DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Consistency in Stateful FaaS Platforms

Author:
Wouter VAN LIL

Supervisor:
Dr. Asterios KATSIFODIMOS
Daily supervisor:
Dr. Burcu ÖZKAN
Daily co-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

August 22, 2023

Declaration of Authorship

I, Wouter VAN LIL, declare that this thesis titled, "Consistency in Stateful FaaS Platforms" 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:

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Consistency in Stateful FaaS Platforms

by Wouter VAN LIL

Serverless computing has allowed developers to write pieces of code comprising solely of the necessary functionality whilst not having to think about the underlying infrastructure. One prominent model is Function-as-a-Service (FaaS), where the code is structured into functions that run based on incoming events. This model was initially stateless, as new function calls can be instantiated at any location and there is no clear consensus on state whenever multiple instances are running simultaneously. Access to external persistent state is slow, making FaaS not suitable for low latency applications. Recent works have found different ways of incorporating state, resulting in Stateful FaaS (SFaaS). With the addition of state, these components are perfectly suited for distributed transactions. SFaaS frameworks try to outperform one another on metrics such as throughput and latency, but less work is performed on consistency.

In this thesis we look at the work on consistency of SFaaS transactions that has been done. We take Jepsen, a framework for testing transactions in distributed systems, and show that it can also be applied to SFaaS transactions. We then proceed to apply it to three SFaaS frameworks. We use Elle as a consistency checker to verify that the three frameworks comply with the consistency level they are advertised as. We have found that two of the tested frameworks do not have the consistency level promised. Facets of the SFaaS frameworks seem to have been overlooked, and diverging from the laid out benchmarking path quickly results in unintended behaviour.

Acknowledgements

Firstly, I would like to thank my supervisor, Asterios Katsifodimos, for assisting me in finding an interesting and relevant thesis subject, directing me towards useful information sources, and guiding the research by discussing the largest decisions to be made.

Additionally, I would like to thank my co-supervisor, Burcu Özkan, for giving feedback during important meetings, bringing knowledge and expertise from a different research group and having a distinct view on the research.

Then, I would like to thank my daily co-supervisor, Kyriakos Psarakis, for always being available to answer the questions I have had during the thesis at short notice and the more frequent meetings discussing progress and short-term goals. This helped in structuring the research and gave motivation during the thesis.

Lastly, I would also like to thank the additional member of my thesis committee, Soham Chakraborty, for taking the time to evaluate my thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

2PC	**2** Phase **C**ommit
2PL	**2** Phase **L**ocking
ACT	**AC**tor **T**ransaction
AWS	**A**mazon **W**eb **S**ervices
CSP	**C**loud **S**ervice **P**rovider
DAAL	**D**istributed **A**tomic **A**ffinity **L**ogging
DSG	**D**irected **S**erialization **G**raph
FaaS	**F**unction **a**s **a** **S**ervice
LXC	**L**inux **C**ontainers
PaaS	**P**latform **a**s **a** **S**ervice
PACT	**P**re-declared **AC**tor **T**ransaction
S2PL	**S**trict **2** **P**hase **L**ocking
RDBMS	**R**elational **D**ata**B**ase **M**anagement **System**
SFaaS	**S**tateful **F**unction **a**s **a** **S**ervice
SSF	**S**erverless **S**tateful **F**unction

Chapter 1

Introduction

Over the past years, there has been a paradigm shift in the way we think about and program applications. The standard used to be creating a single entry point for the entire code base. It would run on a single machine, large segments of the code would depend on other segments and the memory would be shared across the entire application. These monolithic applications are often difficult to maintain, as changing any part of the code can have a large influence on the behaviour of any code depending on it (Dragoni et al., 2017).

The modern approach is characterized by separating the application in several microservices. Each of these is then responsible for taking on a specific task, and coordinating with other microservices to achieve the eventual goal. This removes code dependencies across the separate applications and decreases the size of the code base per service. The microservices communicate through published interfaces, decoupling the implementations. Because of this, changes made to any microservice do not have to be observed by others to continue functioning correctly. This allows for a gradual incremental development process. These microservices can also be scaled separately, where the component that is the bottleneck of the system can be replicated (Dragoni et al., 2017).

This new paradigm has many benefits, but requires a complex deployment environment to get started with. Managing the physical infrastructure, for example the hardware and networking, and the software, to enforce access rules and provide containers, can be challenging for smaller parties (Kamal et al., 2020). Large Cloud Service Providers (CSP), such as Amazon Web Services (AWS), Google Cloud Platform and Microsoft Azure, have made it accessible by taking care of these things, as well as offering scalability, reliability and availability (Setty et al., 2016). This makes them popular options for this serverless computing, allowing developers to focus on the application only and not the infrastructure behind it.

This serverless computing knows different forms, each with a different level of complexity abstracted away. With Platform-as-a-Service (PaaS), the developer has no influence on the underlying hardware or network, but gets an environment to run its code. The hosting environment can be configured and tailored with a choice of what services need to run on the side of the main application (Castro et al., 2019). If we take away more control we end up with Function-as-a-Service (FaaS), where the developer gets the runtime environment and is only required to write functions that can be deployed. The scaling is automatically managed to respond to any volume of incoming requests by instantiating the function in parallel. This however requires us to reason about state. Across multiple invocations, the state of the function is not persisted (Shahrad, Balkind, and Wentzlaff, 2019). The function can be instantiated on a different machine, and there is no evident way to coordinate the state with concurrent execution. They are also unaware of each others existence, making consensus algorithms impossible. There is need for a persistent storage.

The storage of persistent state has traditionally been done by a Relational Database Management System (RDBMS). Such a system provides the developer with ACID guarantees, making the application logic easier to reason about. As the application is scaling, so does the need to store and retrieve data. The scaling of these relational databases happens through increasing the size of the server, which can be disproportionately expensive. The alternative is presented by NoSQL systems, which provide distributed key-value data storage with high availability (Nance et al., 2013). These systems can be scaled horizontally, connecting additional machines to increase the traffic and data it can process (Pritchett, 2008, Minhas et al., 2012). This scaling is exactly in line with the scaling of FaaS, making it a good choice for data storage. But there are also two issues arising with this which we will discuss, the consistency burden on the developer and the latency.

The NoSQL data solutions designed by CSPs are focused on the availability of data (DeCandia et al., 2007). Instead of adhering to the ACID properties, they follow BASE (Fox et al., 1997), substituting consistent state and isolation for soft state and eventual consistency. The scope of an atomic transaction is also limited to a single record (Stonebraker, 2010). The replacement properties lay the burden of faultlessness on the developer correctly writing the code. While not every data operation requires consistency, there can still be some critical parts where it is mandatory. It also helps reasoning about problems as it simplifies concurrency constraints.

The retrieving and storing of state after every function invocation is still a costly operation in terms of time, making it unsuitable for low-latency applications. Function invocations often mutate only part of the data, indicated by a specific supplied key. Approaches have been taken to increase the locality of function and the stored data (Jia and Witchel, 2021, Sreekanti et al., 2020), or persist the function with retrieved data in preparation for another invocation (Bykov et al., 2011). This has lead to Stateful FaaS (SFaaS), where each addressed invocation is now also responsible for part of the state. The persistence of this state is handled by the underlying framework.

SFaaS offers the similar parallelizability and extensive programming control as FaaS, but with the extra benefit of low-latency access to its assigned state. Each component is able to do complex computations on its own data, but has limited direct access by design to the full state now. By compositing different stateful functions this limit is removed, but the problem of concurrency is introduced. The option opens here for distributed transactions that help simplify the reasoning about consistency, and have the potential to benefit from the scaling property of SFaaS itself. There are several (experimental) transaction protocols built on top of stateful functions, promising strong consistency guarantees.

The papers describing all of these SFaaS transaction protocols present metrics displaying throughput or latency, attempting to be the fastest solution yet under the given circumstances. What is not included is any experiments on the consistency. It is often assumed that it is consistent by the design of the system. Testing methods for (distributed) databases have existed for a long time (Chays et al., 2004), and testing methods have found many products claiming more than they deliver (Zheng et al., 2014, Kingsbury and Alvaro, 2020).

1.1 Research questions

This thesis formulates the following questions for its research:

1. What is the current state of testing SFaaS systems for consistency?

2. Do existing SFaaS systems adhere to the promised consistency level?
3. What should a standardized consistency testing format for SFaaS look like?

1.2 Contributions

We set up three existing SFaaS systems, one proprietary and two academic, and apply a testing framework for distributed systems evaluating the transactional consistency. We take the promises made by the theoretical description of the systems, and compare them to the practical implementations.

We show the inadequacy in the prevailing testing methods used by authors on their own work, and identify the weak points. We address the necessity of verifying consistency as support for other claims, and advocate awareness of the implications of dependencies.

We reason about a standardized testing format. We take insight from existing benchmarks, related work and our own experience in setting up the experiments. From this we offer guidelines for a standardized testing format, with more general rules as rules for the specific case of SFaaS.

1.3 Outline

Starting with chapter 2, we summarize the prerequisite knowledge for the following chapters. After this, we give an overview of publications in the same field with similar research in chapter 3. In chapter 4, the testing framework used and the method for verifying consistency are described. The next section, chapter 5, analyses three different SFaaS systems, describes their implementations and applies the testing framework. The findings of the experiments and the implementation process are discussed in chapter 6. Finally, in chapter 7 we conclude the thesis and reflect on the research questions.

Chapter 2

Preliminary

2.1 Database transaction models

2.1.1 ACID model

When database systems underwent a change from having a single user working on the data in batches, to a multi-user interactive system, problems of concurrency required solutions. Most RDBMSs now offer transactional guarantees based on ACID properties (Haerder and Reuter, 1983). These guarantees help the developer reason about concurrency and make writing code easier. Note that the consistency property does not relate to consistency as discussed in this thesis. Rather, the term isolation in database research is more identical to what consistency is in distributed systems research. These properties help reason about transactions and are described as:

- **Atomicity:** A transaction can consist of multiple operations on the database. Either all operations should occur or none at all. In the event of a crash or other failures, we do not want only part of the transaction to go through.
- **Consistency:** The database must always be in a consistent state. If the transaction would violate any constraints placed on the database, it cannot commit.
- **Isolation:** Multiple transactions can be executed concurrently. These transactions should not be affected by intermediate state of other transactions. It should appear that the transactions have run sequentially, without any interleaving.
- **Durability:** Whenever a transaction has committed, it should be persistent. Even when malfunctions occur afterwards, the transaction's alterations should remain.

2.1.2 CAP theorem

While the ACID properties are incredibly useful, there are some problems when we try to scale the database by adding multiple servers. These are given in the CAP theorem (Gilbert and Lynch, 2012). The theorem states that only two out of three can be attainable. The consistency term here is in the context of distributed systems, and is how the term is used in this thesis. The partition tolerance term can be seen as necessary, as we want to scale our database over multiple servers. The choice will then be between consistency and availability.

- **Consistency:** A request will always receive the correct response. No matter what server the request arrives at, the response should be the same, namely the consistent state.

- **Availability:** Every request will eventually get a response. There is always a server that will respond to an incoming message.
- **Partition tolerance:** The database is separated between multiple systems. The communication between these systems can be unreliable. Messages might be delayed or entirely lost.

2.1.3 BASE model

With the aforementioned scalability in mind, BASE arose as a new model. This model does not give strict guarantees such as with ACID. Work has been done in developing weaker consistency levels without giving up much on availability.

- **Basically Available:** Instead of focusing on consistency, the system will spread out data over multiple locations to promote availability. There is most likely always one server that can reply to a request.
- **Soft state:** The requested read of a state might not be reflecting the most recent updates. The state can, but is not required to be consistent. The developer will have to write its code taking this into account.
- **Eventually consistent:** At a certain time in the future, the database will be consistent. Whenever there are no write operations, all components of the database will eventually house consistent data.

2.2 Isolation levels

Database isolation levels have been defined by categorized phenomena (Berenson et al., 1995). These levels help classify and compare database systems. Certain levels will allow or disallow some of phenomena. What these are can be seen in table 2.1. We will see later that the isolation levels we will be using are generalized and therefore slightly different, but the terminology from these definitions is often used. The serializable level indicates that the transactions have occurred in some total ordering. None of the transactions have interleaved their operations. The phenomena determining the isolation level are described as follows:

- **Dirty write:** A transaction writes a value to a specific key, then before committing or aborting, another transaction overwrites that value.
- **Dirty read:** A value written by a transaction is read by another transaction before the write has committed or aborted.
- **Fuzzy read:** Also named non-repeatable read. A read value is overwritten before the read has committed. Subsequent reads of the key show a different value than before.
- **Phantom:** This phenomenon occurs when using predicates does not apply to this thesis.

Isolation Level	Dirty Write	Dirty Read	Fuzzy Read	Phantom
Read Uncommitted	Not Allowed	Allowed	Allowed	Allowed
Read Committed	Not Allowed	Not Allowed	Allowed	Allowed
Repeatable Read	Not Allowed	Not Allowed	Not Allowed	Allowed
Serializable	Not Allowed	Not Allowed	Not Allowed	Not Allowed

TABLE 2.1: Isolation levels defined by their allowed and disallowed phenomena

2.3 Transaction processing techniques

2.3.1 2 Phase Locking

To ensure that a transaction is operating isolation of other transactions, we can make use of 2 Phase Locking (2PL). During the lifetime of a transaction, it is able to acquire locks on specific entries in the database. Whenever it wants to operate on an entry, it requests a lock. The lock is acquired whenever no other transaction currently has the lock. Then the transaction can read and alter the value of the entry. Whenever the transaction is done with the entry, it can release the lock, allowing other transactions to acquire it again.

With 2PL, there are two phases. In the first phase, the expanding phase, the transaction can only acquire locks. Eventually the transaction will acquire all locks of the entries it either reads or modifies. This ensures that no other transaction can interfere. After this, the second phase commences. In the shrinking phase, locks can only be released. The transaction can obtain no more locks.

A distinction can be made between two types of locks. There are read-locks and write-locks. Whenever a transaction acquires a read-lock on an entry, it is only allowed to read the value and not modify it. In this case, whenever another transaction attempts to acquire a read-lock, it can be given as well without causing disruption. This allows transactions reading values to have less conflict. Only when all read-locks are released can a write-lock be obtained for that entry again. Whenever a write-lock is acquired, no other transaction can obtain a lock on the same entry. This holds for read-locks as well as write-locks.

Strict 2 Phase Locking

A variation on this is Strict 2 Phase Locking (S2PL). With this method, the write-locks acquired can only be released after the transaction has either committed or aborted. This prevents any value from being read or overwritten before it is certain the value is correct. This can prevent some phenomena such as a dirty read.

2.3.2 Optimistic Concurrency Control

An alternative to using locking mechanisms is Optimistic Concurrency Control (OCC). Transactions can begin to access a key by making a timestamp at the moment they start. The associated value can then be read and modified. Whenever it is done accessing the value, it verifies that no other transactions have modified it. If it is unchanged during its use, the change made can be committed. Otherwise the value is rolled back. This alternative to using locks is useful whenever there is little contention on keys, as it removes costly locking operations.

Deadlock

Whenever we have multiple transactions contending for the same locks, we can get into a deadlock state. What this means is that a transaction is not able to continue because it is waiting for a lock held by another transaction. This other transaction in turn is waiting for a lock held by the initial transaction. Neither transaction is able to continue. In this example, there are only two transactions. However, this can occur with any number of transactions operating concurrently greater than one. To avoid a permanent blocking state, there are algorithms in use.

The first algorithm is wait-die. Whenever a transaction is started, it obtains a timestamp. Upon attempting to obtain a lock, if the lock is already being held by another transaction, it will compare the timestamps. Whenever the transaction requesting the lock has an older timestamp than that of the transaction holding the lock, it is allowed to wait. If this is not the case, the transaction is aborted. It is not possible for two transactions to wait for each other, since one has a younger timestamp and will abort.

The second algorithm is very similar. The transactions are obtaining a timestamp again. When a lock is requested and it is determined that it is held by another transaction, another comparison is made. This time, if the requesting transaction has a younger timestamp, it is allowed to wait. If its timestamp is older, it will abort the other transaction. This method is not always possible with FaaS, since function instances can not always communicate with each other.

2.3.3 2 Phase Commit

A transaction must either commit or abort. What this means is that either all or none of its operations go through. This can be done with 2 Phase Commit (2PC). A coordinator can start the protocol, and will send to all participants of the transaction a message telling them to prepare the actions for a commit or an abort. The participants will then send back a confirmation that everything is ready for a commit, or send back a message indicating an issue. This is the first phase of the protocol. Whenever the coordinator has received all replies, the second phase starts. If all incoming messages were positive, it proceeds to tell every participant that they can commit the final value. If one or more participants indicate that they want to abort, the coordinator will send all participants a message telling them to abort. This way, all participants either commit or abort.

2.4 Adya histories

2.4.1 History

Much of the related work, as well as the testing framework used in this thesis, extend an existing history model (Adya, Liskov, and O'Neil, 2000). This history model is used to represent the interactions of transactions on the state of the stored data. Based on this history, we can precisely follow the states the database was in and find any inconsistent behaviour. A brief explanation is given here that should be sufficient for understanding this thesis. Some things have been omitted that are deemed irrelevant for this research such as the use of predicates, as to the best of our knowledge there is no study yet about this in combination with SFaaS.

A history consists of two parts. The first is the partial ordering of events from the transactions. This shows what operations have taken place from what transaction. In the upcoming example it can be seen on the left hand side.

The second part is the total version ordering of every object stored in the database. Only the committed versions are in this ordering. If version x_1 was installed before x_2, we denote this with $x_1 \ll x_2$.

If we were to combine these two, we could end up with the example history from the paper, looking as follows:

$$w_1(x_1)w_2(x_2)w_2(y_2)c_1c_2r_3(x_1)w_3(x_3)w_4(y_4)a_4 \qquad [x_2 \ll x_1]$$

We have transaction T_1 writing version x_1, and then committing. T_2 has written version x_2 and y_2 before committing. By looking at the version ordering we can see that T_2 must have happened before T_1. T_3 then reads version x_1 and writes x_3, indicating that it happened after T_1, and is uncommitted as of yet. T_4 aborts.

2.4.2 Dependencies

When we have a history, we can start modeling the dependencies between transactions. When a transaction depends on another, it means that there exists a specific ordering in which the events of the transactions occurred. The reading of a value cannot happen before it has been written. The one event has to have happened before the other. There are three types of dependencies defined in the paper. Two of these also have a predicate based definition, which we will not discuss, the third has only a general dependency description. The requisite types are defined as follows:

- **Directly item-read-depends**: We say that T_j directly item-read-depends on T_i if T_i installs some object version x_i and T_j reads x_i.
- **Directly item-anti-depends**: We say that T_j directly item-anti-depends on transaction T_i if T_i reads some object version x_k and T_j installs x's next version (after x_k) in the version order. Note that the transaction that wrote the later version directly item-anti-depends on the transaction that read the earlier version.
- **Directly Write-Depends**: A transaction T_j directly write-depends on T_i if T_i installs a version x_i and T_j installs x's next version (after x_i) in the version order.

2.4.3 Directed Serialization Graph

Based on the definitions of the dependencies, we can create a Directed Serialization Graph (DSG). Every transaction becomes a node in this graph, and three types of directed edges are added based on the dependencies. Only committed transactions are added to the DSG, so the graph does not contain all information a history does and is not a replacement. A directed edge signifies that the transaction is dependent on the outcome of the other transaction. If we were to turn the example history into such a DSG, it would result in figure 2.1.

2.4.4 Generalized isolation levels

Finally, anomalies can be detected by finding cycles and other inconsistencies in the DSG and the history. The absence of these anomalies define what isolation level can

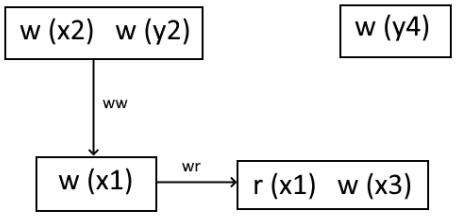

FIGURE 2.1: Example of a DSG using the earlier mentioned history.

be ascribed to the system. These definitions are generalized so that they are also applicable to optimistic and multi-version concurrency control. The phenomena are defined in the paper as follows, slightly altered to disregard predicates:

- **G0: Write Cycles.** A history H exhibits phenomenon G0 if DSG(H) contains a directed cycle consisting entirely of write-dependency edges.
- **G1a: Aborted Reads.** A history H shows phenomenon G1a if it contains an aborted transaction T_1 and a committed transaction T_2 such that T_2 has read some object modified by T_1.
- **G1b: Intermediate Reads.** A history H shows phenomenon G1b if it contains a committed transaction T_2 that has read a version of object x written by transaction T_1 that was not T_1's final modification of x.
- **G1c: Circular Information Flow.** A history H exhibits phenomenon G1c if DSG(H) contains a directed cycle consisting entirely of dependency edges.
- **G2: Anti-dependency Cycles.** A history H exhibits phenomenon G2 if DSG(H) contains a directed cycle with one or more anti-dependency edges

There is a certain similarity with the phenomena described in chapter 2.2. Adya's paper talks about portable levels, since they also carry over to optimistic and multi-version concurrency control. The definitions from the earlier mentioned isolation levels will be used during the tests.

2.5 Actor model

The actor model is a model for concurrent computation where all the computing is done on the primitive type of actor (Hewitt, Bishop, and Steiger, 1973). These actors are self-contained units of computation, and can only communicate with other actors through asynchronous messages. Aside from receiving and sending messages, an actor does not interact with the other actors. Multiple actors can be running concurrently, and actors can send and receive multiple messages. An actor can only be processing a single message simultaneously, any received message during execution is handled later (Agha et al., 1997). The order the messages are received in is non deterministic. Information received or computed can also be stored by the actor, influencing subsequent decisions (Hewitt, 2010). Not a single actor has an overview of the entire system, rather, the collective of agents can function as one.

The actor model has gained interest with the usage of multiple cores, and now again with cloud computing. Since the model is not based on implementation details

but rather gives a general description, it can be useful on any system using concurrency. With multiple cores, each core can run a single actor concurrently. Whenever the computation of an actor is done, it can make way for another actor to process a message on that core. Accessing state is done through sending messages instead of directly accessing it in memory. This way, multiple cores can be utilized with much easier reasoning of concurrency and scalability.

The characteristics of the actor model reflect what cloud computing has to offer, SFaaS in particular. The primary computing unit in the actor model can be translated to the stateful function. How this function retrieves its state is not decided by the actor model, so multiple implementations are feasible. Messages are analogous to FaaS function calls, where no memory is shared but a payload is formed and sent. The scalability of the cloud makes it possible to have many actors operating concurrently almost trivial.

The actor model is used in Microsoft Orleans, one of the tested frameworks in chapter 5.

2.6 SFaaS

To understand fully what SFaaS is about, we start with its foundation, FaaS.

FaaS allows developers to only write code and not worry about the underlying infrastructure. The developer chooses a CSP platform where the function will be served. Whenever a function is called, the request is automatically propagated to an available server. What type of server this is and what software is installed there is to no concern of the developer, the correct infrastructure is provided by the CSP. The cost the developer pays is directly related to the time the function takes to execute. Whenever there are no requests, no costs are made.

When more function calls arrive in temporal proximity, the platform automatically scales the resources allocated. Additional calls can be executed on a different machine in parallel. This is perfect for processing large amounts of data that can be separated in small chunks, each requiring the same independent data modification. Because it is uncertain what to do about state in this situation, it is left up to the developer to make its own data storage calls.

With SFaaS, the state is treated as a first-class citizen. The developer has access and can alter the state without defining its storing and retrieval functionality. Often, attempts are made to make accessing the state faster, since the key-value stores available are usually slow. The implementation of this state can be different in SFaaS frameworks.

In Orleans, one of the systems we will test in chapter 5.1, the actor that is called will persist on the server, and subsequent function calls will arrive at the actor. This way, the state remains locally available, making it very fast. Only when an actor is not accessed for a longer duration is the state written to persistent storage.

In another framework, cloudburst, the state and function execution is disaggregated logically and co-located physically (Sreekanti et al., 2020). This simplifies the development process and provides fast access to the state.

What becomes immediately clear is that this state access needs some rules to allow for consistency. If a SFaaS implementation allows two function calls to operate in parallel, the concurrent accesses should be regulated to prevent certain phenomena. Either the developer should accept this shortcoming, or should look at transactional guarantees that might be offered. Since there are many approaches SFaaS

frameworks can take in state storage, developers should look into the consistency options provided per framework.

Chapter 3

Related Work

The importance of prudence when working with transactions is demonstrated by an attack against concurrent database-backed systems formalized by the name of ACIDRain (Warszawski and Bailis, 2017). In this attack, adversaries will attempt to exploit concurrency anomalies by interacting with the exposed API. There are two types defined of anomalies that can occur.

Level-based isolation anomalies exist when the database has not been set to a serializable isolation level. The default isolation for many databases is often set to a lower level to reduce bottlenecks created by concurrency control (Fekete et al., 2005). The serializability is too often assumed and the isolation settings are left unchanged.

Scope-based isolation anomalies are the application developer's responsibility. They can occur when critical code is not encapsulated by explicit transaction demarcations. Standard database behaviour often executes each operation as an independent transaction, providing no transactional guarantees across the operations.

To find potential vulnerabilities, the database logs are analyzed in an approach called Abstract Anomaly Detection. Each API call is mapped to the transactions that occur because of it, and within each transaction the operations are noted. If two operations are reading and/or writing the same value, a conflicting relation is added between them. Possible anomalies are then found by detecting cycles. This is similar to the approach taken by Adya et al. in chapter 2.4.

Twelve open source eCommerce applications were analysed, and exploits were attempted for the found potential vulnerabilities. Three examples are decrementing the inventory below zero, redeeming vouchers multiple times and adding products to the cart post checkout. With added network delay, these were all detected. This shows the importance of being aware of isolation levels.

Another tool for verifying serializability is the system Cobra. (Tan et al., 2020) It works in a similar fashion to Elle, where a history is inferred by observing transactions made by clients. Cobra works simultaneously with regular use of the database. A different definition is taken for what it means to be a "black-box" verifier; Cobra does not consider the dependency on using appends and requiring a custom workload to be within this category.

The system has a history collector sit in between the client and the database, and record the transactions taking place. The client appends a unique value to each of its writes, and consumes it on a read operation. This enriches the history as for every read operation we can recover which exact write was responsible. At intervals the history collector sends the data over to the verifier, who will take the fragment, turn it into a graph and merge it with the previously verified history.

To actually perform the verification, a SMT solver is used. The encoding of the graph is refined by several techniques, as the unrefined graph would overwhelm the solver. The refinements could all have been omitted if the object versions were

traceable, which would have been the case with appendable lists.

One consistency checker that attempts to do something similar to Elle is Coo (Li, Chen, and Li, 2022). The approach to creating a history has many resemblances, including treating the database system as a black-box. Where Elle only uses three types of directed edges to indicate the dependency type, (r, w, anti) Coo constructs a Partial Order Pair (POP) graph. Both will then detect cycles in this graph, and if one is found this means an anomaly is found.

A survey has been done on the existing anomalies in the literature where 19 have been found, which have been mapped to their own anomaly definitions. A total of 33 anomalies are described by Coo, each with the associated POP combinations. Some existing ones have been merged, such as "Dirty read" and "Aborted read" now falling under the same anomaly "Dirty reads". By trying to fit the anomalies to the POP cycles, some information about the specific anomaly is removed, and other anomalies are not possible to describe in terms of them[1].

The main difference, and Coo's critique on Elle, comes to what test cases are generated. Where Elle will generate random workloads according to settings (parameters), Coo will have many handcrafted cases where each has its own anomaly to detect. In the paper it is stated that Elle is wasting time by doing these random tests.

Both of the systems claim to be sound, and close to complete. Jepsen has reported that an inconsistency has been found in version 12.3 of PostgreSQL on the Serializable Isolation level: "Together, we confirmed that this bug was present in PostgreSQL 9.5.22, 10.13, 11.8, 12.3, and 13; we assume it is present in every extant version." The inconsistency that has been found is named a "write skew" by Coo. What is interesting is that for version 12.4, Coo has reported that PostgreSQL will do a rollback, and fails to detect the full anomaly here. Upon inspecting the release notes of it is mentioned that this error has been fixed at version 12.4 [2].

[1] https://jepsen.io/analyses/mongodb-4.2.6

[2] https://www.postgresql.org/docs/release/12.4/

Chapter 4

Testing framework

To verify the consistency level of different SFaaS systems, we first need to establish a testing setup. This chapter will discuss how we will be making use of Elle and Jepsen. Elle is intended to verify consistency of databases based on observations, Jepsen will help us set up distributed systems ready to be tested. Although this setup is not intended to test SFaaS systems, we can fabricate our distributed frameworks so that the tests fit seamlessly.

4.1 Elle

4.1.1 Consistency checker

Elle is a black-box consistency checker based on a history model defined by Adya et al. (Adya, Liskov, and O'Neil, 2000). An overview of this model can be found in section 2.4. This history is a recording of every state the objects in the database have been in, as well as what the transactions are that have taken place. With this information, we can deduce the order in which the transactions have occurred by looking at what version of the object was installed by what transaction. If a transaction has influence on another transaction, either by modifying or reading a value the other transaction has altered or seen, we speak of a dependency. There are three important dependencies formalized that are used by Elle. These can be seen as slightly simplified versions from those by Adya et al. that are applicable to our research:

- **Directly write-depends:** T_i installs x_i, and T_j installs x's next version.
- **Directly read-depends:** T_i installs x_i, T_j reads x_i.
- **Directly anti-depends:** T_i reads x_i, and T_j installs x's next version.

We can infer these dependencies from the aforementioned history. The transactions and the inferred dependencies are then constructed into a DSG. In this graph, the transactions are represented by vertices and the dependencies by directed edges. Each edge is directed from the dependent transaction toward the transaction it depends on. These edges are named slightly different to their respective dependency formulation, but indicate the cause of the dependency well. The names given are *write-write dependency*, *write-read dependency* and *read-write dependency*, ordered as the formulations listed above.

Additionally, we can also look at the concurrency of the processes submitting the transactions. If a process receives confirmation that a transaction has committed, its next transaction will always be later in terms of real-time. Some consistency levels allow the transactions to be scheduled in a different order, whereas others require them to be in accordance with real-time. Another consistency level even makes

a difference in only applying this rule for the same process. These dependencies are registered as conditional dependencies and are only used in the analysis when checking for specific consistency levels.

We can then start to check the DSG for the presence of a cycle. If a cycle is found, it would imply that all the transactions that are part of the cycle rely on the other transactions in the cycle happening first. Transitively this would even imply that a transaction occurred before itself. None of them were the first to execute, yet still they all finished. A distinction is made between different types of cycles, based on what type of dependencies it contains. The cycles Elle distinguishes are again adaptations of definitions by Adya et al. and as follows:

- **G0:** A cycle comprised entirely of write-write edges.
- **G1c:** A cycle comprised of write-write or write-read edges.
- **G-single:** A cycle with exactly one read-write edge.
- **G2:** A cycle with one or more read-write edges.

Aside from the anomalies detected by looking for cycles, Elle will also check acyclic anomalies defined by Adya et al., as well as ones found by applying Elle to real databases. These last anomalies are defined as:

- **Garbage reads:** A read observes a value which was never written.
- **Duplicate writes:** The trace of a committed read version contains a write of the same argument multiple times.
- **Internal inconsistency:** A transaction reads some value of an object which is incompatible with its own prior reads and writes.

Whenever the checker has found what anomalies have taken place in the history, it can deduce what consistency levels the history allows for. There remains a rather large issue though, we do not have the history.

4.1.2 History acquisition

The actual history necessary for this analysis is not known to the client. The client can only infer the history based on the observations made. It knows what transactions have been sent to the database, and what the response was. This response can be the results of read operations, as well as information about whether the transaction committed or aborted. Even an ambiguous response, such as a timeout, lets the client know that it is uncertain whether the transaction committed or aborted.

Due to the black-box nature of Elle, we have limited information to construct the history. One thing we can improve is to make every value written to a key in the database unique. If there are no two transactions that write the same value, we can recover for every read value what the transaction was that wrote it. This is the **recoverability** property and helps detecting read dependencies. Whenever a read operation takes place, it is known which write operation of a transaction happened before it.

Values being written to a stored object discard any information on previously stored values. By turning the object into a list that we append to, any previously appended values are still present in the version stored last. This introduces **traceability**

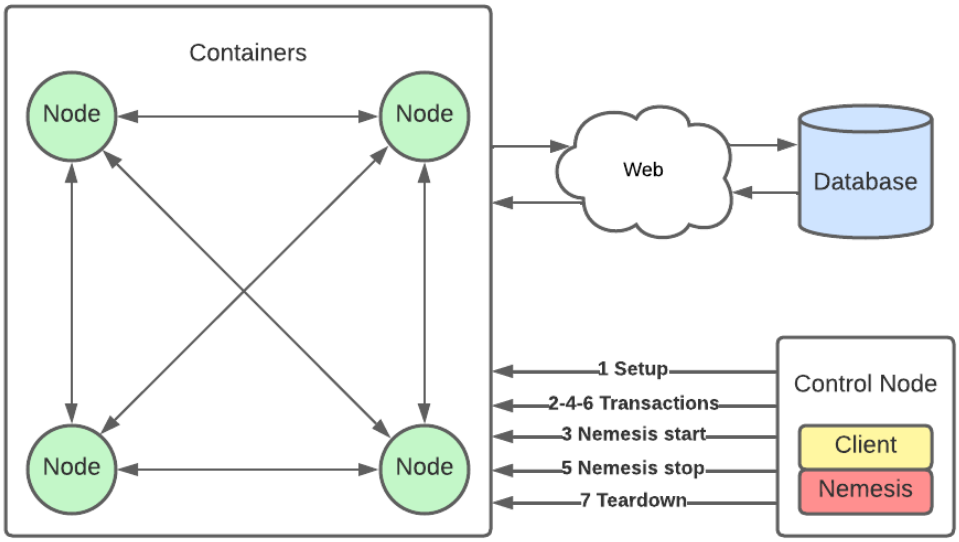

FIGURE 4.1: A general overview of the Jepsen setup.

to the object, where we can see the exact order in which the write operations took place. This results in being able to chart write dependencies, for any write it is clear what the previous write was, as well as anti dependencies, where a write happens after a list has been read.

From the pieces of information received, an attempt is made to reconstruct the possible histories that could have taken place. Whatever information is known by direct observations can be pieced together. Since not all information is available, Elle leaves multiple interpretations possible of the uncertainties. Further observed transactions can then fill in the gaps, and whenever one possibility remains for a previously uncertain transaction it can collapse to its only possible state. If eventually an anomaly is found, it is certain that in every history that is possible it is also present. The anomaly is a subgraph of all DSGs of the possible histories. Whenever all the information is included in the history, we can start to check for anomalies.

4.2 Jepsen

To run the tests on the distributed systems, the Jepsen framework is used. This framework automates many steps that are necessary in running tests against these systems. Each of the automation steps is described below. Jepsen also provides the option to plug in Elle to generate the required transactions and verify the consistency of the observations. A visual overview of the setup is shown in figure 4.1, and can be consulted alongside the description.

4.2.1 Nodes

Since we are testing a distributed system, we must setup multiple connected nodes. Jepsen offers some guiding in setting this up to work with its framework. One of the options provided is using the preset cluster on AWS Marketplace, but this does

not provide the full control running it locally does. A containerized setting is thus preferred, and the recommended option here is using Linux Containers (LXC) [1].

LXC lets multiple containers run using the same kernel [2]. These containers live independently of the Jepsen control node, where the test is orchestrated. At any point we can SSH into the containers and any necessary software can be installed on these systems. The containers' network can be fully managed, creating and removing connections between nodes. After tests, we can also look at the logs stored locally on the nodes.

4.2.2 Setup/Teardown

Whenever a test is started, Jepsen makes a connection to the nodes taking part in the test. A list should be supplied containing the participating nodes. Jepsen can take control of the systems and execute any commands on the operating system, such as downloading and running files. These privileges are then used to setup the database system on each node. These instructions as well as the specifications on how to tear down the database after the test have to be implemented by the user. This gives full control and allows for any system to be tested.

In our case, the nodes will first install the required packages. If the node has installed them before, this will only take a brief moment, while new nodes are ensured to have the correct packages at their disposal. Next, the application we want to test is downloaded as an archive and installed. With this method, it has to be available through a public download link. After the program is started, there is a delay before running the test to allow the client to start. At the end of the test, logs that were made locally on any of the nodes can be automatically copied to the test system for further inspection. The teardown phase cleans everything up.

4.2.3 Client

Once the nodes are setup, they need to receive input to actually perform the operations we want to test. What these operations are, will be determined by a generator. Jepsen provides some generators, but custom ones can be made to tailor to specific needs as well. Since we are making use of Elle to check the final history, it is expected to use the accompanying generator as well. This generator creates transactions, consisting of a list of reads and appends. We can supply parameters indicating how many keys we want active at the same time, the minimum and maximum transaction length, and how many writes we allow per key before turning it inactive. During the test, the transactions are generated on demand. These transactions are then forwarded to the client, where a custom implementation decides how to propagate this information to the nodes.

An example of a transaction that the generator can create looks as follows. This will read the value at key 6, append value 8 to key 2, and append value 1 to key 7.

```
[[:r 6 nil] [:append 2 8] [:append 7 1]]
```

The result could look like this. The read values are returned, and the append operations have no return value.

```
[[:r 6 [2 3 4]] [:append 2 8] [:append 7 1]]
```

[1] https://github.com/jepsen-io/jepsen
[2] https://linuxcontainers.org/lxc/introduction/

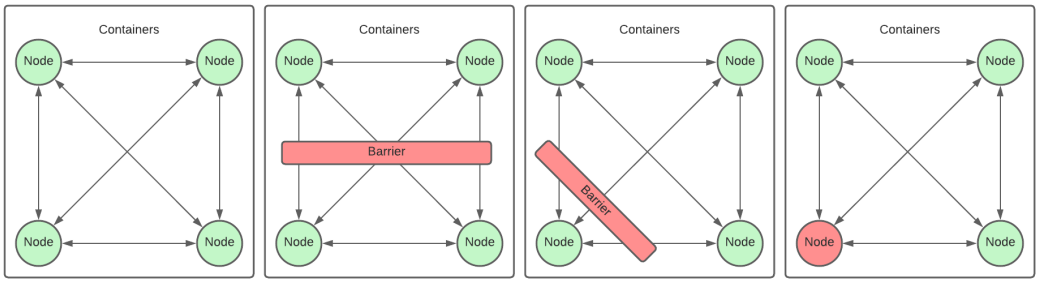

FIGURE 4.2: A depiction of the different nemesis actions. From left to right: inactive, partitioning in two halves, partitioning a single node, stopping a node's process.

Every outgoing transaction is logged, as well as the corresponding returned values. In the case where an error occurs on the database side, and the transaction is aborted, this is also registered.

The Jepsen client needs to find a way to send its transactions to the tested system. The systems that we test are diverse and written in different languages, so finding a universal way of communicating is important. This is also relevant when looking at a general testing setup. All cloud platforms that support FaaS offer an API gateway for HTTP requests. Any other systems that should not be hosted on a FaaS platform, most likely have their own client written to interact with the system. This client is often only available on the programming language the system is written in. Since most programming languages do have a well functioning web server available, incoming HTTP traffic can easily be transformed to a client request.

4.2.4 Nemesis

To test the systems in the event of a failure, a nemesis is introduced. This special client process introduces faults into the cluster. The most used actions in Jepsen's own tests by the nemesis include partitioning the network into two random halves, partitioning a single node and pausing the running process on a node [3]. A visual overview of these actions is given in figure 4.2. Other nemesis actions are similar but require input to select which nodes the action is taken upon. Unless a specific suspected fault is targeted, the general actions are sufficient. In the event of a partitioning, Jepsen prevents any traffic between the containers of the nodes. The containers still have access to outside connections, even in the case of a single node being partitioned. These nemesis strategies are set to activate and deactivate based on a schedule created by the generator. The rules for this schedule can be customised similar to the generator. Nemesis actions are also logged for evaluating the cause of any inconsistencies.

[3] https://github.com/jepsen-io/jepsen

Chapter 5

SFaaS frameworks

The testing setup was applied to three different frameworks that will be discussed in this chapter. These are Microsoft Orleans, Snapper and Beldi. Although the frameworks have a different approach to handling state and computing units, they can all be categorized as SFaaS. Each section starts with an overview of the framework, delves deeper into some aspects that are deemed important for consistency in transactions, and is followed by the implementation details. The sections end with experiments and their outcome.

5.1 Microsoft Orleans

The first framework we will be looking at is Microsoft Orleans. This is a framework that can create distributed applications that allow for elastic scalability (Bykov et al., 2011). It is based on the actor model, which is described in chapter 2. In Orleans, actors are the combination of identity, state and behaviour, and communicate with each other through messages. The identity makes every actor addressable for communication. Through these messages we can call functions on these actors, which are defined in their behaviour. The outcome of the behaviour an actor can be influenced by its internal state.

One notable invention that Orleans has done, is that the actors in its system are virtual actors. These virtual actors are always in existence, and cannot be created or destroyed. Even though there might not be a physical location where the actor currently lives, it can still be accessed as if it did exist somewhere on one of the servers.

To get started we need to host a server. Multiple servers can be started at different times, and together they will automatically orchestrate into a cooperative application. Every server that is part of the application is called a silo, and all of them are connected in a so called cluster. An visual overview can be seen in figure 5.1. These silos can effortlessly be added or removed from the cluster during the deployment of the application to address the computing requirements. The framework will take care of the silo awareness in the cluster.

Each of these silos can host what are called grains, the virtual actors. The functionality of a grain consists of a class with an interface for other grains or external clients to call. What makes this different from stateless functions is that each grain is also associated with a unique key. Whenever a call is made to one of these grains, Orleans will try to find an activation with that specific key that is available in memory on one of the silos. The state of the grain is then persisted through multiple calls. If this is not found, it will activate the grain and place it on one of the suitable silos. The grain placement depends on the strategy chosen and can be customized. Defining the behaviour of grains is where the functionality of the application lies.

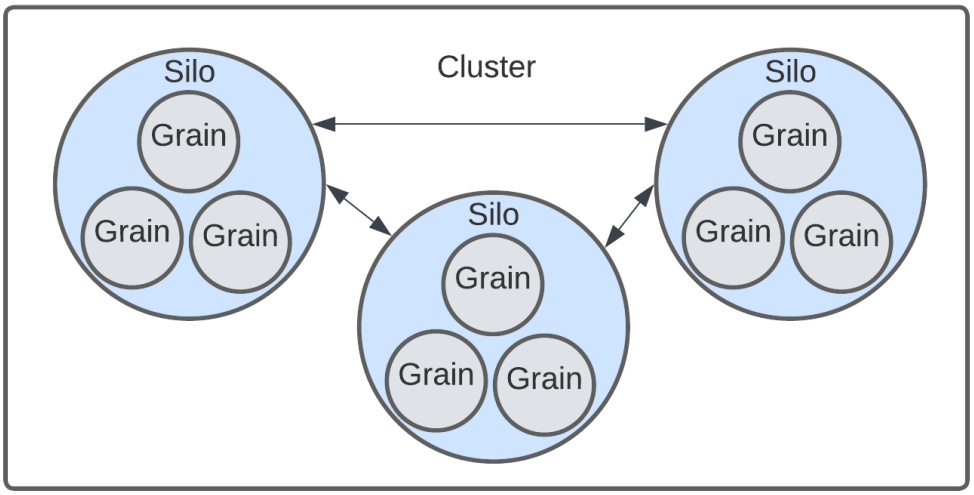

FIGURE 5.1: Relation between clusters, silos and grains.

If a newly activated grain requires its state, it can be retrieved from persistent storage. Examples of given options for this are Azure Storage and DynamoDB. What consistency level is required from any of the persistent storage implementations is not mentioned. Although this can potentially have a huge impact on the consistency, it is not yet relevant as in chapter 5.1.3 we will see a required alternative for transactions.

We will now look at some details of Orleans, which seem vulnerable to cause inconsistencies, either through proper use or through unintended behaviour. For this we will start with the cluster management and continue with the grain directory. After this we will look at the specific transaction implementation provided by Orleans, and what it changes from the general workings.

5.1.1 Cluster management

Whenever we want to connect a new server, we can simply start a silo at any machine. It should then connect to the cluster and start activating grains when called. To make this possible, the silo should first have a way of knowing the locations of the other silos to make first contact, and broadcast its own existence to the members of the cluster. For this there exists a persistent silo membership table[1]. This table stores for each silo how to connect to it by saving its IP address and port. The joining silo writes its own existence to the table and can read the information about the other connected silos. There are several implementations of the silo membership table available, with the default using Azure Table.

To see if silos are still available and alive, the cluster's members will send each other pings over the same port as the regular messages are sent. Every silo will only send messages to a small set of other silos. If a ping is successful, then other messages sent to that silo should not fail. In the event when a ping does not succeed, the sending silo will make a note of this in the silo membership table. When enough

[1] https://learn.microsoft.com/en-us/dotnet/orleans/implementation/cluster-management

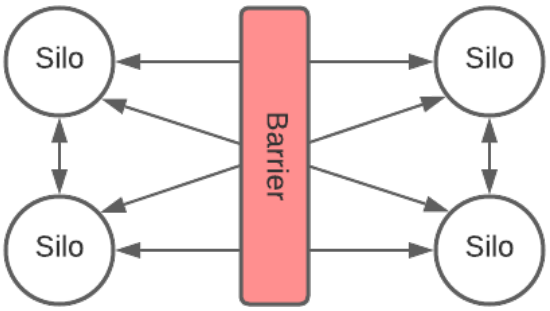

FIGURE 5.2: A cluster of four silos creating a race condition when the network is partitioned in half.

messages and pings have not been answered due to unavailability, the silo will be marked as dead.

Before diving into some examples, an overview of the most important settings can help clarify the protocol even more. These settings can be configured manually but are **defaulted** to:

- Every **10** seconds the silo sends out probe messages
- Whenever **3** messages are unanswered we suspect a dead silo and write it to the membership table
- After **2** silos have suspected a silo to be dead, it is marked as such
- Each silo will probe **3** other silos

Whenever a single silo is entirely stopped or partitioned from the network, it will not be able to communicate with the others and it will eventually be marked as dead. It does not matter if it still has a connection to the persistent storage of the membership table, as this connection is only used to initially publish a silo's existence and get information on other silos, not publish aliveness.

However, when we partition the network in two halves, we can immediately see situations arise where a network partition might interfere with the intended behaviour of the silo membership table. We are creating a race condition for which silos will be marked as dead. We can create an example, visualised in figure 5.2, by taking the default settings and creating a cluster consisting of four silos. Each silo will be probing all other silos for aliveness, and when two silos suspect a possible dead silo, they write it to the membership table. When we split the network into two equal halves, the resulting membership table is dependent on a race condition. Both halves will mark the disjointed silos as dead, but it depends on when the partition happens on which half will succeed.

Silos being marked as dead but still performing operations can cause inconsistencies. Other silos will think it does not hold any active grains, and can activate an already active grain on another silo. Depending on the exact implementation, there are many possibilities of anomalies here.

TransactionOption	Description
Create	A new transaction is started, and a transaction context is created. This context will be passed along to subsequent function calls.
Join	This function call receives a transaction context and becomes part of the transaction.
CreateOrJoin	If a transaction has been started earlier and a transaction context has been passed, join the transaction. Otherwise, start a new one.
Suppress	The call made to this function is not transactional, and the transaction context is not passed along. This function however can be called from within a transaction.
Supported	The transactional context is passed along to the function with this option, but the function itself is not part of the transaction.
NotAllowed	This function cannot be called from within a transaction.

TABLE 5.1: The options for transaction attributes and their corresponding working.

5.1.2 Grain directory

Calls made to a grain require the grain to be activated at one of the silos, or the call will trigger an activation. To get the call to the correct silo, or instantiate the grain on the current silo, it is necessary to have an overview of the active grains and their locations. The mapping between a grain activation and the silo it is residing at is maintained by the grain directory.

The default storage used for this is an in-memory distributed directory[2]. Every silo contains part of a distributed hash table. This directory is eventually consistent, which might pose a problem for transactions. When the cluster is unstable, duplicate activations can occur. There are other implementations of the grain directory and custom ones can be used as well, that might enforce something better than eventual consistency. This would also prevent the duplicate activations.

5.1.3 Transactions

Within this framework, there is an implementation of transactions with ACID guarantees. Upon instantiating the silos and the clients, they should be configured so that they will be using these transactions. When this is done, the callable grain functions need a transaction attribute attached, together with a transaction option defining the behaviour. These behaviours are stated in table 5.1. When the grain that initially started the transaction has finished, the transaction is automatically committed. Any exceptions thrown during the execution of the grain will cause the transaction to abort. The promised consistency level is serializable[3].

This is not the only thing activating transactions changes. By design, all the grains in Orleans have single-threaded execution. Whenever a function is called, all incoming calls are put on hold until the original function has returned. This is even the case when it is waiting on a call made to another grain or any other asynchronous

[2] https://learn.microsoft.com/en-us/dotnet/orleans/host/grain-directory
[3] https://learn.microsoft.com/en-us/dotnet/orleans/overview

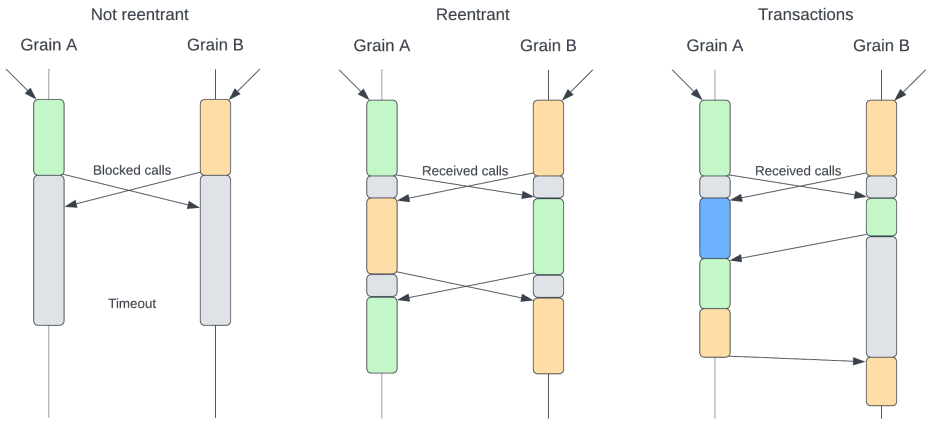

FIGURE 5.3: Three possible settings of reentrancy. Green and orange indicate the path of a request. Gray indicates inactivity. Blue indicates the transaction implementation of a grain.

calls. This makes reasoning about the state of a grain intuitive, as the written function executes as one block. This behaviour occurs when a setting called reentrancy is not allowed. If we were to allow reentrancy, whenever a grain is asynchronously waiting for something to complete, another message can be handled. This requires more complex reasoning about consistent state. With transactions enabled, we must set every grain to be reentrant. This part of functionality will now be handled by the transaction aspect of the grain. Calls made to a grain have their transaction context passed along, but the grain decides when it executes what call. Examples of these settings can be seen in figure 5.3. A deadlock can still occur if two transactions use the same keys.

When using transactions, we also need to configure the data store to use transactional storage[4]. Currently, the only supported implementation is that of Azure Table Storage. What exactly is different about this is not documented, and instructions to create a custom alternative do not exist. The transactional implementation differs from the regular grain storage since it has to be more strict[5][6]. It also adds fields to the stored data, containing a transaction id, a transaction timestamp and the transaction manager. This can be seen when observing the database at the Azure Portal.

5.1.4 Implementation

When looking at the implementation of Orleans and its connecting points, we can track the chronological flow of information. As mentioned in chapter 4.2.3, the Jepsen client will send its transactions through HTTP requests. This will be the starting point of where the Orleans implementation begins. Accompanying the description of the system is a visual representation that can be found in figure 5.4. Orleans has its own client written in C# for communicating with the cluster, now the client

[4] https://learn.microsoft.com/en-us/dotnet/orleans/grains/transactions

[5] https://github.com/dotnet/orleans/blob/main/src/Azure/Orleans.Transactions.AzureStorage/TransactionalState/AzureTableTransactionalStateStorage.cs

[6] https://github.com/dotnet/orleans/blob/main/src/Azure/Orleans.Persistence.AzureStorage/Providers/Storage/AzureTableStorage.cs

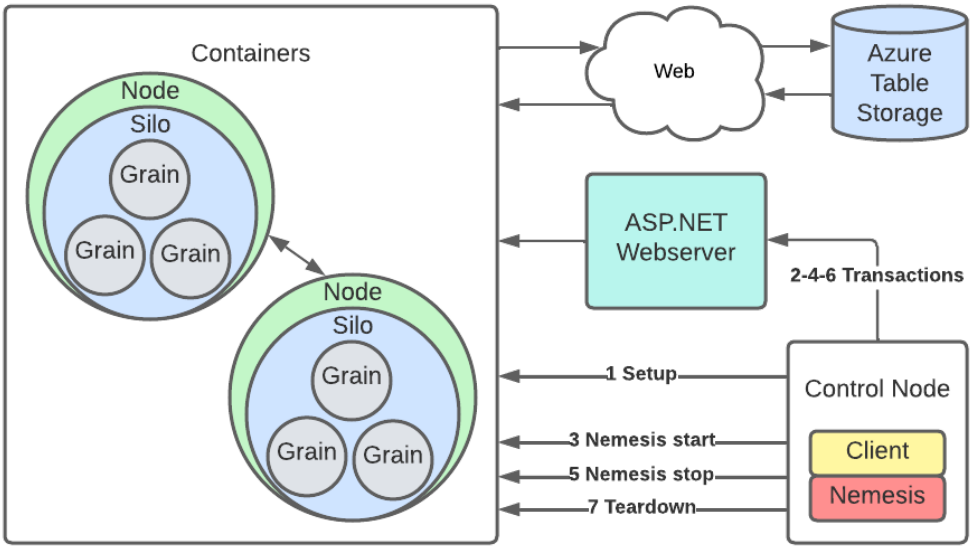

FIGURE 5.4: Orleans overview

only needs to receive the transactions from the Jepsen client. Since we are working with the .NET platform, the web server of choice will be ASP.NET Core[7]. This .NET framework can be used to create web applications that can run on multiple platforms.

The task that the web server needs to complete is kept at a bare minimum, to ensure that most of the work is done by Orleans. When the server is started, it attempts to start a client and connect to the cluster. When it is connected, it can start making calls to grains in the silos. Next, it will listen to HTTP requests coming in. Upon receiving such a request, the transaction string attached as JSON is passed along when making the first grain call to a grain. This grain is the TransactionGrain.

The TransactionGrain is the entry point within the cluster for any transaction. Its task is to parse the message from a string to a list of operations, and propagate those. This grain is marked as a stateless worker, giving it other properties than regular grains. As the name suggests, activations of the grain are stateless and thus have no identifier. This allows for any number of activations to exist simultaneously, either on different silos or the same. On this grain the transaction is also started with the *Create* transaction option, and will eventually be committed or aborted. By parsing the input here instead of on the web server, we avoid a bottleneck as this function now scales with the size of the cluster.

After the parsing has completed, the TransactionGrain can start making grain calls for every operation in the transaction. This is done to the DataGrains, which are responsible for the data associated with their identifier. These are asynchronous calls, but their results are awaited before issuing another grain call. This might slow down transactions, since some of the individual operations could execute at the same time, but the order would not be guaranteed if two operations engaged with the same key. This is a requirement for consistency. In chapters 5.2 and 5.3 an issue arises where an alternative approach coincidentally solves this issue as well.

[7] https://github.com/dotnet/aspnetcore

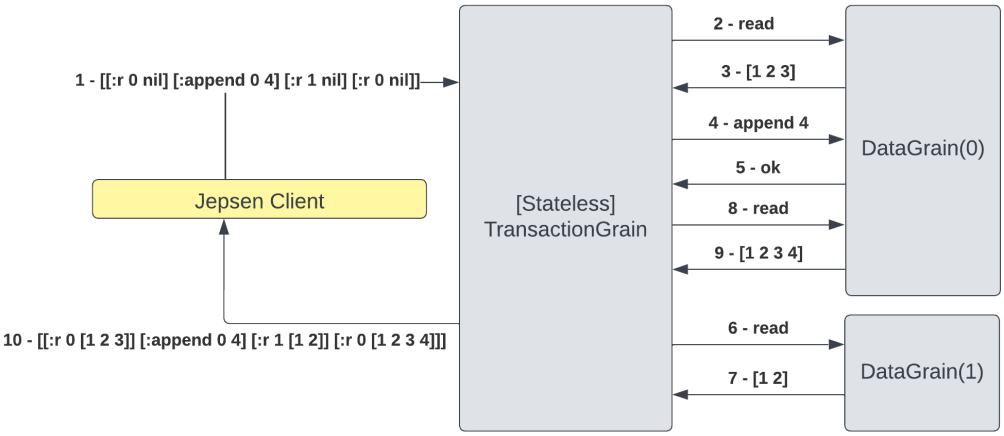

FIGURE 5.5: Example of information flow in a transaction

However, this alternative approach removes some complexity and thus is not implemented here.

There are only two functions implemented for the DataGrain. This is because Elle only requires these two actions. Both functions join an existing transaction. Activations of this type of grain can either read and return the stored list, or append a value to the list and store it. Since we are working with the virtual actor model, we can assume an empty list to exist for every key that does not need to be initialized.

After the TransactionGrain has received responses from all its DataGrain calls, it will format the message back to a string, ready to be sent back by the web server. On the returning of this string, the transaction is committed. The path an example transaction can take is given in figure 5.5.

5.1.5 Experiments

Deadlocks

One of the first observations made is that Orleans is very prone to deadlocks. Running multiple transactions generated by Jepsen that access the same keys shortly after one another results in a standstill. According to the released technical report, wait-die is used to prevent this (Eldeeb and Bernstein, 2016). A transaction requesting a lock held by a transaction with an earlier timestamp should abort. However, this differs from the shipping version of Orleans as a quick experiment easily shows A. Both transactions are aborted, whereas with Wait-Die only the newer one should abort.

That the shipping version differs from the technical report is confirmed on the Orleans discord channel, and through email conversation.

5.2 Snapper

The second framework we will be looking at is Snapper (Liu et al., 2022). To improve the transactions in Orleans, this framework was built on top of it. Snapper uses Orleans' underlying virtual actor model, and extends it with its own transaction implementation. Orleans' transactions are especially slow with high contention, and Snapper tries to solve this while still making use of the actor model. To do so,

a distinction is made between two types of transactions, namely Pre-declared ACtor Transactions (PACT) and ACtor Transactions (ACT). Whether or not it is known beforehand which actors are accessed and how many times they are accessed determines in which category the transaction will belong.

There are two types of actors defined, namely transactional actors (actors) and coordinator actors (coordinators). Transactional actors can be seen as regular actors, they contain user defined logic and have persistent state. The other type is the coordinator actor, which in accord with their name, coordinates transactions. Additionally there is a logger on each server, which can be used by multiple actors and coordinators to reduce IO cost by batching.

5.2.1 PACTs

The main performance increase comes from PACTs, which will be explained first. Before any operation takes place, an overview of which actors will be accessed during the transaction, and how many times they will be accessed has to be supplied. By providing these, Snapper can deterministically schedule the transactions. This additional information is required when starting a PACT compared to an ACT.

The client will first submit a transaction to an actor. This is the actor that will be responsible for committing or aborting the transaction. First it will send the provided information on the participating actors to a coordinator to request the start of a transaction.

The coordinator will batch all incoming requests together until it receives the token. The coordinators use a classical token ring algorithm. The token contains information on what the last assigned transaction id was, and what batch each actor received last. Each queued transaction receives a unique transaction id, and a batch id that is the lowest transaction id of that batch. The coordinator will send subbatches to each participating actor, comprising of the relevant information of the batch for that specific actor. This means only information on the PACTs that actor is part of, and the previous batch id. The token is updated and sent along the ring.

Each actor receives a sub-batch, and can see what its latest scheduled batch is before this one. It can then schedule the sub-batch so that it waits with the execution until the previous batch has completed. As these batches are deterministically scheduled, there is no possibility of an abort by conflict. Because of this, other operations can take place before previous ones are committed. Transactions that can abort by the user will cause a cascading abort, and should be committed as ACTs to avoid performance degradation. To commit a transaction, the coordinator uses 2 phase commit. The messages contain references to the batches instead of separate transactions, so a batch of transactions is committed all at once.

5.2.2 ACTs

The second type of transactions are ACTs, these resemble a more regular approach to transactions. Snapper handles ACTs using pessimistic concurrency control. Whenever the coordinator receives the token, it allocates some id's for ACTs before passing the token on. This way each ACT can receive its unique id without waiting. For the acquisition of locks S2PL is used. when the state of the actor is accessed, the lock is acquired. The locks are released on the second phase of 2PC. To avoid deadlocks it also makes use of wait-die.

5.2.3 Hybrid

To get the both of best transaction types, Snapper uses a hybrid model where both types can be submitted. Both PACTs and ACTs can be executed, with some rules set in place to guarantee correctness. One or more ACTs can only be scheduled in between batches, not during a batch. This ensures that ACTs do not see the results of half executed PACTs and vice versa. Since the PACTs are not expected to abort, the ACTs can start executing immediately after the execution of the PACT is done. In order for the next PACT batch to execute again, all the scheduled ACTs need to have been committed. An aborting ACT would cause a cascading abort if the PACT batches started immediately so it is more efficient to wait here.

With the hybrid approach there are two issues that come to light, the first being deadlocks. Assume we have two actors, A_1 and A_2. Both actors received a PACT batch B_i, and an ACT T_j. The order in which these arrive is non-deterministic. The schedule for A_1 might look like $T_j \rightarrow B_i$, and the schedule for A_2 might look like $B_i \rightarrow T_j$. Actor A_1 cannot start B_i before it has committed T_j, but actor A_2 cannot start B_i since it is expecting a call from A_1 on what to do during the transaction. Multiple actors can be involved in a deadlock. The current implementation to avoid deadlocks uses a simple timeout mechanism, which aborts ACTs to have the minimal amount of cascading aborts.

The second issue has to do with serializability. In the previous example, if A_2 were to call A_1 during the execution of B_i, then both T_j and B_i could run to completion. This would not be a serializable schedule. While running only PACT batches, or only ACTs, serializability would be guaranteed. However, another check has to be done for hybrid schedules. For every ACT, the maximum batch id from batches before it is compared to the minimum batch id of batches scheduled after it. If there are other ACTs before it, this maximum batch id relationship is transitive, encapsulating batches indirectly affecting the ordering as well. If the former is smaller than the latter, there should be no conflict. Otherwise, the schedule is not serializable and the ACT should be aborted.

5.2.4 Implementation

As the implementation of Snapper is built on top of Orleans, the implementation for the testing is similar. Again, we will be making use of a similar ASP.NET web server to handle the incoming HTTP requests from the Jepsen Client. This time, we have two separate endpoints to distinguish between transactions being submitted as ACTs or as PACTs. We use the same C# client to connect to the Orleans cluster.

The calls made to the cluster are different with this framework. Four API's are exposed, replacing standard calls for the ones provided by Snapper. These can be seen in table 5.2. Instead of making a call to a grain where the function is marked to start a transaction, we start the transaction explicitly. This starts either an ACT or a PACT. In the case of a PACT, we first analyse the transaction to list what actors will be accessed and add this data to the call starting the transaction. Calls made to other actors also use this API, passing the transaction context along. Lastly, getting the state is done through a different call, which handles locking.

The information flow of a transaction is identical to that of Orleans, using a TransactionGrain and a DataGrain again. Both PACTs and ACTs follow the same implementation, and await each call before moving onto the next.

Before the experiments can be started, we need to initialize every grain that is going to be used. This ensures that they all start with an empty list as saved state.

ACT	Task<object> **StartTxn**(string startFunc, object FuncInput)
PACT	Task<object> **StartTxn**(string startFunc, object FuncInput, Dictionary<Guid, int> actorAccessInfo)
both	Task<object> CallActor(TxnContext ctx, Guid actorID, FuncCall call)
both	Task<TState> GetState(TxnContext ctx, AccessMode mode)

TABLE 5.2: The snapper API calls replacing Orleans' functionalities

Snapper's state saving mechanism seems incompatible with the virtual actor initialization of Orleans.

5.2.5 Experiments

PACTs

We start by testing PACTs with the least complicated settings. Jepsen will use a single thread to emit only PACTs to the server. All of the transactions are received and processed properly. No inconsistencies have been observed.

The second phase of the experiment introduces the use of multiple threads. Multiple transactions are occurring on the silo concurrently, giving space for overlapping transactions. There are multiple anomalies consistently found when running the experiment with two or more threads. The first that we encounter is a G1c anomaly, which can be seen in figure 5.6. A transaction is reading key 37 written by another transaction, but a cycle appears when it also writes key 30 before that very same transaction. One of the transactions should perform all its operations before the other, but they appear to be interleaved. From the figure it is not immediately clear that the ww-dependency holds, since 21 could have been appended before 20. This dependency is found in the version ordering, which we can see in a different committing transaction. From the following committed transaction, we can see that the values were appended in ascending ordering:

```
[[:r 30 [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21]]
[:append 37 6] [:r 37 [1 2 3 4 5 6]] [:r 37 [1 2 3 4 5 6]]
[:r 36 [1 2 3 4 5]]]
```

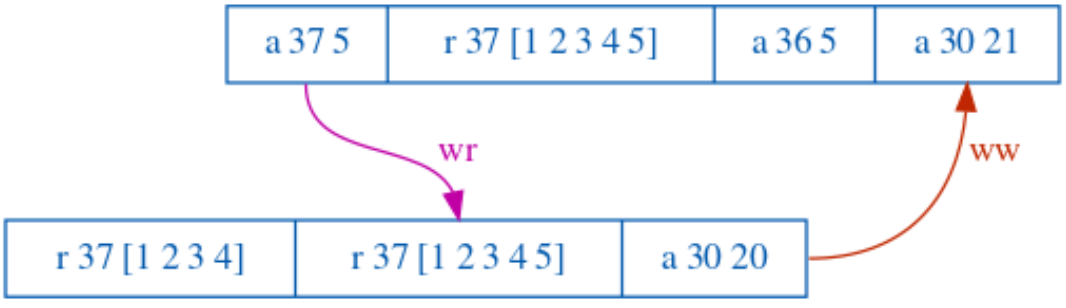

FIGURE 5.6: Dependency graph submitting only PACTs with 2 threads, G1c anomaly

This experiment also yielded some other inconsistencies. In figure 5.7 we observe a G1-single anomaly, a cycle with one read-write edge. A transaction attempts to append two values to key 86, which should happen atomically. Another transaction

observes the first appended number, but not the second. This breaks the *read-atomic* property, where either none or all of the updates, in this case appends, in a transaction should be observed.

In both of the transactions Elle claims the *read-committed* property appears to be broken. Writes from uncommitted transactions should not be observed. The Snapper paper guarantees that PACTs do not abort due to concurrency issues, but users are allowed to abort themselves. The implementation appears to do no check whether any exceptions were thrown and commit the transaction regardless[8]. If PACTs do indeed never abort, then the *read-committed* property has no meaning in this context.

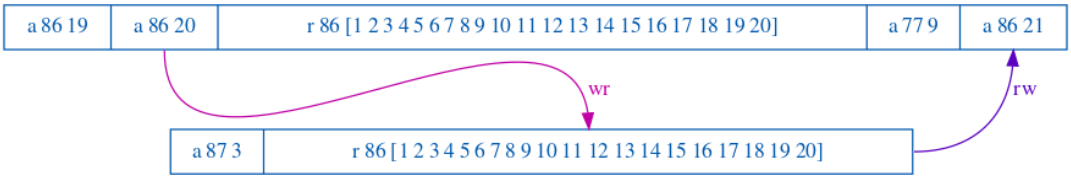

FIGURE 5.7: Dependency graph submitting only PACTs with 5 threads, G1-single anomaly

Over the course of multiple experiments, many of these anomalies have been found. It is interesting to note that all of these findings were describing transactions operating on the same key multiple times. PACTs require us to submit what keys will be accessed, and how often. The system should be aware that it is not done with a transaction yet, and limit accessibility to other transactions. This is not the case, and appears to be where the inconsistencies are coming from.

ACTs

The experiments with ACTs start with a serious impediment, the result of the transaction is never returned. In Snapper's experiments it might not have been necessary to benchmark the system. To get some use out of the experiments we add a return value ourselves, unless the transaction is aborted.

The experiment starts again by using a single thread, this time sending ACTs to the server. Even though there is one transaction happening at the same time, there are still some that are aborting. The unexpected behaviour can be seen in figure 5.8. An abort, marked in yellow, happens whenever a transaction first reads a key, and then tries to append to the same key. When looking at key 6, we can see that the aborted append is still observed by later committing transactions. Elle reports this as a *read-uncommitted*, but this is more likely a *read-aborted*, or G1a anomaly.

In that same figure, we also see the consequences of appending after a read. We see that happening when the value of 11 is appended to key 13. The transaction is committing, but there is a ww-dependency reported. This indicates that the value 11 is written after 32. Since the append was committed, it has to have happened, but since it is not observed, it must be appended last. If we look at the last read of the key, we can see that the value has simply disappeared, or was not appended properly.

```
[[:r 13 [3 7 10 12 13 14 16 18 19 21 23 24 25 29 31 32]]
[:append 11 32]]
```

[8] https://github.com/diku-dk/Snapper-Orleans/blob/main/Concurrency.Implementation/TransactionExecutionGrain.cs

Elle says that in theory the append could happen after all read were done, but it is likely that something went wrong when appending.

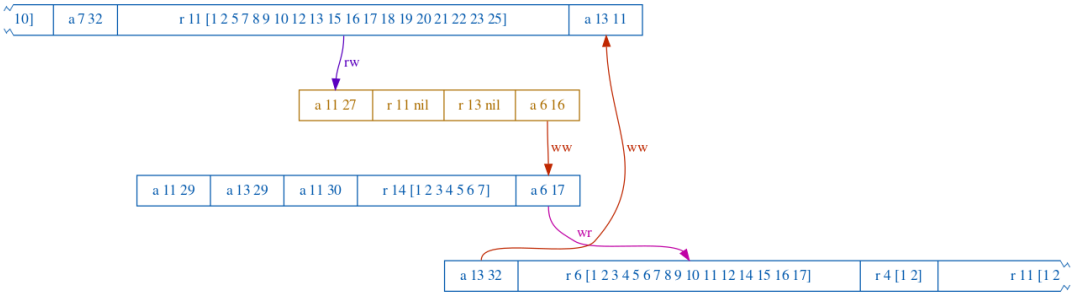

FIGURE 5.8: Dependency graph submitting only ACTs with 1 thread, G1-single anomaly. Unimportant parts of two transactions are left out for visibility.

Grouped operations workaround

The issue of how the system handles multiple accesses of the same key is also occurring in Beldi, and to continue testing we apply a workaround. The operations in a transaction are grouped together by key, and submitted as a single call. The operations are then handled atomically as a single operation. This was not done for PACTs, since the paper explicitly mentions this should not be an issue for those transactions.

With this workaround in place, we continue the experiments. We send ACTs to the server, both single- and multithreaded. There have been no inconsistencies found using the new grouped operations method.

Hybrid

For the testing of hybrid transactions, we will keep the grouped operations fix in use. The ratio of ACTs to PACTs can be adjusted, but for this experiment it is set at 1:1. Whether a transaction is submitted as ACT or PACT is still chosen randomly. This ensures that over the span of the experiment different sequences of transaction types are encountered.

When using a single thread to send the transactions, we encounter no anomalies. Both PACTs and ACTs operate well under these circumstances, and running them hybrid does not appear to introduce extra errors.

With multiple threads, we encounter the same anomalies as we have found using just PACTs. Upon inspection, the transactions that form a cyclic dependency were all submitted as PACTs. From this we reason that the hybrid execution results in no extra errors here either.

5.3 Beldi

The last framework we will be testing is Beldi. Beldi is a library and runtime system that allows for fault tolerant execution of serverless stateful functions. It is an extension of Olive, an earlier released library, with some extra functionalities. To see the full capabilities of Beldi, we need to take a look at Olive first.

5.3.1 Olive

Olive requires the developer to think of pieces of code or function calls to be executed as intents. Intents require some extra rules when implementing, but also receive guarantees from the library. When implemented correctly, Olive will guarantee that the intent is run exactly once. That is, in the case of server crashes at least once, and in the case of multiple calls at most once.

During the execution, every externally visible operation and a possible returned value will be logged. If the intent crashes for any reason, a replay of the intent will know not to perform this operation again, preventing e.g. a counter incrementing twice. Every non-deterministic local operation also has to be logged, as running the operation twice can yield different results. Any other operations can be performed locally again without affecting the outcome.

To perform the writing of the value and the logging of it atomically, Olive also introduces distributed atomic affinity logging. It overcomes the limited atomic operation size by placing the value and the logging in the same table. Concrete implementations are made for Azure table storage and Amazon DynamoDB. This logging provides the at-most-once semantics.

To ensure that intents are run exactly once, we also need at-least-once semantics. This is provided by the liveness requirement, the code can be executed and will eventually finish on some attempt. Code always resulting in an exception is not suited for Olive. The intent collector is also introduced, which will find intents that have not finished executing from the logs, and attempt to run it once more. Eventually the intent will be completed.

An important addition that makes transactions within Olive possible is to use locks with intent. Instead of having a lock give exclusive access to a single process or client, it now gives access to a single intent. It is allowed for multiple instances of the intent to access the locked object, since the exactly once semantics ensure that any externally visible operation only occurs once. If a client crashes, the lock would be lost, but with intents this is not the case.

5.3.2 Improvements

Beldi takes these ideas and improves upon them. The first change made is that the notion of the intent is changed. Where an intent was known as pieces of code to execute before, now it is an invocation of an Serverless Stateful Function (SSF) with specific parameters. The SSFs can now also call other SSFs or even another instance of its own function.

The second change is made to the logging table. Olive's logging requires the Distributed Atomic Affinity Logging (DAAL) to be within the scope where atomic operations can take place. Each CSP has a different implementation of storage, and the size on which atomic operations can take place can become a limiting factor for the DAAL. Beldi introduces a linked DAAL, where each entry in the DAAL is split into multiple entries with each row pointing to the next. The linked DAAL can then be traversed, or if the data store allows scan and projection operations this can be sped up. The databases should also be configured to be linearizable to ensure the retrieved list to form a consistent snapshot.

New modifications to the data introduces new rows to the linked DAAL. With enough time, this linked DAAL would grow to a significant size impacting overhead and storage cost. To prevent this, Beldi comes with a garbage collector. The GC removes old rows concurrently with SSFs accessing them. This is done in steps to

avoid any SSF interacting with the linked DAAL to notice the garbage collection happening.

5.3.3 Transactions

In the initial work on Olive, transactions were implemented using Optimistic Concurrency Control. Beldi has its own implementation using 2PL, wait-die and 2PC.

Any SSF can start a transaction if it is not already part of one. Upon doing so, a transaction context is created containing the transaction id and the the mode. The mode indicates whether the transaction is currently executing, committing or aborting. This context is passed along any subsequent SSF calls.

During the execution phase, any operations on the data will first try to acquire the lock. If it is unable to do so, it will check according to wait-die whether the lock is being held by a transaction that started earlier or later than the current transaction. Wait-die is used because the SSF is unable to kill other running SSFs. If the transaction holding the lock is older, it will wait and try again. Otherwise it will abort the current transaction.

The writes that happen during the execution phase are written to a shadow table. This table is used by the executing transaction only that instantiated it, and ensures that intermediate values are read as well. Garbage collection is also applied here.

The SSF that started the transaction has to eventually reach a part where it ends it, either through committing or aborting. When committing, the shadow table is flushed into the regular storage. On abort, the shadow table is erased. Any other SSFs that are part of the transaction are called as well, with the transaction context containing information on the status of the transaction. Upon commit or abort, the SSFs will omit any of the regular actions taken when called, and instead will perform the same operations on the changed data. The context is also propagated to any SSF calls made.

Beldi promises Opacity, which is strict serializability with aborting transactions not even reading wrong values. This prevents any wrong reads to have an unexpected external outcome.

5.3.4 Implementation

The prototype of Beldi created, works on AWS Lambda and DynamoDB. Three different applications from the open source benchmarking suite DeathStarBench were written to show Beldi's capabilities (Gan et al., 2019). Out of these three, 'Hotel Reservation' uses transactions and was used as guideline for the structure of the implementation to test the consistency. A total of four separate lambda functions were deployed in the tests.

The first registered lambda function is the entry point for any operations. It receives the information about the transaction, and converts it to a list of operations. The transaction is started and for each of the operations, a call is made to a data lambda function. Once the results are received the transaction is committed and the response is formatted and sent back.

The data lambda function receives a single read or append operation. It will read the stored value using the provided API, and in the case of an append it will also store the appended list back using another similar function. In case of a failure during the read or write operations, an error is returned.

The intent collector is scheduled to execute every two minutes, and the garbage collector every minute.

5.3.5 Experiments

Unreleased lock

One of the first encounters with the Beldi API already resulted in some unintended behaviour. The two functions that are required are implemented, namely reading a value and appending a value. When appending a value in a single transaction, all seems to go well. Performing this transaction multiple times results in the list functioning properly when inspecting the database. It is however when we try to only read a value that we run into trouble. The first attempt correctly returns the list, but subsequent attempts run into an error. Upon inspecting the logs, it appears that the value cannot be read from the database. Appending to the same key afterwards has the same outcome. The database is showing that the lock is not being released, even though the transaction is committed.

In Beldi's implementation of Hotel Reservation, the only of the three services containing transactions, there never occurs a read without a write. This is a situation that might have been overlooked by the developers. The solution here is to always write the value back, even in the case of just a read. Applying this method the lock gets released properly.

Grouped operations

The most simple implementation will parse the transaction string, and for each operation will make a synchronous call to the data SSFs. Take the following transaction for example:

```
[[:r 1 nil] [:append 1 10] [:r 1 nil]]
```

We would expect the second read to see the changes made by the append. This is not the case, as the returned value is:

```
[[:r 1 []] [:append 1 10] [:r 1 []]]
```

According to the paper, each call to a SSF should be passed the transaction context along, making it search for changes in the shadow table that were made by the same transaction. In the logs we can read that after the writing has completed successfully, the next read will read an empty list.

To see the scope of the problem, we can also submit the transaction as three separate transactions. The response is awaited before sending the next transaction to ensure the correct order. Here we can see that this indeed returns the expected results. The problem is located at multiple calls made to the same SSF within a transaction.

When applying a workload with Jepsen and verifying the results with Elle, we get the following anomalies reported:

- G0, a write dependency cycle is detected.
- G1c, a dependency cycle consisting of write-write and write-read edges.
- G-nonadjacent, an undiscussed anomaly. It is not well canonicalized in the literature, and thus we will not go into detail for this. A small description is given in the source code [9].
- G-single, a cycle with exactly one read-write edge.

[9] https://github.com/jepsen-io/elle/blob/main/src/elle/consistency_model.clj

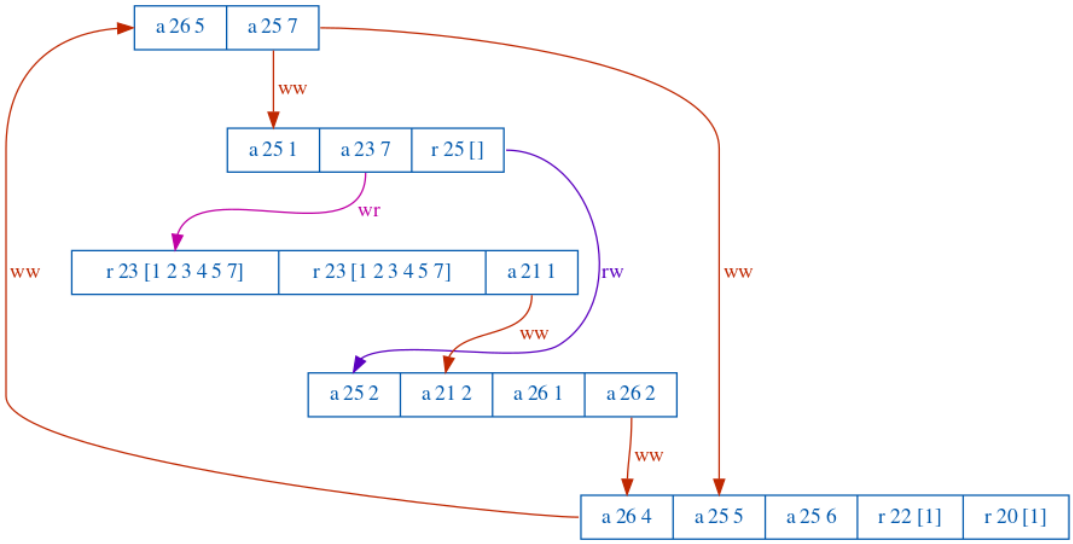

FIGURE 5.9: Unverifiable G1c anomaly detected. The cycle is visible, but exactly which transaction depends on another is not manually verifiable because of disappearing values.

It is difficult to make sense of these anomalies, since some of the most basic features of a database are not working. The graphs generated to show where the dependencies went wrong are strenuous to manually verify, since information is disappearing left and right. An example of a G1c anomaly is shown in figure 5.9. Aside from these anomalies, Elle also indicates that the system is not *read-atomic* and also not *read-uncommitted*. The former is a consistency model that relates to *when* atomic transactions become visible (Cerone, Bernardi, and Gotsman, 2015). *Read-uncommitted* is one of the lowest requirements for consistency that create this unverifiable outcome.

To solve this issue, we can group operations to the same key together. Since the transaction is submitted in a single message, intermediate results are not used anyway and operations with different keys do not interfere. The entry point will divide the operations into buckets, each bucket containing the operations for a single key, and send them out as a single call per bucket. The previously stored list is read by the data SSF, all operations are executed sequentially, and the resulting value is stored back. This gives the expected results for transactions consisting of multiple operations.

Single client

Beldi is made for easy deployment to the cloud environment and runs on Amazon Lambda. Because of this, some of the features of Jepsen are not applicable here. We have no control over the nodes and their interconnectedness. Stopping a process or partitioning the network is not possible. We are limited to testing the system under normal circumstances and can only change our client.

In this experiment we have only one client sending transactions. Whenever a response is received the next transaction will be sent out. Every request that was sent returned a correct response, indicating no unexpected events. When checking the final results with Elle, it appears that there are no anomalies either.

Multiple clients

The more clients we will add, the more interference there will be between transactions contending for the same lock. This can already be seen when adding a second client. Two requests will be sent out simultaneously to the system. Whenever they get a response, a new request is sent immediately without waiting for the other.

During the run, there are considerably more error responses from the SSFs. This appears to be happening first when two transactions involving the same key are running concurrently. The paper describes the use of wait-die, which appears to be a correct explanation when looking at the arrival times in the logs. This cannot be checked on the client side as the timestamps are registered on the SSFs, but AWS provides logs for these.

The errors of this kind are gradually increasing in frequency. Eventually there are also occurrences where the two transactions that are sent out do not overlap, yet an error is still returned. When inspecting the database after the run, we can see many locks that still have an owner. These are the keys where the errors were generated from. During collision it is possible that the lock is not released, making other attempts to access this value futile. The intent collector is periodically running, and is showing an empty table, indicating that all intents have finished. These locks will not be released.

Running the results through the Elle checker again yield no inconsistencies.

Multithreaded

The paper mentions that it has not implemented asynchronous SSF calls but the option to do so is readily available. This is done with the use of goroutines. We will be using a single client to conduct this experiment, with the operations grouped together. These options are chosen for the largest possibility of success. Per bucket of operations a single goroutine is made.

The very first call returns succesfully. Any call made afterwards to a key used before fails. Quickly all the locks become taken and do not get released. There is one exception, whenever only a single key is accessed, the lock is released succesfully.

Again, the consistency is correct, we are rarely able to use a key twice, but it is far from functional.

Chapter 6

Discussion

6.1 One-click reproducibility

In recent times there has been serious concern about the reproducibility of studies (Baker, 2016). Researchers have been unable to reproduce studies performed by others, and even reproducing their own work was not always possible. It might seem that in the field of computer science there is code supplied, exact instructions for the computer to execute, such that reproducilibility is self-evident. This is often not the case. The code might not be published, either because it is still a work in progress, it is owned by a company or it can simply be lost (Hutson, 2018). Even if the code is available, running it might give very different results. We can be running it on a different machine, there might have been some implicit decisions made and even randomness can affect the outcome.

To offer reproducbility, researchers add instructions that need to be followed to the letter. Both Snapper[1] and Beldi[2] have written instructions for achieving the results from the evaluation section in their reports. All of the CSP features used are added as well. They even include the exact procedure to get the visual graphs included. This is excellent proof that the experiments that were run actually lead to these results. If there is still any doubt, the source code is often publicly available to those with the intent to dig deeper.

This might seem comprehensive, but there lies a problem with this method. These instructions allow us to apply the systems to their chosen problems or benchmarks, with their setup only. When attempting to introduce a different problem or a different setup, we run into a large amount of specific cluttering settings, incompatible dependencies and broken promises. This would not be an issue if the experiments were covering all grounds that are required to gain confidence in the solution. Defining this set of experiments remains a challenging task, if possible at all.

In our experiments with Beldi, deviating from the given workloads immediately poses some unexpected problems. This starts out by the given transactional application always applying a write after a read. It is not hard to imagine a transaction containing only two reads being required in an application, and having other externally visible operations that do not interact with the database. It is also never the case in the benchmark that two calls are made to the same SSF, first write a value, and then read it. The writing and reading is always done in a single SSF call. It is instantly clear that there is a consistency anomaly when performing this workload and the read is not affected by the earlier write.

[1] https://github.com/diku-dk/Snapper-Orleans/blob/main/README.md
[2] https://github.com/eniac/Beldi/blob/master/scripts/README.md

6.2 Benchmarking metrics

Research is always comparing itself to its predecessor. We need this to show that the new solution is an actual improvement. Every field of research has its own metrics to measure progress made. For transactions these metrics are usually throughput, the amount of transactions finishing in a certain time frame, and latency, how long a transaction takes to complete (Thomson et al., 2012, Lu et al., 2021, Jia and Witchel, 2021). One key aspect of transactions, consistency, is barely ever tested. It is not a metric that can show an improvement over another solution by itself. Only when the consistency level is similar or more strict than that of another, do metrics like throughput and latency offer improvements. What consistency level is useful is defined by what problems it intends to solve. Snapshot isolation can provide better solutions in terms of throughput and latency than that of serializability, and even though its consistency is not as strong, there are still use cases for it.

This does not mean that it should not be tested though. A solution can claim to have a certain consistency level, but this does not mean that it is actually achieved. Even though the system theoretically should be consistent, the implementation also needs to show this. This is not unlike other metrics. To see why consistency testing matters, we will use throughput as an example. When the description of the solution promises a higher throughput, it still need to show this in practice with at least a prototype. If we are to dismiss the verification of consistency, the prototype might be incorrectly displaying better throughput results. Were the system consistent, it might need many more steps that are not measured in the current outcome.

6.3 Dependencies

In testing these systems, we are not only testing their correctness, but also that of the underlying systems they depend on. The collaboration between all cogs in the machine is required to perform well, since parts depend on one another. If there is one weak link that is not performing as promised, the system might not adhere to its guarantees.

If we take a quick look at the systems tested here, we can immediately find some of these dependencies. We will start by looking at Orleans.

For the grain directory, there are two provided options. Either use Microsoft's own Azure Storage, or the use of Redis. A custom implementation can be made as well. It is mentioned that the eventual consistency of Azure Storage is enough to make Orleans work properly. When Redis is deployed on multiple machines it does not guarantee strong consistency either. What a custom implementation should adhere to is unknown, as the interface to be extended does not offer this information. It appears that the grain directory is not influential for consistency in transactions as discussed in chapter 5.1, but the lack of known requirements of dependencies is illustrated well here.

The second configurable storage that is used in Orleans, is for the persistence

of data. There are multiple options to choose from, each with their own guarantees. Two examples are Azure Blob Storage which uses snapshot isolation[3] and DynamoDB which offers eventual consistency by default[4]. Again, the exact requirements are not known. Whenever we use transactions however, our options are limited. We are confined to a single transactional implementation, one that uses Azure Table Storage. This is said to have strong consistency (Calder et al., 2011).

Beldi is using DynamoDB for storage in the linked DAAL. DynamoDB uses eventual consistency by default, which should not be enough to get serializability. Taking a look at the part of Beldi that handles database operations we can see that every read operation of any kind is set to use the "ConsistentRead" flag [5]. This should provide all reads with the latest version of the data with every prior update having taken place [6].

There are some problems with depending on proprietary software. Since the major CSPs have a large infrastructure, they offer the fastest and cheapest solutions. Building your system on top of this makes it highly dependent. For some services provided there exist research papers, documentation is often available on the CSP's website, but these sources can quickly get outdated with updates to the source code. It might be in the CSP's best interest to keep most relied upon products still functional, but there are no guarantees. In truth, the user has very little idea what is going on at the CSP's servers. If the service were to be updated or shut down, the software depending on it would be useless. If an open source solution that could be self hosted would be depended on, no changes can make the final product faulty.

6.4 Standardization

All of the frameworks tested are reliant on external systems, both for persistent storage and for other overhead. It can be inevitable to make use of such systems. When using them, a definition of what consistency level is required for your framework to function properly is essential. If other features of the specific storage are required as well, this should be explicitly documented. This way, instead of relying on an external system, it is relying on more abstract features.

Having at least two options to choose from is also beneficial. This shows that your framework works with a general solution. The overlap in consistency level that both options give is also a clear indication of what consistency level is required in them.

The debate about whether or not the consistency testing tool used is actually a black-box, meaning that absolutely no changes need to be made to the framework, is not relevant for stateful functions. The main idea of stateful functions is that we can create our own implementation and decide how we alter the data. The by Jepsen required append function is easily appended to the application's list of functionalities.

To generalize a testing framework, it needs to be decoupled from the distributed system as much as possible. The testing framework will generate a workload which needs to reach the distributed system, and a response needs to be sent back. As far as the distributed system is concerned, it is receiving requests from a regular client. The testing framework in turn should not rely on internal information of the distributed

[3] https://learn.microsoft.com/en-us/azure/storage/blobs/concurrency-manage

[4] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

[5] https://github.com/eniac/Beldi/blob/master/pkg/beldilib/dblib.go

[6] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html

system. Not only does this require a specific implementation, it also should not matter to the testing framework what is going on behind the scenes. If the output given by the system is always correct, then the system is consistent.

The communication happening should not prove to be a barrier. Since solutions use different programming languages, we need a universal way of sending data. Three major CSPs all offer triggers for their functions, multiple from actions within the CSP's ecosystem, but also one for HTTP requests. Setting up a web server to handle HTTP requests when running a system locally is also made accessible by the many options available for all of the most popular programming languages. Little knowledge is needed to set this up, and the only functionality the web server has is parsing the input and passing it along. The data can be sent as JSON, which is broadly supported, or even as a string, where custom parsing is required.

Chapter 7

Conclusion

At the start of this thesis, we have formulated three questions to guide this research. In this chapter we aim to answer these questions and conclude our research.

1. What is the current state of testing SFaaS systems for consistency?

The three frameworks we have looked at in this thesis each report on their throughput and latency, but any tests of consistency are left to be desired. Newer work building on top of one of the frameworks omits this as well, and since it is using a similar foundation any inconsistencies are inherited as well. There is little attention given to consistency aside from a theoretical assumption. There are no consistency tests performed. This even going as far as one of our tested frameworks not returning a value at all. We can only hope that at some point during development this was checked, as altering the code is required to verify this ourselves.

2. Do existing SFaaS systems adhere to the promised consistency level?

To answer this question, we have set up three SFaaS systems that are described in chapter 5 and applied the testing framework from chapter 4 to them. The first of the three systems, Orleans, does what it promises without any modifications, albeit with an extremely low throughput when there is more lock contention. Serialization is promised and no evidence was found that this is not the case.

The second system, Snapper, promises serializability as well. With both methods of submitting transactions, issues arise whenever a transaction accesses the same key multiple times. Although it might have been overlooked that it can be desired to access the same key with ACTs, it is required to indicate how many times a key will be accessed with PACTs. Depending on how we look at it, the system's consistency level is either read uncommitted or not even that. With these faults in place it is definitely not serializable.

Lastly, Beldi, also promises serializability. Again we have the same issue of not being able to access the same key multiple times. This again brings the consistency level to below read uncommitted. If we are lenient to this fault, we encounter many locks not being released and keys becoming unusable.

3. What should a standardized consistency testing format for SFaaS look like?

In chapter 6, we touch upon many factors that are required and will benefit a standardized testing format. By decoupling the testing framework from the SFaaS implementation completely, newly developed systems can easily be plugged in and tested. Simple custom functions such as append are almost the only thing required by the developer for the system to be tested. Using broadly used channels of communication such as HTTP should pose no barrier for software written in any language.

Jepsen is also offering nodes on the AWS marketplace to spare a troublesome installation process on your own hardware. Such a testing setup, which does not have to include Jepsen, should leave SFaaS framework developers with no excuse not to test consistency.

Appendix A

Experiments

All code will be available on https://github.com/wvanlil/ConsistencyStatefulFaaS

A.1 Jepsen

It is possible to run entire Jepsen tests on AWS, however the latest version is not available and editing the source code requires more control than this offers. The other option that we have is creating our own containers using LXC.

The process to install the containers is described on Jepsen's Github page[1], but the required steps for Ubuntu 20.04 LTS are given here. When using another operating system or if any of the steps are not working, the page should be referred to.

First we install the required tools:

```
$ apt install lxc debootstrap bridge-utils libvirt-clients
libvirt-daemon-system iptables ebtables dnsmasq-base libxml2-utils
iproute2
```

Next we can create virtual machines. With this command, 5 are generated, but this is variable. The machines will be running debian buster as Jepsen is tested on them.

```
$ for i in {1..5}; do sudo lxc-create -n n$i -t debian --
--release buster; done
```

There are several scripts that help in setting up the containers. These are located in the */Jepsen/jepsen_scripts* folder. In order to start 5 containers, we run the following command:

```
/Jepsen/jepsen_scripts $ bash start.sh 5
```

We can use the stop script to stop these 5 containers:

```
/Jepsen/jepsen_scripts $ bash stop.sh 5
```

To get information on the containers that are currently running:

```
/Jepsen/jepsen_scripts $ bash list.sh
```

If the Jepsen test is unable to find the nodes during the test, we need to create a list with IP-addresses to find them. The following command creates a text file at `/Jepsen/jepsen.elle/resources/node_file.txt`:

[1] https://github.com/jepsen-io/jepsen/blob/main/doc/lxc.md

```
/Jepsen/jepsen_scripts $ bash list.sh | bash create_nodelist.sh
```

These containers are the nodes where the tested systems will be running. We can use the local machine as our control node. This node will send instructions to the containers and send the actual test commands. On this local machine, we need Leiningen installed. Installation instructions can be found on their website [2].

After running the test, we can verify the result that are locally stored. We do this by entering REPL mode:

```
lein repl
```

To apply the checker to the latest history, we run the following commands:

```
(require '[elle.list-append :as a])

(def h (mapv read-string (clojure.string/split-lines (slurp
"store/latest/history.edn"))))

(pprint (a/check {:consistency-models [:serializable], :directory
"out"} h))

scp root@10.0.3.119:/snapper/snapper.log /home/wouter/Documents/Scriptie/Logs/
```

No such file or directory If wget fails, create folder by ssh'ing into the machine.

A.2 Orleans

All components that we will be using with Orleans require some personal settings to be applied. The file `Orleans/Custom/Constants.cs` contains these settings and should be edited.

The connection string is required to connect to the persistent storage and can be found on the Azure website. It is located under `Home->Storage accounts->your storage account->Access keys`. If no storage account is present, one has to be created.

The same service id should be used throughout all experiments, as this is used by some Azure providers. The cluster id can be changed every deployment, and makes sure that all clients and silo's with the same id can communicate.

A.2.1 Jepsen

To run the full jepsen test, we need to create the silo's and the web client to interact with it. We start with building the silo.

```
dotnet build Silo --runtime linux-x64 --no-self-contained
```

The silo should be compressed in a tar file, and uploaded somewhere publicly available so that the containers can freely download it. We start by creating the tar file from the built silo.

```
tar -zcvf file.tar.gz path/to/dir
```

[2] https://leiningen.org/

Next we need to upload it somewhere. For the tests we have used dropbox. Copy the download URL into the clojure tests.

Next we build the web application.

```
dotnet build WebApp --runtime linux-x64 --no-self-contained
```

If we have run an experiment before, we might need to clear the previous stored values. This can be done in the azure portal. In the storage account, we go to storage browser, and we can see two tables. The OrleansSiloInstance keeps track of all silo's, active or dead. Because the silo's are given a sudden stop signal in the tests, this table is not cleaned up well. If any issues arise, this should be manually emptied. The same should be done for the TransactionalState table, to initiate the experiment with a clean start. Otherwise old values can still be read which causes inconsistencies.

When everything is ready, we can start jepsen by calling the clojure file. The Orleans test is set as the main entry point of the project, so we don't need other flags here.

```
lein run --nodes-file ./resources/node_file.txt
```

Whenever at least one silo has started, and populated the available silo table, the web application can be started. There is a 15 seconds window where this needs to happen. In the case of the web application not finding any silo, there is enough time to restart the process during this time period. If there is not enough time, some initial transactions might get lost, but this is not a problem.

A.2.2 Deadlock

The deadlock example requires only one silo and a special client to be run. This can be executed on a single machine. The client automatically starts the required transactions and Jepsen is not necessary. Make sure that both the ASP.NET and .NET runtime are available.

```
apt-get install dotnet-runtime-6.0
apt-get install aspnet-runtime-6.0
```

We can then continue by building the solutions.

```
dotnet build Silo

dotnet build Client
```

We then start the silo. Once this is done we can start the client, who will then connect to the cluster.

```
dotnet Silo.dll
dotnet Client.dll
```

A.3 Snapper

The Snapper experiments are run similar to that of Orleans. We start by building the solutions. Since Snapper is using .NET Core 3.1 we do not need to build for a specific runtime. We build both the Silo and the Client with the server attached.

```
dotnet build SnapperSiloHost
dotnet build ElleSnapperExperimentProcess
```

Next we install the correct runtime. We install the ASP.NET runtime on the local machine. The .NET runtime for the Silo is automatically installed with our test script.

```
apt-get install aspnetcore-runtime-3.1
```

We need to upload our file again, and change the URL in our *snapper.clj* clojure file with a direct download link. We then start one container and run our test file.

```
lein run -m jepsen.snapper test --nodes-file ./resources/node_file.txt
```

On our local machine, once the silo has started, we can start the web server which will connect as a client to the cluster.

```
dotnet ElleSnapperExperimentProcess.dll
```

A.4 Beldi

To start our experiments with Beldi, we first need to upload our code to AWS. This is done using a docker container. The next command will mount the folder containing Beldi's code with our additions to the container. This container has the tools installed for automatically creating AWS Lambda functions.

```
$ docker run -it -v "$(pwd)"/Beldi-master:/root/elle
tauta/beldi:latest /bin/bash
```

All following commands need to be executed on the container.

Next, configure AWS with your credentials. We first need to create an access key. To do so, go to the AWS portal and select Account->Security Credentials->Access keys->Create access key. Next we can run the following command on the container.

```
$ aws configure
```

We need to set the region to us-east-1, since Beldi appears to have some other configurations that are hardcoded. The output format needs to be set to JSON.

A shell script is made to compile the function code, deploy it to the cloud and setup the database. This last part cleans the database, creates the tables and populates them.

```
$ ./scripts/elle/run.sh
```

Next we need to create an endpoint for the function. Go to the AWS portal, and select Lambda->Functions->beldi-dev-elle. Add a trigger and select API Gateway. Create a new API, set the API type to HTTP API and set the security to open. The trigger should have a link to the gateway.

If this link is not working during the tests, there is also a function URL available. This can be used as an alternative.

The URL needs to be added in jepsen.elle/src/jepsen/beldi.clj.

To run Jepsen, we need at least one node running, even though it is not being used. Usually the amount of nodes is linked to the amount of requests concurrently sent, however this can be altered by using the concurrency flag.

```
lein run -m jepsen.beldi test --nodes-file ./resources/node_file.txt
--concurrency 5
```

	groupOps	async
Single operation	false	false
Grouped operation	true	false
Asynchronous	true	true

TABLE A.1: Table showing the possible configurations for experiments.

A.4.1 Settings

For the experiments described in section 5.3.5, we need to adjust some settings according to the experiment. The file Beldi-master/internal/elle/main/main.go contains two variables, groupOps and async, which should be set according to table A.1. After this, the shell script to build and deploy should be run again, which also cleans the database.

A.4.2 Grouping operations

To show the textual example of the grouped operations, a python file sending requests to AWS Lambda is supplied. Make sure the options are set to match table A.1. Then run the python script with the following command.

```
python3 groupop_test.py
```

Bibliography

Adya, Atul, Barbara Liskov, and Patrick O'Neil (2000). "Generalized isolation level definitions". In: *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, pp. 67–78.

Agha, Gul A et al. (1997). "A foundation for actor computation". In: *Journal of functional programming* 7.1, pp. 1–72.

Baker, Monya (2016). "Reproducibility crisis". In: *Nature* 533.26, pp. 353–66.

Berenson, Hal et al. (1995). "A critique of ANSI SQL isolation levels". In: *ACM SIGMOD Record* 24.2, pp. 1–10.

Bykov, Sergey et al. (2011). "Orleans: cloud computing for everyone". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pp. 1–14.

Calder, Brad et al. (2011). "Windows azure storage: a highly available cloud storage service with strong consistency". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 143–157.

Castro, Paul et al. (2019). "The rise of serverless computing". In: *Communications of the ACM* 62.12, pp. 44–54.

Cerone, Andrea, Giovanni Bernardi, and Alexey Gotsman (2015). "A framework for transactional consistency models with atomic visibility". In: *26th International Conference on Concurrency Theory (CONCUR 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Chays, David et al. (2004). "An AGENDA for testing relational database applications". In: *Software Testing, verification and reliability* 14.1, pp. 17–44.

DeCandia, Giuseppe et al. (2007). "Dynamo: Amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6, pp. 205–220.

Dragoni, Nicola et al. (2017). "Microservices: yesterday, today, and tomorrow". In: *Present and ulterior software engineering*, pp. 195–216.

Eldeeb, Tamer and Philip A Bernstein (2016). "Transactions for Distributed Actors in the Cloud". In.

Fekete, Alan et al. (2005). "Making snapshot isolation serializable". In: *ACM Transactions on Database Systems (TODS)* 30.2, pp. 492–528.

Fox, Armando et al. (1997). "Cluster-based scalable network services". In: *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pp. 78–91.

Gan, Yu et al. (2019). "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18.

Gilbert, Seth and Nancy Lynch (2012). "Perspectives on the CAP Theorem". In: *Computer* 45.2, pp. 30–36.

Haerder, Theo and Andreas Reuter (1983). "Principles of transaction-oriented database recovery". In: *ACM computing surveys (CSUR)* 15.4, pp. 287–317.

Hewitt, Carl (2010). "Actor model of computation: scalable robust information systems". In: *arXiv preprint arXiv:1008.1459*.

Hewitt, Carl, Peter Bishop, and Richard Steiger (1973). "A universal modular actor formalism for artificial intelligence". In: *Proceedings of the 3rd international joint conference on Artificial intelligence*, pp. 235–245.

Hutson, Matthew (2018). *Artificial intelligence faces reproducibility crisis*.

Jia, Zhipeng and Emmett Witchel (2021). "Boki: Stateful serverless computing with shared logs". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 691–707.

Kamal, Muhammad Ayoub et al. (2020). "Highlight the features of AWS, GCP and Microsoft Azure that have an impact when choosing a cloud service provider". In: *Int. J. Recent Technol. Eng* 8.5, pp. 4124–4232.

Kingsbury, Kyle and Peter Alvaro (2020). "Elle: Inferring isolation anomalies from experimental observations". In: *arXiv preprint arXiv:2003.10554*.

Li, Haixiang, Yuxing Chen, and Xiaoyan Li (2022). "Coo: Consistency Check for Transactional Databases". In: *arXiv preprint arXiv:2206.14602*.

Liu, Yijian et al. (2022). "Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems". In: *Proceedings of the 2022 International Conference on Management of Data*, pp. 65–78.

Lu, Yi et al. (2021). "Epoch-based commit and replication in distributed OLTP databases". In.

Minhas, Umar Farooq et al. (2012). "Elastic scale-out for partition-based database systems". In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, pp. 281–288.

Nance, Cory et al. (2013). "Nosql vs rdbms-why there is room for both". In.

Pritchett, Dan (2008). "BASE: An Acid Alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability." In: *Queue* 6.3, pp. 48–55.

Setty, Srinath et al. (2016). "Realizing the {Fault-Tolerance} Promise of Cloud Storage Using Locks with Intent". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 501–516.

Shahrad, Mohammad, Jonathan Balkind, and David Wentzlaff (2019). "Architectural implications of function-as-a-service computing". In: *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pp. 1063–1075.

Sreekanti, Vikram et al. (2020). "Cloudburst: Stateful functions-as-a-service". In: *arXiv preprint arXiv:2001.04592*.

Stonebraker, Michael (2010). "SQL databases v. NoSQL databases". In: *Communications of the ACM* 53.4, pp. 10–11.

Tan, Cheng et al. (2020). "Cobra: Making transactional key-value stores verifiably serializable". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp. 63–80.

Thomson, Alexander et al. (2012). "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pp. 1–12.

Warszawski, Todd and Peter Bailis (2017). "Acidrain: Concurrency-related attacks on database-backed web applications". In: *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 5–20.

Zheng, Mai et al. (2014). "Torturing databases for fun and profit". In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 449–464.