A Transaction Platform for Microservices-based Big Data Systems

María Teresa González-Aparicio[a], Muhammad Younas[b], Javier Tuya[a] , Rubén Casado[c]

[a] Polytechnic School of Engineering, University of Oviedo, 33204 Gijón, Spain

[b] School of Engineering, Computing and Mathematics, Oxford Brookes University, Oxford OX33 1HX, UK

[c] Science and Technology Park, Accenture SL, 33203 Gijón, Spain

Abstract

Microservices architecture has increasingly been adopted for building distributed and scalable applications. The premise is that microservices are designed as smaller software components which are easier to be preserved and which enable separation between different components. This paper proposes a new transaction platform for microservices architecture to manage processing of big data stored in a cluster of NoSQL databases. New asynchronous protocols are designed to execute database operations as transactions, and to maintain their correctness and consistency. A prototype system has been developed that simulates London bus service across bus routes. It is evaluated through simulation experiments using big data from 'Transport for London' data service in order to analyse effects of transaction processing on response time and throughput in microservices architecture. The transaction platform reliably processes database operations, and enables data availability and consistency in failure-free and failure-prone environments.

Keywords: Microservices architecture, transaction management, big data, NoSQL database

1. Introduction

Microservices architecture has become a popular platform for developing data-driven applications [1] that run in a complex distributed setup such as cloud, IoT and big data systems. In microservices architecture, an application is implemented as a set of autonomous microservices that represent different business components (or functionalities) and that work together in order to achieve a desired output [2]. The aim is to provide high scalability, availability, maintainability and decentralization of applications and data storages. Microservices can be implemented in different languages and can access data from multiple types of databases including relational (SQL) databases and NoSQL databases. Relational databases require data to be structured in tabular (relational) form, enforce integrity constraints and relationships between tables. They also support transactions with ACID (Atomicity, Consistency, Isolation, Durability) properties in order to ensure strict consistency and allow concurrent executions of transactions. Nevertheless, NoSQL databases follow different data models and provide different level of consistency, availability and efficiency. They do not support transactions like relational databases. They are mainly designed to process large volume of data and generate results in real time such as analysis of millions of tweets or processing of live road traffic data.

In existing literature, different techniques and models have been developed for managing transactions in microservices architecture. A Saga Pattern [3] approach is developed in order to manage local sequential transactions for updating microservices. It follows compensating actions approach that compensate or cancel completed actions in the case of failure. However, this approach does not consider read isolation wherein isolation anomalies can emerge [4]. Authors in [5] propose to improve Saga Pattern approach by setting a constraint that database commits at database layer only if all transactions are totally successful at the cache layer. Therefore, CRUD (Create, Read, Update, Delete) operations are first performed at memory (cache) level. If transactions are successful then effects of their operations are reflected at database level. In our previous work [6], we developed a framework for managing transactions that takes into account contextual information of users and a required level of big data consistency. We carried out a detailed analysis of impact of big data (and its characteristics) on maintaining data consistency.

Transaction management in microservices architecture involves NoSQL big databases which can be deployed in a cloud and IoT setup. A transaction comprises a series of operations (e.g., read/write) that either successfully execute in full or not at all. If one operation fails, then all the operations must be rolled back to keep data in consistent state and maintain correctness of application. However, transaction management in microservices architecture and big databases brings in new challenges [7]. Transactions in a microservices architecture encompasses multiple databases – i.e., each database is associated with a specific microservice. Therefore, managing transactions and maintaining data consistency across multiple databases is a challenging task. In addition, other inherent characteristics (i.e., volume, velocity, variety) of big data complicate the process of managing transactions in microservices architecture. For instance, volume (or size) of data is large and data are generated and consumed at a high speed (velocity) as compared to traditional data. Big data also comes in various structures and formats unlike classical relational database which is well structured and normalized. In addition, NoSQL database technologies scarcely support transactions, and there is a lack of recovery mechanisms such as fallback or rollback. For instance, Redis implements the optimistic locking protocol and a script is considered as a transaction itself; meanwhile BigTable implements transactions at a row level; and Riak and Cassandra rely on conflict-resolution mechanisms.

In this paper, we design and simulate a new two-level transaction management platform for microservices architecture that supports real-time processing of big data which is stored in a cluster of NoSQL key/value databases. We design new asynchronous protocols for executing database operations as transactions at data storage layer in order to maintain consistency of big data. The platform is capable of processing transactional (write/read) operations in a reliable and transparent manner. It handles transaction failure and provides data availability and resilience in case of database failures.

A Docker containerized environment [8] and Redis NoSQL databases are used in the design and simulation of the proposed two-level transaction management platform. Such an environment is chosen as it is most widely used in microservice architecture [1]. The transaction platform is evaluated using data from 'Transport for London' (TfL) data service [9] and London Bus service as an application area. We conduct a series of experiments by simulating London Bus service across bus routes. Specifically, the case study provides a complex environment for developing a transaction platform that requires modern technologies of NoSQL, cloud, IoT and microservices architecture. Experimental results show that the transaction platform maintains consistency of big data and provides a fair response time and throughput in both failure-free and failure-prone environments.

The work is focused on transaction management at the storage layer with microservices to achieve a higher level of storage independence along multiple databases, and data are continuously received from data sources such as sensors, buses, etc. This can be attributed to data stream as in lambda architecture [10], which combines batch and online (stream) processing on data platforms. The rationale for choosing London Bus service is twofold. First, it provides an environment for developing a transaction platform that requires new technologies (such as NoSQL, big data, cloud, IoT) and new architectural design (such as microservices architecture). Second, it's used as a practical application area for the proposed transaction platform. Data can be consistently sent to passengers and other stakeholders through various channels such as display screens at bus stops, on board 'next-stop signage' screen within the buses, websites and mobile apps. This would enable passengers to better plan their travel routes and minimise waiting time at bus stops thus saving millions of pounds every year [11].

Note that the main goal of transaction platform is to ensure application correctness and consistency of data – that is, data remain consistent if written to databases under transactional operations. Conversely, without transactions data can be inconsistent and would have negative impact on applications and users. For instance, in London Bus case study, information (sent to display panels at bus stops or to websites or mobile apps) could be inconsistent due to the following factors such as the frequency of updating bus timetable data, the existence of multi-version (old/new) data, etc. However, maintaining consistency has impact on performance, response time and throughput. Thus, the transaction platform is evaluated in terms of response time and throughput. Such evaluation can be attributed to tradeoff between data consistency and (high) latency [12].

The remainder of this paper is structured as follows. Section 2 reviews and analyses related work. Section 3 explains the design of the transaction-based microservice architecture under study. Section 4 describes the implementation of the execution protocol. Section 5 discusses the experiments in both synchronous and asynchronous mode. Finally, conclusion is presented in section 6.

2. Related work

In microservices architecture, a transaction spans across several services and heterogenous databases [13] that may have different underlying data models and structures. In such an environment, managing transactions that guarantee data consistency and correctness of operations involve more complex design than classical transactions with homogenous single shared database.
Existing research design general models for distributed transactions in a microservices architecture. In [14], authors build a transactional management layer which integrates datatabases with multi-versioning features. Some researchers have developed blockchain platforms [15] that can run smart contracts on separate actors (services) and execute transactions independently. These platforms are claimed to be achieving a higher scalability and throughput. Authors in [16] have developed a model named SagaMAS where every transaction is created as a microservice. It follows a semi-orchestrated asynchronous model where every agent can make a request without waiting for a response. Transactions between microservices are performed through the coordination of

their agents. However, it is not clear whether the above approaches implement database transaction models that ensure consistency of databases and correctness of applications.

Furthermore, various protocols have been developed for the execution of transactions. Authors in [17] adopt the classical two-phase commit protocol (2PC) for managing transactions in a microservice architecture. In this approach, a coordinator controls an execution of a transaction in two phases. In the first phase, participants nodes are asked if a transaction can be committed with a 'yes' or 'no' response. In the second phase, the transaction commits if every node provides a 'yes' response. If any of the nodes provides a 'no' response, then a rollback operation is executed to cancel the effects of completed operations. However, the classical 2PC protocol is less efficient and is not appropriate for large-scale and highly loaded systems [18]. 2PC protocol is also not suitable for big data and NoSQL databases that need high efficiency and scalability.

The work presented in [4] and [13] introduce SAGA pattern to enhance 2PC protocol and related communication between component systems. This work is based on a conventional SAGA project [19] which was proposed to generate software development systems automatically. Transactions based on SAGA pattern are used to update data in different services in a microservice architecture. However, SAGA pattern transactions only satisfy ACD (atomicity, consistency, durability) properties, but not ACID (atomicity, consistency, isolation, durability) [3]. This model allows to read and write data in partially completed transactions. The lack of read-isolation is solved in [5], when a transaction only commits if it has been completed successfully. Another, issue with SAGA pattern is the coordination and maintenance of transactions which work with multiple microservices. The work in [20] proposed a SAGA-based Pilot-Job with several types of applications to be used over a wide range of heterogeneous distributed infrastructures, such as Clouds and traditional Grids/Clusters.

Furthermore, authors in [21] have developed a model for long-lived transactions. This model is called saga transaction model where long transactions are split into a series of sub-transactions that confit in sequence. But the constraint is that all sub-transactions must be successfully completed. If any of them fails the completed sub-transactions need to be rolled back through the execution of compensation transactions (or actions). This model was developed for classical (relational) databases and it may not work for NoSQL big data systems which have different characteristics from classical databases.

3. The transaction platform

This section presents the proposed transaction platform for microservices-based big data systems. First, it presents the overall architecture and main components of the transaction platform. It then illustrates different aspects of the design of storage layer with big databases.

3.1. The architecture

The architecture of the proposed transaction platform is shown in Fig. 1. It comprises different layers which include: client layer, application layer, storage layer and data source layer. It is based on our previous work [6]. Here, we extend that architecture by managing data at data storage layer using transactions and microservices. Moreover, data need to be adapted before their storage in a specific NoSQL (key/value) database such as

Redis, etc. The goal is to manage a polyglot of databases in order to support a wider variety of data storage needs.

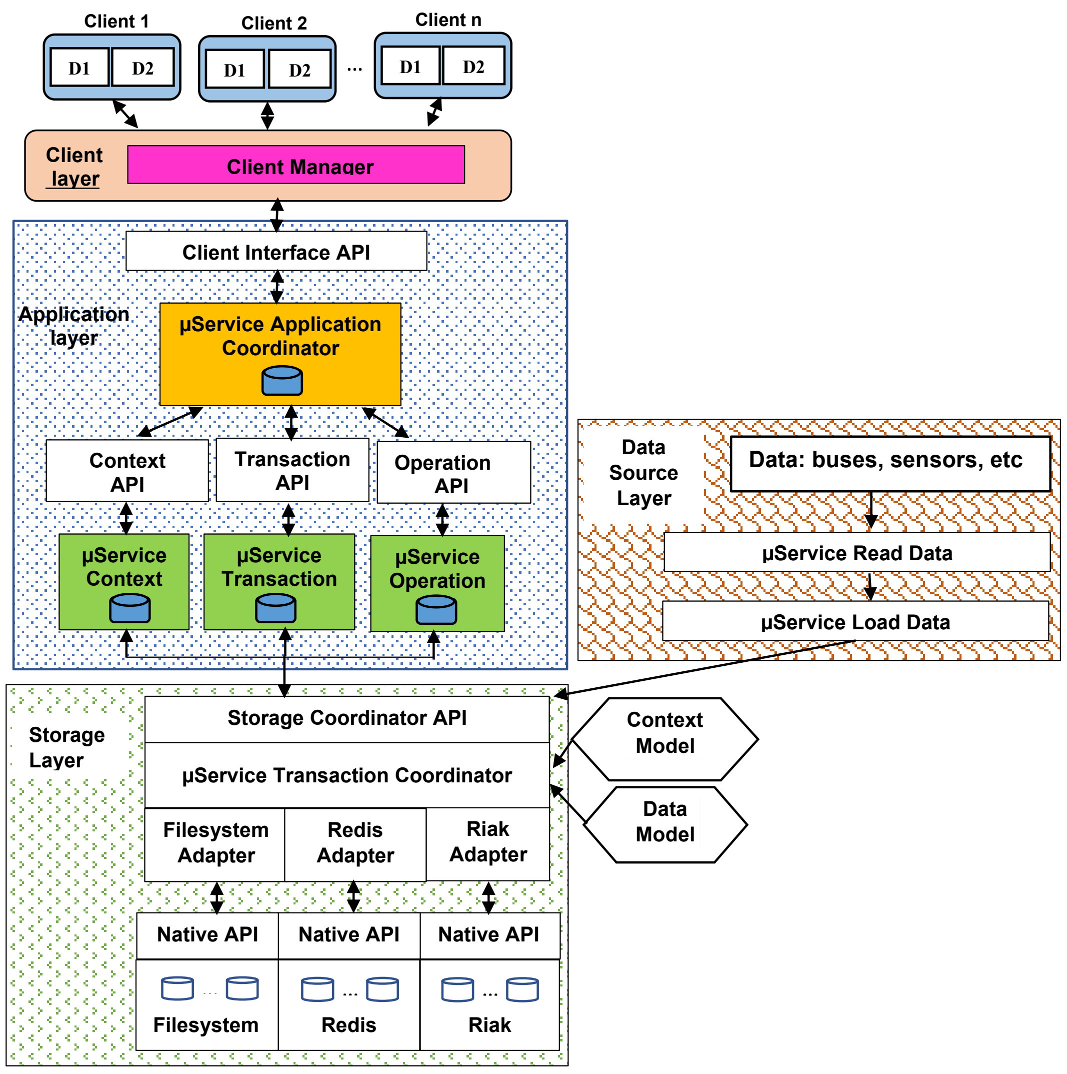


Figure 1. A transaction management system.

a) **Client Layer:** This layer serves as an interaction between client and application layer. It includes a client manager which receives and sends client's requests to the Application Layer. It also gives a response to the client whenever the Application Layer sends back the requested results.

b) **Application Layer (AL):** This layer comprises the following components:
   - **Client Interface API:** It receives requests from the Client Layer and sends to the Application Layer. It also provides responses from the Application Layer which are to be sent to the Client Layer.
   - **µService Application Coordinator:** This module receives and executes client's requests. For every request, it sets an execution context (Context API), whenever an operation is to be executed as a transactional (Transaction API) or non-transactional (Operation API). It will then be run according to its (consistency)

requirements. During this process, a client's request could access any NoSQL database using the *Storage Coordinator API* from the *Storage Layer*.

- **µService Context (Dependencies and Semantic Rules):** It receives, analyses and stores different semantic needs associated with client's requests.
- **µService Operation:** It receives, executes and control the execution of a non-transactional client's requests.
- **µService Transaction:** It receives, executes and control the execution of a transactional client's requests.

c) Storage Layer: It applies transaction management techniques at data source's side at the storage layer. Therefore, CRUD operations for the data to be stored or the ones that come from data sources (sensors, buses, web, etc.) are managed according to transactional criteria.

Data storage deals with databases and stores data that come from *Application Layer* (i.e., data from user's requests) and *Data Source Layer* (i.e., data from sensors, buses, web, etc). A data system is considered more reliable and trustful as it stores one or more instances of the same database type which coexist in case of database failures. Moreover, several copies of the same data will be recorded along different instances or nodes. The purpose is to provide a high data availability and consistency. This process involves several components, each with different roles.

- **Storage Coordinator API:** It provides a communication interface between the µService Load Data and the Application Layer with the µService Storage Transaction Coordinator.
- **Storage User Transaction Coordinator:** It controls the storage of data along one or more database instances at once. In general, data are copied along instances (or nodes) of the same database type (e.g., Redis). But data can also be copied along different (polyglot of) databases (e.g., Redis, Riak, etc). One of the main features of this module is the creation of a transaction in charge of managing and guaranteeing the entire storage process of data.
- **SQL/NoSQL DataBase Adapter:** It adapts source data to the requirements of a database where data are stored. Data adaptation is needed as there are differences in datatypes, schemas, etc., between different databases. Therefore, it is highly likely that source data suffers from different structure alterations (reorganization, modification, deletion, or datatype changes) depending on the type of databases where data are stored. Moreover, context information could be crucial in some client's requests execution environments, i.e., semantic data information (ranges, relationships, conditions, etc). In consequence, it is necessary to deal with information in relation to data type fields and context. The interaction with a specific database is carried out through its own *Native API*.

d) Data Source Layer: It represents different sources of data (e.g., sensors, buses, etc.) from where data (with high velocity) are received. Raw data from external sources are read with a µService Read Data, which are then sent to Storage Layer through µService Load Data.

3.2. Design of a storage layer

The main components of storage layer and their inter-communication flow are represented in Fig. 2. It follows most common model of microservices [22], in which interaction is carried out through REST requests. Data are sent to or read from the service (Controller/Service). When data are sent (post), a write operation is performed in more than one database thus creating multiple copies of the same data. The purpose is to increase system resilience and data availability. Nevertheless, every database works autonomously, i.e., it does not depend on others. Therefore, the goal is to design a model to be in charge of performing write/read operations as part of a transaction. It is made of a coordination process (Cluster Model) to control all database outcomes and to provide a unique response to the user (via Transaction Manager). Every database status and response are controlled with another model (Server Model).

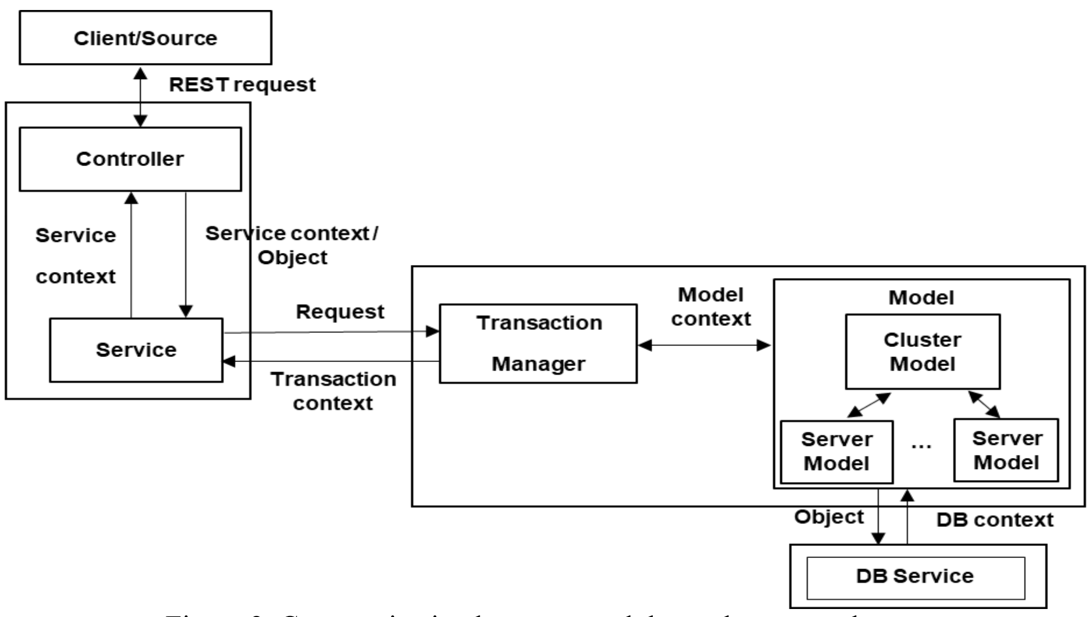


Figure 2. Communication between modules at the storage layer.

3.3. Coordination in a cluster of databases

In the storage layer, a set of databases (N) are grouped in a cluster, which is coordinated according to a cluster's databases model (Fig. 3). Read and write operations are executed as a transaction. For instance, if a write operation writes data to a database, then the same data should be written to every database in the cluster. This is to ensure consistency of data and correction of an application. If failure occurs, then partially completed operations are managed as described in the following sub-sections of fallback and rollback.
Coordination among a cluster of databases applies a quorum policy to ensure a consistent view of data. Indeed, many consistency protocols use a quorum-based principle based on the intersection property [23]. NoSQL databases such as Dynamo, Cassandra, Voldemort and Riak also follow this technique. In our work, the consistency is guaranteed by applying the majority quorum protocol [24] which is explained as follows.

(i) Write operation (denoted w) is used to write data to a database in a cluster. A write operation (w) is considered successful if consistency can be guaranteed under a quorum policy, i.e., data have been written to at least half of the databases that belong to a cluster.

(ii) If N (number of databases) is odd, then a quorum is considered as (N/2)+1. Otherwise, it is considered as N/2. The strongest consistency is reached when data are written to the whole cluster (i.e., w = N). For instance, if N=3 then a quorum is achieved when a write operation is performed in 2 (3/2+1) out of 3 databases, and the strongest consistency is guaranteed when data are written in the three databases (w = 3).

(iii) In relation to a successful execution of a write operation and maintenance of consistency, quorum policy defines three different scenarios. If a write operation is fully completed (i.e., w=N), then it is considered as totally successful (OK_Full_Cluster). If a write operation is executed such that only a quorum is achieved (as above), then it is partially successful (OK_Cluster_Dirty). If quorum cannot be achieved, then it is considered as not successful (Error_Cluster_Dirty).

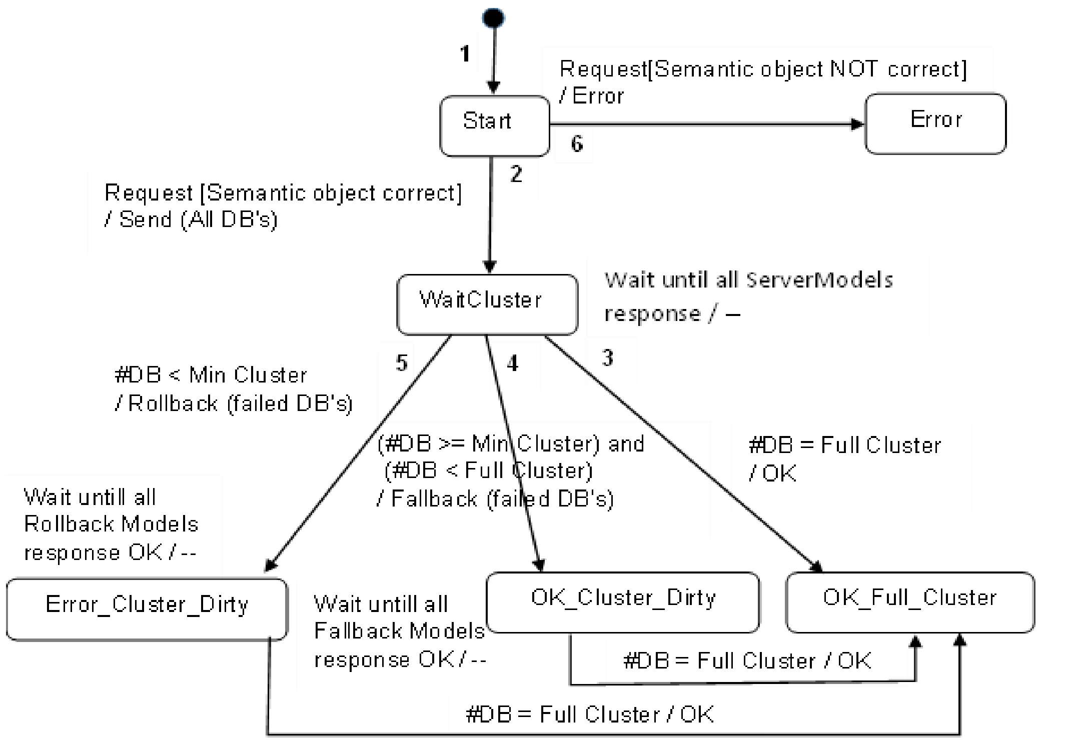


Figure 3. Cluster's databases model.

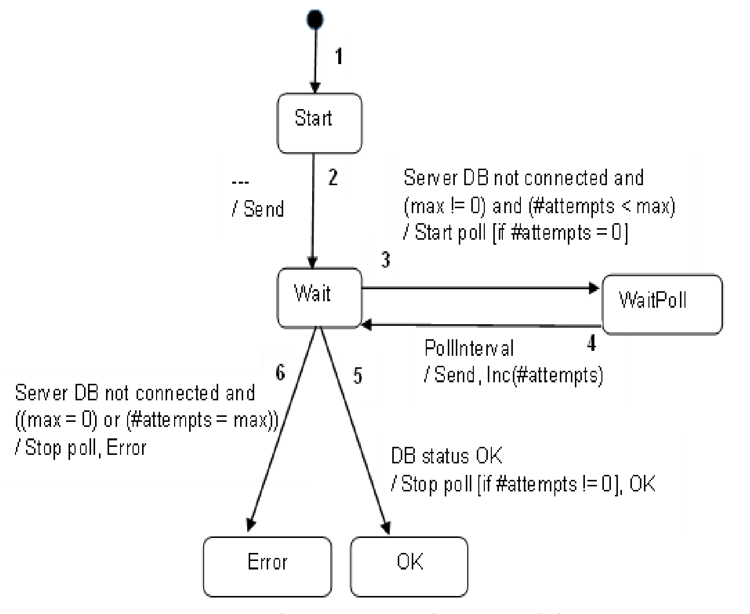
Steps involved in Fig. 3 are represent in Algorithm 1.

Algorithm 1. Algorithm for cluster's databases coordination.

```
//Receiving data from a REST request
//Setting parameters
data = {Set of data from a data source or user's request}
status = Start
DBi = Database [i = 1 … N]
num_responses = 0
quorum = N / 2

//data semantically not correct
if ¬ [data semantically correct]
    then
    status = Error;
    Send an error; Exit.
endif

//data semantically correct
data = [semantically correct]
status = WaitCluster
for i = 1 … N
   Send data to DBi (∈ cluster)
   Wait (get response from DBi)
    num_responses = num_responses + 1
endfor
//Request totally successful
if num_responses = N
     then status = OK_Full_Cluster; Send OK; exit
endif

//Request partially successful
if ((num_responses >= quorum) and (num_responses < N))
     then start a Fallback process for each failure DB
            status = OK_Cluster_Dirty
            Wait until all Fallback processes respond OK
            status = OK_Full_Cluster; send OK; exit
endif

//Request not successful
if (num_responses < quorum)
then start a Rollback process for each successful DB
           status = Error_Cluster_Dirty
           Wait until all Rollback processes respond OK
           status = OK_Full_Cluster; send OK; exit
endif
```

3.4. Write/Read Operations

The write/read operations on each database are performed according to Server Model which is depicted in Fig. 4.



1
Start
2

/ Send
Server DB not connected and
(max != 0) and (#attempts < max)
/ Start poll [if #attempts = 0]
3
Wait
WaitPoll
PollInterval
/ Send, Inc(#attempts)
4
6
Server DB not connected and
((max = 0) or (#attempts = max))
/ Stop poll, Error
5
DB status OK
/ Stop poll [if #attempts != 0], OK
Error
OK

Figure 4. Database model.

Data are written to or read from each database. If a database is unavailable for writing/reading data, a poll mechanism is sparked. Under the poll mechanism, write/read operation is re-executed for a specific interval of time and for a maximum number of attempts. If a maximum number is reached, the write/read operation is considered as temporarily failed. Steps involved in Fig. 4 are represented in Algorithm 2.

Algorithm 2. Server Model algorithm.

```
//Receiving data from the Cluster Model
//Writing/Reading data in a specific database DBi = Database [i = 1 … N] (∈ cluster)

//Setting parameters
status = Start
max = Maximum number of attempts for writing/reading data
#attempts = 0
pollInterval = Specific number of seconds

//Sending process
Write to/Read from DBi
status = Wait
Wait for a response

//No response
if ¬ [DBi respond] and (#attempts = 0)
    then Start a poll to check DBi availability
endif
while ¬ [DBi available] and (#attempts < max) do
        status = WaitPoll
        Wait pollInterval seconds
        Write to/Read from DBi
        #attempts = #attempts +1
        status = Wait
endwhile

//DBi is available or #attempts is over
Stop poll
if DBi available
    then status = OK
        return OK
    else status = Error
        return Error
endif
```

3.5. Fallback model

A fallback model is designed to cater for situation when a quorum is achieved but some databases are still unavailable. Then, a write/read operation keeps trying to be completed until unavailable database becomes available or active. For this reason, a poll mechanism starts in order not to limit the number of attempts to execute write/read operations. This is based on the assumption that an unavailable database is failed temporarily and would eventually recover and become active. The model is represented in Fig. 5.

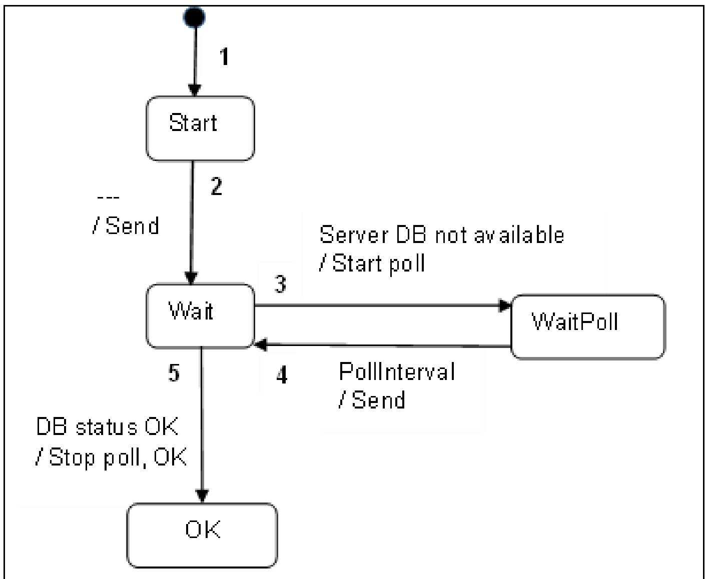


Figure 5. Fallback state transition model.

Steps involved in Fig. 5 are represented in Algorithm 3.

Algorithm 3. Fallback algorithm.

```
//Receiving data from the Cluster Model
//Writing/Reading data in a specific database DBi = Database [i = 1 … N] (∈ cluster)

//Setting parameters
status = Start
pollInterval = Specific number of seconds

//Sending process
Write to/Read from DBi
status = Wait
Wait for a response

//No response
if ¬ [DBi respond]
    then Start a poll to check the availability of DBi
endif
while ¬ [DBi available] do
        status = WaitPoll
        Wait pollInterval seconds
        Write to/Read from DBi
        status = Wait
endwhile

//DBi is available
Stop poll
status = OK
return OK
```

3.6. Rollback model

A rollback model deals with a situation when a quorum is not achieved. In this case, a write/read operation on a cluster of databases is cancelled. If a write operation is already performed on a specific database, then a rollback process is activated which cancels the effects of completed operation in order to bring back that database to a previous state (as a compensation mechanism). If a read operation is executed, then the operation is omitted. This model also considers the possibility that a database becomes unavailable before a cancellation process starts. Therefore, a poll mechanism is sparked with a limitless number of attempts (as in fallback model). This model is depicted in Fig. 6.

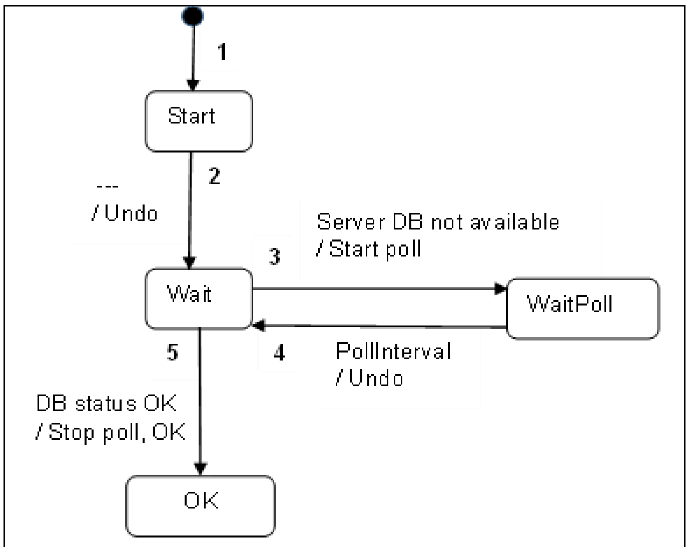


Figure 6. Rollback state transition model.

Steps involved in Fig. 6 are represented in Algorithm 4.

Algorithm 4. Rollback algorithm.

```
//Receiving data from the Cluster Model
//Removing data from a specific database DBi = Database [i = 1 … N] (∈ cluster)

//Setting parameters
status = Start
pollInterval = Specific number of seconds

//Sending process
Remove data from DBi
status = Wait
Wait for a response

//No response
if ¬ [DBi respond]
    then Start a poll to check the availability of DBi
endif
while ¬ [DBi available] do
        status = WaitPoll
        Wait pollInterval seconds
        Remove data from DBi
        status = Wait
endwhile

//DBi is available
Stop poll
status = OK
return OK
```

4. Execution protocol

This section describes an overall execution protocol for different scenarios explained in section 3. Fig. 7. depicts the main steps of the execution protocol. When a new request arrives, it is managed by Transaction Manager (TM). TM creates a general transaction context (Cluster Model (CM)), and a general Cluster Model Transaction (CMT) linked to it. Each main transaction is comprised of several subtransactions, each of which is associated with a database. Thus, a new subtransaction context (Server Model (SM)) is created for each database service that belongs to a cluster, and a subtransaction Server Model Transaction (SMT) linked to it. Subtransaction's contexts are included as part of the main or general transaction context CM. Therefore, we define a Cluster Model Transaction (CMT) that consists of a set of subtransactions called Service Model Transaction (SMT). That is, CMT = (SMT_1, SMT_2, …, SMT_N), where N is the number of databases which are part of the cluster. Every SMT_i ($1 \leq i \leq N$) can perform a write/read operation in a specific key/value NoSQL database i.

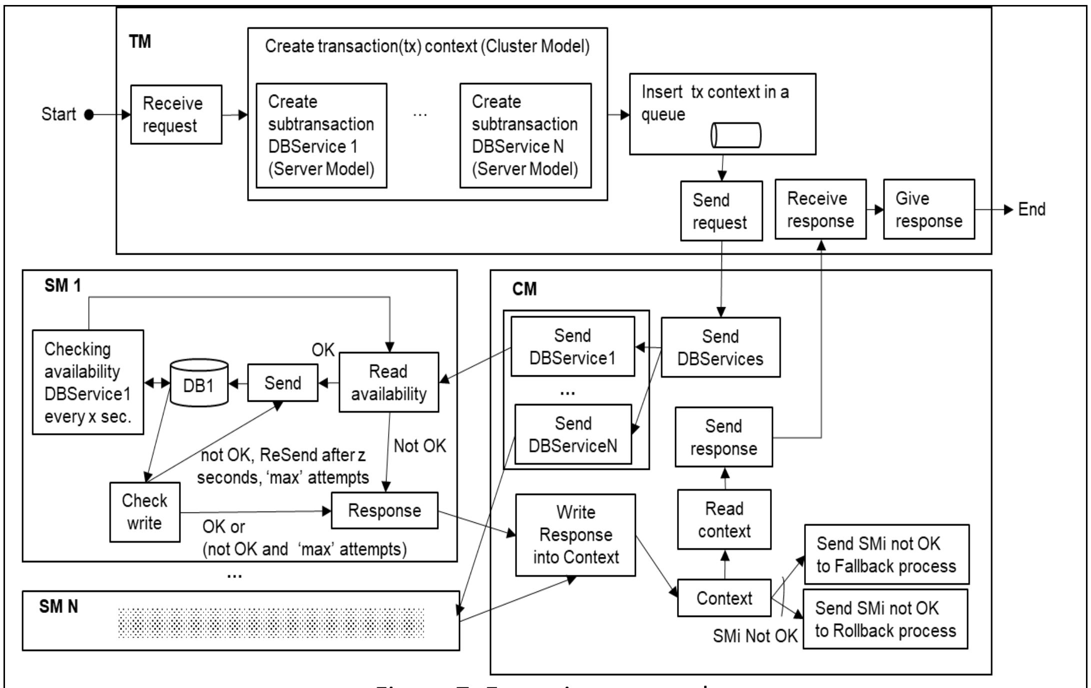

TM
Start
Receive request
Create transaction(tx) context (Cluster Model)
Create subtransaction DBService 1 (Server Model)
...
Create subtransaction DBService N (Server Model)
Insert tx context in a queue
Send request
Receive response
Give response
End
SM 1
Checking availability DBService1 every x sec.
DB1
Send
OK
Read availability
Not OK
not OK, ReSend after z seconds, 'max' attempts
Check write
Response
OK or (not OK and 'max' attempts)
...
SM N
CM
Send DBService1
...
Send DBServiceN
Send DBServices
Send response
Read context
Write Response into Context
Context
Send SMi not OK to Fallback process
Send SMi not OK to Rollback process
SMi Not OK

Figure 7. Execution protocol.

Then, the general context CM is sent to TM and is stored in a queue which is dedicated to saving transaction contexts. Finally, the system waits for a response from TM. CM sends a request to every SM, which provides a response to CM after a read/write operation is executed. A final result is then read from CM and is sent to TM. It has to be noted that every SM works autonomously and asynchronously in relation to others. Indeed, every database service has a process linked to it, which knows the state of the database through periodical checks, i.e., every 'x' seconds. When a SM receives a request, the state of the database is read first into a corresponding process. If it is not active, a negative response is sent immediately to the CM. Otherwise data is sent to the database. This event prevents write operations from being performed in a specific database service in case of temporary unavailability. However, if data could be stored, a positive response (OK) is sent to CM. Otherwise, a response is resent every 'z' seconds for a maximum of 'max' attempts. In final scenario, if data could not be stored, then a 'not Ok' response is sent and a fallback/rollback process is created and stored in a specific queue.

Initially, every database service has two queues linked to it, one for storing fallback processes and another for storing rollback processes. So, every time a CM starts an asynchronous Fallback or Rollback process for every SM which could not store data, it is stored in a corresponding queue. The type of context to be created depends on the number of positive responses, as explained in Section 3.4. At the end, a final response from every database is registered in the general transaction context, CM.

The implementation of a fallback queue process is represented in Fig. 8 This process is applicable to every unavailable database service. An automatic process checks every 'y' seconds if a database service is available, and if there are SM's in the list of fallbacks. If the response is positive, then the list is locked to avoid new SM's from being entered into the list. Every data in the SM list is then sent to the database to be stored. If data could be stored, then SM is removed from the list, otherwise it will remain in the list. This process keeps processing elements from the list until it is fully completed.

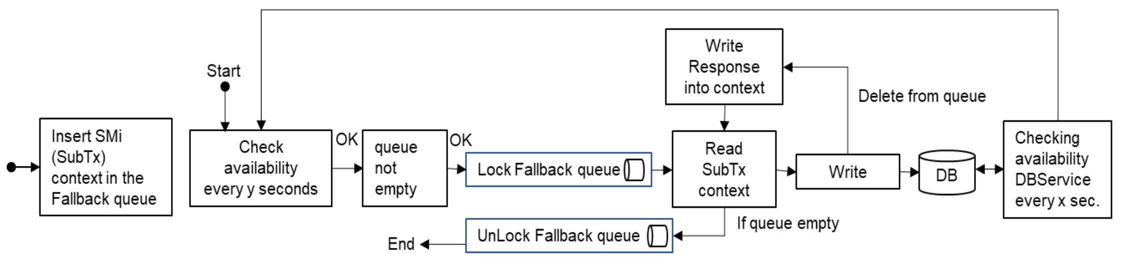

Start
Insert SMi (SubTx) context in the Fallback queue
Check availability every y seconds
OK
queue not empty
OK
Lock Fallback queue
Write Response into context
Read SubTx context
Delete from queue
Write
DB
Checking availability DBService every x sec.
If queue empty
UnLock Fallback queue
End

Figure 8. Fallback queue process.

The implementation of a rollback queue process is represented in Fig. 9. This is applied to every available database service. It follows the same procedure as that of the fallback queue. However, in rollback process data are deleted from a database.

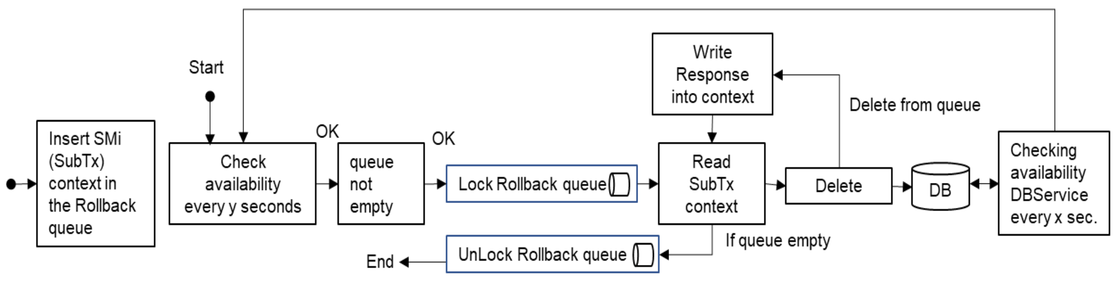

Start
Insert SMi (SubTx) context in the Rollback queue
Check availability every y seconds
OK
queue not empty
OK
Lock Rollback queue
Write Response into context
Read SubTx context
Delete from queue
Delete
DB
Checking availability DBService every x sec.
If queue empty
UnLock Rollback queue
End

Figure 9. Rollback queue process.

5. Simulation-based evaluation

This section presents simulation-based evaluation of the transaction platform. It describes a set of simulation experiments in order to evaluate performance, response time, and consistency of the storage layer.
The experiments have been carried out using a machine with the hardware/software specifications, shown in Table 1.

Table 1. Hardware/software features.

Hardware/software specification	
CPU core	2.4 GHz Intel(R) Core(TM) i7-5500
Operating system	Windows 10 Pro 64 bits
IDE	Eclipse 2020-06
Programming language	Oracle Java 7
Database	NoSQL key/value, Redis 6.2.3
Containerization technology	Docker Engine v20.10.10

Experiments simulate a scenario of data processing using London Bus service as a case study. The simulation data (in JSON format) come from Open Data source that the Council of London provides from a large public transport fleet. We choose the London Bus service and such data in our experimentation for the following reasons. There are more than 9000 buses which operate across more than 600 routes. Such buses generate a large volume of data [25] which arrive at the system in real time at different intervals of time, different speed and with different data structure. In addition, it also justifies the practical application of the transaction platform. For example, data can be consistently sent to passengers and other stakeholders through various channels such as display screens at bus stops, on board 'next-stop signage' screen within the buses, websites and mobile apps. This would enable passengers to better plan their travel routes and minimise waiting time at bus stops thus saving millions of pounds every year [11].

5.1. Data representation and storage

The "Transport for London" (TfL) [9] provides information in relation to arrival prediction times for different bus lines which follow a bus spider maps [26]. Fig. 10 shows a data record with different fields and a response sample.

```
[ { "id": "string",
    "operationType": 0,
    "vehicleId": "string",
    "naptanId": "string",
    "stationName": "string",
    "lineId": "string",
    "lineName": "string",
    "platformName": "string",
    "direction": "string",
    "bearing": "string",
    "destinationNaptanId": "string",
    "destinationName": "string",
    "timestamp": "2022-07-14T16:27:17.473Z",
    "timeToStation": 0,
    "currentLocation": "string",
    "towards": "string",
    "expectedArrival": "2022-07-14T16:27:17.473Z",
    "timeToLive": "2022-07-14T16:27:17.473Z",
    "modeName": "string",
    "timing": {
      "countdownServerAdjustment": "string",
      "source": "2022-07-14T16:27:17.473Z",
      "insert": "2022-07-14T16:27:17.473Z",
      "read": "2022-07-14T16:27:17.473Z",
      "sent": "2022-07-14T16:27:17.473Z",
      "received": "2022-07-14T16:27:17.473Z"  }  } ]
```
Figure 10. A response sample for arrival times in a specific line.

A subset of data from arrival prediction record was taken as a reference for experimental purposes. Moreover, a bus line is covered with more than one vehicle (bus). Fig. 11 represents a generic bus line with 'm' bus stops, and 'n' buses travelling through the same line in both directions, 'inbound' and 'outbound'.

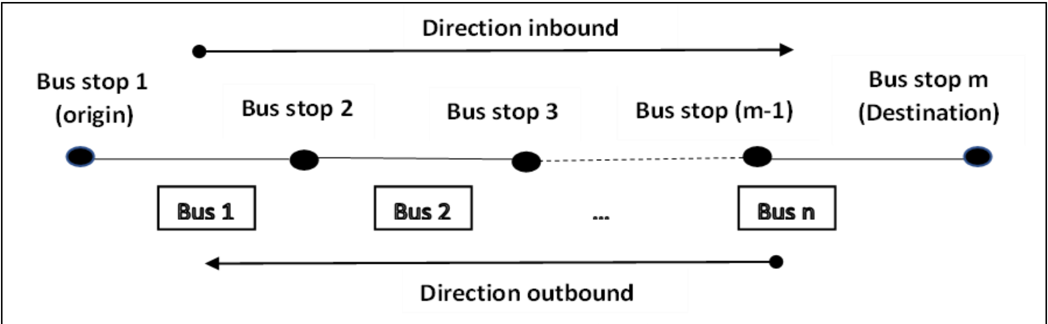


Figure 11. A generic bus line.

In our experiments, every bus is associated with the following data, which is a simpler and cleaner version from the one used in TfL: a bus identifier ('vehicleId'); origin station from where a bus departs at a specific instant of time, the direction ('direction'), and the position from the last origin station ('currentLocation') according to the direction (inbound or outbound).

Bus data are stored in a cluster made of three containers, with a Redis NoSQL key/value database per container. Therefore, data need to be adapted to a key/value format. Indeed, for every bus i ($1 \leq i \leq n$) and every movement j ($1 \leq j$), a key is built up as "Bus"+i+"-M"+j, in order to be unique at every stage of its journey. Secondly, value linked to the key and the bus i, is built up as a union of three data: origin, direction and location of the bus i at a requested instant. Such data are a reduced version of the overall data.

Both write/read operations are performed with a transaction protocol as it is explained in section 4. Then, a Cluster Model Transaction (CMT) and a set of Service Model Transactions (SMT) are created. That is, CMT = (SMT_1, SMT_2, …, SMT_N), where N is the number of databases which are part of the cluster. According to our software specification, every SMT_i ($1 \leq i \leq N$) refers to a write/read operation in a specific Redis NoSQL database i. For instance, a write operation can write data in order to update the current location of a bus on a spider map, or read the database to find out the arrival time of a bus to a bus stop.

5.2. Simulation of the communication process

A general communication process is represented in Fig. 12. It involves (i) sending a request to the service at the storage layer (interval [1, 2]), (ii) processing of request by the microservice (interval [2, 3]), and (iii) sending a response back to the user (interval [3, 4]). When the request arrives at microservice, then a read or write operation has to be performed in the respective databases. If the microservice is fully active, then the operation is executed successfully, and a positive response is returned. Nevertheless, if the microservice is not working properly at full capacity, the microservice needs to be aware of the new faulty situation, acts accordingly, and keeps the user informed with a negative response.

We devise the metrics to empirically evaluate the transaction platform with respect to the response time (i.e., response to Application/Data Source Layer) and throughput under different scenarios such as normal, fallback and rollback. Response time of storage coordinator (denoted RSC) (interval [1, 4]) is calculated using parameters such as Time of Data Request (denoted TDR) (interval [1, 2]), Time of a Write/Read operation in a database (denoted TWR) (interval [2, 3]), Time of a database in a Failure state (denoted TF) (interval [2, 3]), and the Time of a Request Response (denoted TRR) (interval [3, 4]).

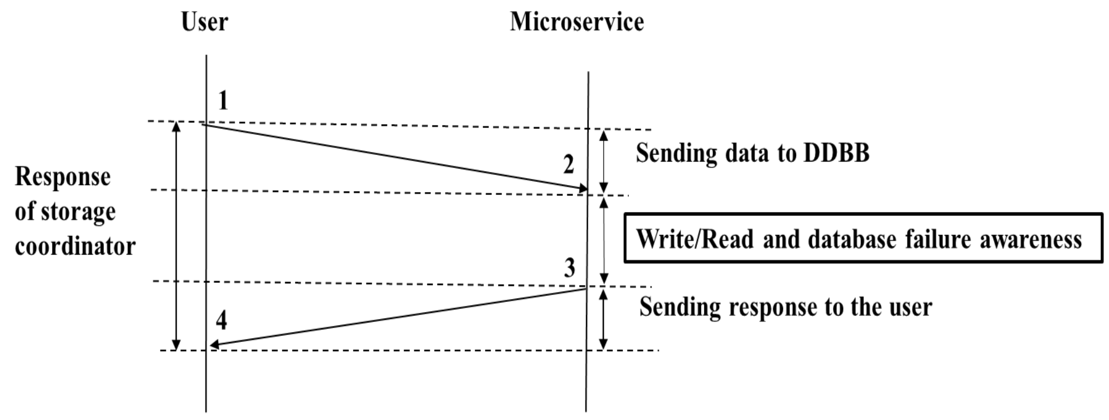

User
Microservice
1
2
3
4
Response of storage coordinator
Sending data to DDBB
Write/Read and database failure awareness
Sending response to the user

Figure 12. A sequence diagram for a request.

This study is focused on how the storage layer manages requests and database failures (interval [2, 3]). This fact establishes how good the quality of the response from the transaction microservice-based architecture is at this level. Therefore, this process has an impact on the response of the storage coordinator, i.e., the total amount of time spent to give back a response to a client at the storage layer since a request is sent (interval [1, 4]).

Note that the main goal of transaction platform is to ensure application correctness and consistency of data – that is, data remain consistent if written to databases under transactional operations. Conversely, without transactions data can be inconsistent and would have negative impact on applications and users. For instance, in London Bus case study, information (sent to display panels at bus stops or to websites or mobile apps) could be inconsistent due to the following factors such as the frequency of updating bus timetable data, the existence of multi-version (old/new) data, etc. However, maintaining consistency has impact on performance, response time and throughput. Thus, the transaction platform is evaluated in terms of response time and throughput. Such evaluation can be attributed to tradeoff between data consistency and (high) latency [12].

5.2.1. Metrics for a normal scenario

It considers a scenario where every write/read request is sent to every database in a sequential order, and the cluster is made of 'p' databases. RSC is calculated as follows.

$$\mathbf{RSC_{normal}} = \mathbf{TDR} + \sum_{i=1}^{p} \mathbf{TWR}_i + \mathbf{TRR} \qquad (1)$$

TDR is time taken to communicate a request from Application Layer/Data Source Layer to Storage layer. TWR_i is time to send data to a database 'i' and do write/read operation. TRR is time taken to send a response to a request.

In case of database failures, if ‘q’ out of ‘p’ is the number of databases which are not available, then RSC is calculated as follows.

$$\mathbf{RSC_{failure}} = \mathbf{TDR} + \sum_{i=1}^{p-q} \mathbf{TWR}_i \qquad + \mathbf{\Delta t1} \ + \mathbf{TRR} \tag{2}$$

Where Δt1 is the time to manage ‘q’ databases which are unavailable.

$$\mathbf{\Delta t1} = \sum_{i=1}^{q} \mathbf{TF}_i \tag{3}$$

TF_i is the time taken in re-executing write/read operations, i.e., to keep trying to write/read data every ‘z’ seconds for a maximum of ‘max’ attempts in database, i.

$$\mathbf{TF_i} = \sum_{j=1}^{max} (\mathbf{z} + \mathbf{TWR_i}) \tag{4}$$

5.2.2. Metrics for a fallback scenario

In the case of fallback scenario, the time for a write/read operation is calculated as follows.

$$\mathbf{RSC_{fallback}} = \mathbf{RSC_{failure}} + \mathbf{\Delta t2} \tag{5}$$

Where Δt2 is the time taken to provide a positive response when unavailable (failed) databases recover. If ‘q’ is the number of unavailable databases then time taken to recover such databases (in sequence) is calculate as follows.

$$\mathbf{\Delta t2} = \sum_{i=1}^{q} \mathbf{TFF}_i \tag{6}$$

TFF_i is the Time that a failed database remains in a Fallback process. Suppose, that a database ‘i’ becomes active after ‘r’ intervals, with a gap of ‘y’ seconds between each interval. If a fallback queue contains ‘s’ requests, then TFF_i is calculated as follows.

$$\mathbf{TFF_i} = \sum_{j=1}^{r} \mathbf{y} + \sum_{j=1}^{s} \mathbf{TWR}_i \tag{7}$$

5.2.3. Metrics for a rollback scenario

If the case of a rollback procedure, the time taken to complete a ‘cancel’ (or delete) operation is calculated as follows.

$$\mathbf{RSC_{fallback}} = \mathbf{RSC_{failure}} + \mathbf{\Delta t3} \tag{8}$$

Where Δt3 is the time taken to provide a positive response when ‘q’ databases recover.

$$\mathbf{\Delta t3} = \sum_{i=1}^{q} \mathbf{TFR}_i \tag{9}$$

TFRi is the time that a failed database remains in a Rollback process. Suppose that a database ‘i’ becomes active after ‘r’ intervals, with a gap of ‘y’ seconds between each interval. If a fallback queue contains ‘s’ requests, then TFRi is calculated as follows.

$$\mathbf{TFR_i} = \sum_{j=1}^{r} \mathbf{y} + \sum_{j=1}^{s} \mathbf{TD}_i \tag{10}$$

TD_i is the time to send a delete request to a database ‘i’ in order to remove a request located in position ‘j’ in a queue.

5.2.4. Metrics for throughput

The throughput of storage coordinator, ThSC, is the number of requests per second which are processed by the storage layer. However, if requests are sent in sequential order, it considers the instant of time taken to send a request (denoted TRS). If there are ‘n’ requests, then ThSC is calculated as follows.

$$\mathbf{ThSC} = n / (\mathbf{TRS}_n - \mathbf{TRS}_1) \tag{11}$$

5.3. Simulation process

The aim of every experiment is to measure the load of the system under two perspectives, both response time and throughput of the storage layer. Therefore, the system is tested with different number of buses. Specifically, the number varies at the power of two (1, 2, 4, 8, …, 512). Table 2 represents a set of parameters used during the experiments.

Table 2. A set of experimental parameters.

Parameters	Meaning
NUMBER_BUSES	Number of buses
NUMBER_STATIONS	Number of bus stops
DISTANCE_BETWEEN_STATIONS	Distance between bus stops
LENGTH_ROUTE	Length of the route
INTERVAL_BETWEEN_BUSES	Distance between buses along the journey
NUMBER_OF_MOVEMENTS	Total number of times that buses are moving along the bus line
DISTANCE_PER_SECOND	Number of metres a bus moves per second
TIME_TO_MOVE_FORWARD	Time a bus spends moving every time is ordered to move
PERIOD_LAUNCH	Interval of time during which all buses have to be moved

During experimentation, the simulation of NUMBER_BUSES movements is performed. A journey takes place along the same bus line from an origin to a destination and vice a versa. Every second all buses data are sent to the system evenly distributed in time, i.e., each bus data are sent every PERIOD_LAUNCH / NUMBER_BUSES seconds. Therefore, a new location for every bus is calculated every second. So, the corresponding set of pairs (key, value) are built up, and sent to the system to be stored in the cluster.
Experiments consider two modes of sending requests; sequential and parallel. A study on both modes has been analysed under three scenarios per mode: no database failures, fallback, and rollback.

5.4. Experimentations of sequential order

In this mode, buses' data are sent in sequential order. Therefore, the interval between requests is the sum of two values, i.e., system response time per bus data and the gap time until the next bus data are sent (PERIOD_LAUNCH / NUMBER_BUSES). Nevertheless, this fact implies that the bus rate per PERIOD_LAUNCH is lower, because it depends on the response time of the storage layer per bus. For instance, if it is required to send 4 buses per second, but the storage layer response time per bus is 0.5 seconds, then the bus data rate is 2 buses data per second instead of 4.

5.4.1. Normal scenario

It considers a failure free scenario where a cluster of NoSQL key/value databases is working in a normal state. Fig. 13 shows mean response time of the storage coordinator, i.e., the time it takes to send a response to the Application/Data Source Layer when data are written to the databases. It can be observed from the experiments that response time is not severely affected with an increase in a workload (number of buses/sec). In other words, it provides a good response time as well as data consistency using transactional (read/write) operations. Fig. 13 shows that response time fluctuates between 0.04 and 0.08 sec with a mean of 0.051 sec approximately. However, when the system enters into a stable stage (i.e., 32 requests per second and onwards) then the response time remains stable around 0.04 sec.

The throughput is graphically represented in Fig. 14. It shows percentage of how many bus requests the storage layer is capable to manage per second. The graph shows a steady level of throughput with respect to an increase in the workload (i.e., number of buses/sec). If the workload is high the throughput is reduced, possibly, due to communication and processing overhead of data sources (buses, sensors, etc), and configuration parameters in both, NoSQL databases and microservices. But with higher workload, the time interval between requests is reduced to a negligible level.

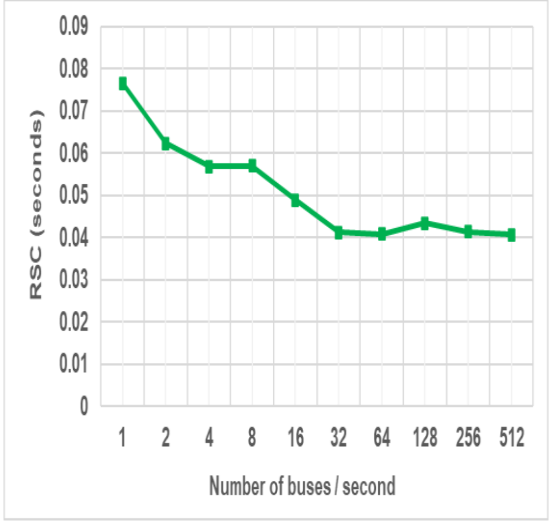


Figure 13. RSC in Normal scenario.

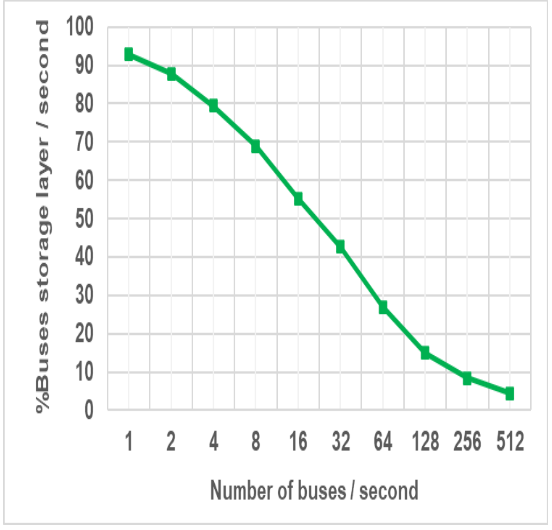


Figure 14. ThSC in Normal scenario.

5.4.2. Fallback scenario

In this scenario, the system starts its execution in a normal state. After a certain period of time, one database stops working. This then triggers fallback process to start executing, which runs for a period of 60 sec (approx). After such period (of fallback), the database starts working again. The different phases are represented in Fig. 15.

Cluster fully active	A database stops	Cluster fully active
Normal	Fallback	Normal

Figure 15. Stages in a fallback scenario.

In a fallback period, when the cluster identifies a database failure, then it keeps in memory those keys which could not be saved totally. It waits for an answer from two (working) databases. Fig. 16 shows that the response time in fallback scenario. It moves from almost 0.09 to 0.05 lower or equal to 1.0 request per second. However, it stabilises from 32 requests/sec and onwards. In relation, workload and throughput, fallback scenario is in line with the normal scenario. As discussed, for the reasons stated above, throughput goes down when workload goes high.

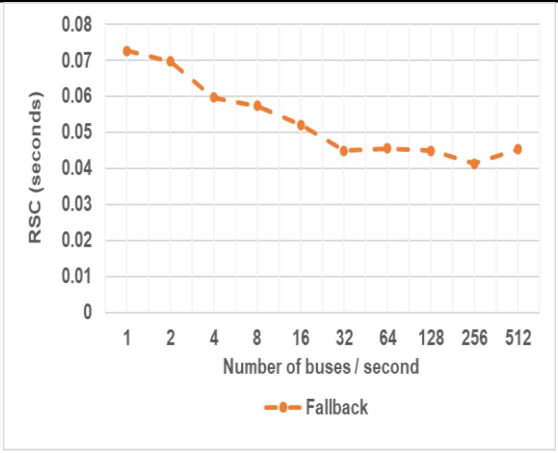


Figure 16. RSC in Fallback scenario.

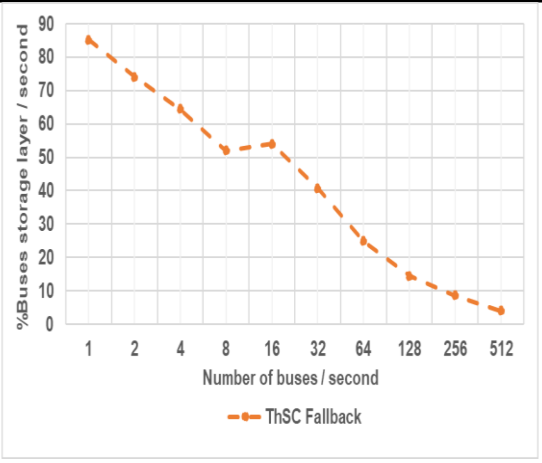


Figure 17. ThSC in Fallback scenario.

5.4.3. Rollback scenario

In this scenario, the system starts its execution in a normal state, but after a certain period of time, two databases stop working one after another. Therefore, a fallback process starts when one database stops, and a rollback process starts when second database stops. After a duration of 60 sec (approx), both databases are restarted in sequential order. A fallback process comes up again after one database gets restarted. Finally, the system recovers to its normal state when the second database is recovered. The different phases are represented in Fig. 18.

Cluster fully active	A container stops	Two containers stop	A container stops	Cluster fully active
Normal	PreFallback	Rollback	PostFallback	Normal

Figure 18. Stages in a Rollback scenario.

Fig. 19 represents the response time in rollback scenario. However, in rollback, response time fluctuates more as compared to the previous scenarios. This could be linked to the time taken in recovering two databases instead of one. In relation to throughput, rollback states seem to follow the same pattern as in previous scenarios. As the workload (16 requests/sec and onwards) increases, the percentage drops substantially.

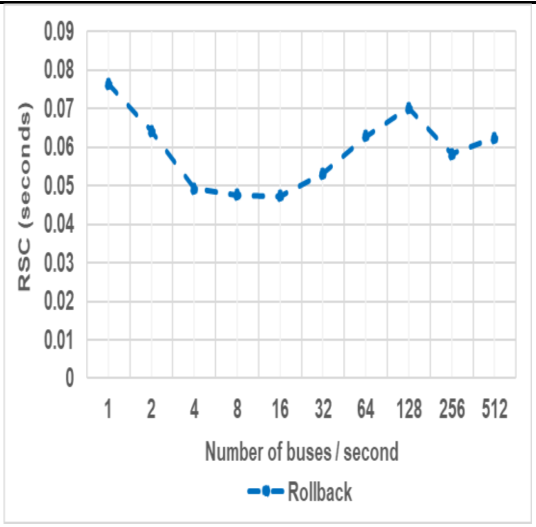


Figure 19. RSC in Rollback scenario.

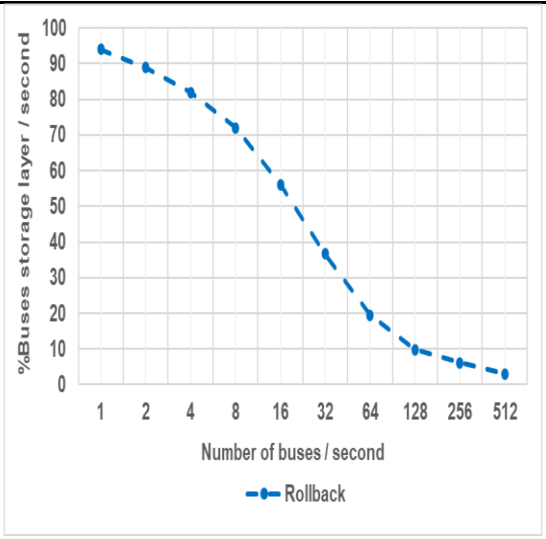


Figure 20. ThSC in Rollback scenario.

5.5. Experimentation of parallel order

In this mode, buses data are sent to the system in parallel order. It generally provides a better response time but may result in race conditions, wherein data (write) operations compete for accessing the same database in order to write or update data.

5.5.1. Normal scenario

Throughput of normal scenario in asynchronous mode is shown in Fig. 22. It shows that the throughput of the system tends to be steady with workload (up to 16 requests/sec). It then starts plummeting with workload (32 requests/sec and onwards). Therefore, the system achieves its maximum capacity between 16 and 32 requests. However, in synchronous mode, the system seems to lose capacity at a constant pace as the number of requests increases. Those numbers are correlated with the time system response (RSC) (Fig. 21). In synchronous mode, it behaves at constant rate always below 0.1 seconds in comparison to the asynchronous mode, where the delay starts increasing exponentially from 32 requests and onwards, and the system seems to be overloaded. Further observation is that RSC tends to be higher in asynchronous mode, as parallel requests are competing for same databases.

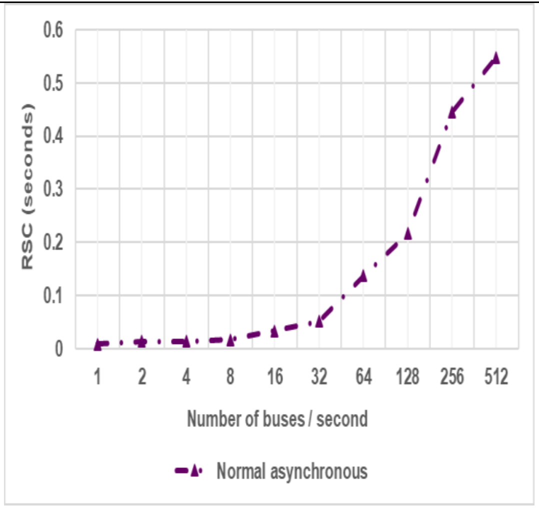


Figure 21. RSC in normal scenario.

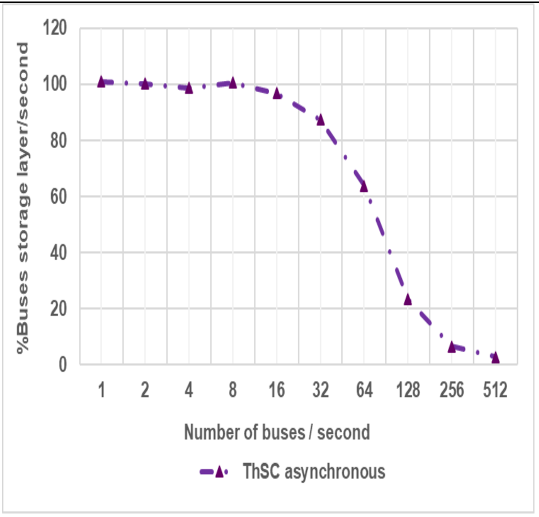


Figure 22. ThSC in normal scenario.

5.5.2. Fallback scenario

As described above, failure of one database triggers fallback process, which runs for a period of 60 sec (approx). After recovery, the failed database starts working again. In this scenario, the RSC tends to be below 0.1 seconds when workload range is between 1-64 requests/sec approximately. Nevertheless, the system starts being overloaded when workload goes higher, i.e., 128 requests/sec and onwards, where the RSC is above 0.3 seconds (Fig. 23). This fact is also reflected in the throughput (Fig. 24). The percentage of requests per second is above 50% from 1 to 64, and below 20% from 128 and onwards.

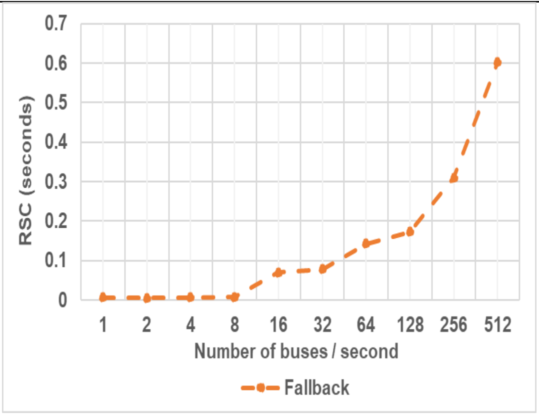


Figure 23. RSC in Fallback scenario.

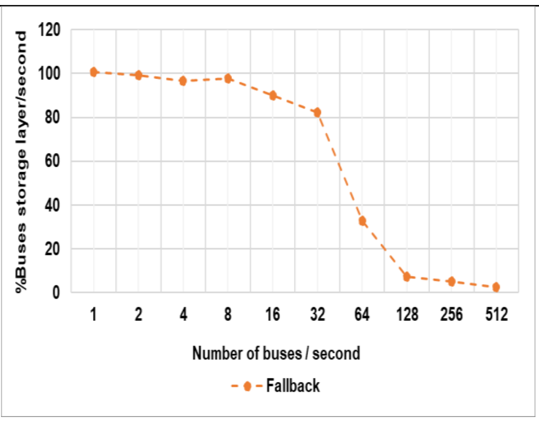


Figure 24. ThSC in Fallback scenario.

5.5.3. Rollback scenario

The execution of rollback scenario follows the same steps as described above. However, response time in rollback of asynchronous mode tends to be more unstable and fluctuates sharply.

In the rollback state, data are written to only active database. During this phase, the system seems to be affected by the increase in the number of requests (due to parallel execution). At the beginning, with workload (e.g. 2 requests/sec) is very low, and the system can manage it without incurring extra overhead. However, it can be observed that as the number of requests increases the system spends more time in providing a response (Fig. 25). Indeed, the system starts being overloaded from 64 requests per second with a throughput below 50% (Fig. 26).

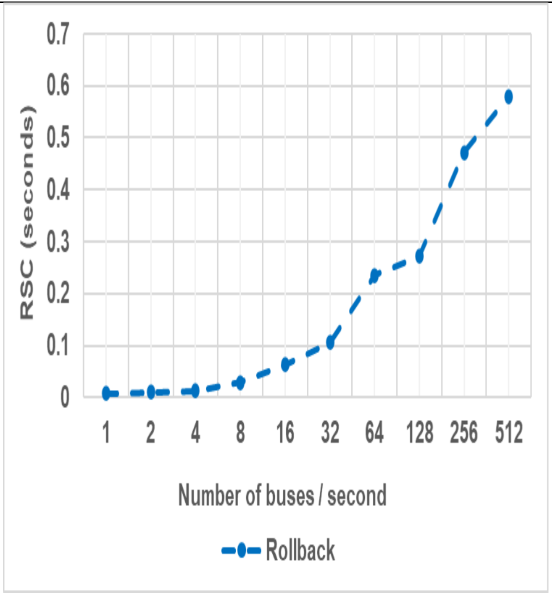


Figure 25. RSC in Rollback scenario.

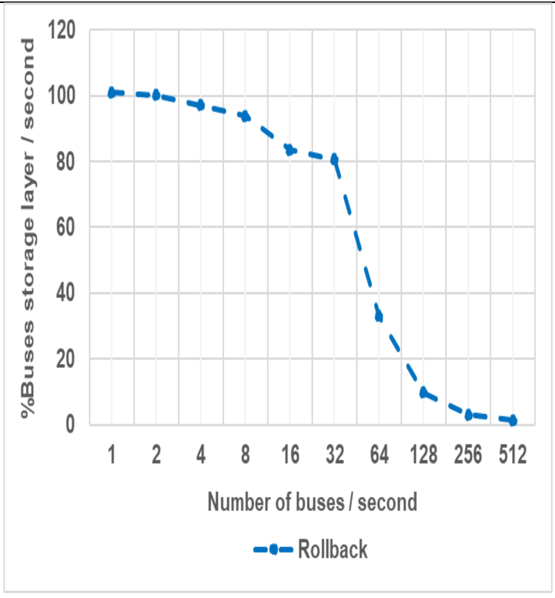


Figure 26. ThSC in Rollback scenario.

6. Conclusion

This paper has designed and developed a new transaction platform and protocols for NoSQL big databases that adopt microservices architecture. The transaction platform manages processing of big data stored in a cluster of NoSQL databases. It is evaluated through the prototype system that simulates London bus service across bus routes. The platform is evaluated through a number of simulated experiments with data from 'Transport for London' data service. Experiments are mainly conducted to evaluate and analyse the response time and throughput when database (write/read) operations are executed as transactions. The transaction platform reliabily processes database operations and enables data availability and consistency in failure-free and failure-prone environment. Thus data (e.g. bus time, location) can be consistently sent to passengers and other stakeholders through various channels such as display screens at bus stops, on board 'next-stop signage' screen within the buses, websites and mobile apps.

Our main objective was that the proposed transaction platform ensures data consistency and application correctness in a complex distributed system using modern technologies such as big data, NoSQL databases and microservices architecture and that it works in a (real) application setup, e.g., London bus service. Moreover, the proposed transaction platform may not be efficient in terms of response time and throughput while using transactions and maintaining data consistency and application correctness. Various factors contribute to the performance of transactions such as underlying network, (NoSQL) databases, transactions delay (write/read operations), and other aspects such as microservices architecture and technology. For instance, using microservices architecture has undeniable advantages, but it has performance implication in the proposed transaction platform. When a monolithic transaction platform or architecture is refactored into microservices architecture then high latency is inevitable.

Acknowledgements

We would like to express our gratitude to the University of Oviedo, Spain, Oxford Brookes University, UK, and the financial sponsor from the Spanish Government, Projects I+D+i 2019 supported by the Ministry of Science, Innovation and Universities (PID2019-105455GB-C32).

References

[1] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, Data management in microservices: State of the practice, challenges, and research directions, arXiv preprint arXiv:2103.00170, 2021, https://doi.org/10.48550/arXiv.2103.00170.

[2] S. Sharma, Mastering Microservices with Java 9: Build domain-driven microservice-based applications with Spring, Spring Cloud, and Angular. Packt Publishing Ltd, 2017.

[3] 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), pp. 483–490, 2019, https://doi.org/10.5220/0007918704830490.

[4] K. S. Ahluwalia and A. Jain, High availability design patterns, in Proceedings of the 2006 conference on Pattern languages of programs, pp. 1–9, 2006, https://doi.org/10.1145/1415472.1415494.

[5] E. Daraghmi, C.-P. Zhang, and S.-M. Yuan, Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture, Applied Sciences, vol. 12, no. 12, p. 6242, 2022, https://doi.org/10.3390/app12126242.

[6] M. T. González-Aparicio, M. Younas, J. Tuya, and R. Casado, Testing of transactional services in NoSQL key-value databases, Future Generation Computer Systems, vol. 80, 2018, https://doi.org/ 10.1016/j.future.2017.07.004.

[7] 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), pp. 2024–2027, 2019, https://doi.org/10.1109/ICDE.2019.00230.

[8] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, Performance evaluation of microservices architectures using containers, in 2015 IEEE 14th International Symposium on Network Computing and Applications, pp. 27–34, 2015, https://doi.org/10.1109/NCA.2015.49.

[9] Transport of London - Unified API, https://tfl.gov.uk/info-for/open-data-users/unified-api#on-this-page-0 (accessed 14th July 2022).

[10] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, Lambda architecture for cost-effective batch and speed big data processing, in 2015 IEEE International Conference on Big Data (Big Data), pp. 2785–2792, 2015, https://doi.org/10.1109/BigData.2015.7364082.

[11] M. Stone and E. Aravopoulou, Improving journeys by opening data: The case of Transport for London (TfL), The Bottom Line, vol. 31, no. 1, pp. 2-15, 2018, https://doi.org/10.1108/BL-12-2017-0035.

[12] D. Abadi, Consistency tradeoffs in modern distributed database system design: CAP is only part of the story, Computer, vol. 45, no. 2, pp. 37–42, 2012, https://doi.org/10.1109/MC.2012.33.

[13] P. J. Sadalage and M. Fowler, NoSQL distilled: a brief guide to the emerging world of polyglot persistence, Pearson Education, 2013.

[14] R. Jiménez Peris, M. Patiño Martínez, I. Brondino, and V. Vianello, Transaction management across data stores, International Journal of High Performance Computing and Networking, pp. 0–10, 2017, https://doi.org/10.1504/IJHPCN.2018.096709.

[15] N. Ranasinghe, K. De Zoysa, and W. K. Ng, SaaS-Microservices-Based Scalable Smart Contract Architecture, in Security in Computing and Communications: 8th International Symposium, SSCC 2020, Chennai, India, October 14--17, 2020, Revised Selected Papers, vol. 1364, p. 228, 2021, https://doi.org/10.1007/978-981-16-0422-5_16.

[16] X. Limón, A. Guerra-Hernández, A. J. Sánchez-García, and J. C. P. Arriaga, SagaMAS: a software framework for distributed transactions in the microservice architecture, in 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT), pp. 50–58, 2018, https://doi.org/10.1109/CONISOFT.2018.8645853.

[17] H. Uyanik and T. Ovatman, Enhancing Two Phase-Commit Protocol for Replicated State Machines, in 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 118–121, 2020, https://doi.org/10.1109/PDP50117.2020.00024.

[18] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, Calvin: fast distributed transactions for partitioned database systems, in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 1–12, 2012, https://doi.org/10.1145/2213836.2213838.

[19] R. H. Campbell and P. G. Richards, SAGA: A system to automate the management of software production, in Proceedings of the May 4-7, 1981, national computer conference, pp. 231–234, 1981, https://doi.org/10.1145/1500412.1500445.

[20] A. Luckow, L. Lacinski, and S. Jha, Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems, in 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 135–144, 2010, https://doi.org/10.1109/CCGRID.2010.91.

[21] H. Garcia-Molina and K. Salem, Sagas, ACM Sigmod Record, vol. 16, no. 3, pp. 249–259, 1987, https://doi.org/10.1145/38714.38742.

[22] N. M. Josuttis, SOA in practice: the art of distributed system design, O'Reilly Media, Inc., 2007.

[23] S. P. Kumar, S. Lefebvre, R. Chiky, and O. Hermant, "Consistency-Latency Trade-Off of the LibRe Protocol: A Detailed Study," in Advances in Knowledge Discovery and Management, Springer, pp. 83–108, 2018, https://doi.org/10.1007/978-3-319-65406-5_4.

[24] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," ACM Trans. Database Syst., vol. 4, no. 2, pp. 180–209, 1979, https://doi.org/10.1145/320071.320076.

[25] London Buses, https://tfl.gov.uk/corporate/about-tfl/what-we-do (accessed 21st July 2022).

[26] Bus spider maps, https://tfl.gov.uk/maps_/bus-spider-maps (accessed 14th July 2022).