Islamic University of Technology (IUT)

Department of Computer Science and Engineering (CSE)

Data Consistency in Large Scale Applications

Authors

Saiful Islam Niloy - 170042065

&

Md. Nishat Ishmum - 170042027

&

Md Ariful Islam - 170042040

Supervisor

Dr. Kamrul Hasan

Professor, Department of CSE

Co-Supervisor

Dr. Hasan Mahmud

Assistant Professor, Department of CSE

A thesis submitted to the Department of CSE

in partial fulfillment of the requirements for the degree of B.Sc. Engineering in SWE

Academic Year: 2020-21

April - 2022

Declaration of Authorship

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by Saiful Islam Niloy, Md. Nishat Ishmum and Md Ariful Islam under the supervision of Dr. Kamrul Hasan, Professor of the Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), Dhaka, Bangladesh. It is also declared that neither of this thesis nor any part of this thesis has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Authors:

Saiful Islam Niloy

Student ID - 170042065

Md. Nishat Ishmum

Student ID - 170042027

Md Ariful Islam

Student ID - 170042040

Co-supervisor:

Dr. Hasan Mahmud

Assistant Professor

Department of Computer Science and Engineering

Islamic University of Technology (IUT)

Supervisor:

Dr. Kamrul Hasan

Professor

Department of Computer Science and Engineering

Islamic University of Technology (IUT)

Acknowledgement

We would like to express our grateful appreciation for **Professor Dr. Kamrul Hasan**, Professor of the Department, Department of Computer Science & Engineering, IUT for being our adviser and mentor. His motivation, suggestions and insights for this research have been invaluable. Without his support and proper guidance this research would never have been possible. His valuable opinion, time and input provided throughout the thesis work, from first phase of thesis topics introduction, subject selection, modification till the project implementation and finalization which helped us to do our thesis work in proper way. We are really grateful to him.

We are also grateful to **Dr. Hasan Mahmud**, Assistant Professor, Department of Computer Science & Engineering, IUT and **Md. Jubair Ibna Mostafa**, Lecturer, Department of Computer Science & Engineering, IUT for their valuable inspection and suggestions on our proposal.

Abstract

One of the challenges in large scale application is data consistency. In this paper, we have studied a few of the algorithms which are used to tackle this issue. We have also conducted some interview sessions with software experts and analyze their judgment regarding this issue. We have tried to analyse this topic based on two major software architecture. Mircoservice and monolithic architecture. Microservice is a trending software architecture for large scale application nowadays. It has many advantages along with some drawbacks. There are many services that communicate with each other to do a single operation. So, maintaining consistency is a big concern here. There are some ways to ensure consistency. Generally, microservice-based application use eventual consistency model. But this model has a chance of data loss. In some cases, data loss can be threatening to the business. To avoid this situation strong consistency model can be used. But algorithms that supports strong consistency comes with performance degradation of software compare to eventual consistency model.

Contents

1 Introduction **3**

1.1 Overview . . . 3

1.2 Problem Statement . . . 5

1.3 Motivation & Scopes . . . 5

1.4 Research Challenges . . . 5

1.5 Thesis Outline . . . 6

2 Background Study **7**

2.1 ACID Property . . . 7

2.2 BASE Property . . . 8

2.3 CAP Theorem . . . 8

2.4 Strong Consistency . . . 9

2.4.1 Two-phase commit (2PC) pattern . . . 9

2.4.2 GRIT . . . 10

2.4.3 Paxos Pattern . . . 10

2.5 Eventual Consistency . . . 11

2.5.1 SAGA Pattern . . . 11

2.5.2 Polling . . . 14

2.5.3 Event Sourcing . . . 15

3 Methodology **16**

3.1 Case Study . . . 16

3.1.1 Stock Management System . . . 17

3.1.2 Flash Sale . . . 18

3.1.3 Location Tracking . . . 18

3.1.4 Online Multiplayer Gaming . . . 18

3.1.5 Profile Picture Change . . . 18

3.1.6 Shopping Cart . . . 19

3.1.7 Social Media . . . 19

3.1.8 Insurance Partner Management System 20
3.1.9 Printing System 21
3.1.10 Reseller Site: Product Posting 21
3.1.11 DNS 22
3.1.12 Sports Score 23
3.1.13 Social Media Reaction Count 23
3.1.14 Product Rating 23
3.1.15 View count on Youtube 23
3.1.16 Salary Management System 24
3.2 Interview Session 25
3.3 Method of Data Collection 25
3.4 Method of Analysis 25
3.5 Evaluation & Justification of this Methodological Choices 26

4 Results & Discussion **27**
4.1 Overview 27
4.2 Project Metric Analysis 27
4.3 Data Consistency Analysis 29
4.4 Limitations 31

5 Conclusion and Future Work **31**

1 Introduction

1.1 Overview

Monolithic software architecture is the most traditional software architecture. Here all the functionality of the software wraped in a single project. If the user base of the project is not large, this architecture works fine. But if the user base increase, scaling needs to be done, functionality modification required. There might be required few functionality scaling in large portion and others in small portion. Monolithic architecture do not support scaling different functionality in different number. Here all the functionality will be scaled in same times. In monolithic architecture, fault isolation is difficult. As bug may arise from all the functionality, it is difficult for a project member have to have idea of all the modules. So, as the project grow, debugging becomes inefficient. Different programming language, stack can be a barrier in monolithic architecture. There might be situation that some functionality is efficient with one language, framework and other functionality with other programming language, framework. Monolithic architecture do not support functionality wise language, stack separation. In monolithic architecture, increase number of team can make things difficult. As all the functionality wraped up in single project, so all the team need to be sync with each other. There might be situation for one teams' modification another teams' functionality is not working. So, to make a simple change also, there is a risk factor. In monolithic architecture, if the application size is already large, it becomes difficult to make changes or adding new things. Because the developer have to have the idea of the related functionality that may have impact of the changes and new addition.

Microservice is a popular service-oriented software architecture for large scale, extensive application. It is a combination of several small services deployed independently, have inter-communication between them worked as a single system. Microservice support independent service scaling in different number of times. For example, in a system, service 1 hits 1000 hits per second and service 2 hits 3000

hits per second on average. So, a system designer a scale service 2 and deploy it in 3 servers using a load balancer and deploy service 1 in one server. Each service has their own codebase as separate project. As a result, if a bug occurs, it is comparatively easier [1] to find which functionality is creating issues and which service is used for this. In this scenario, whole project need not be taken into consideration while debugging as it has already been find out that the service which is responsible for the issue. Separate programming language and separate framework or stack can be used for separate services based on requirement. Because each service deployed independently. So, functionality wise stack can be chosen based on performance, time, resource availability etc. Separate teams can be organised for separate services. Here dependency between teams in comparatively less than in monolithic architecture. Because each service is isolated by terms of codebase and deployment. Faster deployment is possible. Because each service is small compared to the whole project developed in monolithic architecture. As a result, deployment difficulties reduces. Adding changes in system means calling the responsible team to do that. Not all the teams may need to be acknowledged about it. As a result, faster modification can be done compare to monolithic architecture.

Microservice offers a lot of flexibility. But it has some cons also.[2] One of the main challenges of microservice architecture is maintaining consistency among the services. As to handle a request, there might need to call different several services to do the job and response. Among the service calls, there may be one service failed to work and also previous services make some changes to the database of that request. Now it make the system[3] inconsistent. We have discussed about two case study where this type of consistency issue arises. One of these is a part of insurance management system [4]. Other is about the ticket booking system. None of the cases can have hundred percent perfect solution. But based on the scenario each of them trade off with some factor.

1.2 Problem Statement

One of the challenges in large scale application is data consistency. When an application is scaled, there are multiple database replicas. When a new data is inserted to one, it needs to replicate to all the other database replicas. To do so, it takes some time. In this intermediate period of time, the system remains inconsistent. This situation can happen in microservice architecture if two or more service use same data and in monolithic architecture if the system is scaled and there is multiple database replica.

1.3 Motivation & Scopes

There might be different data consistency requirement based on situations. Low consistency requirement can be handled with eventual consistency. Some smart pattern like event sourcing can be used to loss data due to consistency issue here. In some cases, data loss might not be a matter. But in some cases data loss can be deadly. Like in a system of high consistency requirement like flash sale, if it is a distributed system, there might be a scenario where a single product is sold to multiple customer, if consistency is not ensured. c Our research will give a general overview of how industry expert approach this issue in different cases.

1.4 Research Challenges

One of the major challenge in this domain in consistency control in large scale applications, the most part of the research in this domain has happened in software industry. Most of the research work was kept private for business value. So, it was hard to go forward with reverences. Maintaining consistency in distributed architecture is difficult. According to CAP theorem[5], both availability and consistency can not be achieved at the same time. So, it has to trade off based on requirement.

1.5 Thesis Outline

In chapter 1, we have discussed about the introduction of our work. In chapter 2, we have showed the background study we have to do. In chapter 3, we have state our methodology of work. Basically, we have two types of methodology. Case study collection and interview with software experts. In chapter 4, we have shown the outcome and analysis of the interview sessions. In chapter 5, we have draw conclusion and states some scope for future work. The final section of this study contains all the references.

2 Background Study

Consistency means the data integrity remains same before and after the transaction occurs. Without consistency there is a chance of data loss. Also in microservice architecture, some user may get new data and some user may get old data.

There are two type of consistency model.

- Strong Consistency
- Eventual Consistency

Strong consistency model follows ACID property. Eventual consistency model follows BASE property.[6, 7]

2.1 ACID Property

- **Atomicity:** Each transaction is considered as one unit and the entire will be executed at once. Partial transaction is not allowed.
- **Consistency:** Database should be valid before and after every transaction. Example- The total amount before and after the transaction must be maintained. Total before T occurs = 500(user 1) + 200(user 2) = 700. Here 100 transferred from user 1 to user 2. Total after T occurs = 400(user 1) + 300(user 2) = 700.
- **Isolation:** This attribute assures that several transactions can take place at the same time without causing database inconsistencies.
- **Durability:** The updates and modifications to the database are written to disk when the transaction has completed execution, and they survive even if the system fails.

2.2 BASE Property

- **Basically Available:** Ensures data availability by replicating it among database cluster's nodes.
- **Soft-state:** Data may fluctuate over time due to the absence of strong consistency. The developers are in charge of maintaining consistency.
- **Eventual consistency:** Although immediate consistency may not be feasible with BASE, it will be achieved eventually (in a short time).

2.3 CAP Theorem

There is a famous theorem regarding distributed computing giving by Eric Brewer called CAP theorem. It states about database consistency in distributed system. Different approach later comes up for ensuring data consistency considering this theorem.

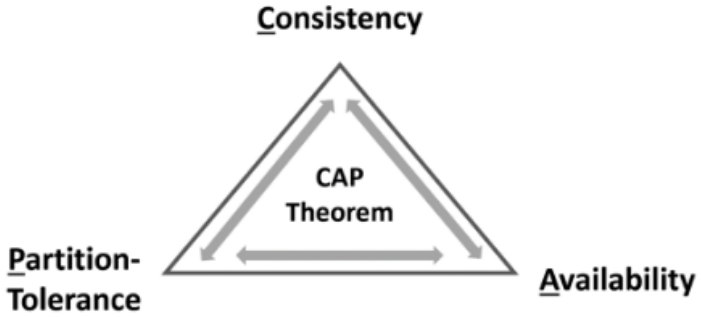

Figure 1: CAP Theorem

According to CAP theorem, in a partitioned distributed data store, both consistency and availability can not be achieved simultaneously. We have to trade off between consistency and availability. [5]

2.4 Strong Consistency

In strong consistency model, user always get latest data. User might get data little bit late but will always get the correct and consistent data.

2.4.1 Two-phase commit (2PC) pattern

Two-phase commit or 2PC pattern is one of the approach of strong consistency model. [8] It is widely used in database systems. It is a blocking protocol used to guarantee that all the transactions are succeeded or failed together in a distributed transaction. In some situation, we can use 2PC for microservices but it is considered impractical within a microservice architecture.

2PC has two phases: a prepare phase and a commit phase. In the prepare phase, all microservices that are going to participate in the transaction will be asked if they are ready for some data change. Once all microservices are prepared, the commit phase will ask all the microservices to make the actual changes. Normally, there needs to be a global coordinator to maintain the lifecycle of the transaction, and the coordinator will need to call the microservices in the prepare and commit phases. 2PC is a very strong consistency protocol. The prepare and commit phases guarantee that the transaction is atomic. The transaction will end with either all microservices returning successfully or all microservices have nothing changed. 2PC allows read-write isolation so the changes on a field are not visible until the change is commited.

2PC can give us strong consistency but it is not recommended for many microservice-based systems because 2PC is synchronous (blocking). The protocol will need to lock the object that will be changed before the transaction completes. So in transaction all participating services will remain block and other can't use those services. The lock could become a system performance bottleneck. Also, it can happen that two transactions mutually lock each other (deadlock) when each transaction requests a lock on a resource the other requires.

2.4.2 GRIT

It is a strong consistency model. In this model, there is three phase to ensure consistency in database.

- **The optimistic execution phase:** When new data is fetched or updated to the database, each database service state fetch and update operation as r-set and w-set.
- **The logical commit phase:** At this stage, transaction manager operate conflict resolution. Then it commits logically if no conflict but abort in case of conflict.
- **The physical materialization phase:** Now transaction is materialised by database and physical commit is made. [9]

2.4.3 Paxos Pattern

Paxos is a consensus based algorithm for database consistency. It works on distributed network in an asynchronous fashion.[10] It has three types of roles:

- **Proposer:** It sends value to the acceptors. The purpose is to get the value agreed.
- **Acceptor:** It receives value from the proposers. Then decides whether to agree on the value.
- **Learner:** When acceptor choose a value, it learns from that.

It has two phases:

Phase 1

- Proposers send prepare message with a version number to acceptors.
- If acceptor receive the prepare message with older version than it already have, it ignores it. When acceptor receive the prepare message with updated version, it ensures it will ignore the future request that has older version.

Phase 2

- When proposer receives a majority of promise messages from the acceptor, it prepare accept message with identifier number n and a value v. Then the proposer send accept message to all the acceptors with the value.
- When acceptor receives the accept message, it accept the value if it has not sent a promise message with greater identifier number. Otherwise, it will ignore the message.
- When the accept message is accepted, the acceptor will store the message and send accepted message to the every proposor and learner.

2.5 Eventual Consistency

In eventual consistency model, data gets replicated to the replica server or other necessary server eventually. Here user will receive data very fast. But there is a chance of getting older data. And in this case, after some period the user will get the updated and consistent data.

2.5.1 SAGA Pattern

Saga pattern is one of the approach of eventual consistency. [11] A Saga is mainly a sequence of local transactions.[12] It is one of the 6 Important Data Management Patterns of microservices. Saga pattern is mainly used in distributed system where atomicity is not in high priority. It can provide eventual consistency. Here each service perform its transaction and publish an event. The subsequent transaction is triggered based on the event published by the previous service. And if any service fail to do it's task, they publish a failed event and based on that a series of compensating transactions execute to undo the impact of all the previous services.

There are two types of Sagas:

- **Orchestration-Based Saga:** In Orchestration-Based saga, there is a orchestrator that manages all the transactions. It maintain which service need to call and if some service failed it knows what need to be done and execute compensation transaction. This orchestrator can also be thought of as a Saga Manager.

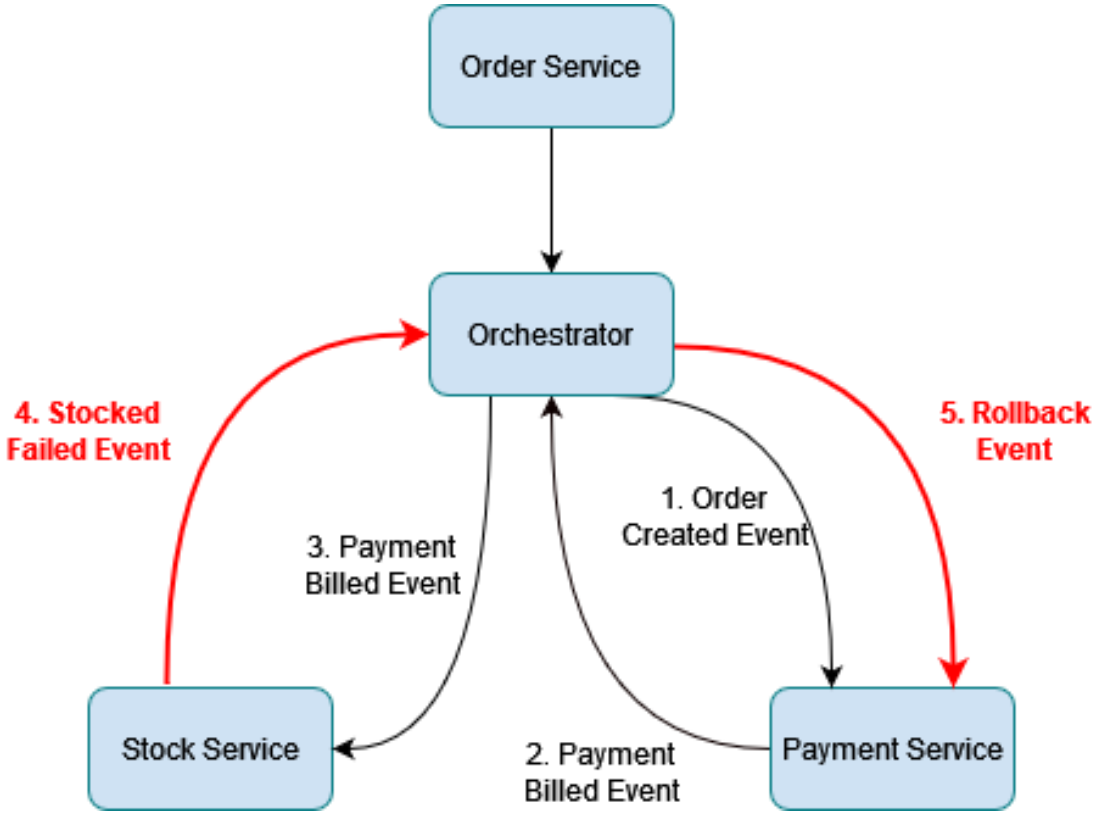

Figure 2: Orchestration-Based Saga

From the figure 2 we can easily understand orchestrator-based saga. Here at first order service is called. It employs the orchestrator and it send order_created_event for the payment service. Payment service do it's task and send success or payment_billed_event event to the orchestrator. Then orchestrator send this event to next service the stock service. It stock service failed to do it's task it send stock_failed_event to the orchestrator. The orchestrator know for which event what to do. So after getting the error

event it send rollback_event to the previous services, in this case it send the rollback_event to payment service and the payment service execute it's compensation function to undo the transaction.[13]

- **Choreography-Based Saga:** In this approach, there is no central orchestrator. Each service do their local transaction and create success or failed event. The other services act upon those events. If it is success event next service do it's task and if it is failed event, it's previous service perform the compensation transaction and publish necessery events. By this way without a coordinator or orchestrator this approach works. In this if the perticipating service series will become long it will be hard and complex. We can implement this if we only have 2-4 services perticipating in the transaction.

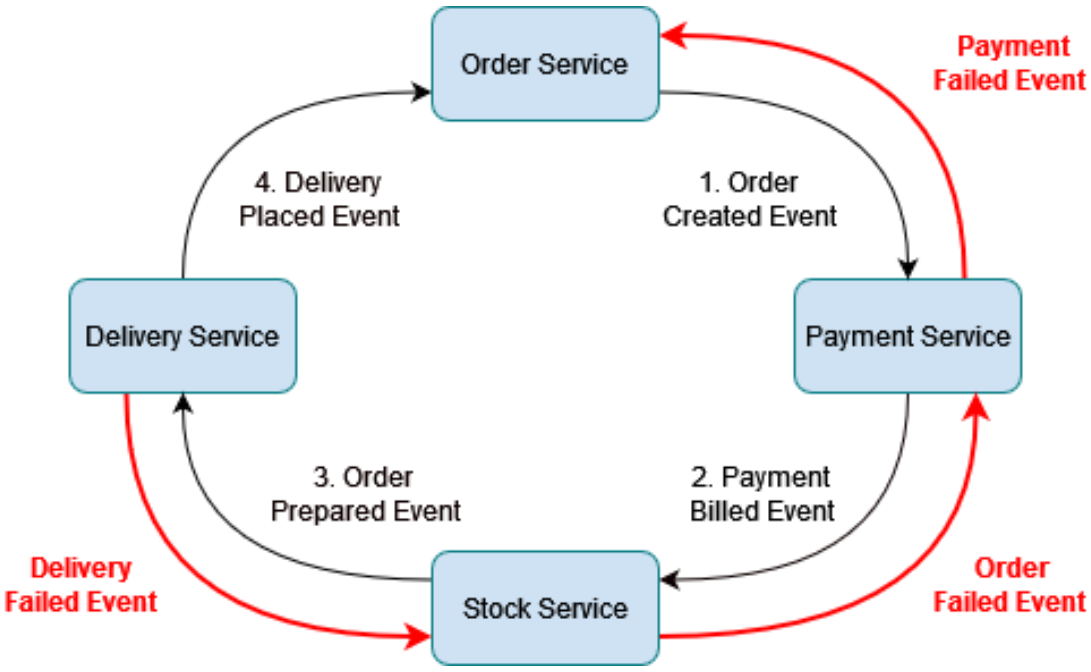

Figure 3: Choreography-Based Saga

In the figure 3 we have a order service. It do it's work and send order_created_event to the message queue. The next service, payment service trigger with the order_created_event event and perform necessery action. Then it send payment_billed_event on success and the next service, stock

service do it's work. It it success it send order_prepared_event event and delivery service trigger and do it's job. After that it send delivery_placed_event to the message queue and from that the order service can know that the actions are performed successfully. If some service failed, let say delivery service failed. Then it will publish delivery_failed_event and by that the previous stock service will trigger and execute it's compensation function. Then it will trigger error event so that previous services can also perform their compensating transactions.

The main benefit of the Saga Pattern is it give us high availability. It can maintain data consistency across multiple services without tight coupling. Also there are no chance of deadlock in this pattern.

However, there are some disadvantage of the Saga Pattern. It can't give us strong consistency so some time we may get old data instead of current one but they will be consistent eventually. Saga pattern increase the complexity from a programming point of view as developers need to design the compensating transactions which can be very hard and complex.[13]

2.5.2 Polling

This model provide us eventual consistency. Here the process is to perform the tasks assigned of synchronizing redundant data to the services that are interested in it. A simple solution is to ask for new data on a regular basis via an interface supplied by the master data service. Multiple data updates can be transferred at the same time using timestamps. The length of the polling interval can be used to regulate the size of the inconsistency window for each interested service separately. Despite the fact that the data sets are ultimately consistent, the time range in which they may diverge is much longer than with a synchronous solution. [4]

2.5.3 Event Sourcing

Instead of saving the most recent status of data into a database, the Event Sourcing pattern allows you to save all events in a database in a sequential manner. The name of this event database is event store. Rather than updating the state of a data record, it appends each modification to a list of events in chronological order. Any event that is triggered is saved in the event store. There will be no updates or deletions to the data, and each event will be saved as a record in the database. If the transaction fails, the failure is recorded in the database as a failure event. Every record entry will be an atomic operation.

The API for adding and retrieving events for an entity is available in the store. The event store functions similarly to a message broker. It announces the events so that other services can be notified and handle them if necessary. It gives services the ability to subscribe to events over an API. When a service saves an event in the event store, it is sent to all subscribers who are interested. This method offers a few advantages. It solves atomicity issues, keeps track of records' history and audits, and may be used with data analytics because historical records are kept.

Instead of keeping the current application state, it might be more useful in some circumstances to save all state transitions and then accumulate them to the current state as needed. This method can also be applied to the challenge of distributing data changes. Events are published when master data changes. The history of all events, unlike the Publish-Subscribe method, is saved in a central, append-only event store. It is accessible to all services, and they can even construct their own local databases from it, as long as they match their own constrained context. This technique delivers a high level of consistency: any data modification is immediately visible to all other components of the system. It's important to note that a central data store is introduced, which microservices try to avoid. This weakens the loose coupling and may be a scaling concern; yet, the data storage's append-only nature allows for great speed. [4]

3 Methodology

We have collected some case studies from internet and our understandings where different consistency and availability requirement has been shown. We have also taken interview sessions of several industry expert about how they approach this topic.

3.1 Case Study

We have done some case study of microservices consistency issue. [14]

Consistency	Cases	Availability
High	Stock Management	Low
	Flash Sale	
	Finacial Transaction	
	Location Tracking	
	Online Multiplayer Gaming	
Moderate	Profile Picture Change	High
	Shopping Cart	
	Social Media Post	
	Insurance Mangement System	
	Printing System	
	Re-seller Site: Product Posting	
Low	DNS	High
	Sports Score	
	Social Media Reaction Count	
	Product Rating	
	Views on Youtube videos	
	Salary Mangement System	

Table 1: Consistency-Availability Wise Category

3.1.1 Stock Management System

In stock management system, there are some stock of service or product. After user order it, this product or service will be booked for the user and system will wait until the next operation like payment. After the payment complete, user will get the product, inventory product or service count will be updated. There can be three services in a stock management system like order service, stock service, payment service which interact each other to complete the task.[6]

- **Order Service:**When a user requests for a product, order service is called. This service initiates the process. Then call the stock service for booking the product. Then the order service calls the payment service to deduct money from the user. After that the product purchase receipt is sent to the user from the order service.
- **Payment Service:** This service deducts money for the user's bank or mfs(Mobile Financial Service) account and sends a success or fail response.
- **Stock Service:**When the order service calls it with the product id, then this service just booked this product for the user and the remaining product count decreases.

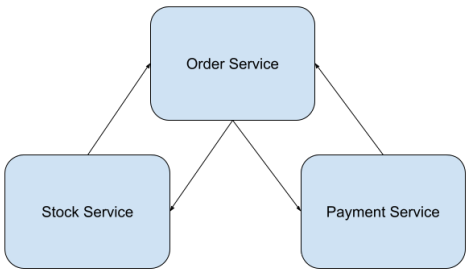

Figure 4: Stock Management System

- **When Inconsistency Can Occured:** To purchase a product, a sequence of operations need to be done. During these operations if any of the services failed inconsistency may occur.
  - If the payment service fails after the product booking, the product is booked in the database but payment has not been done.
  - If the order service fails after the payment from the user account, the user has not got a purchase receipt but payment has already been made.

3.1.2 Flash Sale

Ecommerce shops offers flash sale with huge discount on many occasion. Customers purchases a lot of product in this period. In this case, there might be a scenario that one item can be sold to multiple persons. Because a lot of hit to the server comes this scenario.

3.1.3 Location Tracking

Location tracking means to track someones real time location. In this system, to provide real time location, system needs to offer strong consistency. Otherwise, there might be scenario that, two user is seeing different location of the tracked person.

3.1.4 Online Multiplayer Gaming

People from over the globe play online multiplayer games now a days. It need to provide strong consistency if different server used.

3.1.5 Profile Picture Change

If some person change his profile picture, there needs time to propagate this profile picture to all the servers of the system. As a result, there might be a scenario that, some person from another country is seeing the old profile picture.

3.1.6 Shopping Cart

If user add item to the cart, this data may be stored to multiple servers. As a result, there might be scenario that when user check the cart, some product is missing. User is watching previous state.

3.1.7 Social Media

- **Post Service:** This service is responsible for the user's post information. After the user posted successfully, at first it's information stored on the user's nearest data server.
- **Servers Across the Globe:** For a large scale international software, there need to set up data centers across the globe. If one user stores information, it is first stored on one server. After that replicated to all the other region servers.

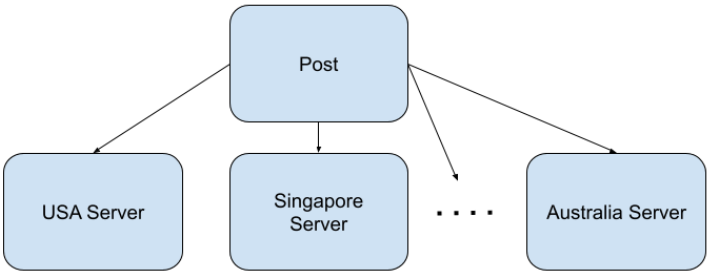

Figure 5: Social Media

- **When Inconsistency Can Occured:** To replicate a user's post to all the servers needs some time. There might be a delay in replication on some servers. So, in this intermediate time the whole system is in an inconsistent state until replicated to all the servers.

  There might be a case that, users' post information replicated to some of the servers but failed in some other region's server. In this scenario, people from some region can see the post, but people from some region can not.

3.1.8 Insurance Partner Management System

- **Partner Service:** This service is responsible for storing partner's detail information.
- **Contract Service:** This service handles the information about contract id, contract type of the user.
- **Communication Service:** This service handles users' contact information like address, email, telephone etc.
- **Account Service:** This service is responsible for user bank account information.

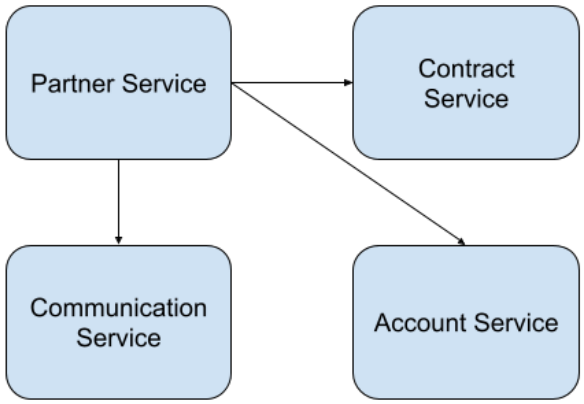

Figure 6: Insurance Partner Management System

- **When Inconsistency Can Occured:** When a new user comes to the system, first the changes are applied to partner service. Then changes propagate to other services. If propagation fails in any of the services, inconsistency problems arise. [4]

3.1.9 Printing System

- **Coordinator Service:** When a user requests to print any document, coordinator service calls the payment service. After that, the printing service is called to print.
- **Payment Service:** This service deduct money from the user account. If it fails for reasons like less money sends a failed response.
- **Printing Service:** This service adds documents to the available printer queue or user chosen printer queue.

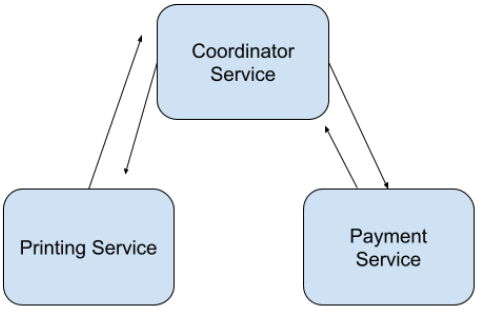

Figure 7: Printing System

- **When Inconsistency Can Occured:** There may be scenarios, like payment service has deducted money from the user. Then printing service may fail because of paper shortage, lack of ink or hardware error of the printer. In this case money has been deducted but printing has not been done. Another case is like if any user clears the queue the money will not be returned.

3.1.10 Reseller Site: Product Posting

- **History Service:** User post information like post time, location and user history like total number of products sold or bought by the user etc handled by this history service.

- **Search Service:** Posts id, tile, thumbnail, short description etc are stored by search service. During searching this service is used. When the post is clicked to see details then another service(Post Details) is called.
- **Post Details Service:** Whole post information (post id, tile ,complete description, image etc.) is stored by the post details service.

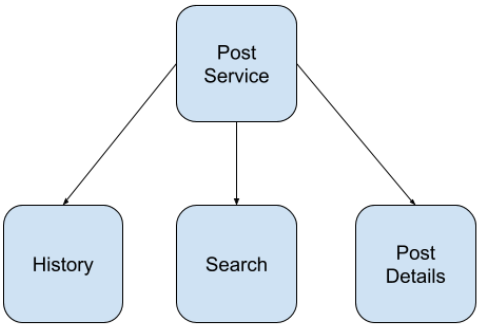

Figure 8: Reseller Site: Product Posting

- **When Inconsistency Can Occured:** When a user posts on the site, the request goes to the post service. Then post service call history, search and product detail service to store each service's required information. But it may happen that one of the services failed to do the operation.
  - If the history service failed, history information will not be updated.
  - If search service failed, this post will not be shown while searching.
  - If the post details service fails, if any user clicks to see details of the post, it will not be shown.

3.1.11 DNS

Domain name server stores the IP address of domains. When a new domain is bought, it replicates its IP address to all the server eventually. As a result, we can not access the domain immediately after buying the address. To be accessible from all over the world, it takes time.

3.1.12 Sports Score

Sports score updates to server may take time. Like servers in different region of the world have to update the latest data. But to replicate latest data to all the server takes time.

3.1.13 Social Media Reaction Count

Social media provides reaction count. Now there might be situation that people from all over the world giving reaction. Now to sum up the reaction to specific server may require time. Now to maintain consistent distributed database at every moment becomes harder.

3.1.14 Product Rating

When consumer rate the product the average product rating changes. But it continuously sum up the rating and updating the average rating count becomes expensive. Moreover in microservice there might be scenario, product rating is used in multiple services. IN that scenario, to replicate the updated rating to the other service takes time.

3.1.15 View count on Youtube

Youtube video is watched from all over the world. As result, view count needs time to count.

3.1.16 Salary Management System

- **Salary Service:** This service works as a coordinator of the other service here. Employee salary transactions are handled by this service. First salary service calls the employee service and takes the bank account information of the employee. Then call the Bank API to transfer salary to the bank account of the employee. If the Bank API returns a success response, then it calls a notification service to notify the user.

- **Employee Service:** This service is responsible for storing employees' information like employee name, rank, contact number, home district, bank account information etc.

- **BANK Api:** This is a third party service. Using this service, requests to the bank can be sent for money transactions.

- **Notification Service:** This service collects phone numbers or email from the employee service. Then sends notification to the employee about the salary transaction.

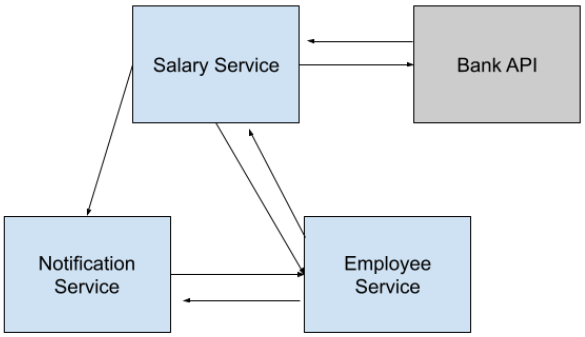

Figure 9: Salary Management System

- **When Inconsistency Can Occured:** Salary service will use other services to send salary and notify the employee. But after a salary transaction, if for some reason notification service fails, the employee will not be notified.

3.2 Interview Session

For solving consistency issue different approach is followed based on business requirement. We conduct formal interview session with software experts and follow certain discussion topic to find a generalised way of thinking regarding this issue.

3.3 Method of Data Collection

We have conducted interview session with software experts. We have set a rule to filter out our sample data. We have chosen working experience of minimum 3 years and microservice working experience of minimum 2 years. During interview session, we have followed predetermined question set. Some of them are close ended and some of them are open ended.

3.4 Method of Analysis

We have designed our interview structure into six section. First section takes personal information like name, email, working experience, designation, number of project database consistency issue faced etc. Second section take project info. Like project architecture, project type, size of database, active user count etc. Then in the next section, we have take their opinion of based on project architecture. If they want to share project experience which has followed monolithic architecture, we have asked question from monolithic section and otherwise from microservice section. Here we asked data consistency issue frequency, monitor tool for inconsistency detection, how this issue can be solved etc. Then if there was a manual intervention section. If any manual participation required we asked question here. At last section, we asked their overall opinion regarding this issue.

3.5 Evaluation & Justification of this Methodological Choices

This issue can solved in various ways. But to explore a general approach, we need to learn it from software experts who have practically handle this issue. To do so, interview session is important. Specially, it is convenient to make the question understandable by explaining it. As a result, chances of getting the appropriate answer increases than other methodology like survey. Also there is a chance of asking follow up question based on the situation. It can be open ended question also.

4 Results & Discussion

In this section, we have presented the outcome of the interview session. We have shown some data in percentage format, discuss about the answers and also present some important quote from the software experts.

4.1 Overview

We have taken interview from software experts. Among them, there were 1 chief technology officer, 1 chief engineer, 3 senior software engineer and 2 software engineer. The total participants in interview session in 7. Almost 57% of the participants have more than 5 years of working experience and 43% participants have less than 5 years of working experience. And 29% participants do not have working experience in microservice, 43% of the participant have more than 3 years of working experience in microservice. Each interview session continued for around a hour.

4.2 Project Metric Analysis

Almost all the participants faced consistency issue at minimum 1 project. Among the projects, 29% of them followed monolithic software architecture and 71% of them followed microservice architecture.

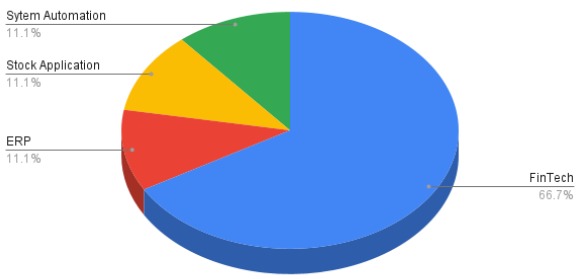

Figure 10: Project Type

Among the projects, 66.7% of the projects were in the category of Fin-Tech and 11% for each of ERP, Stock Application and System Automation. 57% of the projects have more than 100K of active user and 43% of the projects have less than 100K active user. 62.5% of the projects used CQRS[15] approach.

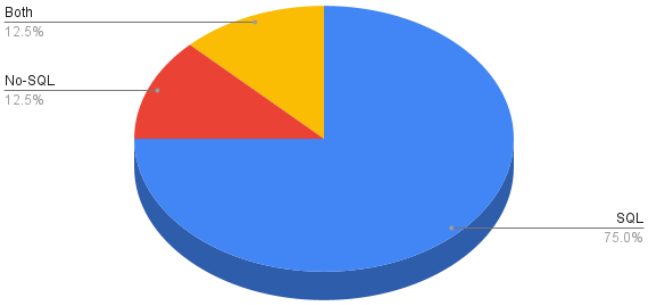

Figure 11: Database Type

75% of the times SQL database used, 12.5% times No-SQL and 12.5% times both type of databases was used in the projects. Another thing is, 37.5% of the project has more than 100GB, 12.5% of the project between 10 to 100GB and 50% of the projects have less than 100GB estimated database size.

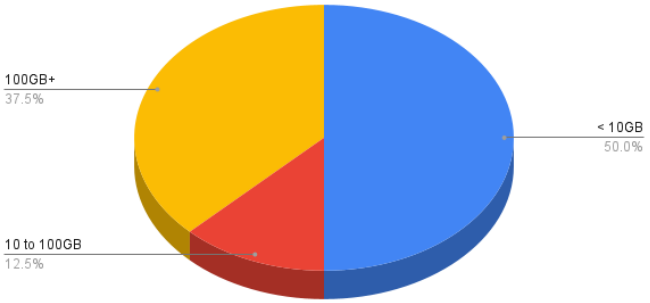

Figure 12: Estimated Database Size

4.3 Data Consistency Analysis

25% of the projects were followed monolithic architecture and 75% of the projects were followed microservice architecture. Among the microservice projects, 50% have less than 10 microservices, 16.7% have less than 50 microservices and 33.3% have less than 100 microservices.

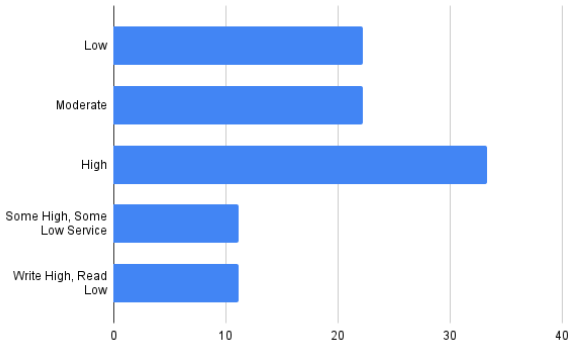

Figure 13: Consistency Requirement

Among the projects, 22% of projects have low consistency requirement, 22% of the projects have moderate consistency requirement, 33% of the projects have high consistency requirement, 11% of the projects have both high and low consistency requirement based of modules or services and 11% of the projects have high consistency requirement for write operation and low consistency requirement for read operation. Most of the projects faced consistency issue. Some of the projects do not faced. The reasons for not facing are handled by application layer in a case, availability was prioritize in another case. As we know, according to CAP theorem, it is necessary to trade off between availability and consistency in distributed approach based on business requirement[5].

About 43% of the project do not faced any consistency issue , 28% project faced 1-10 and 28% projects faced 10+ consistency issue per 10 thousand transaction. Some of the projects used database consistency monitor tool or approaches. One is if an event failed to acknowledge its' success message, several attempt up to threshold limit will be taken. If it fails even after several attempt, it automatically

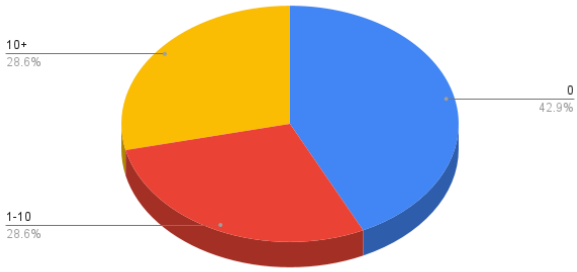

Figure 14: Frequency of Data Consistency per 10K transaction

goes to another special queue for further manual intervention. Another approach is check between databases when new data comes.

Several approach has been used for data consistency requirement application. They are event sourcing, master-slave, event driven architectures and transaction management tools. Some of the projects were depend completely on cloud service provider to ensure database consistency. One of the software expert stated, on their project, client app checks the inconsistency issue. If inconsistency found, it will be acknowledged to server and server will take care further steps. Almost all the software experts do not feel change requirement of software architecture for ensuring database consistency. Some of them told due to tight deadline or not on business requirement they can not think of improvement.

If inconsistency issue occurred, some of them use manual intervention upto certain level like manually trigger an event. Few of them rely on cloud provider to maintain this issue. They do not think to improve this section by automation because of no requirement or for future scope. One of them told, there might need some must manual intervention.

Most of the software experts told eventual consistency is enough to achieve data consistency in microservice architecture. But context of business requirement should be clearly understood for that. Some of them also told that it depend on requirement like high consistency requirement application like FinTech service may not be suitable to use eventual consistency.

Most of the software experts think, monolithic is not the best fit just for strong consistency requirement. Some of them told it depends on on user count. Some of them told, strong consistency can also be ensured by microservice architecture. One of them told, if scalability is not required, monolithic architecture can be a good choice.

One of the software expert told "People should believe in eventual consistency. Everything is eventual consistency in real world." He try to focus that strong consistency is not much found in real life. Almost every problem can be solved by eventual consistency and also guaranteed high availability.

4.4 Limitations

We had lack of dataset collection. We have taken data from 8 software experts. We taken interview with software experts from Bangladesh only. If we take data from other part of the globe, our analysis will be more generalised and clear.

5 Conclusion and Future Work

We have interviewed several software experts. More interviews is required to get a more clear and generalise ideas in this issue. Architecture decision depends on the business requirement, cost vs performance analysis. If the feature is very less and no requirement of scalability, monolithic architecture can be chosen. In that case maintaining strong consistency is easier. In other case we can choose microservice architecture. Here high availability can be achieved with eventual consistency. Eventual consistency can handle almost every kind of requirement.

References

[1] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A systematic literature review," *Information and Software Technology*, vol. 131, p. 106449, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301993

[2] "Microservice Trade-Offs." [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html

[3] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang, "Microservice architecture in reality: An industrial inquiry," in *2019 IEEE international conference on software architecture (ICSA)*. IEEE, 2019, pp. 51–60.

[4] A. Koschel and A. Hausotter, "Keep it in sync! consistency approaches for microservices an insurance case study," 2020.

[5] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, no. 10.1145. Portland, OR, 2000, pp. 343 477–343 502.

[6] D. Kizilpinar, "Data Consistency in Microservices Architecture," May 2021. [Online]. Available: https://medium.com/garantibbva-teknoloji/data-consistency-in-microservices-architecture-5c67e0f65256

[7] "Closer to Consistency in Microservice Architecture - DZone Microservices." [Online]. Available: https://dzone.com/articles/transaction-management-in-microservice-architectur

[8] C. Barthels, I. Müller, K. Taranov, G. Alonso, and T. Hoefler, "Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores," *Proceedings of the VLDB Endowment*, vol. 12, no. 13, pp. 2325–2338, 2019.

[9] G. Zhang, K. Ren, J.-S. Ahn, and S. Ben-Romdhane, "Grit: consistent distributed transactions across polyglot microservices with multiple databases," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 2024–2027.

[10] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[11] P. A. Bernstein and S. Das, "Rethinking eventual consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 923–928.

[12] M. Štefanko, O. Chaloupka, B. Rossi, M. van Sinderen, and L. Maciaszek, "The saga pattern in a reactive microservices environment," in *Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)*. SciTePress Prague, Czech Republic, 2019, pp. 483–490.

[13] K. Malyuga, O. Perl, A. Slapoguzov, and I. Perl, "Fault tolerant central saga orchestrator in restful architecture," in *2020 26th Conference of Open Innovations Association (FRUCT)*, 2020, pp. 278–283.

[14] K. Nath, "Consistency Guarantees in Distributed Systems Explained Simply," May 2021. [Online]. Available: https://kousiknath.medium.com/consistency-guarantees-in-distributed-systems-explained-simply-720caa034116

[15] Z. Long, "Improvement and implementation of a high performance cqrs architecture," in *2017 International Conference on Robots Intelligent System (ICRIS)*, 2017, pp. 170–173.