Delft University of Technology

Masters Thesis

Distributed Dataflow Transactions

Author:
Alexander Walker

Supervisor:
Dr. Asterios Katsifodimos
Daily Supervisor:
Kyriakos Psarakis

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

Student number: 5370558
Thesis committee: Prof.dr.ir. G.J.P.M. Houben, TU Delft, chair
Dr. B. Özkan, TU Delft
Dr. A. Katsifodimos, TU Delft, supervisor

An electronic version of this thesis is available at
https://repository.tudelft.nl/.

August 14, 2022

Declaration of Authorship

I, Alexander Walker, declare that this thesis titled, “Distributed Dataflow Transactions” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: August 14, 2022

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Distributed Dataflow Transactions

by Alexander Walker

As serverless computing grows in popularity, developers are demanding more from existing serverless models. One example is the emergence of Stateful Function as a Service (SFaaS), in which state is added to operators in existing Function as a Service (FaaS) models, to support microservice-type applications while utilising the benefits of a serverless architecture. However, current SFaaS systems cannot provide performant transactions across operators with strong semantics.

In this thesis, we present three conceptual transaction protocols for SFaaS dataflow systems based on Two-phase Commit (2PC), Deterministic Databases and Conflict-free Replicated Datatype (CRDT)s. Based on these insights, we implement Rhea, a deterministic transaction protocol in the prototype SFaaS execution engine, Universalis. Two optimizations are implemented for Rhea: deterministic reordering and a fallback mechanism, to reduce aborts caused by Read-after-write (RAW) and Write-after-write (WAW) dependencies, respectively.

We present a transaction benchmarking client for Universalis that supports workloads from the Transaction Processing Performance Council Benchmark C (TPC-C) and the Yahoo! Cloud Serving Benchmark (YCSB). Using the client, we compare Rhea to a 2PC baseline microservice application. We found that Rhea can provide more than twice the throughput and half the latency of 2PC in the baseline application across a variety of workloads.

Acknowledgements

This work would not have been possible without the support of the supervision team, my friends and family. I would like to thank my fellow students who were part of the building 28 study group, as we all went through the highs and lows of our respective master projects together.

I would like to thank Asterios Katsifodimos for his feedback and supervision throughout the course of the thesis, and introducing me to the topic area in the masters course: Web-scale Data Management. I approached him for a thesis topic because of our interactions in the class, and I'm thrilled that he agreed. I would also like to thank Marios Fragkoulis for his insight in the early phases of the project, which helped to give a direction towards the final implementation.

I would like to thank the thesis committee chair: Geert-Jan Houben, and external supervisor: Burcu Özkan for providing their evaluation of this thesis project, particularly over the busy summer period.

Last but certainly not least, I would like to thank my daily supervisor Kyriakos Psarakis. I consider it a great privilege to have worked with Kyriakos; he was always there for support during the project's more trying times, and I have learned a vast amount from him across duration of the project.

Contents

List of Figures

List of Tables

Acronyms

2PC Two-phase Commit

2PL Two-phase Locking

ACID Atomicity, Consistency, Isolation and Durability

BASE Basic Availability, Soft-state, Eventual consistency

CRDT Conflict-free Replicated Datatype

CRUD Create, Read, Update and Delete

DDBMS Distributed Database Management Systems

FaaS Function as a Service

MVCC Multiversion Concurrency Control

NTP Network Time Protocol

OCC Optimistic Concurrency Control

RAW Read-after-write

S2PL Strict 2-phase Locking

SFaaS Stateful Function as a Service

TID Transaction Identifier

TPC-C Transaction Processing Performance Council Benchmark C

tps transactions per second

WAR Write-after-read

WAW Write-after-write

YCSB Yahoo! Cloud Serving Benchmark

Chapter 1

Introduction

Modern-day computing relies heavily on distributed systems. Since the early 1990s, the wide-scale adoption of the web has driven businesses to strive for increased performance and scalability of their computer systems, beyond the capabilities of pure monolithic systems Steen and Tanenbaum [2016]. The departure from monolithic systems led to the rise in popularity of the microservice architecture, where systems are divided into modules each focused on a limited amount of functionality, enabling them to be scaled horizontally Dragoni et al. [2017].

Traditionally, these systems are deployed on infrastructure hosted by cloud providers such as Microsoft's Azure[1] where customers are charged based on their resource allocation. More recently, there has been an emergence of *serverless computing*, where compute resources are provided and charged on demand, which is more cost-effective for customers and can free up resource capacity for cloud providers Baldini et al. [2017].

A popular model of serverless computing is FaaS, such as AWS Lambda[2] and Google Cloud Functions[3]. In this model, applications are deployed as a "set of stateless functions which are executed in response to user or system-generated events" Shahrad et al. [2019]. In addition to reducing costs, FaaS systems abstract away operational concerns such as provisioning, deployment and scaling from the application developers, enabling them to focus on application-level programming. These advantages can be leveraged in microservice applications by mapping each microservice to a set of functions Eyk et al. [2019]; Garriga [2018]. However, microservices frequently require reading and writing to state, which in FaaS necessitates the application developer manually provisioning, deploying, and scaling an external database, negating the auto-scaling and provisioning benefits of serverless deployment. Furthermore, calls to the state incur network round trips which add unnecessary latency to the application.

As a result of these downsides, the development of Stateful Function as a Service SFaaS systems, such as Flink Statefun[4] and Cloudburst Sreekanti et al. [2020]. These systems couple state to each serverless function, moving the difficult operational and scalability concerns of the state away from the user. However, SFaaS systems alone do not provide guarantees about data consistency, fault tolerance, and durability between functions.

[1] https://azure.microsoft.com/
[2] https://aws.amazon.com/lambda/
[3] https://cloud.google.com/functions/
[4] https://nightlies.apache.org/flink/flink-statefun-docs-stable/

These guarantees can be supported by dataflow systems (synonymous with stream processors), which perform computations through operators on streams of data over time, such as events. Modern dataflow systems such as Apache Flink[5] provide exactly-once processing guarantees while maintaining high throughput and availability Carbone et al. [2015b]. Significant efforts have been made to leverage dataflow execution engines in SFaaS applications to reap the benefits of both paradigms Akhter et al. [2019].

Recent work on SFaaS dataflow systems has focused on improving the developer experience, such as the introduction of novel programming abstractions Zorgdrager et al. [2021]. However, a lack of transactional support inhibits many of the potential use cases for distributed applications Katsifodimos and Fragkoulis [2019]. As a result, we identified that no current SFaaS dataflow system: (i) supports transactions with Atomicity, Consistency, Isolation and Durability (ACID) semantics; (ii) abstracts coordination and execution away from the application developer; and (iii) maintains high throughput (> 1000 transactions per second (tps)).

In this paper, three conceptual transaction protocols for SFaaS systems are presented. Based on these concepts, we provide an implementation of Rhea, a deterministic transaction protocol, within the Universalis SFaaS dataflow execution engine. Additionally, we offer a benchmarking client for Universalis, which is used to evaluate the implementation of Rhea based on operational metrics such as throughput and request latency.

1.1 Research Questions

In light of the aforementioned challenges and based on the lack of transactional support in SFaaS dataflow systems, we provide multiple conceptual transaction protocols, which leads us to the first research question:

> **RQ1:** How can distributed transactions be implemented in SFaaS dataflow systems?

Moreover, distributed transactions often come at a cost to system performance and latency, especially when supporting stricter guarantees. This leads us to our second research question:

> **RQ2:** How do distributed transaction protocols perform in SFaaS dataflow systems?

Transaction isolation is an important property to define to application developers in a transaction processing system. Given this, we define the third research question as:

> **RQ3:** Which isolation guarantees can be supported in distributed transaction protocols within SFaaS dataflow systems?

[5] https://flink.apache.org/

1.2 Contributions

The contributions of this work can be summarised as follows:

- We offer conceptual designs for SFaaS transactions that are based on CRDTs and 2PC.
- Rhea, a deterministic transaction protocol, is designed and implemented in Universalis, a prototype SFaaS execution engine. We implement two optimisations for Rhea, deterministic reordering and a fallback strategy, to reduce RAW and WAW aborts respectively.
- We provide a benchmarking application for Universalis that supports common transaction benchmark workloads and can measure multiple metrics. For comparison with Universalis, we implement a baseline microservice application using the 2PC transaction protocol.
- We evaluate Rhea within Universalis under various conditions and, where feasible, compare the outcomes to the baseline.

1.3 Outline

The structure of the thesis is as follows: in Chapter 2 we present background information on distributed transaction processing. In Chapter 3, we examine the existing literature in the disciplines of Distributed Database Management Systems (DDBMS) and operational dataflow systems. In Chapter 4, we describe the architecture of Universalis, our prototype SFaaS execution engine. In Chapter 5 we present the conceptual protocols designed in this project, and the implementation of the deterministic protocol, Rhea. Chapter 6 includes the experimental design and evaluation findings for Rhea. In Chapter 7, we discuss the outcomes of this project and highlight the constraints we encountered. Finally, in Chapter 8 we conclude by summarising our work in relation to our research questions and proposing future study that could be conducted as a result of this effort.

Chapter 2

Database Transaction Processing

Understanding distributed dataflow transactions requires familiarity with database transaction processing concepts. Although dataflow transactions are heavily based on distributed database transactions, it is important to note that they are not identical to DDBMS transactions because they operate on a higher level similar to microservice transactions and incorporate database state changes with operational user-defined logic.

2.1 ACID Transactions

A *transaction* is an atomic unit of work performed by a database system, typically consisting of multiple operations with an emphasis on data integrity. A bank transfer from account A to account B is a simple example of a transaction. The transfer amount must be deducted from account A and added to account B in one unit of work. In the event of a failure at any stage of the transaction, both account balances should revert to their previous states. Database systems strive to support *ACID* transactions, where ACID is an acronym for:

- **Atomicity**: The entire group of operations in a transaction must be executed in full, or not at all.
- **Consistency**: A transaction's changes must always be read and written to the database in a consistent manner, adhering to all integrity constraints.
- **Isolation**: Concurrent transactions must not impede the execution of a transaction, so that each transaction has the impression that it is being executed independently.
- **Durability**: The result of a transaction is persistently stored, even if the transaction fails.

Although ACID is the most prevalent transactional model in database systems, other models, such as Basic Availability, Soft-state, Eventual consistency (BASE), loosen ACID's constraints to increase flexibility Medjahed et al. [2009]. However, in this thesis, we are only interested in ACID transactions, as they provide application developers with more consistent results.

2.2 Isolation

As alluded to in Section 2.1, database systems must support concurrent transactions. In a database system, concurrency is typically achieved through parallel execution by the database system and multi-user access; however, in distributed databases, network access can add additional concurrency.

This necessitates *isolation levels*, which establish the visibility of concurrent transactions to one another. Higher isolation levels impose more stringent constraints, which reduces performance because fewer transactions can be executed concurrently. Lower isolation levels improve performance at the expense of transaction reliability, as they become more susceptible to *read phenomena*, in which concurrent data changes cause inconsistent reads within the scope of a transaction. Consequently, many database systems permit the application developer to select the isolation level based on their specific needs.

2.2.1 Read phenomena

Typically, isolation levels are classified according to the phenomena that can be experienced at each level. These phenomena are defined by the concept of *commit*, which occurs when a transaction's changes are made permanent following a successful execution. If the database system adheres to ACID's durability property, the committed data is always the most recent view of the data.

Dirty Read

A dirty read occurs when transaction 1 reads data that was modified by transaction 2, but not yet committed. If transaction 2 fails, this can lead to inconsistencies in transaction 1, as transaction 1 will have an incorrect view of the database.

Non-repeatable Read

Transactions may perform multiple reads on the same data throughout their execution. If transaction 1 reads data multiple times and transaction 2 concurrently modifies the same data, transaction 1 may have different read values for the same data during its execution, called non-repeatable reads, which result in inconsistencies.

Phantom Read

Phantom reads occur when new rows are added concurrently by transaction 2 while transaction 1 is running. If transaction 1 operates on a range of data into which transaction 2 has inserted rows, transaction 1 will read the newly inserted rows prior to committing. As with dirty reads, transaction 1 will have an inconsistent view of the database if transaction 2 fails.

2.2.2 Isolation Levels

As defined in the ISO SQL standard Michels et al. [2018], there are 4 levels of isolation. The strongest level is *serializability*, in which transactions must be executed as if they were in a serial order. Depending on the mechanism used to

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads
Serializable	×	×	×
Repeatable Read	×	×	✓
Read Committed	×	✓	✓
Read Uncommitted	✓	✓	✓

Table 2.1: Possible read phenomena from each isolation level

enforce isolation, this may be accomplished by utilising locks to prevent other transactions from reading or writing data used by a transaction.

The second level, *repeatable reads*, utilises row-level read and write locks to prevent concurrent access, but not range-level locks. The third level is *read committed*, in which write locks are held until the end of the transaction but read locks are released after all reads have been performed in all active transactions. The weakest isolation is *read uncommitted*, where no locks are held and transactions can modify data as they execute.

In addition to the four levels outlined in the ISO standard, other isolation levels exist in practice. *Snapshot isolation* guarantees that all reads in a transaction are on a consistent database view, and aborts if the view has been modified by a competing transaction at commit time. Moreover, it can be implemented such that it does not encounter any read phenomena, similar to serialization Berenson et al. [1995].

2.3 Concurrency Control

For database systems to enforce the transaction isolation mentioned in Section 2.2, they utilise concurrency control mechanisms. These methods offer varying levels of performance and isolation and are utilised by numerous database systems.

2.3.1 Two-phase Locking

Two-phase Locking (2PL) is a serializable concurrency control algorithm Eswaran et al. [1976]. The algorithm has two phases: the *growing phase* and the *shrinking phase*. A transaction acquires locks for all the data it needs to access during the growing phase. Exclusive locks are used for writing, while shared locks are used for reading.

Exclusive locks are incompatible with shared and other exclusive locks, so they cannot be held simultaneously. Conversely, shared locks are compatible with other shared locks, so they can be held at the same time. The niaeve strategy for overcoming lock incompatibility is to wait until they are released, but this can result in deadlocks between competing transactions. To circumvent this, the following waiting protocols are utilised:

- **NO_WAIT**: Abort transaction and release all locks if transaction tries to acquire a lock that is incompatible with that held on the data.
- **WAIT_DIE**: Wait for a lock if the transaction timestamp is lower than that of the transaction holding the lock, otherwise abort the transaction.

Once the transaction has completed its data operations and no longer requires a lock, it can release the lock. Once a transaction releases its locks, it enters the shrinking phase, during which it is unable to acquire new locks. Other variants of 2PL impose stricter constraints on the release of locks, for example, in Strict 2-phase Locking (S2PL), the transaction retains its exclusive locks until it commits or aborts.

2.3.2 Optimistic Concurrency Control

2PL is considered a pessimistic concurrency control method due to the use of locking and waits. Optimistic Concurrency Control (OCC) does not rely on such techniques to achieve isolation, instead, it executes transactions concurrently and employs a validation phase to check if the result is serializable Kung and Robinson [1981]. In order to perform this check, the system validates the current transaction against all committed transactions since the transaction started, and all other transactions in the validation phase to check for concurrent accesses. In contentious environments, optimistic concurrency control tends to abort transactions significantly more frequently than pessimistic concurrency control Harding et al. [2017].

2.3.3 Multiversion Concurrency Control

Multiversion Concurrency Control (MVCC) belongs to the timestamp class of concurrency control methods, which use timestamps to determine the order of concurrent transactions. In MVCC, multiple versions of each database record are maintained with a timestamp indicating when they were written so that transactions can operate on a snapshot of the data closest to their execution time Bernstein and Goodman [1983]. As with OCC a validator checks for conflicts based on concurrent writes to the data within a snapshot during the execution of a transaction. When a transaction commits, a new version of the record is appended with a timestamp corresponding to the commit time. MVCC generally facilitates snapshot isolation as discussed in Section 2.2.

2.4 Distributed Transaction Protocols

Supporting *distributed transactions* over data that is stored across a network on multiple constituents while maintaining the ACID properties mentioned in Section 2.1 is one of the greatest challenges in transaction processing. Therefore, a protocol is required to ensure that each property is satisfied while performance is maintained. Distributed transaction protocols often make use of existing concurrency control methods as raised in Section 2.3, to adapt them for DDBMS.

2.4.1 Two-phase Commit

2PC Bernstein and Goodman [1981] is an atomic commit protocol, meaning it can support atomicity and consistency between constituents in a distributed system. 2PC relies on a coordinator to handle the communication between the participants of a transaction, which can be replicated for fault tolerance Gray and Lamport [2006]. In addition, 2PC is designed to overcome many types of failures that can

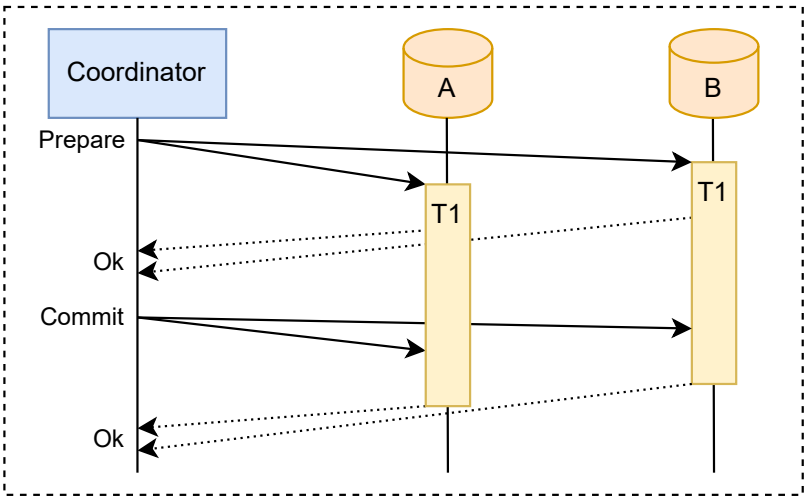


Figure 2.1: Example of a successful two-phase commit execution

occur in distributed settings, primarily due to the prominent use of durable logs that can be read on recovery. The two phases in the protocol are the *prepare phase* and the *commit phase*.

Prepare phase

In 2PC, the client notifies the coordinator that a transaction must be carried out. The coordinator then initiates the transaction by sending a prepare message to all participant nodes, as indicated in Figure 2.1. Upon receiving a prepare message, each node acquires locks on any data required by the transaction on its partition and then notifies the coordinator that it has successfully prepared. The coordinator logs the status of the 2PC execution in a write ahead log so that, in the event of a failure, it can recover to the most recent state.

During this phase, if any problems arise, such as incompatible locks, the participant notifies the coordinator that its preparations were unsuccessful. The coordinator then sends an abort message to all other participants, prompting them to discard their temporary transaction logs. Moreover, if all participants respond with an OK to the coordinator, the execution advances to the commit phase, and the coordinator assumes that the transaction will be committed and cannot be reversed.

Commit phase

In the commit phase, the coordinator sends a message to each participant instructing them to write the transaction effects from the temporary log to the durable storage. As the prepare phase checks for application and integrity level failures, the most common type of failure at this level is a system-level failure, such as a node crashing. In this scenario, the coordinator will detect that a node is offline and ensure that the commit message is sent to it, when it recovers. Generally, 2PC uses two-phase locking to maintain isolation between concurrent transactions, as described in Section 2.3.1, however, there are other implementations of 2PC that utilise different concurrency control mechanisms to maximise performance Lu et al. [2021].

2.4.2 Deterministic Databases

Deterministic databases are a more recent concept in DDBMS, as introduced in Calvin Thomson and Abadi [2010]; Thomson et al. [2012]. The central concept revolves around minimising replica divergence in transactional, partitioned, and replicated systems. Abadi and Faleiro [2018].

The read and write sets of each transaction must be known in advance for deterministic databases, which is not always possible. In cases where this is not possible, the sets are determined by executing the transactions in advance to define the values that will be substituted. Pre-execution can also be used to eliminate nondeterminism from transactions, such as random number generator calls. However, pre-execution cannot always eliminate nondeterminism; consequently, a deterministic database cannot execute every type of transaction.

The primary benefit of these systems is that distributed commit protocols are not required during execution. Deterministic databases define a global serial order of transactions via a *sequencing layer*, which is periodically sent to the schedulers of the participant nodes at set batch intervals. As the order of operations and their outcomes are deterministic, the schedulers in every node will reach the same conclusion regarding which transactions will be committed and which will be aborted. As described in Figure 2.2, the scheduling layer uses the read and write sets of each transaction to generate a lock queue for each data record in order to prevent deadlocks. This also permits numerous optimizations, as transactions can be executed in parallel with relative ease.

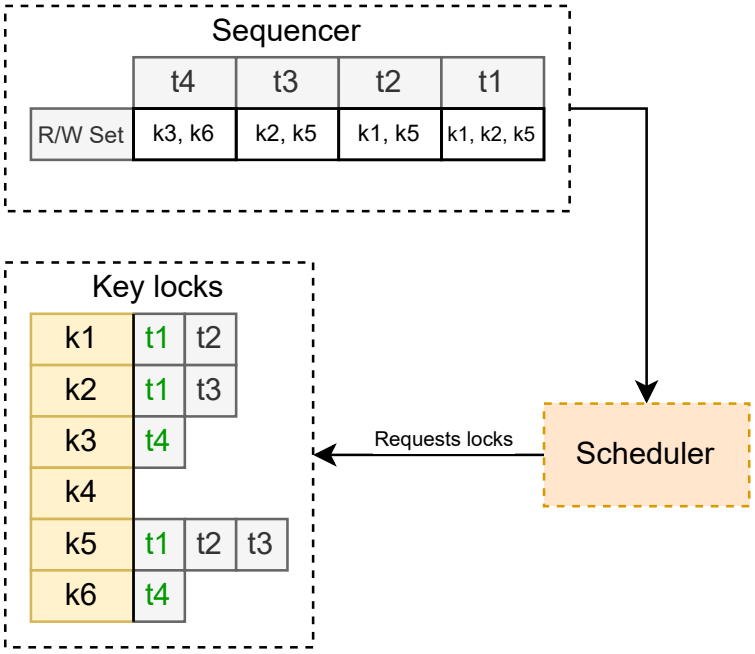


Figure 2.2: Example of deterministic locking, image inspired by Mohammad Roohitavaf [2020]

Chapter 3

Related Work

As the need for larger-scale and more efficient database systems has grown, the field of database systems has continuously advanced. This work is based on a substantial amount of research conducted in this field, specifically concerning distributed database management systems and operational dataflow systems.

3.1 Distributed database management systems

Since the late 1970s, proposals for DDBMS have been published, with a focus on adapting concurrency control techniques from non-distributed systems to distributed databases. This is evident in an early survey by Bernstein and Goodman [1981], which evaluates a variety of concurrency control techniques. Despite surveying over 20 DDBMS concurrency control methods, the authors conclude that they all rely on two fundamental ideas: 2PL and timestamp-ordered concurrency control. The authors chose to evaluate the correctness of the algorithms rather than their performance because, at the time, many algorithms could not be proven correct or did not guarantee correctness. As time has progressed and research into concurrency control methods has advanced the field, the emphasis has shifted from correctness of the protocols to performance, as demonstrated by the survey by Harding et al. [2017]. This survey adds optimistic and deterministic protocols to the classifications of concurrency control methods used in existing DDMBS systems presented by Bernstein and Goodman.

Class	Algorithms	Two-Phase Commit Delay	Multi-Partition Transactions	Low Contention	High Contention
Locking	NO_WAIT, WAIT_DIE	▼	▼	▲	▼
Timestamp	TIMESTAMP, MVCC	▼	▼	▲	▼
Optimistic	OCC	▼	▼	▼	▲
Deterministic	CALVIN	–	▼	▼	▲

Figure 3.1: Summary of results comparing distributed concurrency control algorithms from the paper by Harding et al. [2017]

According to the results displayed in Figure 3.1, locking and timestamp ordering protocols performed relatively better in workloads with low contention but became bottlenecks in workloads with high contention, and vice versa for optimistic and deterministic protocols. In addition, all concurrency techniques exhibited bottlenecks for multi-partition transactions and two-phase commit delays, leading the authors to conclude that all concurrency control methods had varying scalability issues. Despite the unfavourable findings of Harding's survey, transactional DDBMS with satisfactory performance under realistic workloads do exist. One example is Google Cloud Spanner, which enables fully serializable transactions with robust consistency guarantees.

TrueTime, a GPS-synchronized clock, is the most significant innovation within Spanner, allowing any node in the system to generate a globally increasing timestamp. The global timestamp order provides each node with a consistent view of the database when used in conjunction with MVCC. However, Spanner does not provide full support for SQL syntax and is missing features like referential integrity constraints. In addition, supporting a TrueTime-based method would require the creation and maintenance of a GPS-synchronized clock system, which is impractical for the majority of use cases. This results in the use of alternative time synchronisation methods, such as Network Time Protocol (NTP) in CockroachDB Taft et al. [2020]. CockroachDB provides the same guarantees of strong consistency and serializable transactions as Spanner, but the use of the relatively inefficient NTP increases latency by up to 250 milliseconds.

Google Spanner stimulated additional research in the field, such as the emergence of deterministic databases. Calvin was one of the earliest deterministic implementations. Calvin Thomson et al. [2012] is a layer that can be added on top of any database that supports Create, Read, Update and Delete (CRUD) operations in order to facilitate distributed transactions that do not require coordination to commit. Calvin utilises a sequencing layer for this purpose, which collects the read and write sets described in and determines the global order of transactions. This layer can be scaled by partitioning the sequencers and using Paxos Lamport [1998] or Raft Ongaro and Ousterhout [2014] to maintain consensus; however, doing so increases latency and requires the user to provision additional hardware.

Because sequenced transactions are logged persistently and can be replayed by failing nodes upon their recovery, fault tolerance is simplified as a result of Calvin's determinism. Calvin's drawbacks are the requirement to have the read and write sets of every transaction in advance and, as a result, the lack of support for interactive or nested transactions. Numerous later deterministic DDBMS, such as GRIT Zhang et al. [2019] and Aria Lu et al. [2020], were inspired by Calvin. Both systems aim to improve upon Calvin's deficiencies by not requiring the read/write sets for each transaction in advance and by implementing methods to determine the read/write sets.

GRIT employs a global transaction manager across all data partitions and local transaction managers at each node to resolve global and local transaction conflicts, respectively. At commit time, each local transaction manager reports to the global transaction manager which transactions it can commit based on OCC conflict checking. The global transaction manager collects all results and aborts transactions that, according to the local transaction managers, contain at least one conflict.

In Aria Lu et al. [2020], transactions are batched and sequenced to produce ascending Transaction Identifier (TID)s, after which they are optimistically executed in parallel and their results are saved in temporary storage. After the execution phase concludes, the commit phase commences, during which a deterministic conflict detection algorithm is used to determine whether conflicting operations between transactions have occurred. To maintain serializability, Aria checks for RAW and WAW dependencies and aborts any transaction that exhibits one of these dependencies and occurs later in the global order. Conflicting transactions are rescheduled to the following batch so that they can be executed until they commit

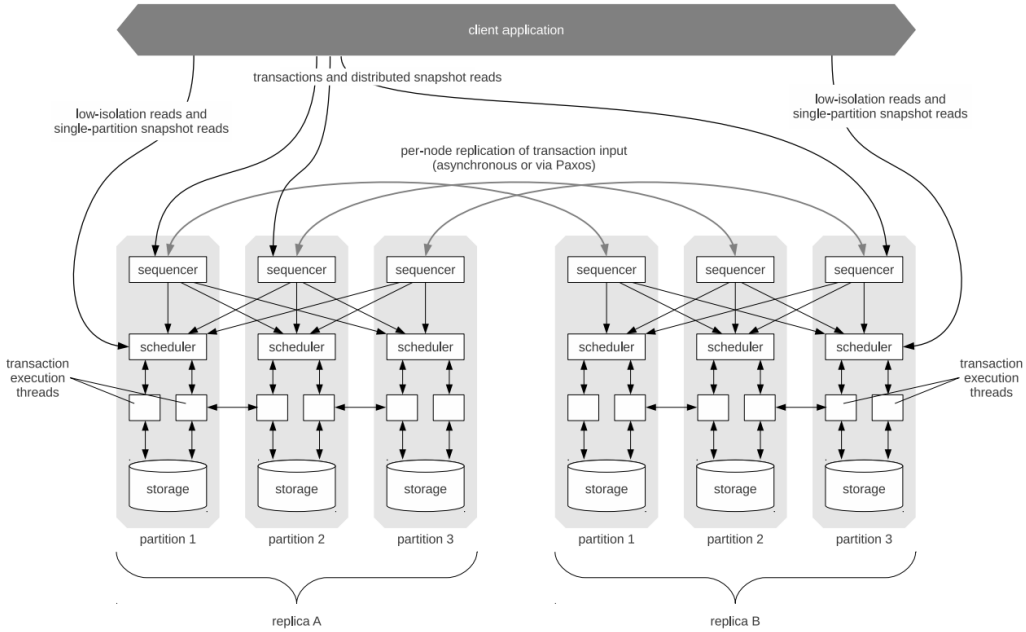


Figure 3.2: System architecture Calvin as displayed in the paper by Thomson et al. [2012]

or fail due to an application or integrity constraint error. The transaction protocol produced in this work, Rhea is heavily influenced by Aria, and is described in Section 5.3.

Aria is used as a comparison in a later paper by Lu et al. [2021], in which an optimised version of 2PC is presented, COCO. COCO aims to decrease the 2PC overhead in replicated systems by batching transactions into epochs and treating the entire epoch as a single 2PC commit unit. The system employs optimistic concurrency control such that, during the prepare phase of 2PC, all transactions in the epoch are executed concurrently without the use of locks. In addition, the commit phase is divided into three distinct phases: (i) locks are acquired for the write sets of each transaction; (ii) read sets are validated for read/write conflicts; and (iii) the transactions are committed to the database. Despite the fact that COCO's evaluated throughput is 2.3 times lower than Aria's in single-node mode, the authors claim that COCO's default configuration of three replicas per partition provides a higher level of availability.

3.2 Operational dataflow systems

As businesses have adopted technologies such as dataflow systems to supplement or replace traditional DDBMS, the need to support operational concerns such as transactions in these systems has increased.

S-Store, which combines streaming functionality with acid transactions, ordered execution, and exactly-once processing guarantees, is described in the paper by Meehan et al. [2015]. To accomplish this, the authors augment an existing DDBMS, H-Store Kallman et al. [2008], which includes transaction processing capabilities, with streaming execution. In S-Store, the application user can define their transactional application in the form of a dataflow graph, in which

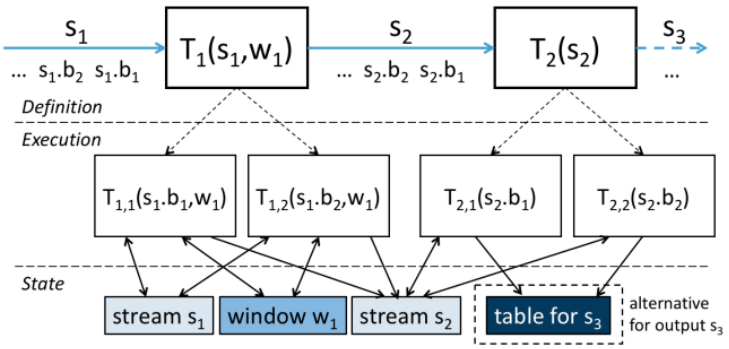


Figure 3.3: Transactional Dataflow Graph in S-store Meehan et al. [2015]

transactions can occur between data streams, which can then be outputted as a stream or to a database table. S-store transactions operate between streams similarly to operators, but they do not provide operator-to-operator transactions and do not contain state, so they cannot support stateful operator transactions.

Cloudburst Sreekanti et al. [2020], which is built on top of the Anna Wu et al. [2018] key-value store, employs a similar architecture for streaming functionality over an existing distributed store. In Cloudburst, users can submit the structure of their functions in the form of a directed graph, which is then applied to the data streams. Allowed are both synchronous and asynchronous function calls, and all functions have complete access to shared state. However, Cloudburst does not provide any guarantees regarding isolation levels for concurrent accesses, which can result in replicas diverging. Cloudburst handles these divergences by default by selecting the most recent write as the final value, but the authors also offer a CRDT Shapiro et al. [2011] based solution to eliminate the replica divergence issue. As a result of the lack of transactional guarantees, Cloudburst is less suited as a platform for distributed dataflow transactions.

Beldi Zhang et al. [2020] implements a runtime that is compatible with existing FaaS platforms like AWS Lambda. Beldi is exposed to the user as an API that can be invoked by user-defined serverless functions to provide access to other functions with transactional guarantees and a key-value store. For fault tolerance, Beldi logs all API accesses to provide a durable record of user actions that can be used to reproduce events after a system failure. The performance evaluation revealed that the authors could not support more than 800 requests per second because AWS's compute service only permits 1,000 concurrent accesses.

Using coordinator functions, serializable and ACID stateful operator transactions are supported via 2PC and SAGAs in the paper by Heus et al. [2021]. The implementations, however, are implemented on the Flink Statefun dataflow system, which was not designed for transactional workloads; as a result, the system's optimization potential is limited when compared to a transactional dataflow system such as Universalis.

Chapter 4

Universalis

In previous work described in Chapter 3, we discovered that serverless dataflow systems that support transactions or transaction-like behaviour do exist. However, these systems frequently have drawbacks, such as poor performance, complex programming models for application developers, and limited transactional guarantees.

We believe this is due to the fact that many of the aforementioned systems are built on top of existing dataflow systems that were not intended for transaction processing. In response, we present Universalis, a SFaaS system designed from the ground up with a dataflow-based execution engine and transaction processing support as one of the primary design goals. At the time of writing, Universalis is a prototype and not a production-ready system. The design elements described in this chapter are therefore susceptible to change. In addition, Universalis is not a direct contribution of this thesis; rather, it is part of Kyriakos Psarakis's ongoing research.

4.1 Design Philosophy

Universalis targets application developers who want to create distributed applications, such as microservice applications. *Accessibility*, *Predictability*, *Performance*, and *Scalability* are the fundamental philosophies that guide the design of Universalis.

4.1.1 Accessibility

The need for accessibility necessitates that Universalis provide interfaces that disrupt application developers' workflow as little as possible. Specifically, the interface must be usable by developers with minimal knowledge of SFaaS systems. Universalis should provide developers with a programming model that allows them to write their applications as they would a standard microservice. Universalis achieves accessibility by providing application developers with a high-level API corresponding to standard CRUD operations. In addition, the internal implementations are concealed from the developer so as not to overwhelm the user with unnecessary information.

4.1.2 Predictability

Universalis must define and provide consistency and transaction guarantees to ensure predictability. This enables developers to construct their applications

around these guarantees without fear of introducing additional bugs due to the system's unpredictable behaviour. In addition, a predictable system can recover from failure in a secure manner, eliminating the need for application developers to implement specific recovery methods. Universalis provides exactly-once processing guarantees and serializable ACID transactions to address predictability. As described in 4.2.5, Universalis makes extensive use of checkpointing to handle failures, so that the system can recover reliably in the event of a failure.

4.1.3 Performance

With a performant execution engine, the application developer can concentrate on optimising application code rather than focusing on lower level details. Due to the fact that Universalis was designed from the ground up, execution and transaction logic have been optimised at a lower level, as described in Chapter 5.

4.1.4 Scalability

Universalis must demonstrate the capacity to scale in the face of increased workloads. Scalability is a core reason why the serverless architecture has become so widely adopted, due to the decoupled nature of the applications, which allow for more elasticity in scaling up and down. Universalis is designed for scalability by employing a highly decoupled and partitioned architecture, as shown in 4.2.

4.2 System Architecture

Universalis is written in Python as this has enabled rapid prototyping of features, despite the performance disadvantage compared to lower-level languages commonly used in distributed systems, such as C++, Go, and Scala. Despite this, Python is one of the most commonly used languages for software development[1], thus building Universalis in Python easy support for applications written in Python. By default Python does not support concurrency, so Universalis makes prominent use of the asyncio [2] library for asynchronous code..

4.2.1 Interface

Universalis provides a Python library which exposes an API that application developers use to interact with the Universalis execution engine. Application developers must define their application as a *stateflow* graph, which is a graph consisting of *operators* as nodes and directed edges to indicate network access between the operators as shown in Figure 4.1. An operator is similar to a microservice, with stateful or stateless functions. As the underlying state of operators are partitioned, the user must also define the number of partitions per operator in the dataflow graph for their stateful functions. An example of code to define a stateflow graph in Universalis can be found in Figure A.1.
Each operator function must be defined as a class that extends the 'Function' or 'StatefulFunction' class in the Universalis Library. The primary distinction

[1] https://insights.stackoverflow.com/survey/2021
[2] https://docs.python.org/3/library/asyncio.html

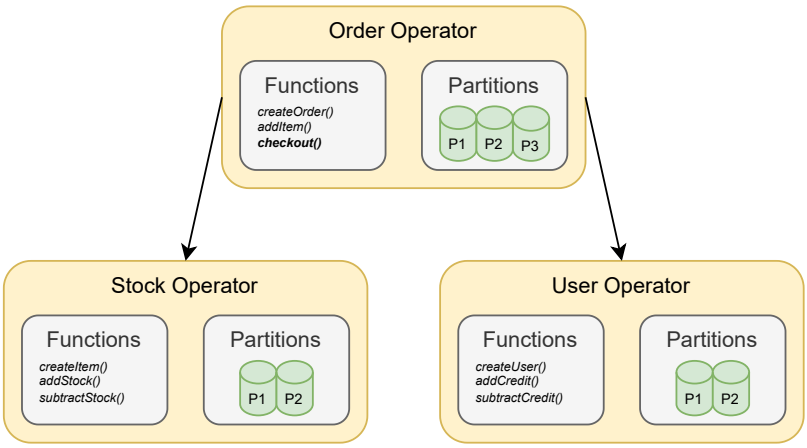


Figure 4.1: Example of a Universalis stateflow graph consisting of 3 stateful operators with differing partitions.

between the two classes is that StatefulFunction permits the use of CRUD methods for state updates. These CRUD methods are fundamental to the implemented transaction protocol described in Section 5.3. As demonstrated in Figure A.2, the function's execution logic must be placed in the 'run()' method for each function type, which will be invoked by the system in response to a user request. In addition, Universalis is compatible with any storage engine that supports CRUD operations. Redis[3], a flexible in-memory key-value store, is utilised by default in Universalis. Redis supports the durability property of ACID by providing a variety of persistence options, such as Redis Database, which periodically logs in-memory data to nonvolatile storage.

4.2.2 Coordinator

In order to submit a stateflow graph, a Universalis execution engine must be deployed on a server. The execution engine contains a coordinator, which is a stateless component responsible for managing the engine's components. The coordinator's primary responsibility is to manage network discovery so that components, as shown in Figure 4.2, can detect new components dynamically when horizontally scaling. The coordinator also instantiates the workers based on the submitted graph, using a deterministic round robin scheduling algorithm to assign operator partitions to the system's workers. Using heartbeat timeouts, the coordinator can also detect and manage component failures. Furthermore, by using Paxos/Raft for consensus, the coordinator can be partitioned and replicated for scalability and fault tolerance.

4.2.3 Event Ingress and Egress

Universalis uses Apache Kafka[4], a distributed event streaming platform, to manage incoming and outgoing messages. Kafka facilitates performant messaging and is partitioned for scalability. Due to the inherent replication and logging of the system, Kafka can also be configured to support exactly once messaging, ensuring

[3] https://redis.io/
[4] https://kafka.apache.org/

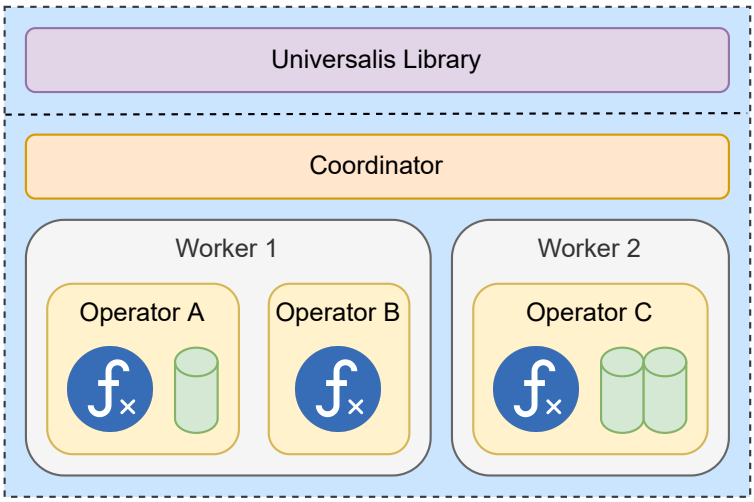


Figure 4.2: Universalis execution engine consisting of two workers with different operator configurations

that messages are delivered. A Kafka cluster is configured with three workers by default in Universalis; however, the application developer can configure this to meet their needs. When functions are sent from an external client to a Universalis instance, they are ingested by the Kafka workers, given a timestamp, and then broadcast so that consumers within the workers can consume them. Messages are forwarded to the Kafka cluster so that they can be output as part of the egress when workers need to return a response to an external participant.

4.2.4 Worker

After a user submits a dataflow graph, the coordinator assigns operators to *workers*. Workers are concurrent Python modules composed of operators and their respective states/functions. Workers are responsible for managing the execution of stateflow functions and any associated transaction logic. The number of workers in the execution engine can be fixed based on the needs of the application developer, or horizontally scaled as the number of operators grows. Moreover, operator partitions can be distributed across multiple workers in order to scale operators. A simple modulo hash function is used to assign partition keys based on the number of partitions defined for an operator.

As highlighted in Figure 4.3, a worker contains a Kafka consumer that is subscribed to function call events for the instance's operators. The events are sent to the transaction layer, which applies the transaction protocol according to the implementation described in Chapter 5. To maintain the ACID properties for all function calls, each event, even if it does not call any remote operators, is treated as a transaction. The transaction layer manages remote function calls, also known as multi-partition transactions, which may require network communication with other worker. If a function is defined to return output, the response is sent to a Kafka producer, which then forwards it to the event egress.

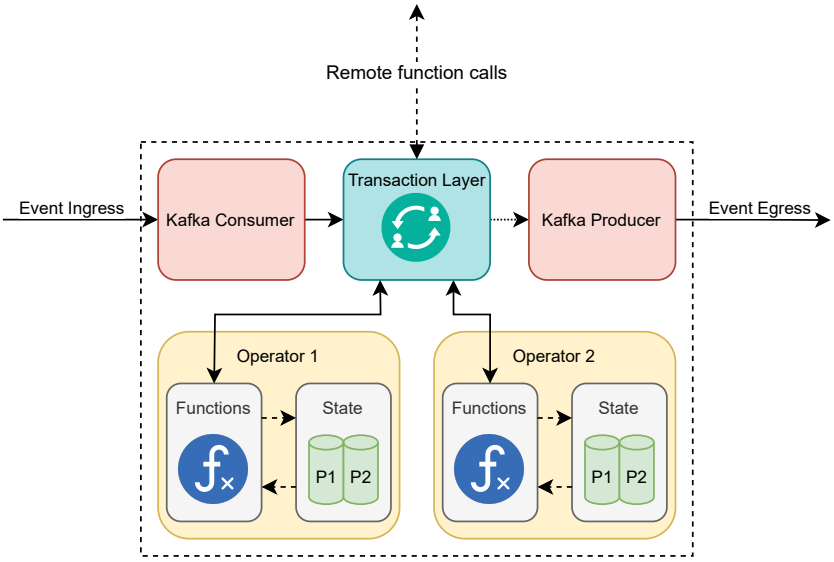


Figure 4.3: Worker flow of execution

4.2.5 Fault Tolerance

Universalis aims to maintain exactly-once guarantees, using a fault tolerance mechanism similar to Apache Flink, however, at the time of writing, the fault tolerance mechanism for Universalis was not fully implemented. The central concept of the mechanism is the periodic insertion of barriers into the streams for use as a marker for checkpoints and as an alignment measure between streams. The state of the underlying dataflow system and corresponding operator states are periodically aligned (if required) and saved to durable storage at checkpoint time.

In the event of a failure at the system level, the coordinator waits for all participants to recover and for each participant to regain their previous checkpointed state. This can be performed efficiently in systems with lower throughput and state, but it can become costly in larger systems with higher throughputs and more state to maintain. Support for exactly-once processing can be enabled by aligning the streams before checkpointing to a durable log. Additional information regarding how fault tolerance operates in Apache Flink and Universalis can be found in the work by Carbone et al. [2015a].

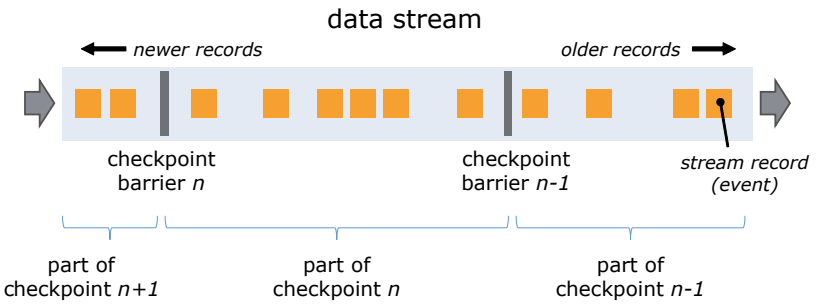


Figure 4.4: Example of a stream barrier from the Apache Flink Documentation[5]

[5] https://nightlies.apache.org/flink/flink-docs-release-1.3/internals/stream_checkpointing.html

Chapter 5

SFaaS Transactions

In prior sections, it was emphasised that transactions in SFaaS systems, such as Universalis, have only been implemented in a handful of instances. In this chapter, we present the initial transaction protocol concepts for Universalis: based on CRDTs Section 5.1 in and 2PC Section 5.2, and provide insight into why they ultimately were not implemented and evaluated in Universalis. In addition, the development of these concepts led to the design and implementation of the deterministic transaction protocol Rhea, which is described in. Section 5.3.

5.1 CRDT Based Approach

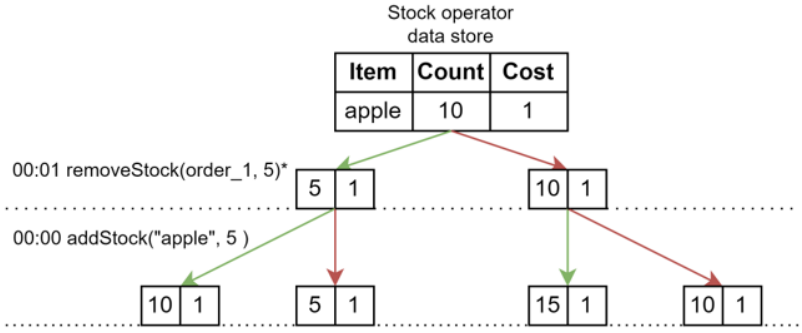


Figure 5.1: Conceptual design of the CRDT based SFaaS transaction protocol: Traga

CRDTs, initially proposed in Shapiro et al. [2011] are targeted at reducing the divergence of replicas in distributed systems, by resolving the merge process of the diverging replicas. To enable this, the state updates that can be applied must be commutative, meaning that the order the operations are applied in does not matter, as the result will always be the same. When the replicas merge, no matter how far they have diverged, the CRDT can easily maintain consistency due to the commutativity of operations.

Traga, a CRDT-based transaction protocol, attempts a similar strategy by treating each row of the key/value store as the state and transactions with state-affecting operations as state operations. As imposing the commutativity constraint on transaction operations would significantly restrict the range of supported transactions, the order of transactions is determined by the timestamp at which they were applied to the state at a given time. When a transaction performs an operation on a row within the store, the resulting state change of the operation is

appended to a tree-like structure on the 'left' branch, while the original value, or the value if the transaction operation fails, is appended to the 'right' branch. State modifications caused by subsequent operations are added to the tree's leaves, as shown in Figure 5.1. If a transaction commits, the operation value stored on the left branch of each sub-tree is merged to the sub-root, tree's the level is deleted, and the sub-trees below rebalance accordingly. When a transaction fails, the same procedure is followed, with the exception that the values stored on the correct subtree branch are merged.

The perceived benefit of Traga was that it did not require costly concurrency control methods to maintain isolation while supporting serializability of transactions.Nevertheless, as transaction operation results were added to the tree's leaves, Traga's memory consumption grew exponentially. High contention transactions that spanned multiple rows and required each row to construct a tree exacerbated these issues. Moreover, when resolving these trees after transactions had been committed, the algorithmic complexity required to rebalance the trees grew exponentially, as each level contained n^2 elements of the elements in the level above. Due to the inherent nature of these issues in the architecture of the concept, we decided not to implement it in Universalis.

5.2 2PC Based Approach

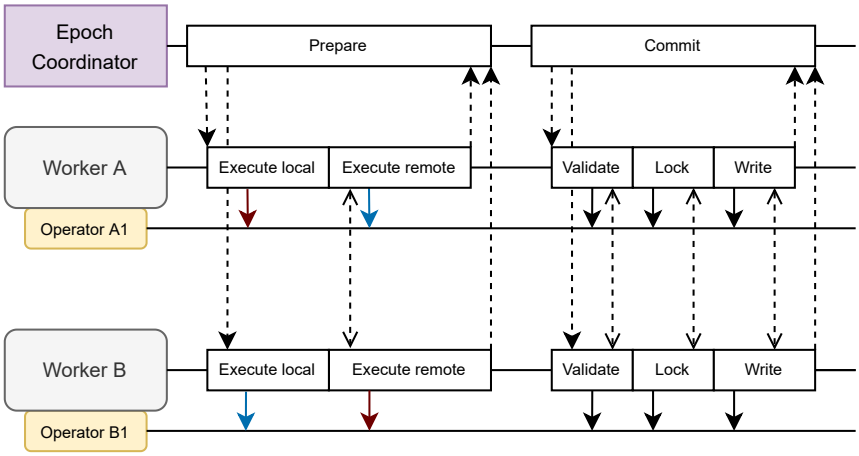


Figure 5.2: Conceptual design of the 2PC based SFaaS transaction protocol: Epoch 2PC

Epoch 2PC is a proposed transaction protocol for Universalis that draws inspiration from the COCO algorithm outlined in Lu et al. [2021]. Even though Universalis is not a replicated system, which is COCO's primary use case, many of the algorithm's features, such as message batching, are still effective. In the Epoch 2PC protocol, incoming transactions are divided into epochs at predetermined intervals. In a coordination protocol similar to the standard 2PC protocol introduced in Figure 2.1, each epoch is then treated as a single commit unit instead of each individual transaction. This method's advantage over the standard 2PC protocol is the ability to batch network calls between constituents to reduce networking overhead. Additionally, Epoch 2PC employs OCC so that locks do not need to be held between the prepare and commit phases, thereby increasing the concurrency

potential. To manage the creation of epochs and the coordination of the currently active epoch, the algorithm employs an external participant known as the *epoch coordinator*. This coordinator determines when the current epoch may transition between the protocol's *execution* and *commit* phases.

5.2.1 Epoch Coordinator

The epoch coordinator is added between the workers and the ingress as a layer. It is not responsible for the execution of transactions, but rather for the advancement of epochs. As with COCO Lu et al. [2021], incoming transactions are assigned a unique TID and batched into epochs every 10 milliseconds. Each epoch is added to a first-in, first-out queue that is persistently logged so that it can be recovered in the event of a failure. The order in which transactions occur in an epoch is used to determine the serial order by the concurrency control methods. Moreover, if any transaction within an epoch fails due to a system failure, the entire epoch is aborted and retried when the system recovers completely, preserving atomicity.

5.2.2 Execution Phase

The epoch coordinator initiates the execution phase by sending a prepare message to all workers that have operators involved in the epoch. The prepare message for each worker contains a list of all required transactions and their corresponding TIDs. The involved workers execute the transaction logic in an optimistic manner, storing the read and write sets locally but not modifying the state. The read values are based on the *snapshot state*, which is the last committed state after the previous epoch.

In the event of an integrity or application level error, such as a negative integer value, the transaction is terminated as an application abort. The developer of the application can define exceptions to be thrown in the event of these aborts, which are directed to the egress so they can be handled appropriately. Any remote function calls made during the execution of a transaction are collected so that they can be sent in bulk to the workers that contain the remote transactions. These transactions are executed by their corresponding workers, who respond to the host worker regarding the success or failure of the remote call. If any remote call for a given transaction fails, the transaction must be terminated. The host worker cannot respond to the epoch coordinator until all remote function calls have returned a response. When the epoch coordinator receives responses from all participating workers, the execution phase for that epoch concludes.

5.2.3 Commit Phase

The epoch coordinator sends a commit message to all participating workers in the current epoch once the execution phase has concluded. Due to the algorithm's optimistic concurrency control, the commit phase is essentially the validation phase. The commit phase is performed in three stages for each worker. Because each stage must be completed before proceeding to the next, they are executed in separate network round trips.

The first stage is to acquire locks for the rows in the write set of all transactions, to prevent conflicts through WAW dependencies. The locks are acquired in order

of TID, such to preserve the serializability property of writes. To remove deadlocks, the NO_WAIT avoidance scheme is used, as discussed in 2.3.1, which performs well in high contention environments Harding et al. [2017]; Lu et al. [2021]. Aborts at this stage are considered concurrency aborts and can be retried in the next epoch, so the worker collects these transactions to send to the epoch coordinator after the commit phase.

Remote function calls must also acquire locks, which are batched and transmitted after the locking phase for host transactions has concluded. As in the execution phase, if any remote function call fails to acquire a lock, the entire transaction is terminated as a concurrency abort and any locks it may have acquired are discarded. After each transaction's write sets have been successfully locked or the transaction has been aborted, the worker begins the validation phase. The validation phase is intended to verify the read sets of each transaction in order to abort any transactions that have RAW dependencies. This is accomplished by ensuring that no rows within the read set of a transaction are locked by another transaction. Similar to the previous phase, if a transaction's validation fails, it must discard its locks and abort as a concurrency abort.

The application stage is the third and final stage. All writes from the remaining transactions are applied to the respective operator states at this stage. After completing this phase, the worker notifies the epoch coordinator that it has executed the commit phase, along with any transactions that were aborted. When all workers have responded to the epoch coordinator, the coordinator prepends any aborted transactions to the next epoch in the queue and prepares to execute the next epoch.

Despite providing advantages over a standard 2PC protocol, the implementation of Epoch 2PC in Universalis necessitated additional provisioning and scaling for the epoch coordinator. This would further complicate the deployment of a Universalis instance, thereby affecting the accessibility aspect of the design philosophy presented in Section 4.1.1. Due to this limitation and the time constraints associated with the project, Epoch 2PC was not implemented or evaluated in this endeavour.

5.3 Deterministic Approach

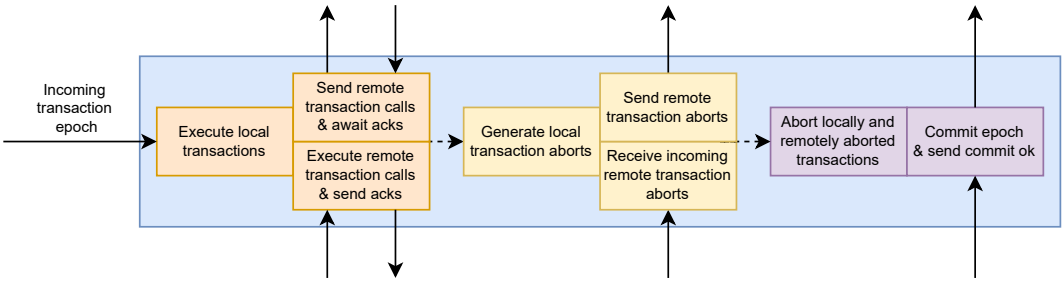


Figure 5.3: Rhea transaction protocol

The final transaction protocol produced in this project was Rhea, which is based on the deterministic database system, Aria Lu et al. [2020]. Compared to previous concepts of transaction protocols, messaging and coordination are drastically reduced, and fault tolerance is simplified, as will be explained in greater detail below.

5.3.1 Design

As described in Epoch 2PC from Section 5.2, Rhea processes transactions in epochs, which are collections of transactions spanning a predetermined time period. The workers process the epoch in two phases: the execution phase and the commit phase. Each worker contains a *sequencer* that determines the global order of transactions, eliminating the need for an external coordinator and allowing the sequencers to scale proportionally with the number of workers. The system ingress delivers incoming transaction calls directly to the workers.

In contrast to other deterministic methods, such as Calvin Thomson et al. [2012], the read and write sets are determined during the algorithm's execution phase, so they do not need to be known beforehand. As the execution phase can be run in parallel, this approach to locking is more efficient than Calvin's singlethreaded approach described in Figure 2.2. After this phase, the protocol can commit deterministically, as the defined read/write sets and deterministic conflict checking algorithm ensure that the same transactions will commit regardless of the order in which they are executed during the commit phase.

Sequencer

In deterministic protocols, such as Calvin Thomson et al. [2012], the sequencing layer is often implemented separately from the execution layer and partitioned in order to scale. This strategy has the disadvantage of requiring costly coordination protocols, such as Paxos Lamport [1998], across partitions, which increase latency and complexity. In Rhea, rather than implementing a completely separate layer, sequencing is performed at the worker level, where each worker contains a sequencer that coordinates epoch progression and determines the global execution order of transactions. The sequencers are capable of producing distinct increasing TIDs without the need for coordination during the sequencing phase.

The sequencing phase proceeds as follows: The sequencer of each worker collects transactions into batches at a predetermined interval of time known as the epoch interval. During this phase, the sequencer assigns a unique ascending TID to each transaction. Given that the coordinator of Universalis assigns a unique identifier to each worker and that each worker stores the number of peer workers in the system, the following formula can be used to determine the TID:

$$\text{TID} = \text{Worker Identifier} + \text{Transaction Count} \times \text{Number Of Workers}$$

The combination of a unique worker identifier multiplied by the number of workers ensures that deterministic and unique ranges of TIDs are assigned to each worker.

Execution Phase

After the sequencer assigns a TID to each transaction, the execution phase begins. In this phase, all the transactions in the epoch for the given worker are executed. The transaction's host worker executes the logic as defined by the user, retrieving the values of any reads from the previously committed state. The keys and values of each transaction's writes are stored alongside the transaction identifier in per-key write reservations on the state. Write reservations may only be made on a key in the state if the TID is less than all other TIDs on the same key; otherwise, the reservation must fail. If the TID is less than the one currently stored for a given key, the new reservation will replace the existing one. Similar to Epoch 2PC, any application level failures that occur at this stage in Rhea are caught and these transactions are aborted, with any error output forwarded to the egress to notify the user.

Remote calls are required to execute the logic and determine if the remote operator can commit for multi-partition and sub-transaction transactions. These operators may be stored on the same worker or on a remote worker, requiring an additional phase after the local execution phase. As they are sub-transactions of the host transaction and must be ordered in the same manner by the remote workers, the workers execute any remote transaction calls under the same TID as the host transaction. Note that if the local execution phase of a host transaction was aborted, this phase is skipped for that transaction.

Remote workers execute the logic of remote transactions identically to local transactions, using the TID of the host transaction to make write reservations. In certain circumstances, remote transactions may trigger additional remote transactions; this is known as transaction chaining. Due to the recursive definition of remote transaction handling, remote transactions are executed similarly to other remote function calls. Because the transaction logic is stored in stateful functions within the operators, the host transaction does not know beforehand how many sub-transactions are contained in a chain or how many acknowledgements it must wait for. As shown in Figure 5.4, each sub-transaction stores a fraction containing its share of the proportion given by the transaction that called it. This fraction is returned to the host transaction if a sub-transaction is the final one in a chain, meaning it does not call any remote transactions. When the total sum of fractions returned equals 1, the host transaction determines that all responses have been received and can proceed to the next phase. The use of fractions rather

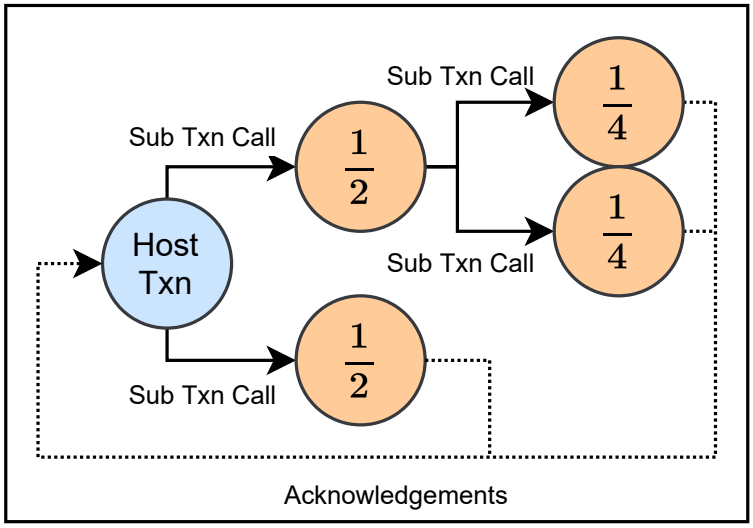


Figure 5.4: Transaction chaining in Rhea. Fractions indicate the proportion of the total chain each sub-transaction represents

than decimal numbers mitigates any potential issues with floating point precision when performing the summation.

If a remote transaction terminates due to an application abort or concurrency abort, the remote worker notifies the host worker that the transaction has terminated. The host will then mark the entire transaction as aborted and notify the related workers. Before moving on to the next phase, each worker must await acknowledgment for every transaction.

Commit Phase

After the execution phase, each worker performs local conflict checking using the TIDs to detect and abort WAW dependencies and RAW dependencies. If a value in the write set has a write followed by a write or a read for a transaction, the transactions with TIDs that are greater than the write are aborted as concurrency aborts. This ensures serializability as transactions cannot observe the result values of subsequent transactions, effectively preserving the serial execution order. As reads have no effect on the stored value, they can occur in any order; no checking is performed unless a write on the same key has occurred. The Aria paper Lu et al. [2020] provides a formal demonstration of serialisation for this conflict resolution method.

After conflict checking, there is a synchronisation point between workers, as each worker cannot begin the commit process until all other workers have signalled their readiness to commit. In the notification message, the workers transmit all local transactions that were aborted due to concurrency errors, allowing any remote transactions that were called on other workers to be aborted as well. Each worker sends remote function commits and waits for a response before committing their local commits, as the transaction must be rolled back if concurrency aborts occur. In addition to sending remote aborts, the commit point is also used to synchronise the transaction count of all workers, with each worker setting its transaction count to the maximum transaction count of all workers. This decreases

the amount transaction counts fluctuate between workers, as the distribution of transactions across workers is frequently not uniform.

In the final step of the commit phase, the write values are made durable in the local state of the operator by updating the values accordingly. The result of a committed or application-aborted transaction can be sent to the event egress after the local values have been committed. Finally, the epoch counter is incremented and aborted concurrency transactions are prepended to the next epoch of transactions.

5.3.2 Implementation

Rhea is implemented in the transaction layer of the workers shown in 4.3. To support write reservations and buffering read/write sets in the state without directly applying the results, the state layer was expanded. The read and write sets are stored in Python dictionaries that map each TID to a set of affected keys and key/value pairs for the read set. Using dictionaries that map each key to an asyncio lock, per-row locks that restrict concurrent access from other co-routines were implemented.

State Operations

```
    async def get(self, key, t_id: int, operator_name: str):
        async with self.read_set_locks[operator_name]:
            if t_id in self.read_sets[operator_name]:
                self.read_sets[operator_name][t_id].add(key)
            else:
                self.read_sets[operator_name][t_id] = {key}
        async with self.writing_to_db_locks[operator_name]:
            db_value = await self.redis_connections[operator_name].get(key)
        if db_value is None:
            if t_id in self.write_sets[operator_name] and key in
            ↪  self.write_sets[operator_name][t_id]:
                return self.write_sets[operator_name][t_id][key]
            else:
                raise ReadUncommitedException(
                    f'Read uncommitted or {key} does not exist in DB'
                )
        else:
            self.reads[operator_name][key] =
            ↪  min(self.reads[operator_name].get(key, t_id), t_id)
            value = msgpack_deserialization(db_value)
            return value
```

Figure 5.5: Overwritten `get` operation in the operator state to support Rhea

As `get` operations to the state within a transactions are read only, the `get` operation in the state was extended to include the TID for the key to be read for that transaction. The reads retrieve the value directly from the state, which contains the key's most recent committed value as of the most recent epoch. Additionally, some transactions may require reading a value that was previously written. In order to facilitate this, the write set of that transaction is examined to retrieve the

previously written value, which is then used as the read result. The implementation of the `get` operation is shown in Figure 5.5.

```
    async def put(self, key, value, t_id: int, operator_name: str):
        async with self.write_set_locks[operator_name]:
            if t_id in self.write_sets[operator_name]:
                self.write_sets[operator_name][t_id][key] = value
            else:
                self.write_sets[operator_name][t_id] = {key: value}
            self.writes[operator_name][key] =
            ↪  min(self.writes[operator_name].get(key, t_id), t_id)
```

Figure 5.6: Overwritten `put` operation in the operator state to support Rhea

To support `put` operations, the TID is added to the write set along with the write's value, as shown in Figure 5.6. As some transactions may write to the same key more than once, the most recent version is persisted in the write set to preserve the serializability of the operation.

```
    async def commit_operator(self, operator_name: str) -> set[int]:
        updates_to_commit = {}
        committed_t_ids = set()
        if len(self.write_sets[operator_name]) == 0:
            return committed_t_ids
        for t_id, ws in self.write_sets[operator_name].items():
            if t_id not in self.aborted_transactions:
                updates_to_commit.update(ws)
                committed_t_ids.add(t_id)
        if updates_to_commit:
            serialized_kv_pairs = {key: msgpack_serialization(value) for key,
            ↪  value in updates_to_commit.items()}
            async with self.writing_to_db_locks[operator_name]:
                await
                ↪  self.redis_connections[operator_name].mset(serialized_kv_pairs)
        return committed_t_ids
```

Figure 5.7: Commit operation in the operator state to support Rhea

According to Section 5.3.1, writes in Rhea are only made durable when a transaction commits. The worker commits per operator by applying to the state any writes in the write set. Redis is used as the backend in this instance, so writes are applied directly to Redis. The committed TIDs are returned so that they can be used in the optimisation of the fallback strategy, as described in Section 5.3.3.

Conflict checking

As described in Section 5.3.1, the conflict checking algorithm detects RAW dependencies that violate serializability. This is accomplished by determining if there are any TIDs with a lower value than the current TID for each key in the transaction's write set. If a conflict is detected, the transaction is added to the list of aborted transactions and returned so that it can be handled by the fallback mechanism described in Section 5.3.3 or retried in the next epoch.

```python
def check_conflicts(self) -> set[int]:
    for operator_name, write_set in self.write_sets.items():
        for t_id, ws in write_set.items():
            ws = write_set[t_id]
            rs = self.read_sets[operator_name].get(t_id, set())
            keys = rs.union(set(ws.keys()))
            for key in keys:
                if key in self.writes[operator_name] and
                ↪  self.writes[operator_name][key] < t_id:
                    self.aborted_transactions.add(t_id)
    return self.aborted_transactions
```

Figure 5.8: WAW and RAW conflict checking algorithm in Rhea

5.3.3 Optimisations

Due to the optimistic parallel execution of transactions, the protocol underlying Aria and Rhea can result in a significant number of aborts under high contention loads. As aborts due to concurrency are appended to the next epoch, this can result in repeated retries of transactions before they are finally executed. In Aria, the authors present two optimisations designed to mitigate issues in high contention environments. This section describes the optimisations: deterministic reordering and the fallback mechanism, as well as how they are adapted for SFaaS transactions in Universalis.

Deterministic Reordering

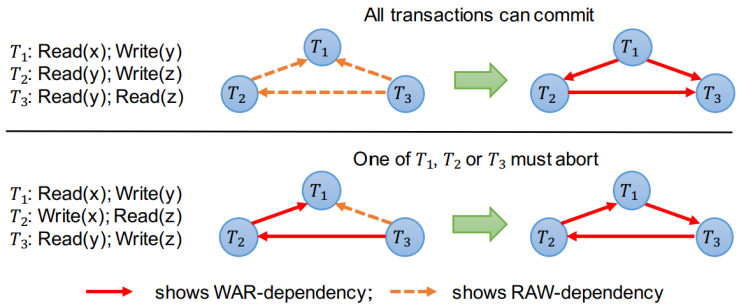


Figure 5.9: Deterministic reordering algorithm in Aria to change RAW dependencies to Write-after-read (WAR) Lu et al. [2020]

By default, the deterministic execution in Rhea can result in a high rate of aborts and a large number of RAW dependencies in high contention workloads. To circumvent this, the authors of Aria Lu et al. [2020] proposed a deterministic reordering scheme that allows cases of RAW dependencies to be converted to Write-after-read (WAR) dependencies without affecting serializability. In the example at the top of Figure 5.9, the given epoch is comprised of three transactions: T_1, T_2 and T_3. T_3 has a RAW dependency on T_1 and T_2, which also has a RAW dependency on transaction T_1. This means that if T_1 commits first, T_2 and T_3 will abort according to the conflict checking algorithm described in Section 5.3.1.

If these transactions are reordered such that $T_3 \rightarrow T_2 \rightarrow T_1$ is the commit order, each of the RAW dependencies becomes a WAR dependency, and all three

transactions in the epoch can commit. Deterministic reordering holds for any transaction with RAW dependencies as long as they do not also have WAR dependencies, as these would become RAW dependencies after reordering and thus would breach serialiazability. The authors of Aria demonstrate that the result is deterministic due to the fact that the reordering is determined solely by the write reservation and read/write sets of the transaction.

```
    def check_conflicts_deterministic_reordering(self) -> set[int]:
        for operator_name, write_set in self.write_sets.items():
            for t_id, ws in write_set.items():
                rs_keys = self.read_sets[operator_name].get(t_id, set())
                ws_keys = set(ws.keys())
                waw = self.has_conflicts(t_id, ws_keys,
                ↪  self.writes[operator_name])
                war = self.has_conflicts(t_id, ws_keys,
                ↪  self.reads[operator_name])
                raw = self.has_conflicts(t_id, rs_keys,
                ↪  self.writes[operator_name])
                if waw or (war and raw):
                    self.aborted_transactions.add(t_id)
        return self.aborted_transactions
```

Figure 5.10: Adapted conflict checking to support deterministic re-ordering. RAW dependencies are allowed if no WAR dependency exists

In order to implement deterministic reordering in Universalis, the conflict checking function presented in Figure 5.8 is modified to allow RAW dependencies if the transaction does not also have WAR dependencies. The implementation code can be found in Figure 5.10, which facilitates the deterministic reordering algorithm.

Fallback Mechanism

While the deterministic reordering algorithm focuses on reducing aborts caused by RAW dependencies, the authors of Aria Lu et al. [2020] propose a fallback strategy that aims to reduce aborts caused by a large number of WAW conflicts.After the commit phase, the read/write sets of all transactions in the epoch are fully determined. The fallback mechanism aims to utilise these dependencies between aborted transactions to generate a serial order that can be re-executed at the end of an epoch, similar to Calvin's Thomson et al. [2012] deterministic locking method. Note that the fallback mechanism is activated after the initial commit phase, meaning that transactions that could commit prior to the fallback phase are not executed.

The fallback mechanism in Universalis is implemented utilising a lock manager that is stored in each operator state and grants read/write locks to transactions in ascending TID order. As part of the fallback method, multi-partition transactions are also executed, necessitating the synchronisation of all workers prior to initiating the fallback procedure. Each worker sends aborted remote functions to the relevant workers so that they have a sequence of aborted transactions that have reads or writes on the state of the worker. Once all workers have responded, dependency graphs can be generated to determine the read and write

set dependencies for each transaction. After calculating these dependencies, they are locked to prevent concurrent access.

Before continuing with multi-partition transactions, workers must confirm that all other workers have locked the related keys to the transaction. When all workers have confirmed that the transaction's keys are locked, the workers can execute and commit their portion of the transaction and release the locks. If a transaction is still unable to commit as part of the fallback, it is added to the next epoch so that it can be retried. In addition, application aborts are still possible during the fallback phase, as the read/write values may have been modified by earlier transactions that committed prior to the fallback. As this optimization requires additional networking calls and computational resources, it is only enabled when the abort rate for a given Epoch exceeds 10%.

```
    def fallback_locking_mechanism(self, t_id, operator_name, key):
        key_id = (operator_name, key)
        if key_id in self.dibs:
            for dep_t_id in self.dibs[key_id]:
                event = asyncio.Event()
                if t_id in self.waiting_on_transactions:
                    self.waiting_on_transactions[t_id][dep_t_id] = event
                else:
                    self.waiting_on_transactions[t_id] = {dep_t_id: event}
                if dep_t_id in self.waiting_on_transactions_index:
                    self.waiting_on_transactions_index[dep_t_id].append(event)
                else:
                    self.waiting_on_transactions_index[dep_t_id] = [event]
            self.dibs[key_id].add(t_id)
        else:
            self.dibs[key_id] = {t_id}
```

Figure 5.11: The fallback locking mechanism operation for Rhea

Chapter 6

Evaluation

Multiple experiments are conducted to assess the Rhea transaction protocol in Universalis. In these tests, the system is subjected to a variety of workloads while metrics are used to evaluate various system aspects.

6.1 Workloads

To create scenarios in which the system can be evaluated, we use a subset of workloads from popular database benchmark specifications. We chose to adapt the YCSB Cooper et al. [2010] with the transactional extension, YCSB+T Dey et al. [2014], and the TPC-C specification Leutenegger and Dias [1993].

6.1.1 YCSB+T

YCSB is a flexible benchmarking specification designed to provide a standard method for comparing contemporary distributed databases. The standard YCSB does not support multi-partition transactions, but it remains a solid benchmark for single-key workloads, an essential workload type for Universalis. In the YCSB, all records to be used in the workload are inserted before the benchmark's operations are executed. We utilised the YCSB+T specification, which enforces atomic transactions in the workloads and adds multi-partition workloads, as the YCSB does not natively support multi-partition transactions. The evaluation functions we utilised are described in in Figure 6.1.

YCSB+T simulates a simple account system, where each row corresponds to a users account, containing an identifier and balance. Operations are performed on these accounts to emulate system usage. YCSB+T introduces the concept of a `ModifyReadWrite` transaction. As with the paper by Heus et al. [2021], we rename this function to `transfer` for simplicity. This function simulates the transfer of funds between accounts. In addition, YCSB+T defines a "closed economy workload" to measure transactional consistency, wherein no money enters or leaves the system during the workload.

After the workload has been completed, the account values for each account inserted are read during a validation phase in order to compute the consistency. These values are used in the following formula, as defined in Dey et al. [2014] to produce an anomaly score, where a score of 0 indicates there are no balance anomalies and the system is consistent:

$$\text{anomaly score} = \frac{|\text{total initial account balance} - \text{total final account balance}|}{\text{number of operations}}$$

Insert(*key*, *value*):

1. Insert the *value* at the given *key*

Update(*key*):

1. Retrieve the value at the given *key*
2. Increment the value by 1
3. Store the new value

Read(*key*):

1. Retrieve the value at the given *key*

Transfer(*key_a*, *key_b*):

1. Retrieve the values stored at *key_a* and *key_b*
2. Subtract 1 from the value stored at *key_a* and increment the value stored at *key_b*
3. If the value at *key_a* becomes negative, return failure to the user, else continue

Figure 6.1: Pseudocode of the implemented operator functions to support the YCSB workloads

In addition, for all YCSB+T, we select transaction keys using a Zipfian distribution. This distribution is a more realistic workload than a uniform workload because it simulates “hot" records, which are frequently accessed keys. In a real-world context, hot records could refer to items on a web store that are more frequently purchased than others, thereby inciting contention.

6.1.2 TPC-C

While the YCSB+T focuses on modern distributed applications, the TPC-C is an established benchmark designed to simulate a "generic wholesale supplier workload" Leutenegger and Dias [1993]. The transactions defined within the TPC-C contain significantly more operations and affect more partitions than the YCSB+T transactions. The TPC-C transactions can be compared to database system transactions rather than microservice transactions, providing insight into how Rhea performs under a traditional transactional workload.

In addition, the TPC-C imposes stringent limitations on the initial seeding of databases prior to executing transactional workloads. The order system is partitioned by warehouses, with each table seeded according to the set cardinalities specified in Table B.1. These cardinalities are used as a factor to control key contention, where decreasing the number of elements per warehouse decreases the key range for workload transactions, thereby increasing contention. In addition, by adding more warehouses, scalability experiments can be conducted to

gain insight into the performance of transactions across an increased number of partitions.

Due to the fact that Universalis does not currently support range queries, only the `NewOrder` and `Payment` transactions have been implemented, as detailed in Figure B.2. However, these two transactions account for 88% of the standard workload mix of the benchmark Lu et al. [2020]. Due to time constraints, we were unable to implement the isolation and consistency experiments specified by the specification for this project.

6.2 Metrics

Focusing on the operational metrics of throughput, latency, and concurrency abort rate, we evaluate the transaction implementation within Universalis.

6.2.1 Latency

Latency, which is synonymous with response time, is the duration a system requires to process a transaction. This provides an indication of the transaction implementation's performance, as longer response times indicate system inefficiency. Response time in Universalis is calculated as the difference between the time a transaction is queued in the Kafka ingress and the time it reaches the system egress. In the baseline application, transaction latency is measured as the duration of an HTTP request between the server and the benchmarking client.

6.2.2 Throughput

In terms of transaction processing, throughput is the number of transactions that a system can process per second. Maximum throughput is used as an indicator of the maximum theoretical performance a system is capable of achieving; however, for operational concerns, response time should also be considered, as a system with a high throughput but a lengthy response time is not user-friendly. Keeping this in mind, we calculate the number of transactions that enter the egress per second, which can only be measured when latency falls within the 90th percentile of the calculated latency.

6.2.3 Abort Rate

The transaction abort rate is an essential metric for optimistic concurrency control methods, such as those used in Rhea. The abort rate is the number of workload terminations that occur per second. This can be used as an indicator of the efficiency of the transaction system's underlying concurrency control methods. When calculating the abort rate, application aborts are excluded because they are based on user-defined application logic and are not inherent to concurrency control methods.

6.3 Benchmarking Clients

In order to evaluate Universalis, a benchmarking client was developed to support the execution of transactional workloads on the system. In addition, we implemented a baseline benchmarking application for use with the various metrics as a baseline measurement.

6.3.1 Universalis Client

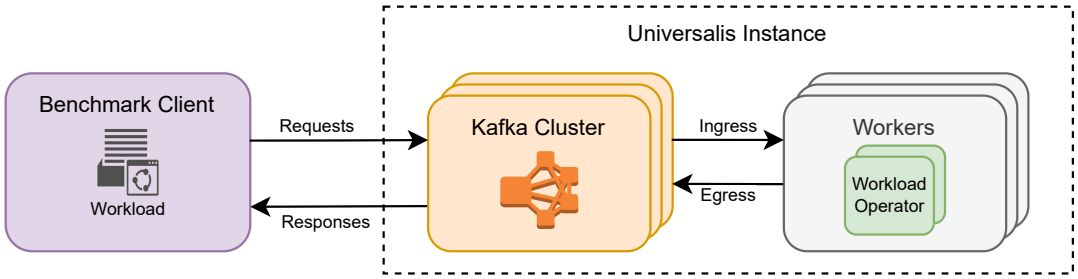


Figure 6.2: Universalis benchmark application

To use the YCSB+T and TPC-C workloads with Universalis, a stateful graph consisting of operators with the required function implementations for the given workload was created, as shown in Appendix B. At the start of a workload's execution, the created graph is submitted to Universalis.

The benchmark client contains the parameters that can be altered for each workload, such as the desired mix of transactions (e.g., read/update mix in acycsb); the distribution of the data; and the number of rows and operations to execute in the benchmark.

In addition, the benchmark client can be scaled to simulate concurrent function calls and increased load on the system. To send messages to Universalis, the benchmark client utilises the Universalis library's built in TCP call functionality. As responses from Universalis are returned asynchronously via a Kafka producer, the client utilises a Kafka consumer to collect the response messages. As each transaction is assigned a request id by Universalis upon ingestion, the benchmark client can link the request and response messages to calculate the metrics.

Before running the transaction mixture for both benchmark workloads, the desired number of rows are seeded with data, allowing the benchmark client to determine which keys are in use for each workload. As the insertions are not strictly part of the workload, they are not measured. After a YCSB+T workload has been executed, a validation phase commences, during which the benchmark client sends read requests for all the keys present in the workload, allowing the values to be checked for consistency.

For each transaction's start and end timestamps, the benchmark client records the timestamp when the message is sent to Universalis and the timestamp assigned by the Kafka consumer when receiving the response. The start and end times are used to calculate the latency and throughput of each transaction by dividing the time into one-second intervals and determining the number of transactions.

6.3.2 Baseline Client

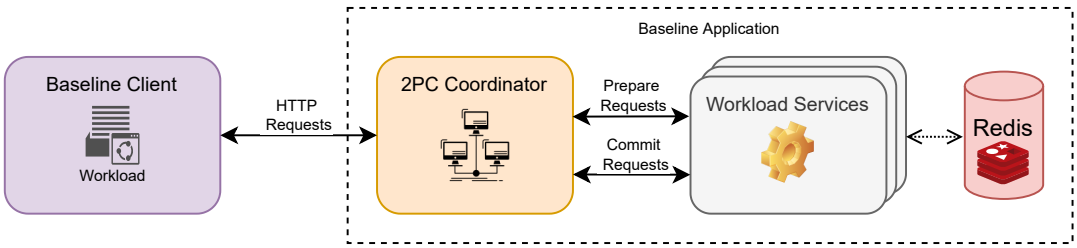


Figure 6.3: Baseline benchmark microservice client

In order to compare Rhea to other application designs, a microservice application supporting distributed transactions via 2PC was developed. This application design is standard for microservices, which are widely utilised in the real world. The baseline application was implemented in Python, with each service utilising the Quart[1] framework to provide asynchronous HTTP endpoints and a shared Redis instance for the state. To provide a fair comparison to Universalis, the microservices in the baseline also make use of concurrent execution of transactions using asyncio. The 2PC coordinator was implemented as a Quart HTTP client, with communication between with each service performed via HTTP. Using a simple modulo partitioning scheme based on the number of services, a distinct key range for the state is assigned to each service.

Moreover, the 2PC coordinator along with NGINX acts as a gateway for incoming requests from the benchmarking clients. The prepare and commit phases of each transaction are implemented using two HTTP round trips, with aborts determined using the HTTP status codes 409 and 500, respectively. Within the services, 2PL, as described in Section 2.3.1 is used as a locking algorithm to provide serializable transactions, with the NO_WAIT deadlock avoidance scheme, as it performs better in high contention and allows for more straightforward measurements of the abort rate compared to other deadlock avoidance algorithms.

The baseline client accepts workload input parameters in the same format as the Universalis benchmarking client described in, making parameter consistency between the two benchmarks effortless. Due to time constraints, only workloads from the YCSB+T was implemented for the baseline client, with each microservice implementing the functions listed in Figure 6.1.

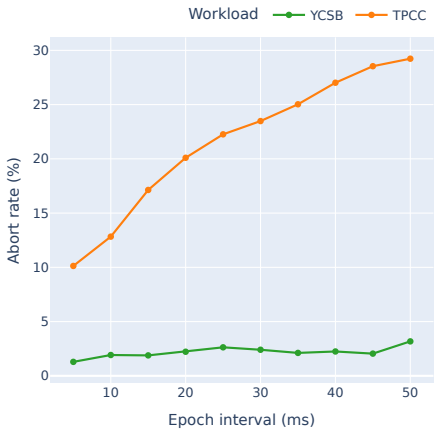
6.4 Experiments

To evaluate Rhea, four experiments were conducted. So that we could evaluate the validity of the results, each experimental workload was repeated five times under identical conditions. The first experiment assessed the effect of modifying the sequencers epoch interval parameter. Next, we assessed the impact of the optimisations introduced in Section 5.3.3. The third experiment compared Rhea's transaction performance to the baseline under varying levels of write contention. Lastly, we compared Rhea's scalability to that of the baseline implementation.

[1] https://quart.palletsprojects.com/en/latest/

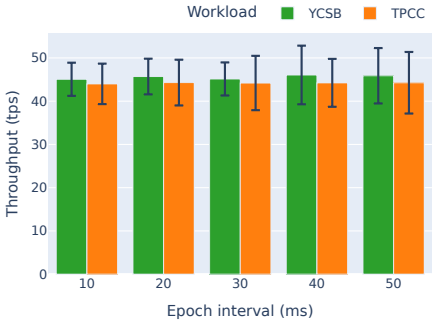
The first three experiments were conducted on a computer with a 32GB RAM and an 8-core, 16-thread AMD Ryzen 3700X processor. Experiment four's scalability tests were conducted on a machine with a 64-core, 128-thread AMD EPYC 7H12 processor and 256GB of RAM.

6.4.1 Epoch Interval



(a) Abort rate

(b) Mean latency with standard deviation



(c) Mean throughput with standard deviation

Figure 6.4: Epoch interval experiments: Comparison in latency and abort rate based on adjusting epoch interval in Universalis w/Rhea

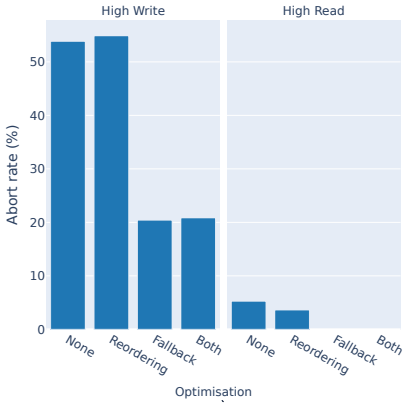
In the first experiment, the epoch interval was varied in increments of 5 milliseconds to determine the effect on the abort rate and latency of Rhea for the YCSB+T and TPC-C workloads. By increasing the duration of the epoch interval, the sequencers in the workers provide more time for transactions to enter the epoch, thereby increasing the number of transactions being processed simultaneously. Two Universalis workers and two partitions per operator were utilised in this experiment, with a constant input request throughput of 50 tps. This allowed for a stable throughput, as shown in Figure 6.4c. The TPC-C workload was executed with 2 warehouses and a scale factor of 100. The YCSB+T workload consisted of 100 rows and 500 operations, of which 80% were read operations and 20% were transfer operations. The outcomes of the experiments can be found in. Figure 6.4.

The first observation is that the TPC-C workload has a higher abort rate and latency than the YCSB+T workload. This is because TPC-C-defined transactions access more keys across partitions than YCSB+T-defined transactions. With a greater number of keys accessed per transaction, the likelihood of conflicts

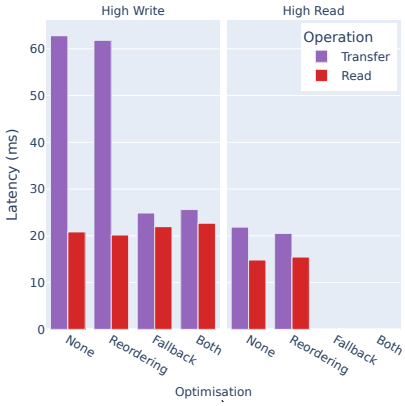
increases. This is especially evident in Figure 6.4a, where the TPC-C results show a much faster increase in the abort rate than the YCSB+T results. Moreover, the observed increase in latency is the result of a greater number of aborts and network round trips for the TPC-C workloads than for the YCSB+T workloads. Consequently, each transaction takes longer to complete because it must wait for responses of remote calls.

Furthermore, in Figure 6.4b, between the 5ms and 10ms interval, latency for both workloads decreases slightly. This is due to the epoch mechanism's overhead. Smaller epoch intervals result in an increase in the number of transactions waiting in the epoch queue. As the synchronisation point between workers adds overhead to each epoch, this overhead is added to the latency caused by transaction processing, resulting in an increase in request latency overall. After this threshold, the latency and abort rate increase for both workloads, as an increasing number of transactions must await the completion of remote function calls.

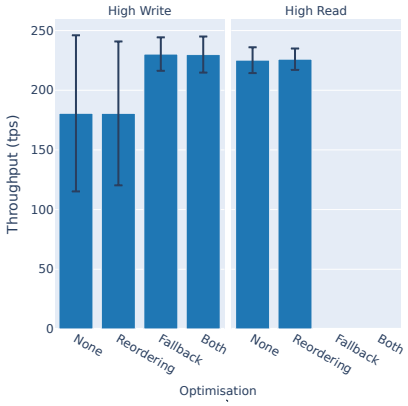
6.4.2 Optimisations



(a) Abort rate



(b) Mean latency



(c) Mean throughput with standard deviation

Figure 6.5: Optimisation experiment results: Comparison in metrics between optimisation strategies in Universalis w/Rhea

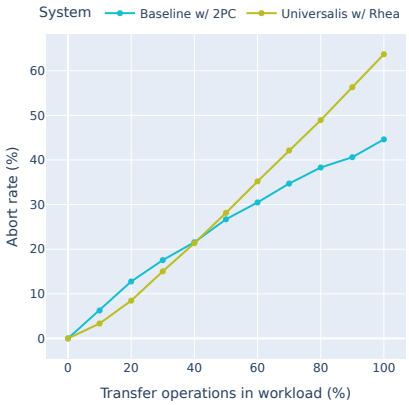
In the second experiment, we examine the impact of implementing the optimisations defined in Section 5.3.3 under two YCSB+T workloads. On Rhea, the workloads were executed under four conditions: no optimizations, deterministic reordering, fallback strategy, and both optimizations. We decided not to run the

fallback strategy for the high read workloads, as there would be very few WAW conflicts and the strategy would not be activated. The A and B YCSB workloads were executed to simulate write-intensive and read-intensive workloads, respectively. To generate multi-partition write transactions, the `update` was replaced with the `transfer` function from YCSB+T. Experiments were conducted with a client request rate of 250 tps, which was well within the maximum throughput tolerance for both types of workloads.

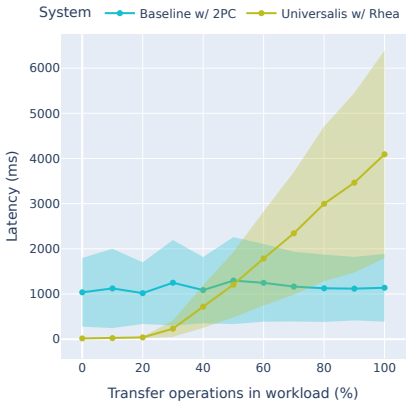
In Figure 6.5a, we observe a significant decrease in the average abort rate with the fallback strategy optimisation under high write contention, which corresponds to the observed decrease in average latency in Figure 6.5b. The fallback mechanism reduces the number of transactions that are aborted and moved to the subsequent epoch. Although the fallback mechanism increases the number of waits for multi-partition transactions, this effect compensates for the sluggishness caused by multiple retries following aborts. As the fallback mechanism is intended to reduce WAW conflicts, this decrease in latency is primarily observed in the `transfer` function. The fallback strategy causes a slight increase in latency for `read` transactions, but this increase is negligible compared to the reduction in latency for `transfer` transactions.

For write-intensive workloads, the deterministic reordering optimisation has a negligible effect on the three metrics in Figure 6.5. This is expected, as the reordering algorithm only reduces conflicts due to RAW dependencies, which occur less frequently in workloads with fewer reads. In read-heavy workloads, this we observe a small reduction in the latency of `transfer` and `read` operations. We observe a small reduction in the latency of `transfer` and `read` operations under read-intensive workloads. However, there is no discernible improvement in throughput, indicating that the deterministic reordering is either ineffective or incorrectly implemented.

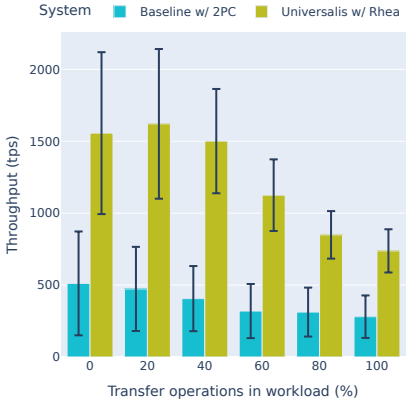
6.4.3 Performance



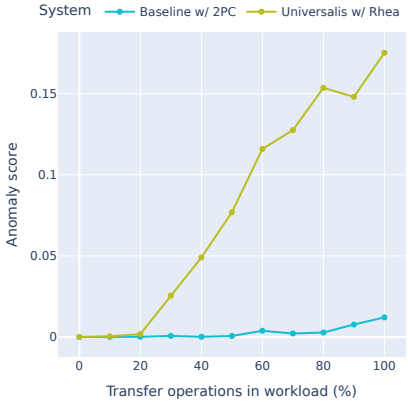
(a) Abort rate



(b) Mean latency with standard deviation



(c) Mean throughput



(d) Anomaly score

Figure 6.6: Performance experiment results: Increasing write contention in workload for baseline system & Universalis w/Rhea

In the third experiment, the transactional performance of Rhea is compared to that of the system introduced in Figure 6.3. To achieve this, we observed the effects of increasing write contention in the YCSB+T workload from 0% to 100%, utilising a combination of the `read` and `transfer` functions to provide a closed economy that can be used for consistency testing. The workload consisted of 1000 rows and 10000 operations. We utilised an input throughput of 2500 tps for Rhea and 1000 tps for the baseline client, as these values represented 80% of the maximum throughput calculated respectively for this workload. The baseline client and Universalis instance for Rhea were assigned two microservice clients and workers, respectively. The fallback mechanism was enabled for this experiment based on the positive results of the fallback mechanism in write-intensive scenarios for Rhea presented in. Section 6.4.2.

As both the baseline client and Rhea use aborts to maintain transaction isolation, they exhibit a similar proportion of aborts as the number of `transfer` operations in the workload increases, as shown in Figure 6.6a. This is because the number of writes increases the number of WAW and RAW dependencies. As write contention increases and Rhea retries aborted transactions, which may abort multiple times in subsequent epochs, we observe a marginally higher abort rate.

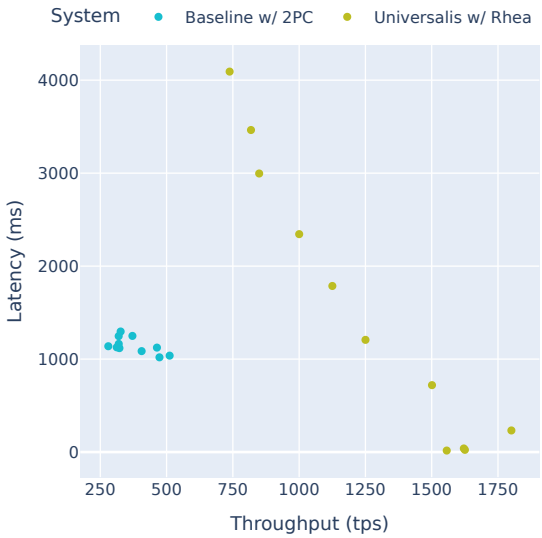


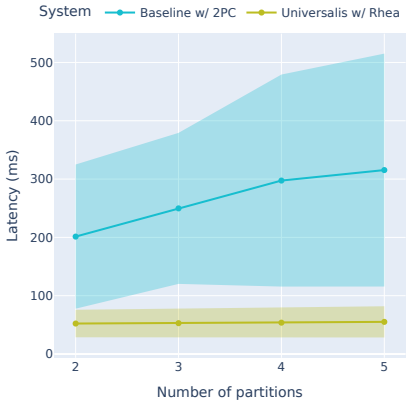
Figure 6.7: Average latency per throughput

As demonstrated in Figure 6.6b, the retrying of aborts in Rhea causes a significant increase in latency as write contention nears 40% compared to the baseline application. At 100% write contention, Rhea's average latency increases to 4 seconds, with a significantly larger standard deviation, because some transactions may be retried multiple times more frequently than others, resulting in a variance in individual latency.

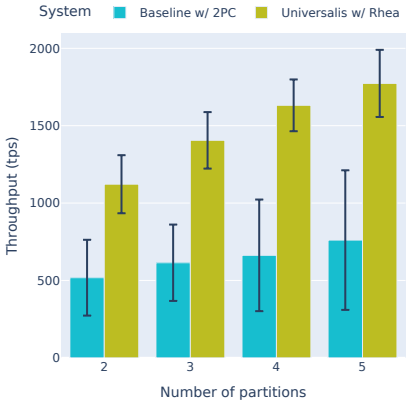
In Figure 6.6b, we observe that the additional latency at high write contention reduces Rhea's throughput to approximately 50% of its peak latency. The baseline application's throughput is significantly lower in all cases when compared to Rhea, primarily due to the substantial overhead of 2PC, as we observe a 50% reduction in throughput at 100% write contention compared to 0%. This effect occurs despite the stable latency because in the baseline application, aborted transactions are not retried. Additionally, as the aborted transactions are not included in the calculation of throughput, fewer transactions are processed per second, which becomes evident when we plot throughput against latency in Figure 6.7.

Lastly, we observe in Figure 6.6d that, under conditions of low write contention, both Rhea and the baseline application experience few consistency anomalies. Few rows are being updated with new values, reducing the likelihood of anomalies occurring. Nevertheless, as the write contention exceeds 30%, we observe a sharp increase in the anomaly score for Rhea, up to a score of approximately 0.15, whereas the anomaly score for the baseline application reaches a maximum of approximately 0.01. The increase in anomalies is likely attributable to the sequencer's lack of support for linearizability of operations. This means that the order in which transactions are applied may not be the same as the order in which they were ingested into the ingress, which may result in inconsistencies as transaction protocols may enforce a different order than the client expects.

6.4.4 Scalability



(a) Mean latency with standard deviation



(b) Mean throughput

In the final experiment, we examined the impact of increasing the number of partitions to scale up transaction protocols. n services and workers were added to the base application and Rhea in order to add n partitions. As in previous experiments, we utilised the YCSB+T workload A, exchanging the `update` and `transfer` functions to simulate multi-partition transactions. To simulate additional load, we also scaled up the client request throughput using the following formula: throughput = number of partitions partitions $\times$ 1000, to simulate additional load. We increased the number of operations linearly with client throughput using the following formula: number of operations = number of partitions $\times$ 10000.

In Figure 6.8a, a 100ms increase in latency is observed as the number of partitions increases from 2 to 5. This is likely due to the 2PC coordinator's increased workload reducing its capacity to fulfil requests to the microservices. In contrast, Rhea experiences no change in latency as the number of partitions increases, indicating that the system remains stable despite the increased load as it scales upwards.

Furthermore, in Figure 6.8b, we observe a linear increase in throughput for both the baseline and Rhea, with the rate of Rhea's throughput increase being greater than that of the baseline, as would be expected due to static latency. In all cases, the throughput values for Rhea and the baseline are significantly lower than the input throughput. In the case of Rhea, where we do not observe a significant increase in latency, this may be due to the inability of the Kafka producers in Universalis to scale.

Chapter 7

Discussion

In this section, we will examine the contributions of this project as described in Section 1.2. In addition, we will discuss the limitations we faced during the process.

7.1 Contributions

In this thesis, we made four contributions to the field of distributed transactions, specifically within SFaaS transactions.

Conceptual SFaaS transaction designs

Due to the exploratory nature of the thesis, multiple transaction protocol avenues were studied. The increasing prevalence of CRDTs in distributed systems and their use in stream processing systems such as Cloudburst Sreekanti et al. [2020] prompted us to attempt a system implementation in Universalis. As demonstrated in Section 5.1, this approach did not succeed primarily due to the method's exponentially increasing complexity. However, as CRDTs are a relatively new concept, there is still the possibility of incorporating them into a transaction processing system. For instance, CRDT-based methods could be used to optimise transactions in which the operations commute, such as addition.

In addition, we presented an Epoch-based 2PC protocol that was not implemented in Universalis due to time constraints. However, the use of message batching in Epoch 2PC could result in a significant performance increase in Universalis compared to a standard 2PC protocol. If Universalis supports replication in the near future, the optimistic concurrency control in Epoch 2PC could be a good fit for the system, as locks are held for shorter durations than in a typical 2PC implementation. Nevertheless, the primary issue with 2PC-based methods is the centralised coordinator, which, despite the fact that it could be partitioned to improve performance and eliminate the single point of failure, would likely experience performance issues when horizontally scaling. In addition, an additional coordinator adds complexity to Universalis, especially in terms of fault tolerance, as the number of elements to be checkpointed increases.

Protocol implementation & optimisations

Rhea, the deterministic protocol that we designed and implemented in Universalis, is heavily based on the Aria Lu et al. [2020] deterministic database, but has been modified for stateful functions. The sequencer is designed for use with

Universalis workers, generating TID based on worker identifiers and transaction counts. Although this method generates unique TIDs between workers, it does not create a linearizable ordering of functions across all workers within an Epoch, which could result in inconsistencies with client input requests.

As the implementation utilises the built-in state operations within Universalis, it has no effect on the user's programming model. This adheres to the accessibility design philosophy of Universalis, as described in Section 4.1.1, as users can create transactional stateful functions without implementing transaction logic.

Both deterministic reordering and the fallback mechanism were implemented in Rhea by adapting them from Aria. Due to the required additional locking and network calls, the fallback mechanism added a significant amount of complexity to the code within the Universalis workers. Moreover, in Section 6.4.2, we noted that the deterministic reordering did not appear to have any effect on the evaluated metrics, which may be due to implementation issues, specifically the reordering algorithm.

Benchmarking applications

The benchmarking application for Universalis enables the calculation of transaction metrics within Universalis. Since Universalis only supports asynchronous endpoints and workloads must be converted to support stateflow graphs, no existing benchmarking client could be used; thus, a custom benchmarking client was created. The client is able to execute workloads based on a configuration file and generate abort rate, latency, and throughput metrics.

The baseline client is based on a microservice architecture, using the 2PC protocol. This serves as a good baseline, as one of the primary demographics for which Universalis is intended are application developers who wish to create microservice-like applications. However, the lack of a TPC-C workload implementation restricts the client's evaluation capabilities, as it does not support large multi-partition transactions.

In addition, although consistency testing has been implemented in both clients for the YCSB+T, it is significantly less comprehensive than consistency testing for other benchmarks. TPC-C provides significantly more rigorous consistency testing than YCSB+T, yielding more accurate results and highlighting instances of inconsistency. In addition, the client lacked isolation tests. Due to the absence of these tests, there is still work to be done in order to bring the benchmark clients up to par with existing benchmarks.

Evaluation of Rhea

In our evaluation of Rhea, we determined that in terms of throughput and latency for YCSB+T workloads, Rhea performs significantly better than the baseline application. In high write contention YCSB+T workloads, despite the substantial benefits of the fallback strategy, we observed a significant increase in average latency of up to 4 seconds per request. As reviewed in Harding et al. [2017], none of the distributed transaction protocols performed well under comparable load conditions. In addition, scenarios with high write contention could be mitigated, for instance by employing dynamic partitioning to reduce key contention.

As the write contention increased beyond 30%, we observed relatively high anomaly scores in the YCSB+T consistency test. These issues could be influenced by the sequencer's lack of linearizability. However, it is also likely that there are bugs within the transaction protocol or Universalis, which arise under specific scenarios induced by high write contention.

7.2 Limitations

Throughout the duration of this project, we were confronted with obstacles that affected the final outcome.

Limited state query support

The absence of extended query support, such as batch queries, range queries, and deletes, which are crucial for many transaction use cases, was a significant limitation of Rhea. In addition, the lack of such queries restricted the evaluation, as many workloads from the YCSB+T and the TPC-C could not be utilised.

Lack of consistency & isolation testing

Consistency testing for the YCSB+T workloads were implemented in both the Universalis and baseline benchmarking clients; however, the anomaly score produced by the closed economy workload is significantly less informative than the multiple metrics produced by the TPC-C consistency tests. This restricted the inferences we could make regarding the consistency of both systems. In addition, the absence of transaction isolation tests, such as those defined in the TPC-C, prevents us from pragmatically validating Rhea's isolation level.

Lack of fault-tolerance mechanism

A major advantage of deterministic protocols is their simplified fault tolerance. We were unable to implement and evaluate the fault tolerance mechanism for Rhea, as Universalis lacked a fully implemented checkpointing mechanism. Due to this, it was not possible to fully implement Rhea's fault tolerance within the project's timeline.

Runtime instability

We encountered runtime issues while executing Universalis with Rhea during the experiments. Particularly, we encountered freezing issues when executing workloads with a large number of rows, such as in the TPC-C workloads with > 4 warehouses and YCSB+T workloads with > 1000 rows; and a large number of partitions (> 5). Due to these issues, we were unable to scale the number of rows with the number of partitions during our scalability tests.

Chapter 8

Conclusion

In the concluding section of this thesis, we summarise our work in relation to the research questions posed in Section 1.1 and outline potential future research.

8.1 Summary

The objective of this thesis was to implement transactional performance in a SFaaS dataflow system. Following a review of the scientific literature in the fields of distributed transaction processing and operational stream processing, we determined that few SFaaS systems implement performant transactions while maintaining a simple programming model. We created two conceptual protocols and implemented a third protocol, Rhea, in the Universalis prototype SFaaS dataflow system. Consequently, we were able to answer our initial research question:

> **How can distributed transactions be implemented in SFaaS dataflow systems?**
>
> A CRDT-based, 2PC-based, and deterministic-based strategy for SFaaS dataflow transactions was presented. While there is still room for the first two approaches to be utilised effectively, we determined that a deterministic-based approach was best suited for SFaaS dataflow systems due to the lack of a distributed coordinator, minimal changes to the programming model, and streamlined fault tolerance.

After implementing the deterministic transaction protocol, Rhea, with two optimizations to reduce transaction aborts, we conducted experiments to evaluate its performance based on the metrics described in Section 6.2. Consequently, we were able to provide an answer our second research question:

How do distributed transaction protocols perform in SFaaS dataflow systems?

Rhea outperformed the 2PC microservice application in performance experiments, achieving more than double the write throughput and half the latency at varying contention levels. However, Rhea's average latency increased exponentially in environments with high write contention, indicating that despite the relatively strong performance, the cost of distributed transactions in SFaaS dataflow systems remains high. In addition, we discovered that there are still issues pertaining to transactional consistency that must be resolved before we can reach a definitive conclusion to this question.

While performance is an important factor in distributed transactions, maintaining a high level of transaction isolation is an equally important property. We address this in our third research question:

Which isolation guarantees can be supported in distributed transaction protocols within SFaaS dataflow systems?

Rhea is an adaptation of the deterministic database Aria Lu et al. [2020], which supports serializable transactions. This isolation level is enforced by disallowing RAW and WAW dependencies, preventing transactions from reading values that were modified during or after their execution. We cannot state with certainty that Rhea supports serializable transactions because we were unable to conduct isolation experiments.

8.2 Future Work

As a result of the work presented in this thesis, we propose numerous future research directions.

Expand the supported state queries

As identified in Section 7.2, Universalis lacks specific state queries, such as range and delete queries. Extending the set of supported queries to include the state would increase the system's use cases and permit the implementation of additional benchmarking workloads for further evaluation.

Explore further transaction protocols

While the purpose of this project was to support serializable transaction protocols due to their reliability for application developers, future research could investigate loosening transaction constraints to improve performance. For instance, future work could examine the implementation of SAGAs in Universalis, which rely on BASE properties to provide eventual consistency.

Enhance benchmarking client

Future research could expand the benchmarking client to support additional workloads, such as the entire YCSB+T and TPC-C workload sets. In addition, addition of the TPC-C consistency and isolation tests would enable more rigorous testing of transaction protocols. Future research could also employ multiple benchmarking clients to simulate multiple users with distinct connections.

Support user-defined isolation levels

As discussed in Section 2.2, increased levels of isolation can negatively affect performance. Numerous DDBMS enable users to set their preferred isolation level, allowing them to optimise their application's performance. Future research could adapt or implement novel protocols that permit developers to determine their own isolation levels.

Improved scalability tests

Universalis is intended to be deployed on large-scale, distributed machines; however, due to the runtime issues discussed in Section 7.2, we were unable to effectively test its scalability. Future research could perform wide scale evaluations on more powerful hardware. Future research could utilise more robust hardware to conduct extensive evaluations. Additionally, research could conduct wide area network (WAN) tests to evaluate global performance.

Appendix A

Universalis Programming Model

```python
from universalis.common.local_state_backends import LocalStateBackend
from universalis.common.operator import Operator
from universalis.common.stateflow_graph import StateflowGraph
from universalis.common.stateflow_ingress import IngressTypes
from universalis.universalis import Universalis


user_operator = Operator('user', n_partitions=2)
stock_operator = Operator('stock', n_partitions=2)
order_operator = Operator('order', n_partitions=3)

g = StateflowGraph('cart', operator_state_backend=LocalStateBackend.REDIS)

user_operator.register_stateful_functions(
    user.CreateUser(),
    user.AddCredit(),
    user.SubtractCredit()
)
g.add_operator(user_operator)

stock_operator.register_stateful_functions(
    stock.CreateItem(),
    stock.AddStock(),
    stock.SubtractStock()
)
g.add_operator(stock_operator)

order_operator.register_stateful_functions(
    order.CreateOrder(),
    order.AddItem(),
    order.Checkout()
)
g.add_operator(order_operator)

g.add_connection(order_operator, user_operator, bidirectional=False)
g.add_connection(order_operator, stock_operator, bidirectional=False)

await universalis.submit(g, user, order, stock)
```

Figure A.1: Definition of the Universalis stateflow graph from Figure 4.1

```python
from universalis.common.operator import StatefulFunction


class CreateItem(StatefulFunction):
    async def run(self, key: str, name: str, price: int):
        await self.state.put(key, {'name': name, 'price': price, 'stock': 0})
        return key


class AddStock(StatefulFunction):
    async def run(self, key: str, stock: int):
        item_data = await self.state.get(key)
        item_data['stock'] += stock
        await self.state.put(key, item_data)
        return item_data


class SubtractStock(StatefulFunction):
    async def run(self, key: str, stock: int):
        item_data = await self.state.get(key)
        item_data['stock'] -= stock
        await self.state.put(key, item_data)
        return item_data
```

Figure A.2: Example stock operator definition in Universalis

```python
from universalis.common.operator import StatefulFunction


class CreateUser(StatefulFunction):
    async def run(self, key: str, name: str):
        await self.state.put(key, {'name': name, 'credit': 0})
        return key


class AddCredit(StatefulFunction):
    async def run(self, key: str, credit: int):
        user_data = await self.state.get(key)
        user_data['credit'] += credit
        await self.state.put(key, user_data)
        return user_data


class SubtractCredit(StatefulFunction):
    async def run(self, key: str, credit: int):
        user_data = await self.state.get(key)
        user_data['credit'] -= credit
        await self.state.put(key, user_data)
        return user_data
```

Figure A.3: Example user operator definition in Universalis

Appendix B

Benchmarking

```python
from universalis.common.stateful_function import StatefulFunction


class NotEnoughCredit(Exception):
    pass


class Insert(StatefulFunction):
    async def run(self, key: str):
        await self.put(key, 1)
        return key


class Read(StatefulFunction):
    async def run(self, key: str):
        data = await self.get(key)
        return data


class Update(StatefulFunction):
    async def run(self, key: str):
        new_value = await self.get(key)
        new_value += 1

        await self.put(key, new_value)
        return key


class Transfer(StatefulFunction):
    async def run(self, key_a: str, key_b: str):
        value_a = await self.get(key_a)

        self.call_remote_async(
            function=Update,
            key=key_b,
            params=(key_b,)
        )

        value_a -= 1

        if value_a < 0:
            raise NotEnoughCredit(f'Not enough credit for user: {key_a}')

        await self.put(key_a, value_a)

        return key_a
```

Figure B.1: Function definitions for code for YCSB operator in Universalis

NewOrder():

1. Select(whouse-id) from Warehouse
2. Select(dist-id, whouse-id) from District
3. Update(dist-id, whouse-id) in District
4. Select(customer-id, dist-id, whouse-id) from Customer
5. Insert into Order
6. Insert into New-Order
7. For each item (10 items):
    - Select(item-id) from Item
    - Select(item-id, whouse-id) from Stock
    - Update(item-id, whouse-id) in Stock
    - Insert into Order-Line
8. Commit

Payment():

1. Select(whouse-id) from Warehouse
2. Select(dist-id, whouse-id) from District
3. Select(customer-id, dist-id, whouse-id) from Customer
4. Update(whouse-id) in Warehouse
5. Update(dist-id, whouse-id) in District
6. Update(customer-id,dist-id,whouse-id) in Customer
7. Insert into History
8. Commit

Figure B.2: Pseudocode of TPC-Cs *NewOrder* and *Payment* functions

Table Name	**Cardinality**
Warehouse	1
District	10
Customer	30k
History	30k
Order	30k
New-Order	9k
Order-Line	300k
Stock	100k
Item*	100k

Table B.1: TPC-C table cardinality per warehouse. *Item does not scale with the number of warehouses

Bibliography

D. J. Abadi and J. M. Faleiro. An Overview of Deterministic Database Systems. *Commun. ACM*, 61(9):78–88, Aug. 2018. ISSN 0001-0782. doi: 10.1145/3181853. URL https://doi.org/10.1145/3181853. Publisher: Association for Computing Machinery.

A. Akhter, M. Fragkoulis, and A. Katsifodimos. Stateful Functions as a Service in Action. *Proc. VLDB Endow.*, 12(12):1890–1893, Aug. 2019. ISSN 2150-8097. doi: 10.14778/3352063.3352092. URL https://doi.org/10.14778/3352063.3352092. Publisher: VLDB Endowment.

I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. Serverless computing: Current trends and open problems. *Research Advances in Cloud Computing*, pages 1–20, Dec. 2017. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/978-981-10-5026-8_1. ISBN: 9789811050268 Publisher: Springer Singapore.

H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 1–10, New York, NY, USA, May 1995. Association for Computing Machinery. ISBN 978-0-89791-731-5. doi: 10.1145/223784.223785. URL http://doi.org/10.1145/223784.223785.

P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981. ISSN 0360-0300. doi: 10.1145/356842.356846. URL https://doi.org/10.1145/356842.356846. Place: New York, NY, USA Publisher: Association for Computing Machinery.

P. A. Bernstein and N. Goodman. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983. ISSN 0362-5915. doi: 10.1145/319996.319998. URL https://doi-org.tudelft.idm.oclc.org/10.1145/319996.319998. Place: New York, NY, USA Publisher: Association for Computing Machinery.

P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows, June 2015a. URL http://arxiv.org/abs/1506.08603. arXiv:1506.08603 [cs].

P. Carbone, A. Katsifodimos, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38, June 2015b.

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152. event-place: Indianapolis, Indiana, USA.

A. Dey, A. Fekete, R. Nambiar, and U. Röhm. YCSB+T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230, 2014. doi: 10.1109/ICDEW.2014.6818330.

N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, Today, and Tomorrow. *Present and Ulterior Software Engineering*, pages 195–216, Nov. 2017. URL https://link-springer-com.tudelft.idm.oclc.org/chapter/10.1007/978-3-319-67425-4_12. ISBN: 9783319674254 Publisher: Springer, Cham.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19 (11):624–633, Nov. 1976. ISSN 0001-0782. doi: 10.1145/360363.360369. URL http://doi.org/10.1145/360363.360369.

E. v. Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. L. Abad, and A. Iosup. The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms. *IEEE Internet Computing*, 23(6):7–18, 2019. doi: 10.1109/MIC.2019.2952061.

M. Garriga. Towards a Taxonomy of Microservices Architectures. In M. C. Antonio and Roveri, editors, *Software Engineering and Formal Methods*, pages 203–218. Springer International Publishing, 2018. ISBN 978-3-319-74781-1.

J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, Mar. 2006. ISSN 0362-5915. doi: 10.1145/1132863.1132867. URL https://doi.org/10.1145/1132863.1132867.

R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.*, 10(5):553–564, Jan. 2017. ISSN 2150-8097. doi: 10.14778/3055540.3055548. URL https://doi.org/10.14778/3055540.3055548. Publisher: VLDB Endowment.

M. d. Heus, K. Psarakis, M. Fragkoulis, and A. Katsifodimos. Distributed Transactions on Serverless Stateful Functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, pages 31–42. Association for Computing Machinery, 2021. ISBN 978-1-4503-8555-8. doi: 10.1145/3465480.3466920. URL https://doi.org/10.1145/3465480.3466920.

R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, Aug. 2008. ISSN 2150-8097. doi: 10.

14778/1454159.1454211. URL https://doi.org/10.14778/1454159.1454211. Publisher: VLDB Endowment.

A. Katsifodimos and M. Fragkoulis. Operational stream processing: Towards scalable and consistent event-driven applications. *Advances in Database Technology - EDBT*, 2019-March:682–685, 2019. ISSN 23672005. doi: 10.5441/002/EDBT.2019.86. ISBN: 9783893180813 Publisher: OpenProceedings.org.

H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981. ISSN 0362-5915. doi: 10.1145/319566.319567. URL https://doi-org.tudelft.idm.oclc.org/10.1145/319566.319567. Place: New York, NY, USA Publisher: Association for Computing Machinery.

L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi-org.tudelft.idm.oclc.org/10.1145/279227.279229. Place: New York, NY, USA Publisher: Association for Computing Machinery.

S. T. Leutenegger and D. Dias. A Modeling Study of the TPC-C Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 22–31, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0-89791-592-5. doi: 10.1145/170035.170042. URL https://doi.org/10.1145/170035.170042. event-place: Washington, D.C., USA.

Y. Lu, X. Yu, L. Cao, and S. Madden. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.*, 13(12):2047–2060, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407808. URL https://doi.org/10.14778/3407790.3407808. Publisher: VLDB Endowment.

Y. Lu, X. Yu, L. Cao, and S. Madden. Epoch-Based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.*, 14(5):743–756, Jan. 2021. ISSN 2150-8097. doi: 10.14778/3446095.3446098. URL https://doi.org/10.14778/3446095.3446098. Publisher: VLDB Endowment.

B. Medjahed, M. Ouzzani, and A. K. Elmagarmid. Generalization of ACID Properties. In L. LIU and M. T. ÖZSU, editors, *Encyclopedia of Database Systems*, pages 1221–1222. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_736. URL https://doi.org/10.1007/978-0-387-39940-9_736.

J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.*, 8(13):2134–2145, Sept. 2015. ISSN 2150-8097. doi: 10.14778/2831360.2831367. URL https://doi.org/10.14778/2831360.2831367. Publisher: VLDB Endowment.

J. Michels, K. Hare, K. Kulkarni, C. Zuzarte, Z. Liu, B. Hammerschmidt, and F. Zemke. The new and improved SQL:2016 standard. *ACM SIGMOD Record*, 47:51–60, Dec. 2018. doi: 10.1145/3299887.3299897.

Mohammad Roohitavaf. Calvin, the Magic of Determinism, Sept. 2020. URL https://www.mydistributed.systems/2020/08/calvin.html.

D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2. event-place: Philadelphia, PA.

M. Shahrad, J. Balkind, and D. Wentzlaff. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075. Association for Computing Machinery, 2019. ISBN 978-1-4503-6938-1. doi: 10.1145/3352460.3358296. URL https://doi.org/10.1145/3352460.3358296.

M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24549-7. event-place: Grenoble, France.

V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407836. URL https://doi.org/10.14778/3407790.3407836. Publisher: VLDB Endowment.

M. Steen and A. S. Tanenbaum. A Brief Introduction to Distributed Systems. *Computing*, 98(10):967–1009, Oct. 2016. ISSN 0010-485X. doi: 10.1007/s00607-016-0508-7. URL https://doi.org/10.1007/s00607-016-0508-7. Publisher: Springer-Verlag.

R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, Portland OR USA, June 2020. ACM. ISBN 978-1-4503-6735-6. doi: 10.1145/3318464.3386134. URL https://dl.acm.org/doi/10.1145/3318464.3386134.

A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, Sept. 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920855. URL https://doi.org/10.14778/1920841.1920855.

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*, pages 1–12. Association for Computing Machinery, 2012. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213838. URL https://doi.org/10.1145/2213836.2213838.

C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 401–412, 2018. doi: 10.1109/ICDE.2018.00044.

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)*, pages 2024–2027, 2019. doi: 10.1109/ICDE.2019.00230.

H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, Nov. 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

W. Zorgdrager, K. Psarakis, M. Fragkoulis, E. Visser, and A. Katsifodimos. Stateful Entities: Object-oriented Cloud Applications as Distributed Dataflows. Nov. 2021. doi: 10.48550/arxiv.2112.00710. URL https://arxiv.org/abs/2112.00710v1.