and very hard to maintains
* Hard to debug and trace transactions to resolve consistency issues
* In above example, under extreme conditions, if a billing service might fail to commit after payment system commits. This may cause data inconsistency across the Microservices.
* The XA with 2 PC is not useful in case of “Use case II” mentioned in section 2.1.

## 3.2 Using Service Orchestrator Pattern

Orchestration is the traditional way of handling interactions between different services. With orchestration, there is typically one controller that acts as the “orchestrator” of the overall service interactions. This typically follows a request/response type pattern. In our problem statement Microservices using Orchestrator can be illustrated as follows. The Customer service becomes a orchestrator to orchestrate the checkout.

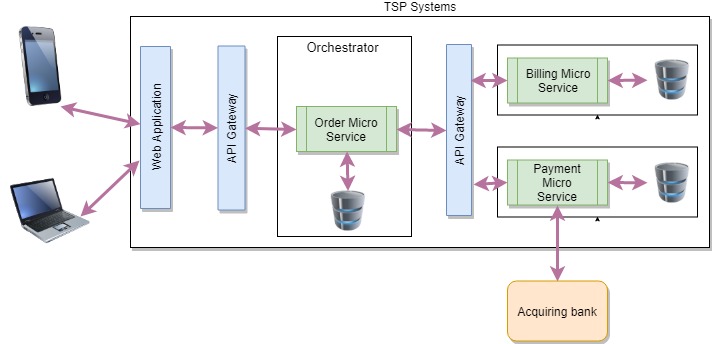


Figure 11: Order Checkout through Orchestrator

**Benefits**

* Provides a better way to control the flow in application when there is synchronous processing
* Easy to manage exceptional flows. E.g. In our problem statement if updating billing system fails, then orchestrator can update Order Microservice to cancel order and update Payment Microservice to cancel authorization and refund payment if payment was successful.

**Drawbacks/Tradeoffs**

* Couples the services together creating dependencies
* If central shared instance of the orchestrator is used for all requests, then the orchestrator is a single point of failure
* It violates couple of Microservices architecture principles like Isolate Failure and Deploy independently as Orchestrator will depend of services

## 3.3 Using Service choreography Pattern

In a choreography pattern, services doesn’t talk to another service in order to instruct an action. Instead each service is observing its environment and acts on events. In real world choreography pattern, Services are connected to a message bus and subscribe channels they are interested in. Once an event series occurs that matters to the service, the service performs the appropriate action. Now it is easy to add new services to the architecture; you simply have to connect them to the message bus.

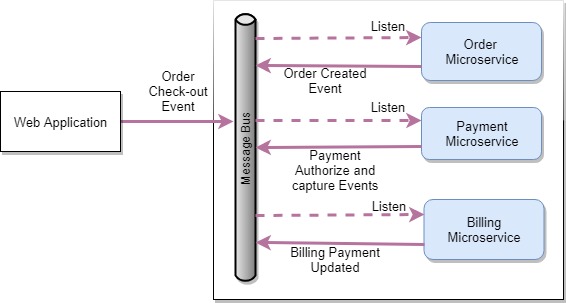


Figure 12: Service Choreography for order checkout

The above architecture remove the direct dependencies between services. Services use an event stream for asynchronous communication of events. Multiple services can consume the same events, do some processing, and then produce their own events back into the event stream, all at the same time. The asynchronous nature of architecture removes the blocking or waiting.

**Benefits**

* Enable faster end to end processing without any blocking
* Service can start consuming new events without direct dependency between services
* Aligns well with continuous delivery as there is no direct dependency between services
* Control is distributed and no single point of failure

**Drawbacks/Tradeoffs**

* Async programming is often significant mind shift for developer and code is hard to read by just looking at it. No information available about producer of the events
* Each service need to maintain casual order between messages (per customer) to complete the business process as message bus does not guarantee about sequence of message delivery
* In some stringent requirements from finance industry to reduce a fraud, an interactive (Synchronous) response are required from customer which is not achievable in async architecture. E.g. In case 3D secure requirement, end user is challenged by issuer bank for entering One Time Password where customer is navigated to issuer network and authorize has to be suspended till 3D Secure approval is processed successfully

## 3.4 Using Mix strategy – Hybrid Pattern

A one-size-fits-all approach doesn’t work well in software architecture. This is true with choreography and orchestration patterns. In our problem statement we have a mix of synchronous and asynchronous processing; either synchronous blocks of asynchronous activities or vice versa. In these situations, a hybrid patterns that adds value and solves end to end problem of eventual consistency and achieve business objectives. The following diagram depicts the architecture where we have to use Orchestration patter along with Choreography pattern to business goals.

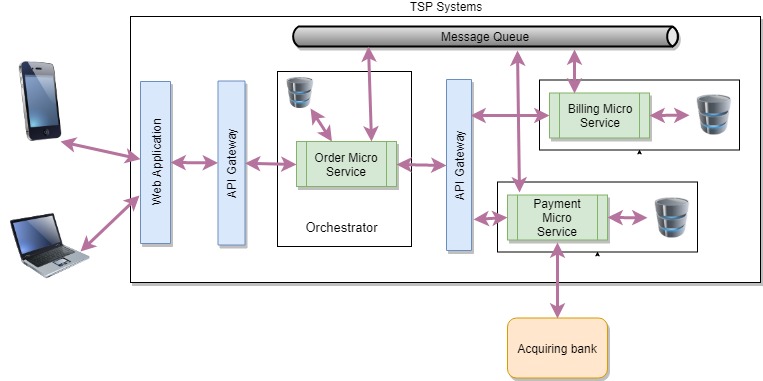


Figure 13: Hybrid Architecture

**Benefits**

* Along with the benefits of both architecture pattern this hybrid approach provides the better eventual consistency.
* Easy to manage exceptional flows. E.g. In our problem statement if updating billing system fails, then orchestrator can update “Order” microservice to cancel order and update “Payment” microservice to cancel authorization and refund payment.
* The message queue decouples the direct dependency on mircoservices for maintaining eventual consistency. E.g. If payment settlement (from acquirer) is failed for a captured payment then payment microservice can notify all subscribers about failed transaction then each microservice can take a corrective action.
* In future any new microservice can subscribe to queue for listening to events and act accordingly without any changes in publisher

**Drawbacks/Tradeoffs**

* Each service need to maintain casual order between messages (per customer) to complete the business process as message bus does not guarantee about sequence of message delivery
* The “Order” microservice will have dependency on “Billing” and “Payment” microservice. This will break the microservice principal “deploy independently”.

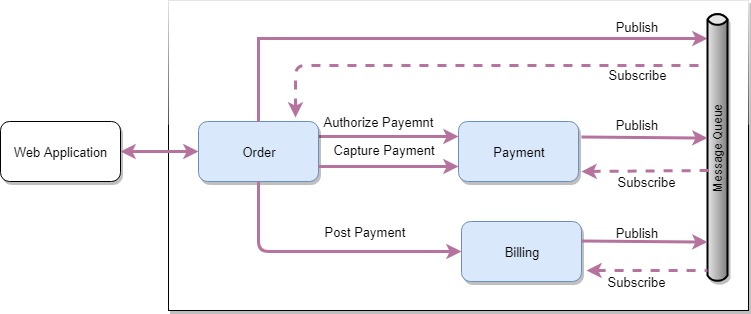


Figure 14: Hybrid Architecture – Process flow

The above diagram depicts the information flow in hybrid architecture. This is a final architecture which is used for building Proof-Of-Concept (PoC). In this architecture, the web channel (customer care application) will trigger the order checkout flow by calling “Order” microservice. The “Order” microservice will be used as orchestrator for calling other Microservices depending on business use case. The typical process flow is described in above diagram. Each microservice will be a publisher as well as subscriber of the message queue. If any microservice failed to process request, then it will publish a message into the queue along with context information. Then subscribers of the queue can take corrective actions using context information available in event received from queue. The “Payment” microservice will be responsible for processing settlement file from acquirer. The “Payment” microservice will publish message into the queue for failed payments, so other microservices can take corrective action. If payment is rejected then associated order also has to be rejected and billing has to be updated about rejected payment.

# Chapter 4: Proof-Of-Concept documentation

This chapter describes the background, objective design and result of Proof-Of-Concept carried to check eventual consistency in microservice architecture. This POC will demonstrate the use of hybrid architecture in achieving eventual consistency across multiple microservices.

## 4.1 Objective

The primary objective of this PoC is to demonstrate the feasibility of hybrid architecture to achieve eventual consistency across microservices as described in previous chapter. This PoC will evaluate two uses cases mentioned in beginning of document. The one use case targets real time use case of financial transaction and another will target offline use case of payment settlement. Also this PoC will evaluate eventual consistency in case of exception handling.

**Out of scope**

* The Payment Card Industry Data Security Standard (PCI DSS) will be out of scope which has direct impact on architecture.
* Authentication & Authorization for microservices
* Microservice orchestration using tools like Docker swamp, Kubernetes

## 4.2 System design details

The PoC is developed using Spring Boot technology and all Microservices are REST services having standard operation like POST, GET, PUT. The “Order” microservice is used as orchestrator for order checkout. The following diagram depicts the “Payment” microservice class diagram. Other microservices are developed using similar design. The “PaymentService” is a RestController which has methods like POST/PUT/GET which are called by “Order” microservice. The “PaymentService” uses “Repository” for interaction with database. The “EventProducer” is used by service for publishing event on Kafka queue. The “KafkaEventListener” is a listener used by application to listen to Kafka queue. When any new event is published on the queue, the Spring Kafka framework calls listen method on “KafkaEventListener”. This listener calls “EventHandler” for delegating event handling based on event source. E.g. “OrderEventHandler” is called when event source is “Order”. These event handlers plays major role in maintaining eventual consistency in service and call necessary third party services. E.g. if system receives “ORDER\_CANCELLED” then the system calls “AcquirerAdapter” for cancelling authorization on card to unblock amount. The event handler also uses repository for storing event in database and maintain action taken on event. The adapters are used by service for interacting other services. It uses load balanced “RestTesmplate” for calling remote REST services. The load balanced “RestTemplate” uses Eureka server & Ribbon for discovery and load balancing service remote calls.

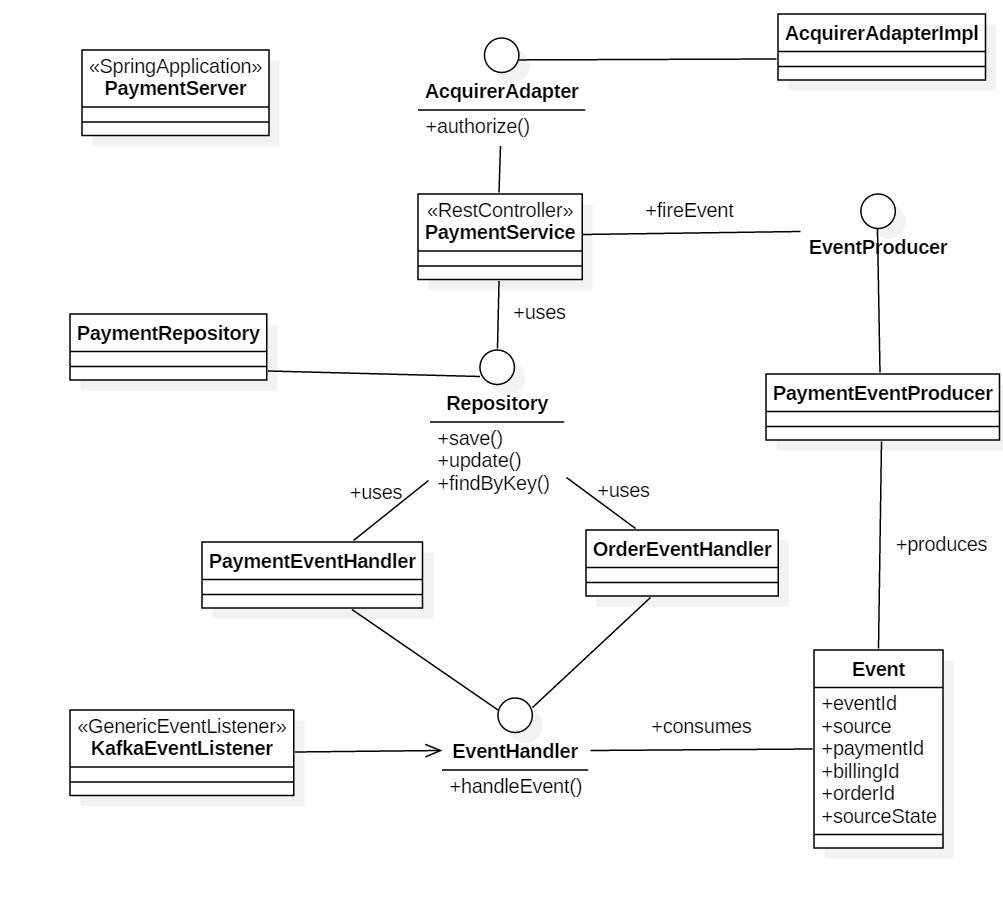


Figure 15: Class diagram - Payment microservice

Each service saves the business object pertaining to transaction as a local entity. These business objects has depicted in following diagram. The state of each business object is maintained (eventual consistency) using hybrid model.

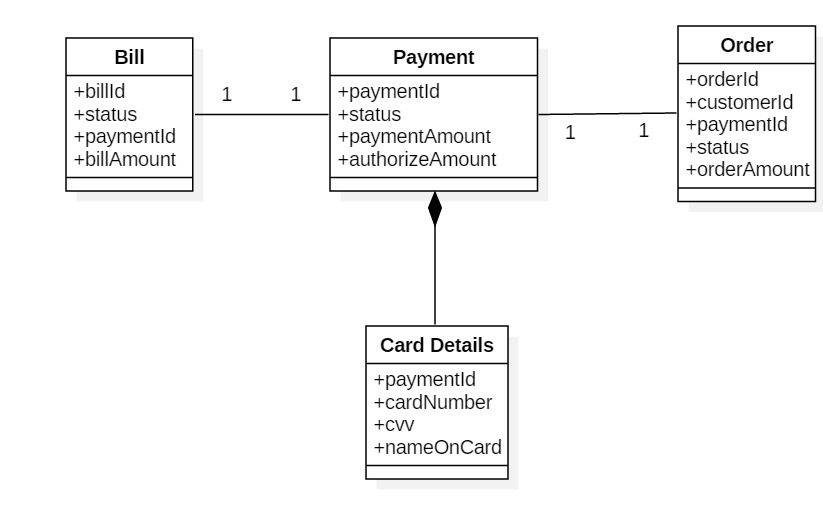


Figure 16: Business Object for PoC

The following image depicts the deployment diagram for the PoC.In total 5, boot applications are deployed and started namely, “Payment”, “Order”, “Billing”, “Acquirer” microservices and “EurekaServer” which is a special type of Spring boot application which is sued as registry for microservices. This EurekaServer is also used by Ribbon for load balancing service calls.

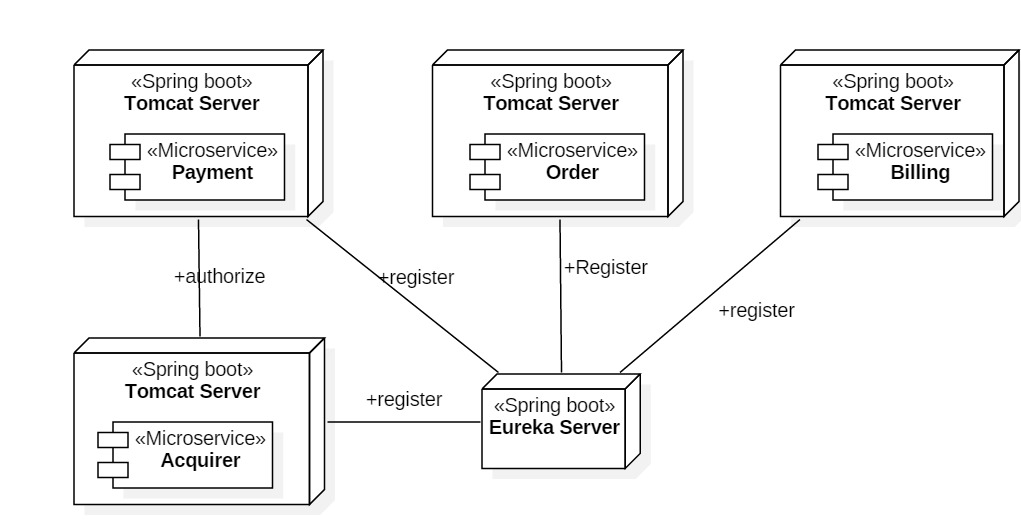


Figure 17: Deployment Diagram

This service accept payment information along with order information. Then “Order” microservice calls “Billing” microservice to initialize bill. Then it calls “Payment” microservice for authorize and capture of payment. If payment capture is successful then it calls bill microservice to confirm payment and sends back success status to caller. The following depicts the sequence diagram for order checkout. It also highlights the places where event is triggered on each state change.

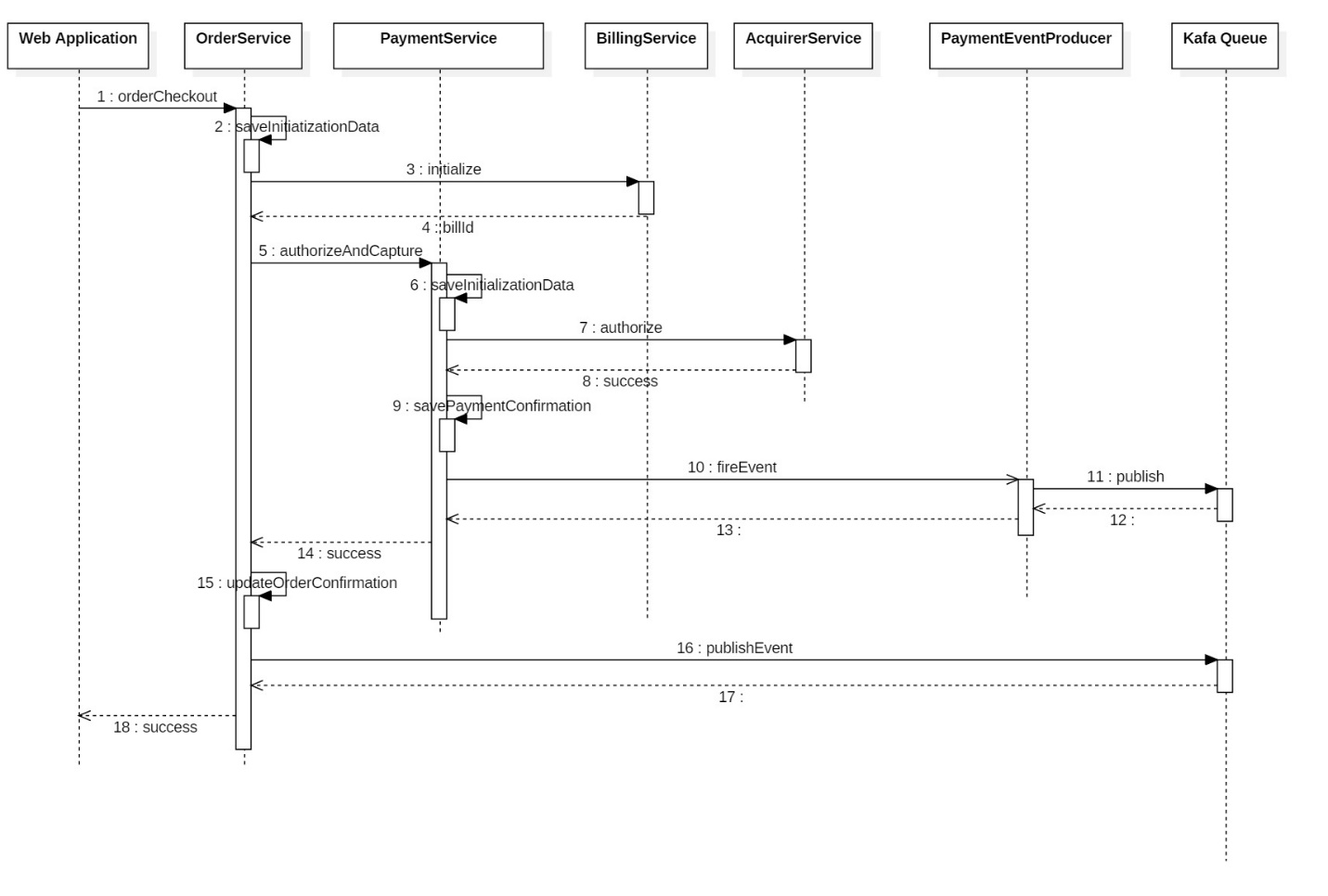


Figure 18: Order Checkout sequence diagram

The following sequence diagram depicts a typical event handling flow. The “onMessage” method from KafkaEventListener is called when any new event is available on Kafka queue. This method first deserializes the event and checks the source. If event source is same service then it ignores event and just saves event in local database. If source is other service then it calls respective event handler. E.g. if event source is “Order” service then it calls “OrderEventHandler” for handling event. Let’s say event is “Order Cancelled” then “OrderEventHander” cancels payment authorization in acquirer and updates local payment status and order status. Then it fires the “Payment Cancelled” event. Then “OrderEventHandler” saves event in local database with state as “event handled” and it finishes event handling.

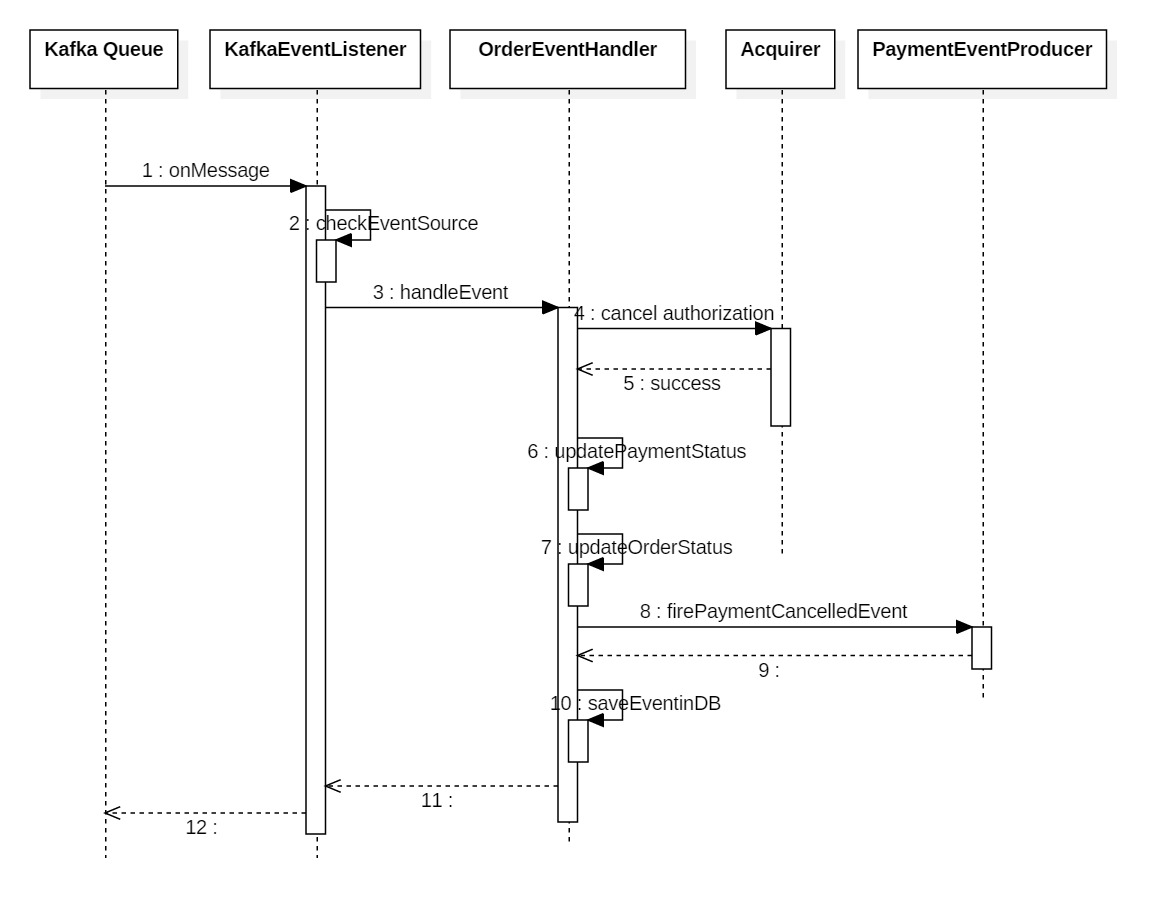


Figure 19: Event handling sequence diagram

The above mentioned PoC code is available on GIT Hub along with readme file which explains steps to start Spring boot applications (microservices) along with configuring Kafka. It also has automated tests written in “FlowTest” component which can be executed once all servers are up and running. The details about code are as follows:

**PoC Code**: <https://github.com/pkytech/BITS/tree/master/dissertation>.

**PoC README**: <https://github.com/pkytech/BITS/blob/master/README.md>

## 4.3 Test Cases & Result

### 4.3.1 Test Cases

|  |  |  |  |
| --- | --- | --- | --- |
| TC ID | Test Case | Expectation | Result |
| 1 | If complete transaction of order creation, payment authorize and capture along with acquirer interaction, billing update is successful then | All microservice should have consistent state as described below  Order state should be in ORDER\_CONFIMED state,  Payment should be in PAYMENT\_SUCCESSFUL state | Passed |
| 2 | Acquirer interaction failed due to network error or validation | All microservice should have consistent state as described below  Order state should be in ORDER\_CANCELLED state,  Payment should be in  PAYMENT\_AUTHORIZATION\_FAILED state | Passed |
| 3 | Acquirer rejects payment during offline payment settlement | All microservice should have consistent state as described below  Order state should be in ORDER\_REJECTED state  Payment should be  PAYMENT\_REJECTED state | Passed |
| 4 | If acquirer sends confirmation for the payment during offline payment settlement | All microservice should have consistent state as described below  Order state should be in ORDER\_CONFIMED state,  Payment should be in PAYMENT\_SUCCESSFUL state | Passed |

Table 3: Test Cases

### 4.3.2 Architectural Observations and Actions Taken

* The two methods, “authorize” and “capture” were merged into single operation to avoid additional eventual consistency handling in case user abandons checkout after “authorize”. If user abandons checkout after “authorization” then authorization with acquirer also need to be cancelled otherwise that amount will be blocked on user’s card
* Even if we use orchestrator for real time payment processing, each microservice should publish state into the queue. This eases out the additional exception handling in orchestrator layer and each service can take corrective action. This also reduces the response time.
* Even if we have a queue for notifying each service about state, if event handler fails to handle event due to system error then system should log the failure and should have a process to correct failed events. The PoC has persisted all events with action taken. If event handler fails to process event then it is stored in DB as ACTION\_FAILED state.
* All event should have context information about event like originator, payment identifier, order identifier, bill identifier, customer identifier and state of originator
* In case acquirer identifies the authorize request as a candidate for 3D secure authorization, the payment has to be suspended till user is authenticated. Then each service is called again with 3D secure authorization details. So each operation should be idempotent.

# Summary

The hybrid architecture can be adopted in developing Microservices real time financial system for Communication Service Providers (CSP). Along with adopting to new architecture some business processes has to be altered for maintaining eventual consistency. In the initial business flow, the “authorization” and “capture” call were different business flows. This triggers to have more exception handling for maintaining eventual consistency between “authorize” and “capture”. But if we merge “authorize” and “ca

pture” into single call then it reduces the effort for additional exception handling for maintaining eventual consistency. Also each writable microservice should have idempotent check for avoiding duplicate transactions in the system which causes consistency issues.

# Conclusions and Recommendations

After coming

# Directions for Future Work

* The Microservices developed using Spring boot framework has to be containerize (use Docker) for using automated deployment instance management like kubernetes, Docker Swamp
* Along with adopting to microservice architecture, the CSP has to adopt to new DevOps process and tools (E.g. RedHat OpenShit for CI & CD, Docker) for continuous delivery and deployment
* Instead of using Relational Database Management System like Oracle, SQL Server for read only data or data for analytics should use NoSQL database like MongoDB, Couch DB which provides better cost effectiveness and availability (CAP).
* Improve code coverage by developing more JUnit/TestNG tests using frameworks like mockito for mocking autowired beans.
* Microservices should use auditing, tracing, logging for tracing user request. This will help for grievance handling

# Bibliography

1. **BOOK**
   1. Lucas Krause, **Microservices: Patterns and Applications,** 2015, (<http://microservicesbook.io>)
   2. Shahir Daya, Shishir Narain, **Microservices from Theory to Practice**, IBM Red Book, 2015 (<https://www.redbooks.ibm.com/redbooks/pdfs/sg248275.pdf>)
   3. Sam Newman, **Building Microservices**, 2015, O’Reilly Media
2. **SCHOLARLY JOURNAL ARTICLES**
3. **Technology documentation**
   1. Kafka Installation - <https://kafka.apache.org/quickstart>
   2. Spring Cloud Framework - <http://projects.spring.io/spring-cloud/#quick-start>
   3. Spring Boot Framework - <https://spring.io/guides/gs/spring-boot/>
   4. Spring Kafka Integration - <https://docs.spring.io/spring-kafka/docs/2.1.4.RELEASE/reference/html/>
   5. Hibernate 5 with JPA - <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>
   6. Eureka Architecture - <https://github.com/Netflix/eureka/wiki/Eureka-2.0-Architecture-Overview>
   7. Load Balancing with Ribbon and Eureka - <https://spring.io/guides/gs/client-side-load-balancing/>
   8. HSQLDB Documentation - <http://hsqldb.org/doc/guide/>
4. **OTHER ARTICLES**
   1. ELECTRONIC PAYMENT SYSTEMS 101 by Amitabh Saxena, Managing Director, Digital Disruptions (<https://microlinks.org/sites/default/files/resource/files/Electronic_Payment_Systems_101.pdf>)
   2. Microservices
      1. Martin Fowler - <https://martinfowler.com/articles/microservices.html>
      2. Chris Richardson - <http://microservices.io/patterns/microservices.html>
   3. Domain model By Martin Fowler - <https://martinfowler.com/eaaCatalog/domainModel.html>
   4. Distributed Transaction (XA) - <http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf>
   5. 3D Secure protocol from EMVCo - <https://www.emvco.com/wp-content/plugins/pmpro-customizations/oy-getfile.php?u=/wp-content/uploads/documents/EMVCo_3DS_Spec_210_1017.pdf>

# Checklist of items for the Final Dissertation Report

This checklist is to be attached as the last page of the report.

**This checklist is to be duly completed, verified and signed by the student.**

|  |  |  |
| --- | --- | --- |
|  | **Is the final report neatly formatted with all the elements required for a technical Report?** | Yes / No |
|  | Is the Cover page in proper format as given in Annexure A? | Yes / No |
|  | Is the Title page (Inner cover page) in proper format? | Yes / No |
|  | (a) Is the Certificate from the Supervisor in proper format?  (b) Has it been signed by the Supervisor? | Yes / No  Yes / No |
|  | Is the Abstract included in the report properly written within one page? Have the technical keywords been specified properly? | Yes / No  Yes / No |
|  | Is the title of your report appropriate? **The title should be adequately descriptive, precise and must reflect scope of the actual work done.** Uncommon abbreviations / Acronyms should not be used in the title | Yes / No |
|  | Have you included the List of abbreviations / Acronyms? | Yes / No |
|  | Does the Report contain a summary of the literature survey? | Yes / No |
|  | Does the Table of Contents include page numbers?   1. Are the Pages numbered properly? (Ch. 1 should start on Page # 1) 2. Are the Figures numbered properly? (Figure Numbers and Figure Titles should be at the bottom of the figures) 3. Are the Tables numbered properly? (Table Numbers and Table Titles should be at the top of the tables) 4. Are the Captions for the Figures and Tables proper? 5. Are the Appendices numbered properly? Are their titles appropriate | Yes / No  Yes / No  Yes / No  Yes / No  Yes / No  Yes / No |
|  | Is the conclusion of the Report based on discussion of the work? | Yes / No |
|  | Are References or Bibliography given at the end of the Report?  Have the References been cited properly inside the text of the Report?  Are all the references cited in the body of the report | Yes / No  Yes / No  Yes / No |
|  | Is the report format and content according to the guidelines? The report should not be a mere printout of a Power Point Presentation, or a user manual. Source code of software need not be included in the report. | Yes / No |

**Declaration by Student:**

I certify that I have properly verified all the items in this checklist and ensure that the report is in proper format as specified in the course handout.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Place: \_\_ PUNE \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Signature of the Student**

**Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Name: \_ Tushar Dilip Phadke \_\_\_\_\_\_\_\_\_\_\_\_**

**ID No.: \_ 2016HT12516 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**