DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Distributed Transactions on Serverless Stateful Functions using Coordinator Functions

Author:
Martijn de Heus

Supervisor:
Dr. Asterios KATSIFODIMOS

Co-supervisor:
Dr. Marios FRAGKOULIS

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

in the

Web Information Systems Group
Software Technology

May 16, 2021

Declaration of Authorship

I, Martijn de Heus, declare that this thesis titled, Distributed Transactions on Serverless Stateful Functions using Coordinator Functions 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: 16 may 2021

... an evolution that parallels the transition from assembly language to high-level programming languages.

Jonas et al.

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Distributed Transactions on Serverless Stateful Functions using Coordinator Functions

by Martijn de Heus

Lately, serverless computing has gained much attention. Function-as-a-Service (FaaS) is the most prominent serverless model. The FaaS model allows cloud users to write simple functions, and the cloud provider takes care of the deployment, maintenance, and scalability of those functions. However, FaaS can be improved upon significantly. Existing FaaS services do not solve some significant and challenging cloud computing problems regarding state and communication, such as function-to-function calls and transactions. Work has been done to improve this early iteration of serverless computing by incorporating state in the equation; leading to Stateful Function-as-a-Service (SFaaS). However, transactions on SFaaS remain an open problem, even though they are crucial to allow more use cases to operate using SFaaS.

This thesis examines existing SFaaS systems and their data correctness guarantees. Based on existing SFaaS systems, this thesis proposes a programming model for both serializable transactions and sagas orchestrations using coordinator functions. The programming model is implemented in Apache Flink StateFun. The implementation is based on existing research on traditional database technology.

The implementation's performance is evaluated using an enhanced version of the *Yahoo! Cloud Systems Benchmark* (YCSB); YCSB is extended to include a transactional operation. The implementation reaches 1840 sagas workflows per second and 1200 serializable transactions at sub-200ms latencies on 5 workers with 2 CPUs each.

Acknowledgements

I would like to thank my supervisor, Dr. Asterios Katsifodimos, for guiding me towards interesting research areas and asking critical questions throughout my process. I would also like to thank Dr. Marios Fragkoulis for the frequent discussions that often motivated me and led me forward.

Furthermore, I would like to thank all the contributors to both the Apache Flink project and the Apache Flink StateFun project for building these systems that are the foundation of the implementation in my thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

10 years ago, the cloud was an extremely promising technology capable of reshaping IT operations (Armbrust et al., 2009). Cloud providers use economies of scale to reduce computing costs and simplify computing for cloud users. For example, cloud users may benefit from on-demand compute services provided and maintained by cloud providers. Besides greatly simplifying operational challenges, cloud computing can also reduce or remove up-front investment requirements and lower the entry barrier for technology startups significantly, boosting innovation (Gupta, Seetharaman, and Raj, 2013).

However, the promise of cloud computing has only partly been fulfilled. Operations in the cloud are not yet as simple as promised, and in some cases, operations became even more complex. This unfulfilled promise has recently led to "serverless" cloud computing becoming increasingly popular. A Berkeley View on Serverless Computing states: "It provides an interface that greatly simplifies cloud programming, and represents an evolution that parallels the transition from assembly language to high-level programming languages" (Jonas et al., 2019). Serverless computing allows cloud users to deploy applications on cloud infrastructure using high-level APIs while abstracting away most operational concerns, including deployment, auto-scaling and resource management. Meanwhile, it provides an opportunity for cloud providers to reduce costs by optimizing resource usage.

Out of different models that may be described as serverless computing, Function-as-a-Service (FaaS) has gained the most attention (Jonas et al., 2019; Baldini et al., 2017a; Hellerstein et al., 2018; Eyk et al., 2017; Baldini et al., 2017b; Akhter, Fragkoulis, and Katsifodimos, 2019). FaaS allows users to deploy a simple function taking input, processing this input, and resulting in output(s). The functions themselves are typically stateless, simplifying the cloud provider's operations since they can run functions anywhere without any dependencies. However, this also means a function requiring state has to do the following: (1) fetch data from an external database, (2) perform some logic based on the read data and the input of the function, and (3) write the data back to an external database, all within a function's execution. This means functions are idle while waiting for external databases' response, introducing extra latency and defeating one of the purposes of FaaS to optimize resource usage. This also introduces unnecessary generic code to communicate with the external system, defeating the purpose of FaaS to abstract this away and allow the developer to focus on code implementing business logic only. Besides this, state management challenges such as state management across functions (e.g., efficient function-to-function calls and transactions across functions) and fault-tolerant exactly-once semantics remain open problems.

1.1 Stateful Function-as-a-Service

To improve upon FaaS, provide a more complete programming model, and allow more classes of applications and use cases to run using serverless computing, there is ongoing work to create Stateful Function-as-a-Service (SFaaS) systems. Most notable systems are StateFun[1], Cloudstate[2], Cloudburst[3] (Sreekanti et al., 2020), Beldi[4] (Zhang et al., 2020) and Azure Durable Functions[5].

1.1.1 Transactions

These systems each have different approaches and varying programming models. StateFun, Cloudstate, and Azure Durable Functions provide a relatively similar programming model encapsulating state within a function instance. In contrast, Cloudburst and Beldi provide functions access to shared mutable state in a distributed key-value store. The different systems have different consistency guarantees. Beldi (Zhang et al., 2020) is the only system that supports multi-key transactions with isolation guarantees. Other systems lack any notion of transactions, even though transactions are described as one of the most significant challenges with modern micro-services-like architectures (Katsifodimos and Fragkoulis, 2019), and would allow many additional use cases to be implemented on SFaaS systems. SFaaS systems do provide an opportunity to coordinate state management and implement transactions within the centralized SFaaS back-end, something that is currently impossible with popular decentralized micro-services architectures.

1.2 Research questions

To shape this thesis, the following research questions are formulated:

1. What SFaaS implementations exist today, and what are their data correctness guarantees?
2. What are shared features of existing SFaaS implementations, and how can they be used to find a conceptual solution for transactions that can be implemented in SFaaS systems?
3. What programming model can be used to implement transactions in SFaaS systems intuitively?
4. What isolation guarantees can be provided for transactions in SFaaS systems, and how do these affect performance?
5. How can SFaaS systems, with and without transactions, be evaluated?

1.3 Contributions

The thesis and its contributions are structured as follows. Chapter 2 describes preliminary concepts to provide common definitions throughout the thesis. After this, the thesis makes the following contributions:

[1]https://statefun.io
[2]https://cloudstate.io/
[3]https://github.com/hydro-project/cloudburst
[4]`https://github.com/eniac/Beldi`
[5]https://docs.microsoft.com/en-us/azure/azure-functions/durable/

1. The current state of SFaaS systems is summarized and discussed in chapter 3.
2. Two fundamental properties provided by multiple SFaaS systems that form a basis to implement transactions are identified and formalized in section 4.1.
3. A concept and programming model to implement transactions in SFaaS systems with these fundamental properties is presented (section 4.2).
4. The concept is implemented in StateFun, including serializable transactions and sagas workflows (chapter 5). The implementation is publicly available on GitHub[6].
5. An approach to evaluate cloud systems with transactions is presented in section 6.1.1.
6. The performance of the introduced transactions in StateFun is evaluated in chapter 6.

After these contributions, the conclusion (chapter 7) summarizes the work by answering the posed research questions. Chapter 8 discusses the results, presents avenues for future research, and describes practical ideas to further improve Flink StateFun.

[6]https://github.com/delftdata/flink-statefun-transactions

Chapter 2

Preliminaries

This chapter introduces some fundamental concepts and definitions that are required to understand the rest of this thesis. Firstly, this chapter introduces definitions of different levels of data correctness. Next, this chapter presents well-known approaches to achieve these levels of data correctness. This chapter also describes fault-tolerance in distributed systems, including techniques to implement fault-tolerance and reliable communication in distributed systems. This chapter concludes with a short description of benchmarks used to evaluate data systems.

2.1 Data correctness

When building an application, application developers face decisions regarding the trade-off between data correctness and performance. It is essential to understand this dilemma thoroughly to make an informed decision. Over the years, terminology precisely defining different levels of data correctness have been introduced. This terminology should also enable clear communication between application developers and engineers developing the underlying systems. The levels of data correctness are separable in two categories; consistency levels (also often referred to as consistency models) and isolation levels. Application developers (and system implementors) often slightly misunderstand these levels and use them interchangeable or combined into a single concept. This section aims to clarify these concepts by describing each individually (sections 2.1.1 and 2.1.2) before reflecting on their relationship (section 2.1.3).

Firstly, consistency levels describe how different clients (or processes using the data) observe data in distributed systems. This applies to distributed systems where multiple processes interact with the data (and the system may replicate the data). Specifically, consistency levels describe how these different processes observe the data and writes made by other processes.

Secondly, isolation levels describe the execution of *transactions* (or sets of operations). Specifically, isolation levels describe the semantics of concurrent transactions. Transactions and isolation levels also apply to traditional non-distributed single-process systems, unlike consistency levels.

2.1.1 Consistency levels

Consistency is a broad term and used in many different contexts. Even in data management and distributed systems, two popular concepts include the word *consistency*; A<u>C</u>ID transactions (Haerder and Reuter, 1983) and the <u>C</u>AP theorem (Gilbert and Lynch, 2002). Isolation levels describe ACID transactions, and section 2.1.2 describes the definition of consistency in this context. The consistency levels discussed here relate to *consistency* as defined by the CAP theorem. The CAP theorem refers to

Concept	Definition
Consistency	Consistency means that any two clients (or processes) should always read the same data at any point in time, even if connected to different nodes. This guarantee corresponds to linearizability as described below.
Availability	Availability means that clients should always receive a (non-error) response from the system, even if some system nodes die. Availability is loosely interpretable as the time it takes a system to respond and usable as intuition for the trade-off between performance and correctness (or consistency).
Partition tolerance	Partition tolerance means the system continues to function even in the case of *network partitions*. A network partition means there is no communication between two nodes in the system. Network partitions are usually outside the developers' control, and therefore, partition tolerance is usually a requirement for distributed systems.

TABLE 2.1: Definitions of guarantees in the CAP theorem

consistency, availability, and partition tolerance guarantees. Table 2.1 includes the definitions of the CAP guarantees. The CAP theorem states that distributed systems can only ever comply with two of the three guarantees. Note that the CAP theorem includes no reference to transactions; therefore the consistency levels do not apply to transactions.

Consistency levels, as discussed here, provide more insight into the consistency guarantee as defined by the CAP theorem. They describe different compliance levels with the consistency guarantee and allow developers and users to reason about the guarantees of the system they are using or developing. The CAP theorem applies to systems that may replicate data across nodes to increase performance and availability. Clients may connect to different nodes containing replicas of the same shared data. Consistency guarantees describe whether all clients read the same data, even though they might be connected to other instances that each have a copy of the data.

Below 5 different consistency levels are described including example schedules. In the example schedules, `W(X, 1)` represents a write operation writing `1` to `X`. `R(Y): 2` represents a read operation where `2` is read from `Y`. Before the operations start, all data is `0`. The consistency levels are introduced in decreasing order of strictness. Every example schedule is also a valid schedule for consistency levels introduced after; or in other words, a stricter consistency level allows a strict subset of schedules permitted by a more lenient consistency level.

Strict consistency

Strict consistency is the strictest model of consistency. It ensures that any process or client always observes the most-recent write for an item. It does this by ensuring a global ordering of both reads and writes based on real-time. Figure 2.1 shows an

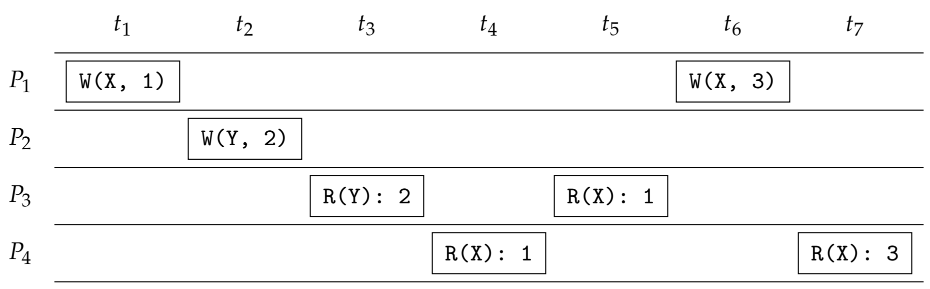


FIGURE 2.1: Example schedule that is strictly consistent

example schedule of strict consistency. In this schedule, read operations always read the value that is most recently written.

This consistency level is fully deterministic and allows for straightforward reasoning about the behavior of a system. Strict consistency depends on a global ordering of all operations. This global ordering is often very costly (in terms of performance) or even impossible to implement.

Linearizability

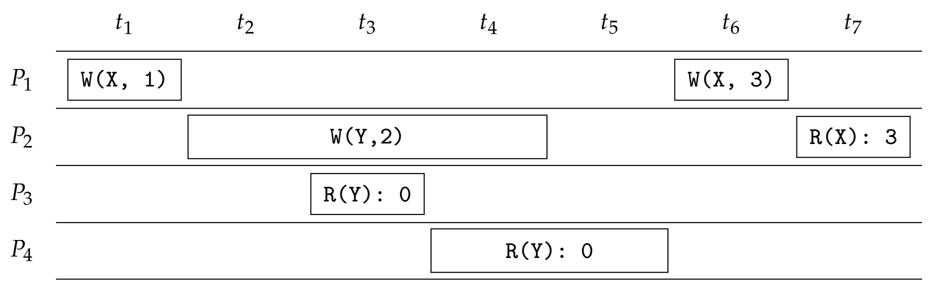


FIGURE 2.2: Example schedule that is linearizable

Linearizability (Herlihy and Wing, 1990), also called *atomic consistency* (Lamport, 1986), is a consistency level in-between sequential consistency and strict consistency. The CAP theorem refers to this consistency level with its consistency guarantee. This consistency level definition is different from the other consistency levels as it acknowledges that operations may take time and may overlap (or, in other words, be performed concurrently). Linearizability allows these overlapping operations to be ordered before or after each other (in the global order) independent of which operation started or finished earlier. Therefore, linearizability is not deterministic, like strict consistency. However, like strict consistency, a single global order of all operations based on real-time still exists; so if one process has observed or written a value, all following processes should observe the same data. The introduction of overlapping operations makes this consistency level more applicable to real-world systems and practical than strict consistency.

Figure 2.2 shows a linearizable schedule. In this schedule, P_3 and P_4 may read both 2 and 0 as these read operations overlap with `W(Y, 2)` in P_2. This is a difference with strict consistency. However, if P_3 would read 2 at t_3, P_4 also has to read 2 as the operation of P_3 finishes before the operation of P_4 starts.

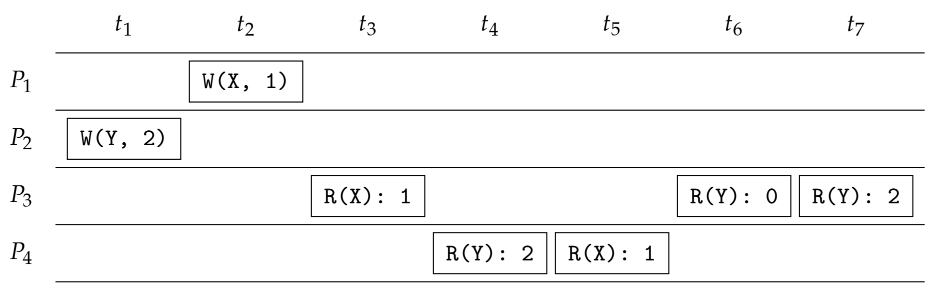


FIGURE 2.3: Example schedule that is sequentially consistent

Sequential consistency

Sequential consistency is a weaker form of consistency than strict consistency and linearizability. Sequential consistency is based on a definition by Lamport: "The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." (Lamport, 1979). In other words, sequential consistency ensures that there is some single global ordering of operations. However, this ordering is not necessarily based on real-time and may also not necessarily be observed by all processes instantly as for strict consistency.

In figure 2.3, it can be observed that in the global ordering `W(X, 1)` precedes `W(Y,2)` because P_3 can observe `0` at t_6 while P_3 has already observed `1` at t_3. This ordering means that since P_4 has already observed `2` at t_4, P_4 must also observe `1` at t_5. If P_4 would not have observed `1` at t_5, this schedule would not be sequentially consistent.

A difference with strict consistency and linearizability in this example schedule is that P_3 may still observe `0` at t_6.

Causal consistency

Causal consistency is a weaker level of consistency than sequential consistency. Where sequentially consistent schedules require a global ordering for all operations, causally consistent schedules only require this relative ordering for causally-related operations (Ahamad et al., 1995). Causally-related operations are based on the *happens-before* relation as defined by Lamport (Lamport, 1978).

Figure 2.4 shows a schedule that is causally consistent. Figure 2.4 shows that `W(Y, 2)` (t_4) is causally dependent on `R(X): 1` (t_3). This means that after P_4 has observed 2 at t_7, P_4 now must observe `1` at t_8 since it observed an operation that was causally dependent on this operation.

This schedule is not sequentially consistent. P_5 observes `0` at t_6 while observing `1` at t_5, meaning it observed the ordering as `W(Z,1)` precedes `W(X, 1)`. Later, P_4 observes `1` at t_8 and `0` at t_9, meaning it observed the ordering as `W(Z, 1)` precedes `W(X, 1)`. This is a violation of sequential consistency, but since `W(Z, 1)` and `W(X, 1)` are not causally dependent they can be observed in any order under causal consistency.

The causal consistency level can be implemented in a system while maintaining the availability and partition tolerance guarantees as defined by the CAP theorem. This is not the case for stricter consistency levels. This makes causal consistency a popular choice for systems.

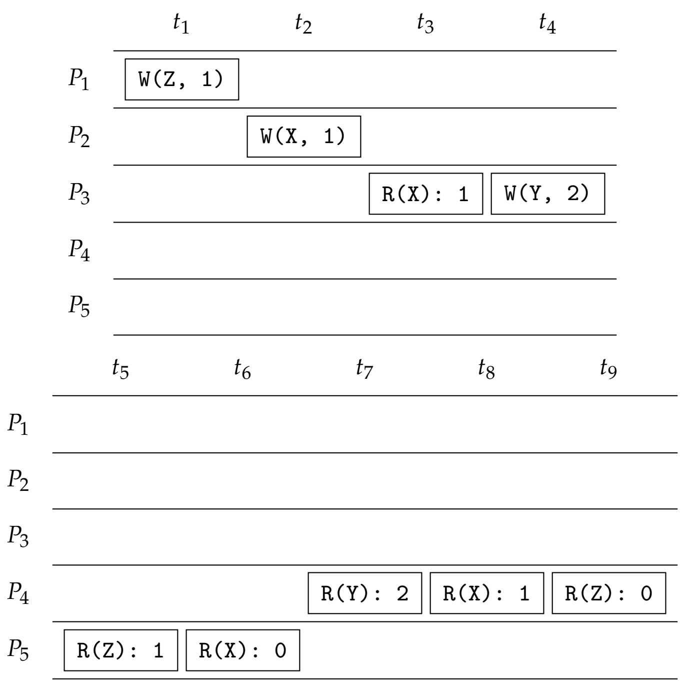


FIGURE 2.4: Example schedule that is causally consistent

Eventual consistency

Eventual consistency is a very weak consistency level. It provides no guarantee on order in any way. The only guarantee it does provide is that when writes stop, all processes will agree on each data item's value at some point in time.

Besides the five consistency levels introduced here, other consistency levels exist based on the data visible to any client or process during its current session (Terry et al., 1994). These are sometimes referred to as client-centric consistency levels and include *monotonic reads* and *read-your-writes*. These models are not considered in this thesis.

Data loss and CRDTs

A system with a weak consistency level is vulnerable to data loss. Data loss may happen when two replicas of the data diverge. The basic consistency guarantee of *eventual consistency*, to which all other consistency levels must also adhere, requires that these diverged replicas converge at some point. So, diverged replicas must be merged at some point. The simplest and a commonly used approach to merge replicas is *last-writer wins*. When using last-writer wins, the system keeps one of the replica's state and discards the other versions. What replica to keep is usually based on time, even though time in distributed systems is not accurate. Discarding replicas can cause data loss if a discarded replica processed an operation that was not processed by the retained replica.

A solution to this data loss are conflict-free replicated data types (CRDTs) (Shapiro et al., 2011). CRDTs are data types that can be merged using a function that is commutative, associative, and idempotent. This means that diverged replicas can always be merged, independent of any order, leading to the same results and without losing any data. A simple example of a CRDT is a set. Merging two sets always leads to the same results, independent of the order and even when performing the operation multiple times. CRDTs can be used in weak consistency levels for higher performance while still guaranteeing no data loss. These replicated data structures may still be in diverged states during operation, so there is no other guarantee for consistency.

2.1.2 Isolation levels

As described before, consistency levels apply to single operations in distributed systems, whereas isolation levels apply to *transactions* in both non-distributed systems and distributed systems. Transactions are sets of operations. A simple example of a transaction is the transfer of x credit from one user to another. This would result in the following set of operations: (1) read balance i of user A to ensure $i > x$, (2) write $i - x$ back to the balance of user A, (3) read balance j of user B and (4) write $j + x$ back to the balance of user B. Problems may be caused when these operations interleave with other operations. For example, if in between (3) and (4) another process subtracts all credit from the balance of user B, the system will be in an incorrect state after $j + x$ is written at (4).

The ACID properties define properties transactions (sets of operations) should comply with to prevent incorrect state and maintain data correctness (Haerder and Reuter, 1983). These properties are described in table 2.2. The completion of a transaction may result in a *committed* transaction, meaning the transaction was successful, or an *aborted* transaction, meaning the transaction failed.

Property	**Description**
Atomicity	Atomicity means that the set of operations in a transaction should be treated as a single atomic unit. This atomic unit should fail (*abort*) or succeed (*commit*) as a whole. For example, in case of a failed operation because of a server crash, all other operations part of the transaction should also fail (leaving the system unchanged).
Consistency	Consistency in ACID refers to a consistent state of the database. Depending on the application, rules may exist to which the system must comply. A transaction should bring the system from one valid state to another valid state.
Isolation	Isolation means that even though transactions may be executed concurrently, they must result in the same database state as if they would be executed sequentially.
Durability	Durability means that once a transaction has been completed, the resulting state should be *durable*. In other words, the resulting database state must always be recoverable, even in situations like system crashes.

TABLE 2.2: Descriptions of properties for ACID transactions

Especially guaranteeing isolation turned out to be a challenging problem to solve, particularly in the context of the trade-off against performance. Isolation levels define subtle differences in compliance with the isolation property, similar to how consistency levels apply to the consistency guarantee of the CAP theorem.

The strictest isolation level is serializability. Serializability means the result of any concurrently performed transactions is exactly the same as if these transactions were performed in some sequential order by a single process. The other isolation levels are best described in terms of anomalies that may occur when a system implements that isolation level. Table 2.3 shows four different isolation levels as defined by an SQL standard (Berenson et al., 1995). The isolation levels are shown in decreasing order of strength and include their anomalies. These different anomalies are described below.

Isolation level	Dirty read	Non-repeatable read	Phantom read
Serializable	+	+	+
Repeatable read	+	+	-
Read committed	+	-	-
Read uncommitted	-	-	-

TABLE 2.3: Isolation levels presented with their exposure to anomalies (-)

Dirty read

A *dirty read* refers to reading a value written as part of a running transaction that may still abort. The read value may therefore be incorrect or *dirty*. If this occurs, there is no isolation between transactions, and reading an incorrect value may have many consequences depending on the application. The read committed and higher isolation levels are not vulnerable to this anomaly.

An example of a solution to this is using an exclusive lock. An exclusive lock prevents other operations from reading and updating a written value until the running transaction completes. Instead of directly reading the value, the operations may queue to be processed when the lock is released (when the running transaction completes).

Non-repeatable read

A *non-repeatable read* refers to reading two different values for a data item as part of the same transaction. A non-repeatable read occurs when another operation (or transaction) has updated a value between the two reads performed by a transaction. Depending on what decisions are made based on the read data, this may lead to an inconsistent state. The repeatable read and stricter isolation levels are not vulnerable to this anomaly.

An example of a solution to this is using shared locks. A shared lock prevents other operations (or transactions) from updating a read value until the lock is released (after the running transactions complete). Other transactions may still read this value.

Phantom read

Phantom reads are very similar to non-repeatable reads, but they apply to aggregated data instead of single-key data. For example, when both the minimum and the average value of a set of data items are queried. In this case, simply locking the existing data items is not sufficient since new data items that are also part of the aggregated set may be added in between. The strictest isolation level, serializability, protects against this anomaly.

More subtleties and anomalies have been discovered since the introduction of these isolation levels, specifically to dissect the serializable level into more precise concepts (Attar, Bernstein, and Goodman, 1984). However, these are not meaningful for this thesis and are not described to avoid over-complication.

2.1.3 Relationship between consistency and isolation levels

After dissecting both consistency levels and isolation levels, it is noticeable that they are different; consistency levels consider how *operations* are observed by other processes in distributed systems, while isolation levels specify how *transactions* are run concurrently while providing isolation guarantees. Isolation levels apply to all systems, whereas consistency levels only apply to distributed systems.

However, it is also apparent that they are related when transactions are performed in distributed systems. If a transaction is performed in an eventually consistent system, it would be impossible to provide any guarantees as a client may observe the results in any order. So, consistency level guarantees are required to perform transactions that provide some isolation guarantees in a distributed setting. However, there are no guarantees about transactions in a strictly consistent system, meaning some isolation level is still required to perform transactions in strictly consistent distributed systems.

The combination of the consistency level and the isolation level is required to reason on a distributed system's data correctness. This has lead to many terms combining both a consistency level and an isolation level. An example of this is *strict serializability* or *strong one-copy serializability* (Bailis et al., 2013), which is the combination of linearizability (consistency level) and serializability (isolation level). This is considered a very strong data correctness model.

2.2 Distributed transaction protocols

Most modern cloud systems optimize performance over data correctness by implementing low consistency and isolation levels or providing no guarantees at all. This led to systems in which it is difficult to reason about correctness and that are hard to use. Besides simplifying these systems, implementing stronger correctness guarantees also enables the use of these systems by a wider variety of applications (Cattell, 2011).

This does not mean no options to implement data correctness exist. Consensus protocols like Paxos (Lamport, 1998) and Raft (Ongaro and Ousterhout, 2014)) can be used to agree on the global order of operations and achieve high consistency levels. However, these protocols are not directly applicable for transactions and isolation guarantees and are therefore not described in this thesis. Options to implement transactions and isolation levels have also been researched in traditional database systems for decades (Haerder and Reuter, 1983). This section introduces different relating concepts and techniques.

2.2.1 Two-phase locking

Two-phase locking is a concurrency control protocol that is used to implement the serializable or repeatable read isolation levels (Bernstein, Hadzilacos, and Goodman, 1987). Two-phase locking is not necessarily a distributed protocol but can be applied in both a non-distributed and a distributed setting. Two-phase locking consists of two phases, the growing phase where locks are obtained and the shrinking phase where locks are released. Two types of locks can be distinguished, a shared lock and an exclusive lock.

Shared lock

A shared lock is sometimes also referred to as a read lock. Once a transaction reads a data item, it obtains a shared lock on the data item. The lock is released when the transaction completes. This shared lock prevents other operations from writing to this data item. However, they may still read the data item. If another transaction reads the data item, this transaction will also obtain a shared lock on the data item. Only once all shared locks are released, a write operation may be performed on the data item.

Exclusive lock

An exclusive lock is sometimes also referred to as a write lock. Once a transaction performs a write operation on a data item, it obtains an exclusive lock. It may only do so when no shared or exclusive locks are active on the data item. This exclusive lock prevents both read and write operations. The write lock is released when the transaction completes.

The rule of two-phase locking is that a transaction may no longer obtain other locks once a lock is released. So, transactions acquire locks during their lifetime (growing phase) and release the locks when they either commit or abort (shrinking phase). Whenever a lock is released, no new locks may be acquired. This ensures that no other operations have unsafe access to data items in a running transaction, ensuring isolation.

Depending on the lock's scope, either the repeatable read or serializable isolation level can be implemented using locks. In the example of a single table, locks on individual rows are sufficient to ensure the repeatable read isolation level, but not necessarily the serializable isolation level as this also considers aggregates of rows. Another option would be to lock the entire table to ensure serializability. This implementation choice not only affects the isolation level but may also have significant effects on the performance. Two-phase locking is a form of pessimistic locking.

Deadlocks

A problem that occurs when using two-phase locking is deadlocks (Coffman, Elphick, and Shoshani, 1971). A deadlock can occur when concurrent transactions try to obtain locks on the same data items. The simplest example of a deadlock considers two transactions A and B. Both A and B require an exclusive lock on data items x and y. Transaction A first acquires an exclusive lock on data item x while transaction B acquires an exclusive lock on data item y. When transaction A tries to obtain a lock on data item y it can not do so and has to wait until transaction B completes. At the same time, transaction B tries to acquire a lock on data item x but can not do so and has to wait until transaction A completes. Both of these transactions can now no longer complete and the locks will never be released. This is a deadlock. A deadlock may also occur on a bigger scale involving more transaction and data items. Deadlocks may be ignored, prevented, or detected and then dealt with.

When a deadlock is ignored, a timeout must be set for a transaction to be aborted artificially and retried as otherwise deadlocks will continue to accumulate and the entire system will come to a halt. This approach may cause significant overhead on the system.

Wait-die and *wound-wait* approaches may be used to prevent deadlocks. When a transaction tries to acquire a lock on an already locked item, the wait-die approach checks both transactions' timestamp. When the new transaction's timestamp is lower than the timestamp of the transaction holding the current lock, the new transaction is allowed to wait. When the new transaction's timestamp is higher, the new transaction is killed. The wound-wait approach works the other way around. The new transaction is allowed to wait when its timestamp is higher than the timestamp of the transaction holding the current lock. When the new transaction's timestamp is lower, the transaction holding the current lock is killed. Both these approaches prevent deadlocks and are easy to implement. A downside of these approaches is that transactions that do not necessarily lead to deadlocks are also killed.

A wait-for graph may be constructed to detect deadlocks. In this graph, nodes are transactions, and directed edges represent transactions blocked by other transactions. Whenever a cycle in this graph exists, a deadlock exists. In the simple example shown above, there would be two nodes A and B that both have a directed edge to the other node resulting in a cycle. This is more complex to implement than wait-die or wound-wait, but does not unnecessarily kill transactions.

2.2.2 Two-phase commit

Two-phase commit is an atomic commit protocol for distributed systems (Gray, 1978). It coordinates a transaction across different instances and ensures the operations are atomic. To do this the two-phase commit protocol relies on a single *coordinator*. Instances that are involved in the atomic transaction are called *participants*. Two-phase commit has two phases; the *prepare* phase and the *commit* phase. In the prepare phase, (1) the coordinator sends *prepare* messages to all participants to make them aware of the upcoming operation they need to perform. Then, (2) the participants reply to the coordinator whether they are able to perform the operation. The coordinator collects the replies from all participants. The coordinator can now *abort* the transaction or *commit* the transaction in the *commit* phase. If all participants replied that they can perform the operation, the coordinator commits the transaction and (3) sends a *commit* message to all participants. (4) The participants then perform the operation. When any single participant replied they can not perform the operation or failed to reply, the coordinator aborts the transaction. To abort the transaction, (3) the coordinator sends *abort* messages to all participants and (4) the participants do not perform the operation.

The two-phase commit protocol is originally solely used for atomicity. However, it can be extended with the two-phase locking protocol to implement serializability in a distributed setting. To do this, participants lock data items involved in the transaction between the *prepare* message and the *commit* or *abort* message. So, during the *prepare* phase, locks across different instances are acquired (growing phase), and during the *commit* phase, these locks are released (shrinking phase). Even though two-phase commit can be used in this way to implement strong isolation levels, its primary function is to ensure the atomicity of a transaction across different instances and it should not be confused with two-phase locking. The combination of two-phase commit and two-phase locking is also vulnerable to deadlocks. Deadlocks in a distributed setting are even harder to detect, since they require extra network communication to detect and resolve them.

2.2.3 Sagas

Two-phase commit can be quite costly, which is why the sagas protocol (or pattern) was introduced (Garcia-Molina and Salem, 1987). Sagas is also an atomic commit protocol for distributed systems. While initially introduced for long-lived transactions, the increased performance and more straightforward implementation of the sagas protocol has led to it being widely used across distributed applications and specifically in popular micro-service architectures. Sagas do not lock and, therefore, often perform better than two-phase commit. However, this also means that they do not provide any isolation guarantees and only provide a *read uncommitted* isolation level. Sagas is also often used to describe the orchestration of a workflow consisting of operations on various distributed applications and not necessarily always as a transaction in the strict sense.

Sagas relies on *compensating* operations. When using sagas, operations that are part of a transaction are performed on different instances. Whenever any operation fails, sagas ensure that compensating operations are performed for operations that are part of the same transaction, and that succeeded. Compensating operations, in this case, undo any effect that the original operation had. When using sagas, there is no locking between the original and the compensation operations. This means that other operations may be performed on the data item in between the original operation and the compensating operation.

Sagas are usually applied in event-driven systems. This means an operation is often an incremental change to the state (for example *addCredit(*`5`*)*) rather than a complete overwrite (for example *setCredit(*`10`*)*). When implementing sagas, compensating operations should always succeed. Otherwise, the system will not return to a consistent state (consistent as by the ACID definition). There are two popular approaches to implement sagas.

Choreography

When using choreography to implement sagas, there is no orchestrator, but instances that are part of the transaction communicate to complete the transaction. To do this, a single instance that successfully completes an operation calls the next instance to perform an operation. This is chained forward to include operations over multiple instances. When an instance fails to perform its operation, they call the previous instance to perform its compensating operation. This is then chained backwards to all previous instances to perform their compensating operations.

A benefit of choreography is that it does not depend on a single instance. However, a downside is that all instances need to be aware of the transaction and that the transaction details are spread across different instances. Instances may have separate code bases (for example, for micro-services), meaning that a single transaction's details are spread across several code bases.

Orchestration

Orchestration does rely on a single instance; the orchestrator. This orchestrator is aware of all instances, operations, and compensating operations of the transaction. The orchestrator messages instances that participate in the transaction with the operation they should perform. It collects the replies of the participants. If no operations failed, the orchestrator is done. However, if any operation failed, the orchestrator should message all instances that successfully performed an operation to perform a compensating operation. This shares some resemblance with two-phase commit where there is also a single coordinator, however, sagas do not do any locking, and two-phase commit does not require compensating operations.

This approach does depend on a single instance. A benefit of orchestration is that all logic of the transaction is centralized in the orchestrator, and participants only have to be aware of their own operations.

2.2.4 Calvin

A more recent approach to implement transactions with strong isolation *and* consistency guarantees is Calvin (Thomson et al., 2012). For example, FaunaDB[1] implements a protocol based on this to implement distributed transactions within their system (Freels, 2018). Calvin requires *replicated and partitioned* storage systems to agree on a single deterministic order of processing transactions before processing them. It does this by introducing an initial sequencing component in all instances. This sequencing layer accepts incoming transactions and decides on a single deterministic global order with all other instances using the Paxos consensus algorithm. After this order has been decided, each instance performs the transactions in this order. Because all transactions are performed in the same order, every replica will arrive at the same results. This additional sequencing step does increase latency. Another limitation of this approach is that the entire read/write set of the transaction must be known ahead of time to ensure a correct deterministic order (Ren, Thomson, and Abadi, 2014).

Google Spanner[2] has implemented a similar approach where agreement on the order is required before processing. Spanner uses a *TrueTime* mechanism to ensure the ordering of transactions across machines (Corbett et al., 2013). However, *TrueTime* requires atomic clocks and GPS, often not available and not feasible in many cases.

Both Calvin and Google Spanner are examples of approaches that optimize data correctness over performance by design.

2.3 Processing guarantees

Besides data correctness, communication is also a considerable problem in distributed systems, because the network is unreliable (Bailis and Kingsbury, 2014). When computer A sends a message to computer B, two problems commonly occur. Firstly, the message of computer A may never arrive at computer B. Secondly, the response of computer B may never arrive at computer A. So when computer A does not receive a response from computer B, two scenarios are possible; either the message from A to B did not arrive or the response message from B to A did not arrive. In this case computer A does not know whether computer B processed the message

[1]https://fauna.com/
[2]https://cloud.google.com/spanner

and does not know whether the message should be send again. There are different solutions to this problem along with different levels of processing guarantees. Processing guarantees reflect how often the receiver has actively processed the message and how often the message may have affected its state. Similar to the data correctness levels, increasing the processing guarantees comes at performance costs. These levels are described below according to the example communication between computer *A* and computer *B* sketched above.

At-most-once processing

At-most-once processing means a message may either never be processed or processed once. In this case, computer *A* sends the message to computer *B* once and does not care about the response. This means the message may either arrive once or not at all. The state of the receiver may therefore either reflect the message or not reflect the message.

At-least-once processing

At-least-once processing means a message may either be processed once or multiple times. To achieve this, computer *A* keeps re-sending the message to computer *B* until it receives a response. The message may arrive multiple times at computer *B*, resulting in at-least-once processing. At-least-once processing guarantees the state of the receiver is updated by the message once or multiple times.

Exactly-once processing

Exactly-once processing means the state of the receiver will always be updated exactly-once by the message. This can be implemented by attaching a unique incremental identifier to the message. Then, the same approach as at-least-once processing can be used on the side of the sender. However, because of the unique incremental identifier, the receiver can distinguish messages it has already received before and ensure it will only process a message once.

Exactly-once processing can also be achieved by sending idempotent messages while using at-least-once processing. Idempotent messages can be applied multiple times without changing the result. An example of this is adding an item to a set. If the item was already added to the set, the resulting set would not change when performing the operation again. In this case, idempotent messages should be implemented at the application level rather than the system ensuring exactly-once processing itself.

2.4 Fault tolerance

Fault tolerance is a term that describes the property of a system that can cope with failures without losing data. An example of a failure is a computer crashing. Failures are a common problem in large distributed systems, so fault tolerance is an important property. This section describes two common approach to achieve fault tolerance.

2.4.1 Active Replication

An approach to achieve a fault-tolerant system is replication. Replication is commonly implemented in traditional relational databases and means a copy of the database is always running in the background as a backup. This is called primary-secondary replication. In case of a failure in the database's primary copy, the secondary copy can directly step in and take over. This does require the primary and secondary copy to maintain the same state, meaning that any update to the database now becomes a distributed transaction consisting of an update to the primary and the secondary instance. This can be achieved with a two-phase commit, but this decreases performance significantly.

2.4.2 Rollback recovery

Another solution that can be used to achieve fault tolerance is rollback recovery (Carbone et al., 2017). Rollback recovery is commonly implemented in stream processing systems. A system's input must be persisted somewhere to use rollback recovery, for example, in Apache Kafka. After a failure, the system can always reach the same state by processing the input again. However, this may take very long in long-running applications, leading to the introduction of checkpointing. Checkpointing mechanisms periodically store a snapshot of the system in a persistent storage system, such as a distributed file system. Whenever a failure occurs, the system can restart from this checkpoint and only process the input that arrived after the checkpoint to reach the same state it had before.

2.5 Stateful stream processing

Stateful stream processing systems are an abstraction that provides some guarantees out-of-the-box. Streams are data in motion. This data can be read, processed, and forwarded by stream processors. Stateful stream processors keep aggregate state based on the data they have processed. Stream processors can read and process data produced by various data sources, such as message queues like Kafka, relational databases (using change data capture), or direct HTTP events. Stream processors may output processed data to various sinks, such as Kafka, relational databases, or document stores like mongoDB[3]. A well-known stateful stream processing system is Apache Flink.

2.5.1 Fault-tolerance and delivery guarantees

Stream processing systems commonly use rollback recovery and checkpointing to provide fault-tolerance guarantees. On top of this, stream processors may also extend processing guarantees to delivery guarantees. These delivery guarantees are not strictly limited to stream processing systems but are most common in stream processing systems. Delivery guarantees require a durable source (or input messages). Kafka is an example of a durable source. Delivery guarantees ensure the entire system provides at-least-once or exactly-once semantics from every input message at the source to every output in the sinks, including any state updates to stateful streaming operators in between.

[3]https://mongodb.com/

2.6 Benchmarking

The recent developments in cloud technology and the increasing usage of Web applications have led to an extreme variety of data-related use cases. These varying use cases have resulted in many diverse systems, ranging from document stores such as mongoDB to distributed file systems like Amazon S3[4]. Each of these systems offers its own trade-offs and may be beneficial for some use cases and perform less well for other use cases.

2.6.1 YCSB

The large variety of use cases and systems makes it difficult to compare systems using a standardized benchmark. The *Yahoo! Cloud Serving Benchmark* (YCSB) is an effort to provide a flexible yet standardized benchmark definition for cloud systems (Cooper et al., 2010). YCSB is extremely straightforward, and therefore its results are easily interpretable and explainable. Since many novel systems do not provide strong data correctness guarantees and specifically no transactions (Cattell, 2011), YCSB can not be used to evaluate performance on transaction workloads.

2.6.2 TPC-C

On the other hand, the TPC-C benchmark has existed since 1992 (Raab, 1993). TPC-C has been used to benchmark fully-fledged relational database management systems, including transactions. This was possible since these systems' requirements were well-defined, and most systems included the same or similar features (as use cases were also less diverse). However, TPC-C requires the system to have many additional features that are currently not commonly supported in modern cloud systems. Because of this, TPC-C is not fit to benchmark these systems.

2.6.3 DeathStarBench

DeathStarBench is a more recently introduced benchmark. DeathStarBench introduces five applications (including a social media network and a banking system) and their associated micro-services designs. DeathStarBench aims to evaluate the performance of micro-services-like architectures throughout the technology stack, from application framework to network overhead to hardware design. However, the micro-services introduced by DeathStarBench are complex, and it is not apparent how to reason about the system's behavior based on the results.

Both TPC-C and DeathStarBench are complex and not easily interpretable, and YCSB is extremely simple. This calls for a benchmark that covers the ground in-between. It should require fewer features than TPC-C and be easier to interpret than DeathStarBench, but still evaluate the performance of transactions. Some attempts were made to introduce a benchmark like this (Dey et al., 2014), but none are widely adopted. This lack of adoption may be because many systems still lack support for transactions.

[4]https://aws.amazon.com/aws/s3/

Chapter 3

Serverless computing

The chapter presents related work on serverless computing. Firstly, the basic definitions in cloud computing are given. Secondly, the concept of serverless computing is presented. Next, a novel serverless model, stateful functions-as-a-service, is discussed, including implementations of this model.

3.1 Cloud computing

Cloud computing has gained much popularity recently and is a promising model for computation. Cloud computing's central idea is to outsource operational IT concerns to either public cloud providers (*public cloud*) or a specialized department within an organization (*private cloud*). The cloud provider provides services to its users. According to "A Berkeley View on Cloud Computing" (Armbrust et al., 2009) this has the following hardware-related advantages:

1. *The illusion of infinite computing resources available on demand*. This means users no longer have to plan ahead and provision enough resources for their upcoming load because they can quickly get these resources on demand.
2. *The elimination of an up-front commitment by Cloud users*. This is specifically related to public clouds. Users can rent available compute resources and services by the hour and do not have to make up-front investments in IT infrastructure.
3. *The ability to pay for use of computing on a short-term basis as needed*. This allows for a lot of flexibility for cloud users. They can use resources while requiring them and simply release them when they are no longer needed.

These benefits are beneficial for many organizations. The services that cloud providers usually provide can be categorized into the following three categories.

Infrastructure-as-a-Service (IaaS)

Infrastructure-as-a-Service (IaaS) is the simplest form of cloud services. It provides users with simple compute resources on-demand. These resources usually include CPU, memory and storage. The most well-known example of this is AWS EC2 instances[1].

[1] https://aws.amazon.com/ec2/

Platform-as-a-Service (PaaS)

Platform-as-a-Service (PaaS) is a more complex form of cloud services. The cloud provider hosts some software on their infrastructure and allows users to interact with and use it without actually maintaining the software itself. Examples of this include a hosted and maintained version of Kafka[2], a hosted version of Kubernetes[3], or even fully managed relational database systems[4]. PaaS may also be scaled up and down by the cloud provider automatically based on the cloud user's load. Because of this, PaaS alleviates many operational concerns for the cloud user. They no longer have to provision IaaS instances and deploy software on top of it. Besides this, cloud providers often provide availability and reliability guarantees for the services they offer.

Software-as-a-Service (SaaS)

Software-as-a-Service (SaaS) are complete hosted software deployments made available to users and are directly usable. Examples of this are Salesforce [5] and Dropbox[6]. SaaS may directly solve and optimize processes for cloud users.

The examples mostly referred to AWS, but many other big cloud providers offer similar services, such as Microsoft Azure or Google Cloud Platform.

3.2 Serverless computing

The cloud has already simplified many operational concerns; however, not all promises have yet been fulfilled. Using the cloud and realizing its full potential requires thorough knowledge of provided services, and orchestrating and configuring cloud services in a large system is not easy. The *serverless computing* model aims to solve this. A Berkeley View on Serverless Computing states the following about serverless computing: "It provides an interface that greatly simplifies cloud programming, and represents and evolution that parallels the transaction from assembly language to high-level programming languages" (Jonas et al., 2019).

Serverless computing models should provide the user with the ability to deploy simple code and automatically leverage all benefits the cloud has to offer. It should be able to auto-scale deployments of the user code, even to zero, and the user should only pay for the actual execution of the supplied code. This model can be described as a PaaS model that is extremely simple to use. Currently, several services are available that are described as serverless.

3.2.1 Function-as-a-Service

The most well-known serverless model is Function-as-a-Service (FaaS). It has become increasingly popular over the last years (Jonas et al., 2019; Baldini et al., 2017a; Hellerstein et al., 2018; Eyk et al., 2017; Baldini et al., 2017b; Akhter, Fragkoulis, and Katsifodimos, 2019). A popular commercial implementation of FaaS is AWS

[2]https://aws.amazon.com/msk/
[3]https://aws.amazon.com/eks/
[4]https://aws.amazon.com/rds/
[5]https://www.salesforce.com/
[6]https://www.dropbox.com/

Lambda[7], and OpenLambda[8] (Hendrickson et al., 2016) is a popular open-source implementation. FaaS allows cloud users to upload code in the form of a simple function to the cloud. The cloud user can configure some triggers and input to invoke the function. The cloud provider then deploys an ephemeral instance of that function, takes the input, and executes the function. The functions are usually stateless so that they can be deployed anywhere and executed entirely in parallel, making it easy for the cloud provider to auto-scale the function. Cloud users can now solely focus on the application logic, knowing that the cloud provider takes care of all operational concerns.

Limitations of FaaS

Even though the FaaS model is very popular and a great step towards serverless systems, there are some limitations. These limitations are described in a paper by Hellerstein et al. (2018). The following four limitations are mentioned:

1. **Limited Lifetimes.** Current available FaaS services only allow very short-lived function executions.

2. **I/O bottlenecks.** Available FaaS services have no access to state. This means network communication has to be used to fetch data and to store state updates. The network that functions can use is often slow.

3. **Communication Through Slow Storage.** Functions are not directly accessible through network. Functions are only accessible through intermediate services or gateways. When a function calls another function, communication is done via a slow storage system rather than a direct network call.

4. **No Specialised Hardware.** Current FaaS offerings do not allow cloud users to specify specific hardware requirements for functions and workloads. An example of this is when access to a GPU would significantly increase the performance of training machine learning models.

Problems one and four are not inherent problems with the computation model but have more to do with configuration. However, problems two and three seem not easily solvable.

Another problem with currently available FaaS services is that they are not fault-tolerant or reliable. Whenever an execution of a function fails, either the function stops, and the execution will be incomplete, or the function will be executed again with the same input. When the function fails, it could have already written some data to an external datastore but not yet performed all its write operations. Stopping the execution will leave the datastore with only half of the writes. Executing the function again will lead to double results unless the operations are idempotent (currently recommended by the cloud providers). Both these scenarios are undesirable.

SAND is a more recent approach to implement FaaS (Akkus et al., 2018). SAND acknowledges the problems described earlier and specifically aims to improve function communication related to problem three. SAND deploys functions that are contained in the same *application* in the same container. This greatly improves function

[7] `https://aws.amazon.com/lambda/`

[8] `https://github.com/open-lambda/open-lambda`

communication within the application because (1) no new container has to be deployed for the called function, and (2) the message to the called function can be sent locally on the same instance, decreasing latency. This greatly improves the performance of workloads that require function communication but does not solve the data problem described as problem 2.

3.2.2 Serverless data storage

Besides serverless computation models, cloud providers also offer data storage with serverless properties. Two storage models are commonly offered by cloud providers. Services like AWS DynamoDB or Azure Cosmos DB provide auto-scaling *key-value storage* and services like AWS S3 offer auto-scaling *file storage*. Both these key-value storage and file storage systems have a payment model based on usage. These models are extremely flexible, but do not provide the user with a lot of guarantees regarding transactions or even consistency in some cases.

3.3 Stateful Function-as-a-Service

A new paradigm called Stateful Function-as-a-Service aims to find solutions to both problems 2 and 3 of FaaS services as described above. Stateful Function-as-a-Service offers the same possibilities as FaaS but also allows the function access to managed state. This access to state addresses problem 2. Besides that, they simplify invoking other functions improving communication between functions, and address problem 3.

This chapter discusses four existing open-source SFaaS systems, namely StateFun, Cloudstate, Cloudburst, and Beldi. The differences between these approaches are mostly caused directly by the choice of architecture and the enabling technology for the systems. This chapter introduces these architectures (3.3.1), discusses their programming models (3.3.2), and consistency guarantees (3.3.3).

3.3.1 Architecture

Each of the four mentioned SFaaS systems is based on different technologies. This section describes the architectures of the different systems.

Cloudstate

Figure 3.1 shows *Cloudstate*'s architecture. Cloudstate is based on Akka[10]. It builds on top of the actor model (Hewitt, Bishop, and Steiger, 1973) and specifically on *Akka Cluster* and *Akka Persistence* (or stateful actors). Cloudstate runs in kubernetes[11].

Cloudstate requires a sidecar to be deployed in the same pod as a user-defined function, as shown in figure 3.1. These sidecars form an Akka cluster, and each sidecar maintains some stateful actors based on the user-defined function deployed with it. A user-defined function may have multiple instances that each encapsulate their own state. Function instances are uniquely identified by an ID. All function

[10]https://akka.io/
[11]https://kubernetes.io/
[11]https://cloudstate.io/docs/cloudstate-solution.html

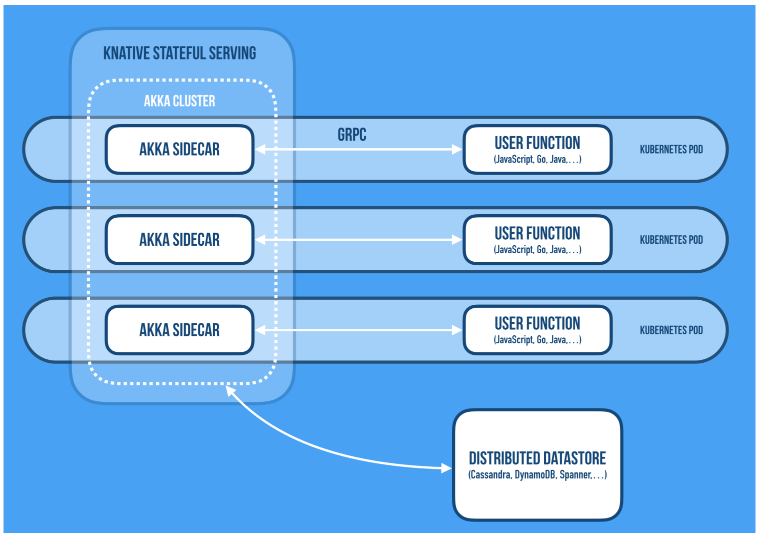


FIGURE 3.1: Cloudstate architecture[9]

instances are represented by stateful actors. If multiple instances of the same user-defined function are deployed, the function instances may be partitioned or replicated across the deployed instances. Cloudstate provides two types of stateful functions; event-sourced and CRDT. In the case of event-sourced functions, only a single stateful actor per function instance exists, and these are partitioned across multiple instances of the user-defined function. For CRDT functions, multiple stateful actors per function instance may exist, so these are both partitioned and replicated across multiple instances of the user-defined function.

The stateful actors communicate with user-defined functions over gRPC[12]. The user-defined functions get called with an incoming message and the state of the function instance the message is addressed to (uniquely identified by the ID). The results of the function invocation are then passed back to the sidecar to handle. The sidecar is responsible for the state management and the communication between functions. Because the state management is done in the same pod as the function executes, compute and state are co-located in Cloudstate, allowing for low latency.

The stateful functions (or stateful actors representing them) within the cluster can communicate between themselves to deliver messages from one function to another. Messages between function instances in Cloudstate have exactly-once guarantees.

To achieve fault-tolerance Cloudstate relies on a separate distributed datastore that persists all the state of the stateful actors and allows recovery in case of a failure. The persistence of Cloudstate is based on rollback recovery and a checkpointing mechanism is implemented to prevent having to process all events again to get to the current state.

StateFun

StateFun is based on Apache Flink[14], a stateful stream processing system. Similar to Cloudstate, multiple instances of each user-defined function can exist, each encapsulating its own state. However, in StateFun, these are represented as stateful

[12]https://grpc.io/
[14]https://flink.apache.org/

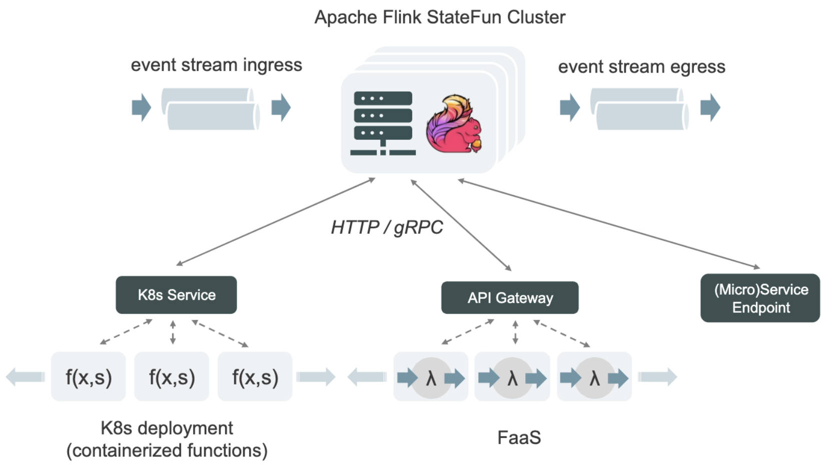


FIGURE 3.2: StateFun architecture[13]

operators in a streaming graph instead of stateful actors. Figure 3.2 shows a sketch of StateFun's architecture.

StateFun supports both internal stateful functions and external stateful functions. Internal stateful functions run in the same JVM as the stream processor (Flink) itself and have direct access to state. This is simply an abstraction on top of a stateful operators. External stateful functions communicate with an internal stateful operator through a HTTP interface similar to the gRPC interface used for Cloudstate. The external functions are called with the message and the state for the addressed function instance. This allows the external functions to be entirely stateless themselves. External functions can be deployed in the same pod as StateFun workers similar to Cloudstate for co-located compute and state or elsewhere for example in AWS Lambda allowing for separate scaling of compute and state. StateFun supports both internal stateful functions and external stateful functions. Internal stateful functions (called embedded functions) run in the same JVM as the stream processor (Flink) and directly access the state. These embedded functions are simply an abstraction over stateful operators. External functions communicate with an internal stateful operator through an HTTP interface similar to the gRPC interface used for Cloudstate. The external functions are called with the message and the state for the addressed function instance, allowing the external functions to be entirely stateless. External functions can be deployed in the same pod as StateFun workers (similar to Cloudstate) for co-located compute and state or elsewhere, for example, in AWS Lambda, allowing for separate scaling of compute and state.

To allow for dynamic communication between functions, StateFun implements a slight change in the stream processing system to support cyclic graphs, rather than only supporting static DAGs.

StateFun directly inherits Flink's fault-tolerance and exactly-once implementation. The fault-tolerance is based on rollback recovery and checkpointing. The exactly-once semantics require an atomic commit to the data sources (ingress in StateFun terminology) and sinks (egresses in StateFun terminology). This approach requires durable input such as Apache Kafka and a distributed file system to persist the system's checkpoints.

[14]https://ci.apache.org/projects/flink/flink-statefun-docs-release-2.2/concepts/distributed_architecture.html

Cloudburst

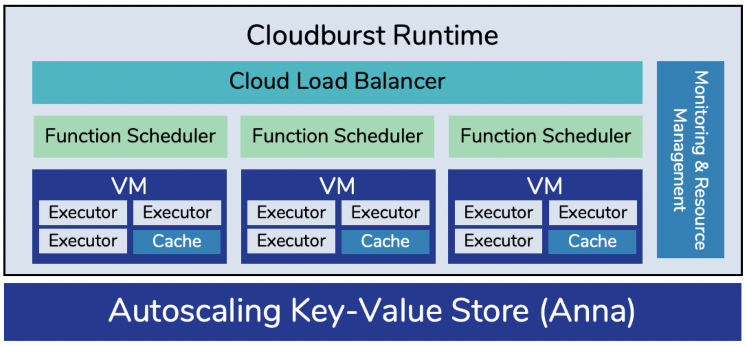


FIGURE 3.3: Cloudburst architecture (Sreekanti et al., 2020)

Cloudburst (Sreekanti et al., 2020) takes a different approach. Figure 3.3 shows Cloudburst's architecture. Cloudburst is based on Anna (Wu et al., 2018), an auto-scaling distributed key-value store. Together Anna and Cloudburst are part of the Hydro Project[15], a research project of the RISE Lab at UC Berkeley. Functions are executed in an entirely separate cloud runtime from Anna but have access to Anna's mutable shared state. The state in Anna is also cached in the cloud runtime instances for low-latency access.

In this architecture, functions have arbitrary access to any state in Anna by key, making it very different from Cloudstate and StateFun. The system tries to execute a function on an instance that has the function's required state cached. Functions can be called synchronously or asynchronously from within other functions. Cloudburst allows users to compose functions by combining arbitrary functions and representing them as DAGs. Then Cloudburst passes the results of a function along the DAG edges towards the next function in the composition.

Beldi

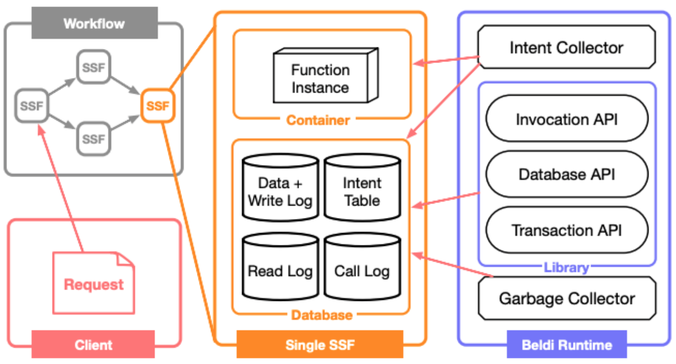


FIGURE 3.4: Beldi architecture (Zhang et al., 2020)

Beldi (Zhang et al., 2020) is built on top of currently available serverless services provided by cloud providers. It uses Function-as-a-Service services and serverless key-value stores to implement its stateful Function-as-a-Service model. Beldi aims to make FaaS workflows fault-tolerant with exactly-once guarantees and allow transactions for FaaS workflows. Beldi relies on a serverless key-value store that provides

[15]https://hydro-project.github.io/

linearizability and atomic updates (i.e., transactions) on the scope of a single key. Serverless key-value stores like AWS DynamoDB and Azure Cosmos DB provide these requirements.

Beldi introduces a runtime that itself runs on FaaS services and uses serverless key-value stores for internal state and bookkeeping. Figure 3.4 shows the architecture. The application developer can invoke other functions and access state in the key-value store through the Beldi runtime. Therefore, the Beldi runtime has complete control over how the data is stored, what metadata is kept, and when to invoke functions. This runtime keeps logs for all of the user's functions in the key-value store. These logs allow Beldi to re-invoke functions that failed initially. Besides keeping logs, the Beldi runtime also appends metadata about recent writes and the status of any lock to all data stored by the user in the key-value store. Because they append this meta-data to the same key as the actual data, Beldi can leverage the key-value store's atomic update feature to commit both state updates and bookkeeping status atomically. Beldi supports both fault-tolerance (exactly-once semantics) and transactions using serverless services based on the logs and the metadata appended to the data items.

To prevent the logs and metadata from growing indefinitely, Beldi implements a garbage collector that cleans up the logs and metadata periodically.

3.3.2 Programming model

The most significant difference in programming models among these systems is the access to state. Both StateFun and Cloudstate encapsulate state within a specific function instance (uniquely identified by a user-defined function and an ID). In contrast, Cloudburst allows any function access to any state stored in Anna, even dynamically. Beldi allows both models; functions can keep their own state, but some state can be accessed by arbitrary functions.

This section compares the programming models for these systems.

Cloudstate

Cloudstate has a quite bloated programming model. It requires a specific definition of a gRPC contract in a separate Protobuf file. The complete code required is very explicit and lengthy. Figure 3.5 shows a sketch of the Python code for a simple event-sourced shopping cart function. The full code is included in appendix A. The state kept by cloudstate for each function instance is defined in the class at line 2. The function at line 4 initializes the state the first time the function is run for a particular ID. The function defined in line 10 is used to serialize the data to store as part of a snapshot, and the function defined in line 16 is used to recover the state from the snapshot.

Cloudstate uses events for persistence. The processing of an incoming message and the event are separated. In figure 3.5, the function to process the command is defined in line 30. This function does not directly change any state but emits an event at line 36. This event is then handled by the function defined at line 20. The event is persisted, and the event handler function can be used to recover the state based on earlier persisted events in case of a failure.

StateFun

[16] https://pypi.org/project/cloudstate/

```python
@dataclass
class ShoppingCartState:
    entity_id: str
    cart: MutableMapping[str, LineItem] = field(default_factory=dict)

def init(entity_id: str) -> ShoppingCartState:
    return ShoppingCartState(entity_id)

@entity.snapshot()
def snapshot(state: ShoppingCartState):
    cart = DomainCart()
    cart.items = [to_domain_line_item(item) for item in state.cart.values()]
    return cart

@entity.snapshot_handler()
def handle_snapshot(state: ShoppingCartState, domain_cart: DomainCart):
    state.cart = {domain_item.productId: to_line_item(domain_item) for domain_item in domain_cart.items}

@entity.event_handler(ItemAdded)
def item_added(state: ShoppingCartState, event: ItemAdded):
    cart = state.cart
    if event.item.productId in cart:
        item = cart[event.item.productId]
        item.quantity = item.quantity + event.item.quantity
    else:
        item = to_line_item(event.item)
        cart[item.product_id] = item

@entity.command_handler("AddItem")
def add_item(item: AddLineItem, ctx: EventSourcedCommandContext):
    if item.quantity <= 0:
        ctx.fail("Cannot add negative quantity of to item {}".format(item.productId))
    else:
        item_added_event = ItemAdded()
        item_added_event.item.CopyFrom(to_domain_line_item(item))
        ctx.emit(item_added_event)
```

FIGURE 3.5: Example of an event-sourced function in Cloudstate[16]

Similar to Cloudstate, StateFun encapsulates state within function instances. Figure 3.6 shows an example function in StateFun. StateFun's programming model is a lot simpler than Cloudstates. The users are in complete control of the data object and need to deserialize it from Protobuf messages themselves. All operations related to state management and function communication are wrapped in the context object that the user can access.

In the example, the state is deserialized in line 3. The state update is confirmed in line 9. In line 14, StateFun sends an egress message to Kafka.

Cloudburst

Cloudburst's model of access to arbitrary shared mutable state is flexible. Cloudburst also allows synchronously calling other functions and composing a function from multiple other functions as a DAG. Figure 3.7 shows an example using Cloudburst taken from the paper introducing Cloudburst (Sreekanti et al., 2020).

In the example, a function is registered in line 6. This function will execute in the Cloudburst runtime shown in figure 3.3. This function can be executed synchronously (line 8) or asynchronously (line 11). Any Cloudburst code has access to arbitrary state in Anna, as in lines 3 and 4.

[17]https://ci.apache.org/projects/flink/flink-statefun-docs-release-2.2/getting-started/python_walkthrough.html

```python
@functions.bind("example/greeter")
def greet(context, greet_request: GreetRequest):
    state = context.state('seen_count').unpack(SeenCount)
    if not state:
        state = SeenCount()
        state.seen = 1
    else:
        state.seen += 1
    context.state('seen_count').pack(state)

    response = compute_greeting(greet_request.name, state.seen)

    egress_message = kafka_egress_record(topic="greetings", key=greet_request.name, value=response)
    context.pack_and_send_egress("example/greets", egress_message)
```

FIGURE 3.6: Example of a function in StateFun[17]

```python
from cloudburst import *
cloud = CloudburstClient(cloudburst_addr, my_ip)
cloud.put("key", 2)
reference = CloudburstReference("key")
def sqfun(x): return x * x
sq = cloud.register(sqfun, name="square")

print("result: %d" % (sq(reference))
> result: 4

future = sq(3, store_in_kvs=True)
print("result: %d" % (future.get())
> result: 9
```

FIGURE 3.7: Example of a function in Cloudburst (Sreekanti et al., 2020)

Beldi

Beldi Library	Description
`read(k) → v`	Read operation
`write(k,v)`	Write operation
`condWrite(k, v, c) → T/F`	Write if c is true
`syncInvoke(s, params) → v`	Calls s and waits for answer
`asyncInvoke(s, params)`	Calls s without waiting
`lock()`	Acquire a lock
`unlock()`	Release a lock
`begin_tx()`	Begin a transaction
`end_tx()`	End a transaction

LISTING 3.1: Beldi's API for functions (Zhang et al., 2020)

Figure 3.1 shows the API of Beldi. These functions can be called from any serverless function. Functions can read state and write state back to the key-value store and synchronously and asynchronously call other functions. All of these operations are fault-tolerant and guarantee exactly-once semantics, unlike standard serverless services.

Beldi also introduces both a low-level API to implement transactions with locks and a high-level API where the application developer can begin and end a transaction. All operations between the `begin_txn` and `end_txn` function calls are included in the transaction, even across different function executions (when `syncInvoke` is used).

3.3.3 Data correctness guarantees

This last section describes the data correctness guarantees provided by the discussed systems.

StateFun

StateFun provides the most straightforward consistency guarantees. Firstly, it provides exactly-once (also more accurately referred to as effectively-once) processing guarantees for all messaging and state updates throughout the system, from ingresses (e.g., Kafka) to internal function invocations and egresses (e.g., Kafka) (Carbone et al., 2017; Carbone et al., 2015). In StateFun, only a single instance of each stateful operator representing a function instance exists, processing all messages; there is no replication (except for snapshots for fault-tolerance). This stateful operator does not process messages in parallel, resulting in isolation based on a single stateful operator (a single function instance or state associated with a single key or address). This isolation results in *linearizability* of all operations on the data and *serializability* for all transactions limited to a single-key. However, all invocations between functions are entirely asynchronous, and there exist no isolation guarantees for more complex operations consisting of multiple function invocations.

Cloudstate

Cloudstate supports two modes of consistency. Firstly, event-sourced entities provide the same guarantees as StateFun functions do. Event-sourced entities are partitioned, and exactly one instance of each entity is active at any given time. This instance processes messages sequentially, ensuring isolation. Secondly, Cloudstate supports CRDT (Conflict-free Replicated Data Types) (Shapiro et al., 2011) entities. CRDT entities are replicated across the Cloudstate cluster allowing for high availability. These replications may diverge, meaning this model does not ensure isolation, even for a single entity. This results in *eventual consistency*. However, CRDT entities are protected from data loss as they are limited to CRDTs. This does limit the data types that can be used in these entities, while on the other hand, event-sourced entities can encapsulate arbitrary data types. Cloudstate provides the same exactly-once semantics as StateFun for messaging throughout the system. Similar to StateFun, it does not guarantee any isolation guarantees across multiple invocations either.

Cloudburst

Cloudburst has a different model of data correctness. This model is based on the CALM theory (Hellerstein and Alvaro, 2019; Ameloot et al., 2015) proposed by the RISE Lab. The CALM theory defines a formal set of applications that can be run coordination-free. This means it does not support *sequential consistency* or higher consistency levels but can perform significantly better since coordination is never required in a critical path (only in background tasks). Cloudburst allows caching, as shown in figure 3.3, and therefore has many state replicas. Cloudburst only implements *eventual consistency* for single operations.

Cloudburst does support function compositions as DAGs that can provide more guarantees. Edges in these DAGs represent causal relationships and are used to implement *causal consistency* in Cloudburst. Besides this, the paper introducing Cloudburst (Sreekanti et al., 2020) claims the system provides the *repeatable read* consistency level. This claim is confusing because of the *repeatable read* isolation level. Cloudburst does not provide the *repeatable read* isolation level. The authors mean by *repeatable read* that within the execution of a single DAG, no non-repeatable read anomaly can occur. This is done by passing previously read or written values for keys along with the execution of the DAG and using these values as opposed to the latest available value.

Cloudburst has no notion of transactions at all and does not implement any isolation level. Since Cloudburst allows for arbitrary access to any state in the key-value store or even to cached state, concurrent DAGs may perform operations on the same state, causing the state to diverge across replicas. To merge diverged data, Cloudburst by default uses *last-writer wins* (LWW) for arbitrary data types. The timestamp is created coordination-free by concatenating the local system clock and the node's ID. Even though this does provide *eventual consistency* or *causal consistency*, this may result in data loss. Cloudburst and Anna can be used for CRDTs to merge diverged data without data loss, but this is not the default behavior.

Beldi

Beldi aims to provide fault-tolerance and transactions for serverless services. It achieves this by firstly ensuring exactly-once semantics for standard function operations such as invoking other functions or reading and writing to the key-value store. This is already a notable improvement over current serverless offerings as these usually depend on the developer implementing idempotent operations in serverless functions.

Beldi goes one step further to introduce transactions to serverless environments. The authors claim they provide opacity isolation level, which is similar to serializability for transactions. Supporting serializable transactions is a significant improvement over current serverless offerings and the other systems discussed in the section.

Chapter 4

Coordinator Functions Approach

This chapter introduces the concept of coordinator functions to implement transactions on top of Stateful Function-as-a-Service systems. The coordinator functions introduced can be used to implement both serializable transactions with two-phase locking and two-phase commit, and sagas workflows with compensating actions. Coordinator functions do assume the underlying Stateful Function-as-a-Service system already provides some guarantees (described in section 4.1). Coordinator functions are described in section 4.2.

4.1 Assumptions and Requirements

Out of the four discussed SFaaS systems, two have many similarities; StateFun and Cloudstate event-sourced entities. This thesis's coordinator functions apply to those two systems due to their shared characteristics. These characteristics are introduced below and are also the requirements for systems where coordinator functions apply.

Firstly, both systems encapsulate state within function instances. Encapsulating state within function instances is a common model for SFaaS; Azure Durable Functions[1] also uses it.

Secondly, opposed to Cloudburst and Cloudstate CRDT entities, StateFun and Cloudstate event-sourced entities have the following two fundamental characteristics that greatly simplify implementing transactions (Beldi is not discussed as it already supports transactions).

4.1.1 Exactly-once guarantees

StateFun and Cloudstate event-sourced entities provide exactly-once semantics and fault-tolerance. Because of these features, atomicity becomes easy to achieve; even when the system fails halfway through a transaction, the underlying fault-tolerant exactly-once semantics guarantee it finishes the transaction. These features also guarantee durability, as any data processing in the system is already durable. Besides this, exactly-once guarantees alleviate the need to implement complex network communication protocols for distributed transactions as the existing messaging can be used for reliable communication.

4.1.2 Linearizability

The invocations for a single function instance are linearizable. Function invocations both read the function state and possibly write a new value. The combination of these operations can be described as a single-key transaction. This transaction

[1] `https://docs.microsoft.com/en-us/azure/azure-functions/durable/`

should also be performed in isolation. Strictly, this means that the function instances should be both linearizable and serializable. However, since function invocations naturally encapsulate both a read and a possible write, the term linearizability is used. Linearizability is required as function invocations need reliable access to the correct state for transactions to be implemented. StateFun and Cloudstate event-sourced entities are linearizable; they have only a single replica of the state and invocations are handled in a first-in-first-out (FIFO) manner by a single process internally. Both systems also maintain this linearizable FIFO order while the remote stateless function executes function invocations.

These primitives provided by the underlying system make it easy to reason about the system's behavior and implement transactions. The only existing implementation of transactions in serverless, Beldi, relies on similar features for transactions; linearizability of the key-value store and Beldi itself implements fault-tolerant exactly-once guarantees before implementing transactions on top of that.

4.2 Coordinator Functions

As described, coordinator functions apply to SFaaS systems where function instances encapsulate state. This encapsulation provides benefits as different developer teams or even different organizations may independently work on different stateful functions, similar to micro-services. However, encapsulating the state in function instances make it complex to access state in multiple function instances to, for example, aggregate state or perform a multi-key transaction. Achieving these multi-function instance operations now requires complicated logic and messaging between function instances, especially since Cloudstate and StateFun only support calling functions asynchronously (and not synchronously).

Azure Durable Functions provides a similar state encapsulation model for their stateful functions, but Azure Durable Functions also supports *orchestrator functions* for common application patterns. Orchestrator functions allow stitching together synchronous calls to function instances, greatly simplifying the implementation of workflows across multiple function instances (such as sagas). Orchestrator functions do not support transactions with isolation guarantees.

The approach to use coordinator functions was inspired by orchestrator functions; it is possible to simply use other stateful functions as *orchestrators* (for sagas) and *coordinators* (for two-phase commit). This thesis introduces coordinator functions that greatly simplify transactions across various function instances, both with serializability using two-phase locking and two-phase commit and without isolation guarantees using sagas. Figure 4.1 shows a general picture of how coordinator functions work. Sections 4.2.2 and 4.2.3 describe this in more detail for serializable transactions and sagas respectively. First, section 4.2.1 introduces the definition of transactions in SFaaS along with the programming model. Section 4.2.4 describes changes required to original functions to participate in transactions using coordinator functions.

4.2.1 Programming Model for Transactions in SFaaS

Coordinator functions are a specialized type of user-defined function that developers can use to implement transactions involving arbitrary function instances based

```python
def serializable_transfer(context, message: Transfer):
    subtract_credit = SubtractCreditMessage(amount = message.amount)

    context.2pc_invocation("account_function",
                                            message.debtor,
                                            subtract_credit)
    add_credit = AddCreditMessage(amount = message.amount)
    context.2pc_invocation("account_function",
                                            message.creditor,
                                            add_credit)
```

LISTING 4.1: Two-phase commit coordinator function.

```python
def sagas_transfer(context, message: Transfer):
    subtract_credit = SubtractCreditMessage(amount=message.amount)
    add_credit = AddCreditMessage(amount=message.amount)
    context.saga_invocation_pair("ycsb-example/account_function",
                                      message.debtor,
                                      subtract_credit,
                                      add_credit)
    context.saga_invocation_pair("ycsb-example/account_function",
                                      message.creditor,
                                      add_credit,
                                      subtract_credit)
```

LISTING 4.2: Saga coordinator function.

on an input message. From a user perspective, the coordinator functions are stateless, but internally they keep the state of the ongoing transaction and communicate with involved function instances transparently. The coordinator functions can rely on the existing exactly-once fault-tolerant messaging between stateful function instances, greatly simplifying durability and atomicity.

The serializable transactions implemented in this thesis are slightly different from traditional transactions; rather than transactions being a set of operations, transactions are now a set of functions invocations, as seen in listing 4.1. The atomicity of transactions covers all side effects caused by the included function invocations. So, state updates and any function invocations to other function instances (or other side effects) are committed or aborted atomically. This does not apply to sagas (listing 4.2); the developer is still responsible for compensating all cascading effects caused by the original invocation with the compensating invocation.

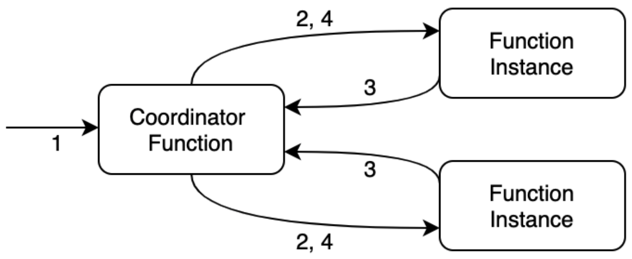


FIGURE 4.1: Communication pattern for coordinator functions

4.2.2 Two-Phase Commit Functions

Listing 4.1 shows an example of a two-phase commit function. The user can add function invocations to the transaction. Besides this, the user can also define desired side effects to perform after the transaction completes. The transactions can complete in three different scenarios; successful, failed, or retry-able. The user can add side effects to perform in each of these cases, as seen later in listing 5.2.

Figure 4.1 shows the general flow of a transaction. The coordinator function serves the role of the coordinator in two-phase commit. After receiving a message (1), the coordinator function invokes the involved function instances with a *prepare* message (2). The involved function instances do not directly apply the side effects caused by the invocation and store them separately instead. At this point, the function instance also locks and stops processing new invocations. The function instances send a message to the coordinator function whether they succeeded (3). Based on its received messages, the coordinator function either commits or aborts the transaction (4). The transaction completes as retry-able if the transaction could not complete due to a deadlock. All of the communication described here is entirely transparent to the user.

4.2.3 Sagas Functions

The sagas coordinator performs the role of the orchestrator in sagas. The coordinator function defines a set of function invocation and compensating function invocation pairs, as seen in listing 4.2, and side effects to perform after completing the sagas. Since sagas do not lock, there is no possibility of a deadlock, and sagas either succeed or fail. The coordinator function calls the function instances with the initial function invocations (2 in figure 4.1). The invoked function instances return to the stateful operator whether the function invocation was completed successfully or failed (3). Because sagas do not require isolation, the function instance can continue processing function invocations. Based on its received messages, the coordinator function can decide whether to send the compensating invocations to the involved function instances. Similar to for two-phase commit coordinator functions, all the communication is transparent to the user.

This model does require stateful functions to implement logic to handle compensating invocations for any invocation that may be part of a transaction, however, this is likely often already the case (e.g. accepting `AddCredit` and `SubtractCredit` messages or `CreateUser` and `DeleteUser` messages).

4.2.4 Regular Stateful Functions

As described, the coordinator functions allow regular stateful functions to be executed as part of a transaction. The aim is to be able to include any arbitrary function in a transaction. This would allow for loose coupling between functions and coordinator functions and the ability to compose complex behavior using simple functions.

For these features, there is however an important piece of information that is required by the system; the success or failure of a function invocation. Some function invocations may fail based on the state of the function instance. A simple example of this is a `SubtractCredit(5)` message to a user function instance with `balance=1` in its state. This should represent a failed execution and if the `SubtractCredit(5)` message was part of a transaction, the entire transaction should fail. This is similar to the concept of integrity constraints in relational databases.

Besides this, regular functions should also participate in the communication with coordinators transparently and implement the correct isolation semantics.

Chapter 5

Implementation

This chapter describes the implementation of the coordinator functions in Flink StateFun. First, it describes the internals of StateFun in section 5.1, then section 5.2 describes changes made to the system to implement coordinator functions.

5.1 StateFun internals

The coordinator functions this paper introduces are implemented in Flink Stateful Functions, also going by the name of StateFun. It is necessary to understand the internals of StateFun to interpret the addition of this thesis.

5.1.1 Functions

Functions are at the core of StateFun. User-defined functions are uniquely described by a *function type* that consists of a namespace and a name. StateFun function types define state managed by StateFun. Multiple *function instances* of any function type can exist in parallel and are uniquely identified by an *ID*. This means any *function instance* can be uniquely addressed by the combination of the function type and the ID, together referred to as the *address*. The invocation of a function type and its accompanying state are always scoped to the current address. So, function instances may be loosely compared to rows in a relational database or documents in a document store. Function instances also share similarities with stateful actors in actor programming models.

Table 5.1 shows how *function instances* can be invoked in StateFun.

Method	**Description**
Ingresses	Events outside StateFun can enter the system through ingresses. Examples of ingress systems are Apache Kafka and AWS Kinesis. For some ingresses, delivery guarantees span across the ingresses (and egresses). This does require replayable ingresses.
Internal messages	Function instances may be invoked by other function instances or even by themselves.

TABLE 5.1: Methods to invoke StateFun functions

A function invocation may perform any combination of one or more of the side effects presented in table 5.2. These side effects are managed by StateFun and the configured delivery guarantees apply.

Method	Description
State updates	The stateful functions have access to their state and may update this state.
Egresses	Similar to how ingresses create entrypoints to the StateFun system, egresses are controlled exits from the system.
Invocations	Function instances may invoke other function instances. Other function instances may be invoked with any Protocol Buffer (Protobuf)[1] message by their unique address (type and ID).
Delayed invocations	Function instances may invoke other function instances with a delay. This delay is handled within the StateFun cluster. This may be used, for example, for internal timeouts or reminder notification systems.

TABLE 5.2: Side effects caused by StateFun functions

Besides these defined side effects, developers are free to cause other side effects in a function's execution (for example by calling a third-party system), however these side effects are not managed by StateFun and exactly-once guarantees do not apply out-of-the-box.

5.1.2 Streaming graph

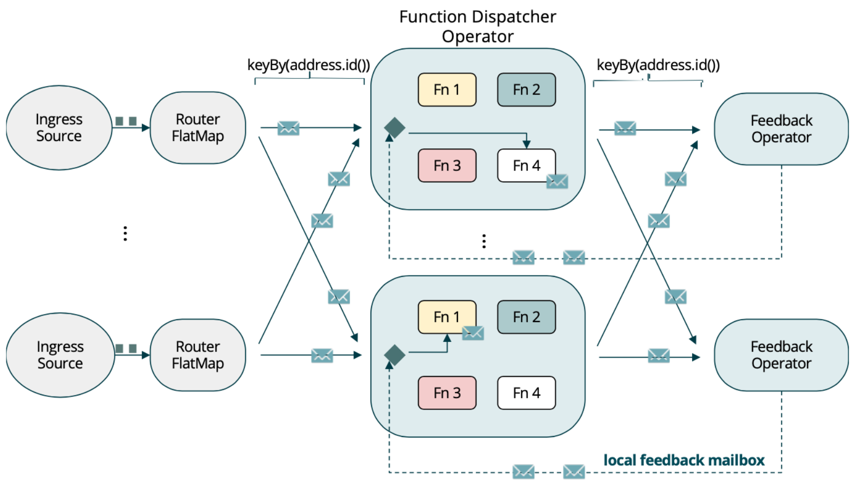


FIGURE 5.1: StateFun stream processing graph (Tai, 2020)

StateFun is powered by stream processing using Flink. It uses a quite simple underlying stream graph to handle messaging and state. Figure 5.1 shows this underlying stream graph (Tai, 2020). It consists of three operators; the ingress and router operator, the function dispatcher, and the feedback operator.

Firstly, the *Ingress and Router Operators* accept messages from ingresses, convert them into *envelopes* used for internal messaging, and route them to the correct Function Dispatcher Operators using a simple *.keyBy* based on the address.

Secondly, the *Function Dispatcher Operators* hold the state of the function instances. Each Flink worker has an internal representation of all function types, however, the

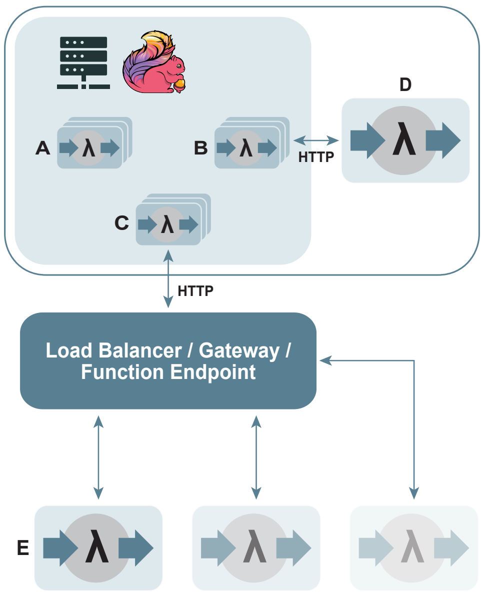


FIGURE 5.2: Stateful Functions deployment styles

state of different instances is partitioned across the workers. The *.keyBy* operation ensures the messages are routed to the worker responsible for the state of the function instance. The Function Dispatcher Operator then loads the function type and invokes the function with the message and the state of the addressed function instance. After the function execution, egress messages are published to the appropriate egresses and function invocations passed to the feedback operator. The outbound function invocations get passed to the feedback operator through another *.keyBy* operator based on their addresses.

Thirdly, *Feedback Operators* are co-located on the StateFun workers with the Function Dispatcher Operators. The Feedback Operator receives the function invocations from other function instances that are addressed to their co-located Function Dispatcher Operator because of the network shuffle caused by the second *.keyBy*. The message is then put in a *local feedback mailbox* to which the Function Dispatcher Operator has access. This is the mechanism that allows dynamic messaging in StateFun based on stream processing.

This streaming graph directly inherits Flink's fault-tolerance mechanisms with rollback recovery and checkpointing to distributed file systems. Besides this, it also supports end-to-end exactly-once guarantees based on Flink's source connectors for ingresses and and sink connectors for egresses. This is particularly useful since Flink already supports a range of data sources and sinks.

5.1.3 Deployment styles

StateFun identifies three deployment styles for functions. These are visualized in figure 5.2.

Embedded functions

Embedded functions run inside the same JVM as the Statefun worker. The functions have direct access to state and are directly invoked by messages in the StateFun cluster. In figure 5.2 embedded functions are shown as **A**.

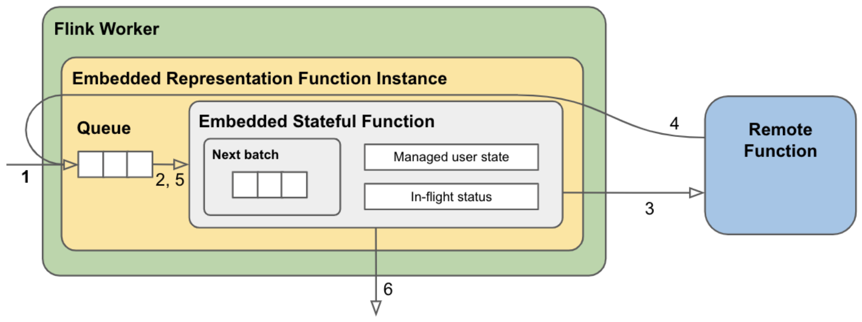


FIGURE 5.3: Original communication flow for remote functions

```
1  message InvocationBatchRequest {
2      Address target = 1;
3      repeated PersistedValue state = 2;
4      repeated Invocation invocations = 3;
5  }
6
7  message InvocationResponse {
8      repeated PersistedValueMutation state_mutations = 1;
9      repeated Invocation outgoing_messages = 2;
10     repeated DelayedInvocation delayed_invocations = 3;
11     repeated EgressMessage outgoing_egresses = 4;
12 }
```

LISTING 5.1: Protocol buffer for regular function

Remote functions

StateFun integrates with existing FaaS services using remote functions. Remote functions are deployed entirely separate from the StateFun cluster, for example on AWS Lambda. These remote functions are shown in figure 5.2 at **E**. Remote functions are internally represented by embedded functions, shown in figure 5.2 as **C**. The remote function and representative embedded function instance are also shown in figure 5.3. The embedded function is responsible for communicating with the remote function and state management. The boxes in the grey box represent the state kept by the embedded function. Exactly-once guarantees apply to this state.

Messages arrive at the embedded function through a queue in StateFun (message 1 in figure 5.3). The embedded functions handle messages one by one in first-in-first-out order. When a new function invocation arrives (message 2), and if the remote function is currently not executing other invocations, the incoming invocation is sent to the remote function. If the remote function is busy processing other function invocations, the message is appended to the `next batch` to be sent to the remote function to ensure isolation and linearizability. The embedded function keeps track of invocations currently being executed by the remote function with the `in-flight status`, shown in figure 5.3.

The embedded function sends either a single invocation or a batch of invocations to the remote function with the `InvocationBatchRequest` message, as seen in listing 5.1 (message 3 in figure 5.3). Besides the invocation(s) to the function (line 4), the message also contains the current state (line 3) to provide the stateless remote function with access to the state. The remote function returns with an `InvocationResponse` message containing the updated state and any side effects caused by the invocation(s) (message 4). When the embedded function processes the response (message 5), the `managed user state` is updated, the side effects are performed (message 6), and any batched invocations are sent to the remote function with the updated `managed user state`.

The remote functions can be implemented in any programming language. A Python SDK is currently actively maintained as part of StateFun.

Co-located functions

Co-located functions are deployed on the same machine as a StateFun worker, for example as a sidecar in a kubernetes pod. This is shown in figure 5.2 at **D**. This is more performant than remote functions since the StateFun worker can invoke the function through local network. Communication with co-located functions is done via the same protocol and co-located functions are internally represented the same as the remote functions. The embedded function in figure 5.2 is represented by **B** (note that **B** is the same as **C**). However, co-locating functions do not provide the possibility of separately scaling the computation layer.

Remote and co-located functions can be added to the StateFun cluster through a *module* YAML-file containing a (load-balanced) service endpoint and the state to be managed by StateFun along with optional configuration settings.

5.1.4 Event-driven database

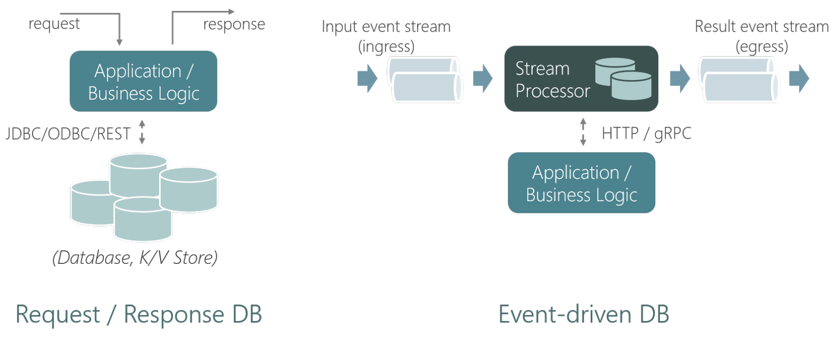


FIGURE 5.4: Event-driven database model (Ewen, 2020)

The StateFun project inverts the relationship between the database and the application. Rather than have the application call the database, the database now calls the application. This is represented in figure 5.4. This model combines state management with messaging and can guarantee exactly-once semantics across both. This solves two complex issues in distributed systems (state management and messaging), greatly simplifying scalable cloud applications.

5.2 Implementation Coordinator Functions

This section describes the implementation of the addition of coordinator functions to StateFun. For this thesis, coordinator functions are implemented for remote and co-located functions, though this approach can also be applied to embedded functions.

Function	Description
Both coordinator function classes	
`send_on_success`(*type*, *id*, *message*)	Sends a message to another function instance if the transaction is successful
`send_after_on_success`(*delay*, *type*, *id*, *message*)	Sends a delayed message if the transaction is successful
`send_egress_on_success`(*type*, *egress_message*)	Sends a message to an egress if the transaction is successful
`send_on_failure`(*type*, *id*, *message*)	Sends a message to another function instance if the transaction failed
`send_after_on_failure`(*delay*, *type*, *id*, *message*)	Sends a delayed message if the transaction failed
`send_egress_on_failure`(*type*, *egress_message*)	Sends a message to an egress if the transaction failed
Two-phase commit functions	
`2pc_invocation`(*type*, *id*, *message*)	Add a function invocation to the transaction
`send_on_retryable`(*type*, *id*, *message*)	Sends a message if the transaction aborted because of a deadlock
`send_after_on_retryable`(*delay*, *type*, *id*, *message*)	Sends a delayed message if the transaction aborted because of a deadlock
`send_egress_on_retryable`(*type*, *egress_message*)	Sends a message to an egress if the transaction aborted because of a deadlock
Sagas functions	
`saga_invocation_pair`(*type*, *id*, *message*, *compensating_message*)	Add a pair of a message and a compensating message to the transaction
Regular functions	
`FunctionInvocationException`	Raised to fail the function invocation

LISTING 5.2: Coordinator functions' Python API.

5.2.1 Representative Embedded Functions

The embedded functions representing the remote function are used to implement coordinator functions for this thesis. B and C in figure 5.2 and the grey box in figure 5.3 show these embedded functions. The logic for transactions is implemented in these embedded functions, hidden from the user and using the existing exactly-once guarantees. Some changes in the user's programming model were required to implement transactions, as shown in listing 5.2.

5.2.2 Updates to Regular Functions

```
1  message InvocationResponse {
2      repeated PersistedValueMutation state_mutations = 1;
3      repeated Invocation outgoing_messages = 2;
4      repeated DelayedInvocation delayed_invocations = 3;
5      repeated EgressMessage outgoing_egresses = 4;
6  +   repeated bool failed_invocations = 5;
7  }
```

LISTING 5.3: New protocol buffer for regular function

For arbitrary function instances to be involved in serializable transactions and sagas, they need to be able to communicate to the embedded function that an invocation failed based on the instance state. An extra Boolean field is added to the Protobuf response of the remote function to communicate these failures, as seen in line 6 in listing 5.3. Since invocations may be batched, a list (`repeated`) field is used.

The developer also needs to be able to fail a function invocation explicitly. The developer can throw an introduced Python exception, the `FunctionInvocationException`, to do this. Programmers may define their own subclasses of this exception for specific failures. When an exception is thrown, all side effects created by this invocation up to that point are discarded. If an invocation part of a transaction or sagas, the invocation will fail and return no side effects. However, for a regular invocation, the developer may define an exception handler to perform any side effects. In the example in figure 5.4, when the invocation is part of a transaction or sagas, it will directly fail when no user exists. However, in a normal execution, it will execute the function defined on line 18 and produce a message to an egress to reply to the incoming message.

```python
class NotFoundException(FunctionInvocationException):
    def __init__(self, request_id):
        super().__init__(request_id)

@functions.bind("example/user_function")
async def account_function(context, message: Read):
    # Get state
    user = context.state('user').unpack(User)

    if not user:
        raise NotFoundException(message.request_id)
    else:
        response = Response(request_id=request_id, status_code=200, state=user)
        egress_message = kafka_egress_record(topic="responses", key=request_id, value=response)
        context.pack_and_send_egress("ycsb-example/kafka-egress", egress_message)

@functions.bind_exception_handler(NotFoundException)
async def handle_not_found(context, request_id):
    response = Response(request_id=request_id, status_code=404)
    egress_message = kafka_egress_record(topic="responses", key=request_id, value=response)
    context.pack_and_send_egress("ycsb-example/kafka-egress", egress_message)
```

LISTING 5.4: Sample programming model for regular function

The batching mechanism also requires some changes, namely, additional bookkeeping for sagas and isolation for serializable transactions. These changes are described in sections 5.2.5 and 5.2.4.

5.2.3 Coordinator Functions

The coordinator functions are also deployed as remote functions. StateFun can distinguish between regular functions, two-phase commit coordinator functions, and sagas coordinator functions by their function classes, as described in the module configuration YAML-file. Coordinator functions are represented by different standardized embedded functions than regular functions. The embedded representations of coordinator functions communicate with the remote function through a different Protobuf interface. The remote coordinator functions also have a different programming model than regular functions; the coordinator functions are stateless from the developer's perspective. The embedded representations of the coordinator functions do keep internal state to keep track of the progress of the transaction. Figure 5.5 shows the general communication flow for sagas and serializable transactions. Messages 1, 2, and 3 are the same in both cases; the coordinator function instance is simply messaged and calls the remote function with the invocation. The messages annotated with a * are not always present in both scenarios.

5.2.4 Two-phase Commit Function

The two-phase commit coordinator function allows serializable transactions in StateFun using two-phase commit and two-phase locking. The developer can describe transactions based on an input message using the functions described in listing 5.2. Listing 5.5 shows an example of the code for a serializable transaction. The programming model is intentionally extremely simple; the developer can simply add function invocations to the transaction and include the desired side effects for each completion scenario based on the input message.

The two-phase commit coordinator function also requires a new Protobuf response message from the remote function to the embedded function. Listing 5.6

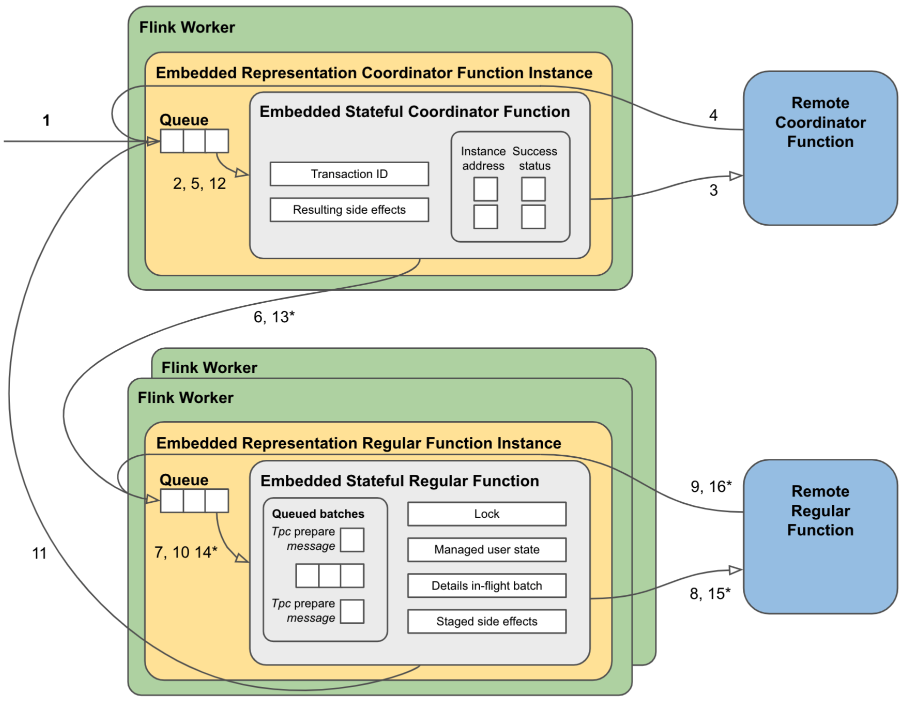


FIGURE 5.5: Communication flow for transactions

```python
def handle_transfer(context, message: Transfer):
    # Send messages
    subtract_credit = SubtractCredit(amount = message.amount)
    context.pack_and_send_atomic_invocation("ycsb/account_function",
                                            message.outgoing_id,
                                            subtract_credit)
    add_credit = AddCredit(amount = message.amount)
    context.pack_and_send_atomic_invocation("ycsb/account_function",
                                            message.incoming_id,
                                            add_credit)

    # Send on success
    response = Response(request_id=message.request_id, status_code=200)
    egress_message = kafka_egress_record(topic="responses",
                                         key=request_id, value=response)
    context.pack_and_send_egress_on_success("ycsb/kafka-egress", egress_message)

    # Send on failure
    response = Response(request_id=message.request_id, status_code=422)
    egress_message = kafka_egress_record(topic="responses",
                                         key=request_id, value=response)
    context.pack_and_send_egress_on_failure("ycsb/kafka-egress", egress_message)

    # Send on retryable (e.g. deadlock)
    response = Response(request_id=message.request_id, status_code=401)
    egress_message = kafka_egress_record(topic="responses",
                                         key=request_id, value=response)
    context.pack_and_send_egress_on_retryable("ycsb/kafka-egress", egress_message)
```

LISTING 5.5: Sample programming model for two-phase commit function

shows this Protobuf message. It includes an arbitrary number of invocations to include in the transaction and side effects based on the different completion scenarios. Figure 5.5 shows this message as message 4.

```
message TpcFunctionInvocationResponse {
    repeated Invocation transactional_invocations = 1;
    InvocationResponse success_response = 2;
    InvocationResponse failure_response = 3;
    InvocationResponse retryable_response = 4;
}
```

FIGURE 5.6: Protocol buffer response of two-phase commit functions

The transactions' communication flow is shown in figure 5.5. After receiving the details of the to-be-performed transaction in message 5, the two-phase commit with two-phase locking protocol starts:

`PREPARE` **& two-phase lock growing phase** Firstly, the embedded coordinator function instance stores the side effects to perform on completion and a map with the involved function instances and a boolean for the completion status. Then, the coordinator function instance sends an invocation to all involved function instances (message 6). The receiving function instances can identify this invocation as a `PREPARE` message and do not batch it with other invocations to ensure isolation. If a batch already exists in the embedded function instance, the function invocation is queued after this batch (as seen in figure 5.5). The concept of queued batches does complicate the batching mechanism as it used to be a simple append-only batch. When the invocation is sent to the remote function (message 8), the details of the transaction (the ID and the address of the coordinator function instance) are stored in the `details in-flight batch`. The function instance performs the isolated `PREPARE` function invocations as normal (message 8, 9 and 10). After the embedded function receives the result of the function invocation (message 10), it sends a message to the coordinator function (message 11) based on the result. If the function invocation was successful, the function instance now stores the function invocation results as `staged side effects` and sets its `lock`. It does not continue processing requests to wait for either a `COMMIT` or `ABORT` message. If the function invocation failed (i.e., a `FunctionInvocationException` was thrown during the execution at the remote function), the function instance continues processing requests because it knows the transaction will be aborted.

`ABORT` **& two-phase lock shrinking phase** When the coordinator function instance receives a failure from any function instance (message 12), it directly aborts the transaction. The coordinator function instance sends an `ABORT` message to all involved function instances (message 13) and performs the side effects for a failed transaction. The function instances receiving an `ABORT` message discard any side effects stored in the `staged side effects`, release the `lock`, and continue processing other invocations. Function instances may also receive an `ABORT` message while the `PREPARE` message is still queued or in progress. The `PREPARE` message is then removed from the queue, or the upcoming response is ignored, respectively.

`COMMIT` **& two-phase lock shrinking phase** When the coordinator function receives a success response from all the involved functions (message 12), it commits the transaction. It performs the appropriate side effects and sends a `COMMIT` message to all involved function instances (message 13). When the function instance receives a

COMMIT message, it performs the staged side effects, releases the lock, and continues processing other requests.

Deadlock detection

Deadlocks may occur when using two-phase commit with two-phase locking. Since the participants in the transaction are distributed, a method to deal with distributed deadlocks is required. A wait-die deadlock prevention approach fits the system well; however, this would unnecessarily abort transactions. Instead, the Chandy-Misra-Haas algorithm is used to detect distributed deadlocks in a decentralized way.

Figure 5.7 shows how the Chandy-Misra-Haas algorithm is implemented. Function instances that are part of a transaction notify their coordinator instance with the addresses of the coordinators of any PREPARE messages queued in front of them after they receive message 7 in figure 5.5. This is message 1 in figure 5.7. The coordinator function instance then sends a probe to the coordinator instances of any blocking transactions, notifying them they are waiting for them (message 2 in figure 5.7). Other coordinator function instances forward these probes to any coordinators that they are waiting for themselves (messages 3 and 4 in figure 5.7). In the case of a deadlock, a coordinator function would always receive a probe sent by itself, detecting they are in a deadlock. The coordinator function keeps track of which functions it is currently blocking (or already sent probes for) to prevent probes from being forwarded indefinitely.

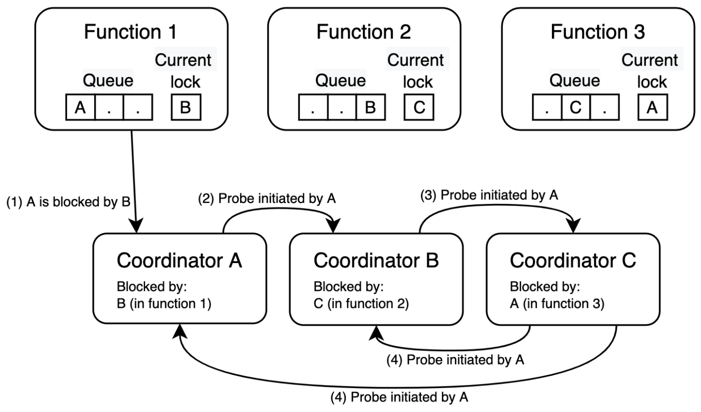


FIGURE 5.7: Deadlock detection visualisation

Whenever a deadlock is detected, it immediately completes as a retry-able transaction; it aborts the transaction and performs the appropriate side effects. A two-phase commit function may send itself a delayed invocation with the same message (and possibly a counter attached) using the retry-able side effects to perform a retry. This is left to the developer, so the system remains flexible.

5.2.5 Sagas Function

The second introduced function class is the sagas function. The sagas function provides a standardized way to implement sagas workflows. Sagas workflows can be described as transactions with the *read uncommitted* isolation level (no isolation at all). This method does not require locking and thereby aims for higher performance compared to the two-phase commit function.

```python
1  def handle_transfer(context, message: Transfer):
2      # Send messages
3      subtract_credit = SubtractCredit(amount=message.amount)
4      add_credit = AddCredit(amount=message.amount)
5      context.pack_and_send_invocation_pair("ycsb-example/account_function",
6                                            message.outgoing_id,
7                                            subtract_credit,
8                                            add_credit)
9      context.pack_and_send_invocation_pair("ycsb-example/account_function",
10                                           message.incoming_id,
11                                           add_credit,
12                                           subtract_credit)
```

LISTING 5.6: Sample programming model for sagas function

```
message SagasFunctionInvocationResponse {
    repeated SagasFunctionPair invocation_pairs = 1;
    InvocationResponse success_response = 2;
    InvocationResponse failure_response = 3;
}

message SagasFunctionPair {
    Invocation initial_message = 1;
    Invocation compensating_message = 2;
}
```

LISTING 5.7: Protocol buffer response of sagas functions

The programming model for the sagas coordinator function is very similar to that of the two-phase commit function. Listing 5.2 shows the available functions for a sagas function and listing 5.6 shows an example. Defining the side effects is not shown in this figure as it is the same as for two-phase commit functions, without side effects for retry-able cases since there are no deadlocks using sagas. The most important detail is in lines 7 and 8 and lines 11 and 12, where the function explicitly requires both an initial message and a compensating message to be set.

Figure 5.7 shows the Protobuf response of a sagas remote function. As opposed to the list of single invocations added to a transaction using the two-phase commit protocol, the sagas function requires invocation pairs.

Figure 5.5 also shows the communication flow of sagas transaction:

Initial function invocations When the embedded sagas coordinator instance returns from the stateless remote function (message 5), it stores a map of the involved function instances with their completion status (`NULL` for now). It stores the side effects to be performed at completion and sends the initial invocations to their function instances (message 6). The function instance can append this sagas function instance to the batches since they do not require isolation. However, the batch should keep track of at what indices sagas invocations are (along with the address of their coordinator instance) to identify the completion status when the result returns from the remote function instance (message 10). These indices and their respective coordinator instance addresses are kept in the `details in-flight batch` while the invocations are sent to the remote function (message 8). When the response arrives, the function instance messages the coordinator function instances of any sagas invocations in the batch with the completion status. If the coordinator function instance receives a success message from all involved function instances (messages 12), it performs the appropriate side effects, and then it is done.

Compensating function invocations When any involved function instance returns a failure to the coordinator, the coordinator can directly perform the appropriate side effects. The coordinator function should also send the appropriate compensating invocation to any function instances that executed its invocation successfully. The coordinator now has to wait until all function instances have returned, and if any return successfully, the coordinator instance should send its compensating invocation. The compensating invocation is then processed as a regular invocation to the function instance. The coordinator instance is done when it received a message from all function instances.

Chapter 6

Evaluation

This chapter describes the evaluation of the coordinator functions. Firstly, the experimental setup, including the used benchmark, is presented. Secondly, the experiments and their results are presented.

6.1 Experimental setup

This section starts by introducing the workload that is used to evaluate the system. Secondly, the application and the infrastructure that is used are described. Lastly, the evaluation metrics are described.

6.1.1 Workload

As described in chapter 2, there currently is no widely-used benchmark to evaluate cloud systems, including transactions. For the evaluation of coordinator function a small adaption is made to the popular *Yahoo! Cloud Serving Benchmark* (YCSB) (Cooper et al., 2010).

In YCSB, the first step is to insert records into the system. Every record has a unique ID and several fields. After the insertion stage, operations are performed on the inserted state. YCSB defines `read` and `write` operations as part of their core workloads. The proportions of these operations in the complete workload may vary to evaluate the system's different properties.

Since StateFun with coordinator functions also supports multi-address transactions, a new multi-address operation is added based on an extension introduced by Dey et al. (2014). This operation is called a `transfer`. The operation atomically subtracts `balance` from one address and adds this to another. This means that records also include an integer `balance` field.

Including the addition of the transfer operations, the workload can consist of the following three operations.

- `read`: The `read` operation reads the state associated with a single key and outputs it to the egress.
- `write`: The `write` operation updates a field associated with a single key and outputs a *success* message to the egress.
- `transfer`: The `transfer` operation requires two keys and a specified amount. It subtracts the amount from the balance of one key and adds it to the other. Depending on the transaction's result, it either outputs a success message or a failure message to the egress.

YCSB defines multiple distributions to determine the record ID for the next operation. The experiments done for this thesis use a uniform distribution. The uniform distribution simplifies the interpretation of the results and the reasoning about the system's properties. It should be taken into account that this does not necessarily represent real-world use cases, in which often some keys are more popular than others.

YCSB allows to variably define the number of fields and the size of the values associated with each field. In this evaluation, all records have ten fields containing a single random string of 128 bits and a single integer field. Any `WRITE` operation simply replaces the random string for one field.

6.1.2 Application

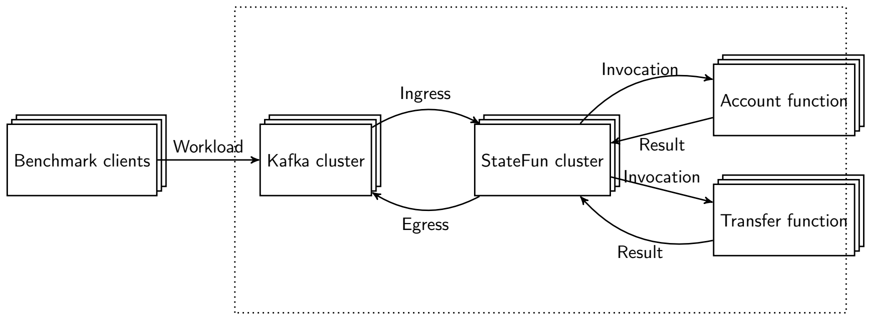


FIGURE 6.1: Application architecture

To support the operations defined in 6.1.1, a StateFun system is implemented with the following two functions:

Account function

The account function is a regular function containing the record state for each key (or address). It processes messages to read the state, update the fields, and subtract or add balance as part of a transaction. It throws an exception and fails the transaction if the key does not exist or if there is not enough balance to subtract the transaction amount.

Transfer function

The transfer function is a coordinator function that takes a message consisting of two different keys and an amount. It will define a transaction consisting of two function invocations, one to each of the function keys. This function is both implemented as a two-phase commit function and as a sagas function.

Figure 6.1 shows a diagram of the system under test. The workload is published in the Kafka cluster. StateFun reads from Kafka as an ingress and invokes the appropriate functions. The result of each ingress message is published to Kafka as an egress.

The system under test is deployed on an HPC cloud with instances with up to 80 vCPUs (SurfSara[1]). In this cloud, two VMs are run with enough vCPU to support the system under test's configuration. The two VMs form a Kubernetes cluster

[1] https://userinfo.surfsara.nl/systems/hpc-cloud

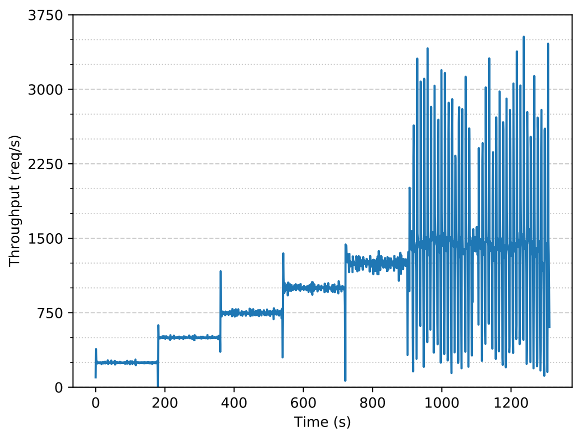


FIGURE 6.2: Graph showing when the maximum throughput is reached

(with an additional master node) to simplify deployment and management of the system's different components. All components shown in figure 6.1 can be horizontally scaled as necessary. The Kafka cluster is given enough resources to ensure it can handle the load. Since the remote functions may be deployed on FaaS services such as AWS Lambda, they are also deployed on enough resources to handle the load on them. This ensures the bottleneck of the system is the StateFun cluster. The number of StateFun workers and the CPU available to them is varied across different experiments.

6.1.3 Evaluation metrics

The system is evaluated using two metrics. The maximum throughput shows the number of workload operations the system can handle per second, and the latency shows the time it takes to process the operation.

The maximum throughput of each workload and system configuration is found by steadily increasing the input throughput created by the benchmark clients in Kafka until the StateFun cluster can no longer consistently handle the load. The output throughput of StateFun in Kafka is measured and compared against the input throughput to see whether the system can handle the load. At some point, the output throughput starts fluctuating as seen in figure 6.2 between 1250 and 1500, and the output throughput drops far below the input throughput; at this point, we define this value as the maximum throughput for the configuration.

The Kafka event time for the ingress and egress event of correlated operations is used to measure end-to-end latency. The latency is dependent on the throughput. In the experiments, the throughput is often set to 80% of the maximum throughput to allow consistent operation of StateFun and measure the latency. When comparing latencies, the different throughput rates at which the latency was measured should be considered.

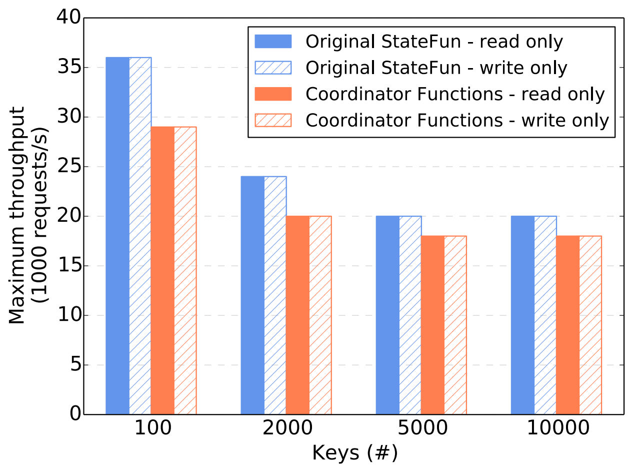


FIGURE 6.3: Maximum throughput for the original StateFun vs. StateFun with coordinator functions

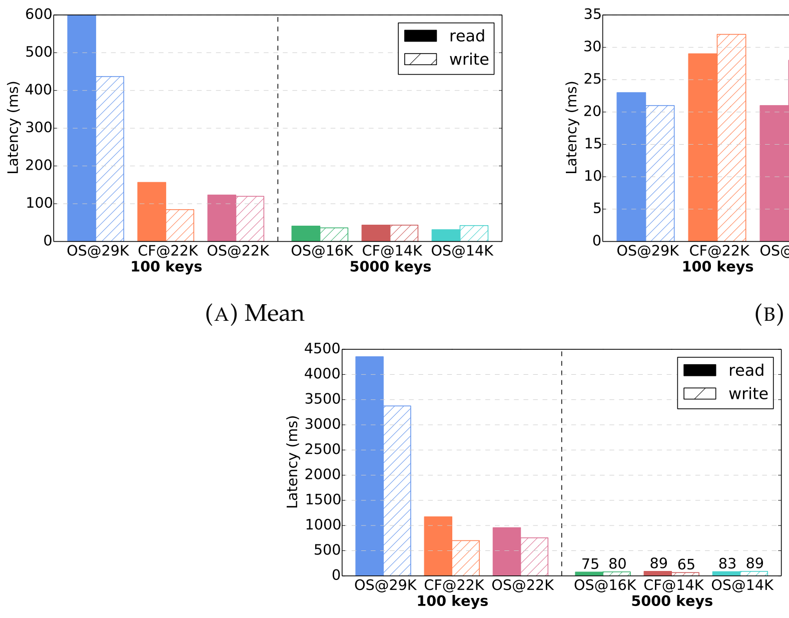
6.2 Experiments

This section presents the results of five experiments. Firstly, the system including coordinator functions is compared against the original StateFun to evaluate the overhead caused by the changes. Secondly, the overhead of transactions using both sagas and two-phase commit is evaluated and compared. Thirdly, the overhead of unsuccessful transactions is compared for sagas and two-phase commit. Fourthly, the time that two-phase commit holds locks and needs to detect deadlocks is measured. Lastly, the scalability of the system is evaluated.

6.2.1 Experiment 1: System overhead

In this first experiment, the performance of the StateFun system with coordinator functions is compared against the original StateFun system on workloads without any transactions. Specifically, the overhead of the additional logic and internal state that is added to regular functions to enable transactions is evaluated. Figure 6.3 shows the maximum throughput achieved by the original system and the system with coordinator functions for various numbers of keys (or function instances). Figure 6.4 shows the different latencies for these systems across workloads. The experiments are performed using 3 StateFun workers, each with 4 CPUs.

The first observation that can be made when looking only at the original StateFun system is the decrease in throughput when more function instances exist. This is due to the batching mechanism. Sending batched messages can increase the performance of StateFun in three different ways. Firstly, less network communication between StateFun and the remote function is required. Secondly, the side effects caused by batched invocations are already bundled at the remote function and only cause a single incoming response message in StateFun as opposed to multiple. Thirdly, only the state at the end of all batched `write` invocations is returned to StateFun to be stored. Batching does increase latency for the higher percentiles, which can be seen in figures 6.4c and 6.4a. We also observe that no differences exist anymore between 5000 and 10000 keys, meaning that there is no noticeable batching in the system at this point.



(A) Mean

(B) Median

(C) 95th percentile

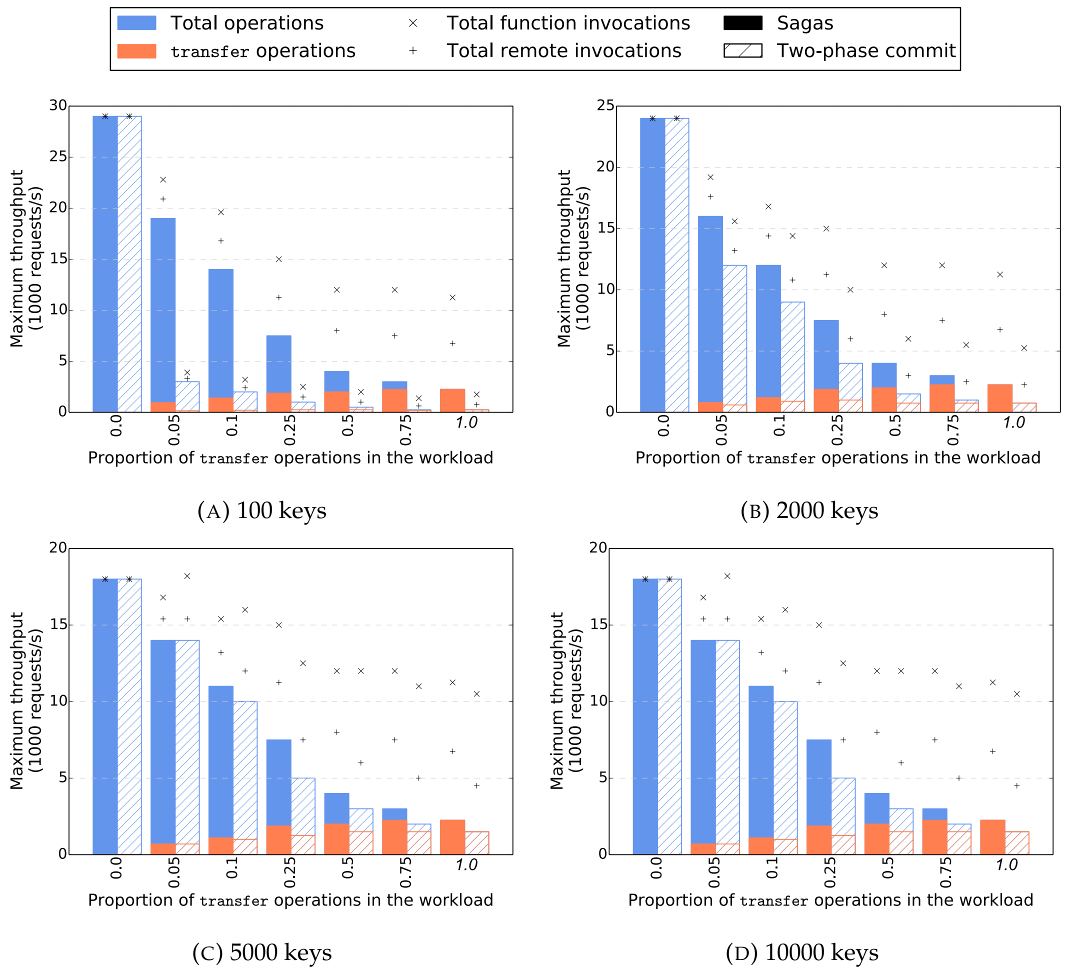
FIGURE 6.4: Graphs comparing latencies of orginal StateFun (OS) and StateFun with coordinator functions (CF) at different throughputs for read-only and write-only workloads

Figure 6.3 also shows that there is no noticeable throughput difference between a workload with only `read` and a workload with only `write` operations. This is because both operations need to access the remote function, making the communication layer the bottleneck. StateFun may improve the throughput of `read` operations by allowing read-only requests for state associated with remote functions to directly fetch state from the internal embedded function without calling the remote function. A downside of this approach would be that there would no longer be any control of the state's read model.

Figures 6.4a and 6.4c show higher latency for the `read` workload as opposed to the `write` workload at 100 keys. This difference is caused by the batching mechanism. For a `write` workload, only the final state after a batch has executed on the remote function needs to be returned to the StateFun cluster from the remote function. For the `read` workload, the state has to be serialized multiple times as a response to each `read` operation. This results in higher latency for `read` operations when batches become bigger.

When comparing the performance of the system with coordinator functions against the system without coordinator functions in figure 6.3, a decrease of 20% in throughput can be observed for 100 keys. This decrease lowers to 10% when the number of keys increases, i.e., when there is fewer batching. This decrease is caused by the changed batching mechanism that allows for isolated function invocations and adds some bookkeeping for function invocations part of a saga or a serializable transaction.

Lastly, the latencies in figure 6.4 are slightly higher for the system with coordinator functions. The added latency is caused by the additional logic required the implement the coordinator functions. Figures 6.4a and 6.4c show higher latencies for the original system at 29K throughput. However, this is only caused by batching; when the original system is run at the same throughput (22K) as the system with



(A) 100 keys

(B) 2000 keys

(C) 5000 keys

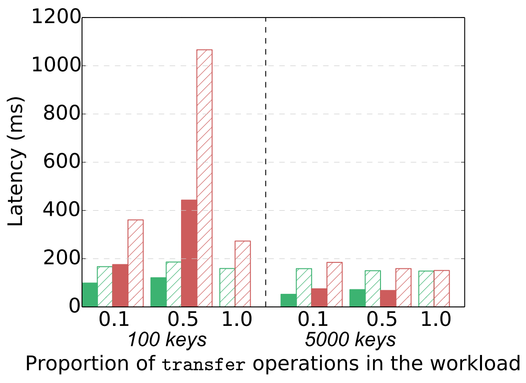
(D) 10000 keys

FIGURE 6.5: Maximum throughput for workloads with increasing proportions of `transfer` operations in the workload

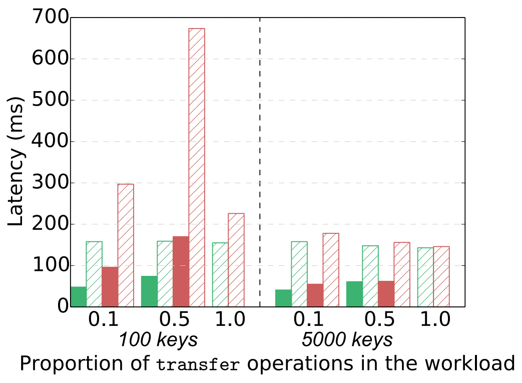
coordinator functions, it performs slightly better, as seen in figure 6.4.

6.2.2 Experiment 2: Transactional overhead

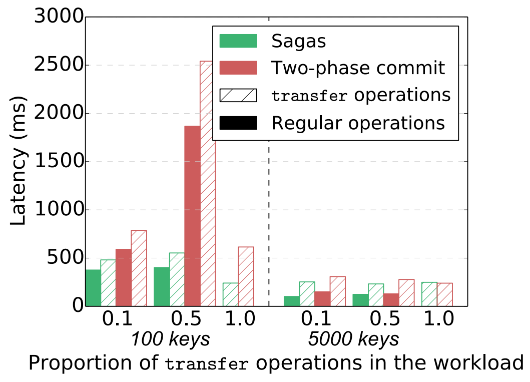
The second experiment shows the performance of transactions in the system. In these experiments, a certain proportion of the workload is `transfer` operations and the remaining proportion is equally shared between `read` and `write` operations. The experiments are performed on 3 StateFun workers with 4 CPUs each. Figure 6.5 shows the achieved throughput when increasing the proportions of `transfer` operations both using sagas and two-phase commit. It also shows the absolute number of `transfer` operations in the workload. For context, it shows indicators for the absolute amount of total internal function invocations (taking into account additional internal invocations required for serializable transactions and sagas) and the absolute amount of total remote function invocations (each `transfer` operation causes 3 remote function invocations). In the evaluation for two-phase commit functions, messages sent to detect deadlocks are not included in the total invocations number; therefore, the indicator is a lower bound. Figure 6.6 shows the measured latencies. In the experiments, the accounts are given enough balance to ensure all transactions succeeded.



(A) Mean



(B) Median



(C) 95th percentile

100 keys					
Sagas			**Tpc**		
0.1	0.5	1.0	0.1	0.5	1.0
11K	3K	2K	1.5K	0.4K	0.2K

5000 keys					
Sagas			**Tpc**		
0.1	0.5	1.0	0.1	0.5	1.0
9K	3K	2K	8K	2K	1.2K

(D) Throughputs at which latency was measured

FIGURE 6.6: Graphs comparing latencies for sagas and two-phase commit coordinator function for different keys and transaction proportions in the workload at 80% of the respective maximum throughputs

The first observation from figures 6.5a and 6.5b is that for few keys, sagas perform much better than two-phase commit. This difference has two reasons: 1) sagas can still benefit from the batching mechanism since they do not require isolation, and 2) the locking for in two-phase commit severely limits the throughput. However, it is also interesting that for a higher number of keys in figure 6.5c and 6.5d, two-phase commit performs comparably to sagas even though it provides much stronger guarantees. This is because there is less contention on a single function instance, decreasing the effect of locking and no benefit of batching as already seen in figure 6.3. For these experiments, a uniform key access distribution is used, while in real-world systems, this is likely skewed towards more access to some popular keys.

A second observation from figure 6.5 is that the total function invocations still drops when the proportion of transactions increases. The total function invocations account for the additional messaging required to coordinate transactions. This result is slightly unexpected and means that the internal coordinator function's logic and state management is more resource-consuming than regular functions, leading the overall throughput of workloads with a high proportion of `transfer` operations to be relatively low.

Figure 6.6 shows the latency at lower keys is much higher for two-phase commit functions. Even though the sagas functions leverage batching, the two-phase commit function's locking is costlier. As expected, figure 6.6 also shows higher latency

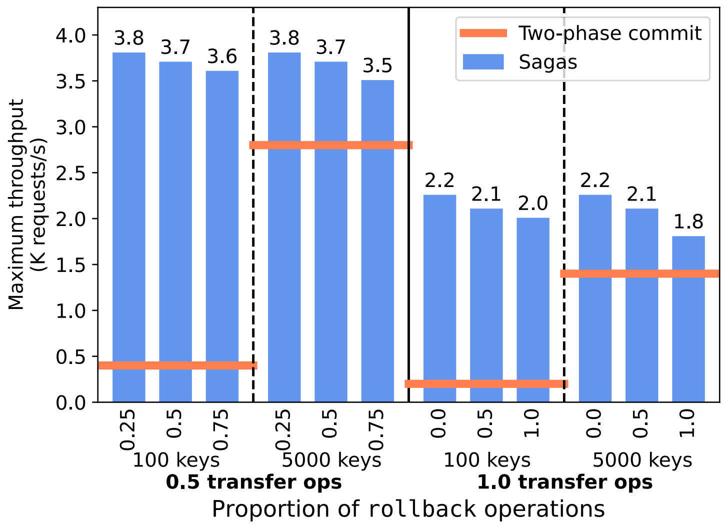


FIGURE 6.7: Throughput with different proportions of rolled back `transfer` operations for workloads with 50% and 100% `transfer` operations

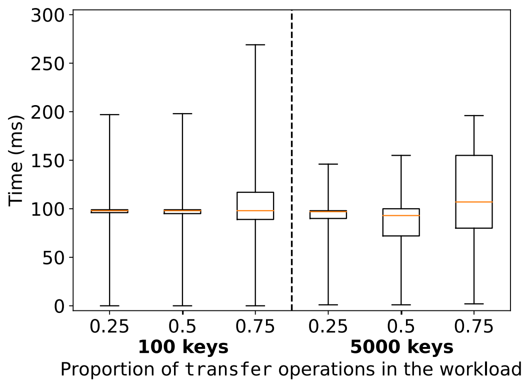
for `transfer` operations as opposed to other operations. This result is expected because `transfer` operations require access to two remote functions sequentially (first to the coordinator functions and then, in parallel, to two regular functions) and require additional messaging. Figure 6.6 shows few differences in latencies for 5000 keys.

6.2.3 Experiment 3: Rollback overhead

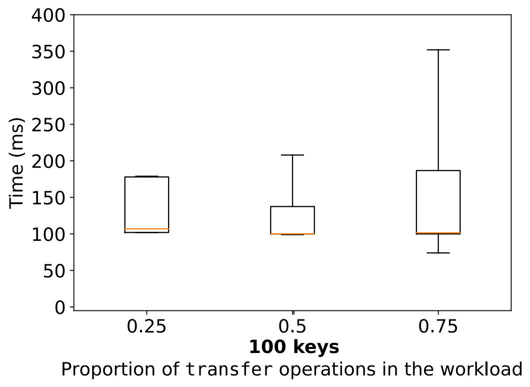
The third experiment provides insight in the overhead of rollbacks associated with transfers. Figure 6.7 shows the maximum throughput for workloads with 50% and 100% `transfer` operations where a different proportions of `transfer` operations fail for sagas and two-phase commit coordinator functions. The experiments are run using 3 Flink workers, each with access to 4 CPUs.

When using two-phase commit, a rollback does not significantly increase the load on the system. This is expected as the two-phase commit coordinator function needs to send a second message anyways, either an `abort` or `commit` message.

For sagas `transfer` operations, an increased proportion of `transfer` operations to be rolled back decreases the throughput. This is expected as the rollback of a saga `transfer` requires additional compensating messages to be send in the system that are not necessary in the case of a successful `transfer`. It can be observed that the difference is quite small when there are only 50% `transfer` operations in the workload. But when the workload consists only of `transfer` operations, the difference in throughput becomes more evident. Between 0% rollback `transfer` operations and 100% rollback `transfer` operations for 5000 keys, the throughput drops by 20%. This is more noticeable than the 10% drop in throughput for 100 keys. The difference in the decreased throughput can be explained by the fact that compensating operations can still benefit from batching for 100 keys, and the effect of batching is less noticeable for 5000 keys.



(A) Time regular locks are held



(B) Time to detect deadlocks

Keys	Transfer proportion	Deadlocks / `transfer` ops
100	0.25	9/12014 (0.07%)
	0.5	27/24107 (0.11%)
	0.75	82/35875 (0.22%)
5000	0.25	0/60121
	0.5	0/120089
	0.75	0/179794

(C) Deadlock frequency

FIGURE 6.8: Details of locking behaviour for a workload for 100 and 5000 keys with various proportions of transfers without rollbacks. The boxplots show the 5th and 95th percentiles.

This shows that the throughput of sagas `transfer` operations suffers when more operations have to be compensated, however it also shows that sagas still outperforms two-phase commit in the case of only rolled back operations.

6.2.4 Experiment 4: Lock timing

In the fourth experiment, the behaviour of locking and deadlocks when using two-phase commit coordinator functions is measured. Figure 6.8 shows the results. The experiments are run for 100 and 5000 keys for different proportions of `transfer` operations in the workload. The experiments are run using 3 Flink workers, each with access to 4 CPUs.

In figure 6.8a, we see that there is few difference in the median across the different workloads. The time that is shown is measured between the moment the function instance sends the response to the `prepare` messages to the coordinator function until the function instance receives either a `commit` or `abort` message and it sends the next batch to the remote function. This is the additional time the function instance is idle, as opposed to just in between batches when there are no `transfer` operations (or sagas are used). It can be observed that when the proportion of `transfer` operations is higher, the higher percentiles of the time locks are held increases significantly.

Figure 6.8c shows the number of deadlocks as opposed to the total number of `transfer` operations in the workload. It shows that no deadlocks occurred for workloads on 5000 keys, this makes sense since the contention is low. For 100 keys, figure

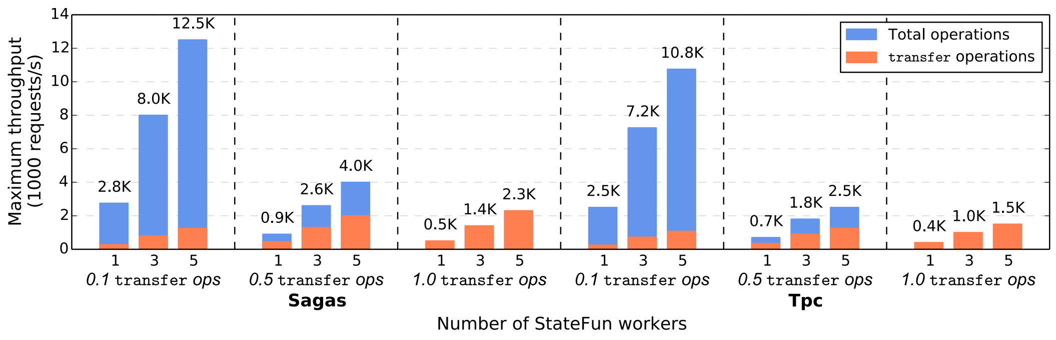


FIGURE 6.9: Maximum throughput for the system with 5000 keys for different numbers of StateFun workers for workloads with different proportions of `transfer` operations

6.8c shows an increasing number of deadlocks when the proportion of `transfer` operations in the workload increases. However, the percentage of deadlocks across all `transfer` operations is still small.

Figure 6.8b shows the time it takes to detect a deadlock. This time is measured between the response to the `prepare` message of a function instance to the coordinator function instance and the moment the coordinator function concludes it is in a deadlock and aborts the transaction. This corresponds to the exact time to perform the Chandy-Misra-Haas algorithm. Figure 6.8b shows that the median of the time this takes is similar. However it also shows the higher percentile of time to detect a deadlock is higher when there are more transfer operations in the workload.

6.2.5 Experiment 5: Scalability

In the last experiment, the effect of additional StateFun workers on the performance is evaluated. These experiments are performed with StateFun workers with 2 CPUs. Figure 6.9 shows the results.

Figure 6.9 shows that sagas' scalability efficiency from 1 to 5 workers is 90% consistently for any proportion of transactions. This result shows that sagas can leverage some of the parallelism provided by StateFun and Flink.

For two-phase commit transactions, the scalability efficiency from 1 to 5 workers starts at 87% at 10% `transfer` operations and drops to 75% for 100% `transfer operations`. This result shows that two-phase commit transactions leverage some of the parallelism provided by StateFun and Flink.

The difference in the scalability efficiency between sagas and two-phase commit is explainable because two-phase commit functions require an additional `COMMIT` message for successful transactions. This `COMMIT` message has to go through the network, decreasing the performance for more workers. The sagas coordinator does not need to send compensating messages for successful sagas. It is interesting for future research to evaluate the scalability when more sagas fail, so more compensating messages over network are required.

Chapter 7

Conclusion

To conclude this thesis, this chapter attempts to answer the research questions posed in the introduction.

1. What SFaaS implementations exist today, and what are their data correctness guarantees?

Chapter 3 discusses four existing implementations of SFaaS systems. Beldi is introduced in October 2020 and implements fault tolerance exactly-once invocations and transactions using existing serverless services such as AWS Lambda and DynamoDB. The exactly-once guarantees only span those two services.

Cloudstate and StateFun also provide fault-tolerant exactly-once semantics but only support transactions on the scope of a single function instance (or *linearizability* of function invocations). However, the exactly-once semantics of Cloudstate and StateFun do span several different data sources and sinks.

Cloudstate also provides a second model that replicates state for availability but only supports *causal consistency*. This model is limited to conflict-free data types to avoid any data loss. Cloudburst provides a more flexible programming model than Cloudstate and StateFun and a weaker consistency level (up to *causal consistency*).

Besides Beldi, there is no notion of transactions in any of the current SFaaS systems.

2. What are shared features of existing SFaaS implementations, and how can they be used to find a conceptual solution for transactions that can be implemented in SFaaS systems?

In chapter 4.1, a standard model was defined where stateful functions encapsulate some state. Multiple instances of this stateful function may exist simultaneously, each encapsulating its own state. These function instances are comparable to rows in a database or keys in a key-value store. This model is used by StateFun and Cloudstate, as well as Azure Durable Functions.

Chapter 4 also identifies two other features that StateFun and Cloudstate (and Azure Durable Functions) share and provide fundamental building blocks to implement transactions; fault-tolerant exactly-once guarantees and linearizable function invocations for each function instance. These two features are the exact same assumptions the only current implementation of transactions, Beldi, makes. Beldi relies on a linearizable key-value store with atomic operations for a single key and implements fault-tolerant exactly-once messaging before implementing transactions on top of that.

The coordinator functions that this thesis introduces rely on SFaaS systems where function instances encapsulate state and that provide exactly-once guarantees and linearizable invocations per function instance.

3. What programming model can be used to implement transactions in SFaaS systems intuitively?

To implement transactions in SFaaS, this thesis introduces using simple coordinator functions. Coordinator functions themselves are stateless but can be used to define transactions with an arbitrary number of function invocations to other function instances. The coordinator functions statically compute the definition of the transaction or sagas up-front based on an input message.

Besides introducing coordinator functions, the programming model of regular stateful functions is extended to allow them to fail explicitly based on their state. The Python API of StateFun is extended with a `FunctionInvocationException` to do this.

Listing 5.2 shows the complete API for coordinator functions and the addition to the regular functions.

4. What isolation guarantees can be provided for transactions in SFaaS systems, and how do these affect performance?

This thesis evaluates implementations of transactions at the *read uncommitted* level using sagas and at *serializable* level using two-phase commit and two-phase locking in chapter 6.

The introduction of transactional functionality in StateFun decreases the performance of non-transactional workloads using a uniform access distribution by at most 20% for few keys (100), lowering to 10% for more keys (2000+).

The performance of sagas is lower than expected but still acceptable considering the extra functionality provided by the simple coordinator function programming model. The performance of *serializable* transactions is very low for few keys. This low performance is easily explained by the locking required and the lack of batching. Interestingly, *serializable* transactions perform comparatively with sagas on systems with more keys. It should be taken into account that a uniform access distribution is used.

Experiment 3 shows the overhead of compensating function invocations required to rollback sagas. This means the performance difference between serializable transaction and sagas is smaller for workloads with many transactions and where transactions are likely the fail.

The scaling efficiency of StateFun with coordinator functions is reasonable. The system can leverage the underlying parallelization of StateFun and Flink. Sagas achieve a consistent scaling efficiency of 90% for 1-5 workers. *Serializable* transactions perform less well, the scaling efficiency for 1-5 workers drops from 87% for a workload with 10% transaction to 75% for fully-transactional workloads.

5. How can SFaaS systems, with and without transactions, be evaluated?

There was no very-well suited benchmark found to evaluate SFaaS systems. The lack of a standard benchmark is likely because SFaaS can be applied to a vast variety of use cases. The evaluation in this thesis is based on an extension of *YCSB*. *YCSB* is a straightforward benchmark definition that can provide some intuition on the system's behavior since the results are easy to interpret. However, it does not cover any more complex use cases.

For this thesis, a small addition to the YCSB benchmark was made; a `transfer` operation was added. This `transfer` operation represents a transaction across two keys (or function instances). The `transfer` operation is used to evaluate the performance of transactions. The addition of `transfer` operations is not enough to make YCSB suitable to evaluate a SFaaS system fully but does provide basic intuition on the performance of transactions.

Chapter 8

Discussion

This chapter discusses the contributions of this thesis (section 8.1) and how they relate to other research in the same area (3). It identifies avenues for future research (section 8.3) and presents ideas to improve Flink StateFun in the near-future based on concepts introduced in this thesis (section 8.4).

8.1 Contributions

This thesis introduces coordinator functions to implement *serializable* transactions and sagas workflows on SFaaS systems and an accompanying implementation in StateFun.

The introduction of coordinator functions and the additional logic and state management associated with them introduces an affordable overhead of 10% on non-transactional workloads. The introduced programming model is simple to use. It is close to orchestrator functions already implemented in Azure Durable Functions and proven sound. The isolation guarantees provided by coordinator functions are also easily understandable for developers. The sagas' performance is solid; even though sagas introduce overhead, this is acceptable as it allows and simplifies significantly more complex use cases. The performance of *serializable* transaction is less reliable since it is dependent on locking and can not leverage the batching mechanism of StateFun. Still, this is acceptable due to the strong guarantees *serializable* transactions provide and the simplicity of implementing them using the introduced programming model. Experiment 3 shows that sagas introduce some additional overhead for failed transaction, this means *serializable* transaction become even more viable for workloads where many transactions are expected to fail. The scalability of both sagas and serializable transaction is acceptable. Understandably, the scalability efficiency decreases as the operations access state across partitions requiring additional network communication.

The implementation of coordinator functions is based on the remote function deployment style provided by StateFun. This fits the direction of Flink StateFun well since the maintainers have recently announced they are moving to a *remote functions first* design[1] for their upcoming release and the future.

The benchmark introduced and used in this thesis was sufficient to evaluate the implemented coordinator function's behavior. However, it is not yet suited to evaluate general SFaaS systems. It is currently unclear how SFaaS will behave and what they will provide to developers in the future, as it is still a very early stage. This uncertainty becomes apparent when comparing StateFun and Cloudstate with Cloudburst that has completely different semantics.

[1]http://apache-flink-mailing-list-archive.1008284.n3.nabble.com/DISCUSS-Releasing-Stateful-Functions-3-0-0-td49708.html

8.2 Related work

Besides the SFaaS systems presented in chapter 3, there is other promising research related to this thesis topic.

Firstly, Calvin (Thomson et al., 2012) was already introduced in 2012. It is an example of an approach that is optimized both for performance and data correctness. Calvin, or solutions similar to Calvin, can solve similar problems around data correctness and performance that coordinator functions aim to solve in this thesis and may perform better. The strategy of optimizing for performance and data correctness up-front may lead to more promising results than adapting a system optimized for performance to implement data correctness (as done in this thesis).

Secondly, research done on Remote Direct Memory Access (RDMA) is also interesting. New hardware and low-level computer engineering developments enable traditional relational databases to scale significantly better (Barthels et al., 2019). Previously, traditional relational databases were thought not to be the solution to big-data-related problems due to limited scalability. However, this new technology may enable the use of relational databases for big data workloads. Relational databases are a mature technology with decades of research behind them to make them feature-rich and easy-to-use. Also, many developers are familiar with relational databases, meaning that if relational databases can scale efficiently to handle bigger workloads, they will (again) be a good option for state management. Researchers have also used RDMA to improve stream and batch processing systems such as Apache Spark and Apache Storm (MacArthur and Russell, 2014; Zhang et al., 2021; Lu et al., 2014).

8.3 Future work

After writing this thesis, I can identify four exciting avenues to continue research.

Firstly, adopting a system that is optimized for performance to provide data correctness guarantees, as done in this thesis, may not always lead to good results. It seems more natural to design a system with both data correctness and performance in mind from the start rather than to optimize for performance first and handle data correctness in hindsight. An example of this is Calvin. More research could be done to design scalable and cloud-native systems and architectures, starting from a data correctness perspective.

Secondly, Cloudburst's approach and the RISE Lab's research developing Cloudburst take is to find applications that can run without strong consistency levels and coordination for improved performance. They formalized a set of applications that can run without coordination (CALM theory (Ameloot et al., 2015)). They could expand on this and try to find ways to model applications to run without coordination. This sounds close to impossible but would be very interesting if it leads to usable results. Another opportunity from taking this approach is to optimize first for workloads that can run completely coordination-free, and after that, define a way to implement parts of a workload that require coordination within that system. The result could look similar to how Cloudstate offers two different models (event-sourced and CRDT) depending on the data consistency requirements.

Thirdly, continuing research on RMDA to improve relational databases for big data workloads can lead to valuable results. It is also interesting to research how RMDA can optimize big-data processing architectures such as stream processing systems.

Lastly, a direction of future research can be to define the goals of SFaaS better. To better understand the goals and challenges of SFaaS, a range of sample applications can be developed using SFaaS, modeling different use cases. The performance of different existing SFaaS may be compared on these sample applications. The raw performance is interesting to compare and their semantics, ease-of-use, and fit for the specific use case. This analysis should provide insight into how SFaaS systems should behave. Another approach is to address the lack of meaningful benchmarks for these systems. The development of benchmarks forces us to think of generic features and behavior desirable in SFaaS systems and structure and validate past ideas (such as the coordinator functions introduced in this thesis).

8.4 Future Improvements for Flink StateFun

Flink StateFun is an exciting system with powerful primitives, such as linearizable operations on function instances and exactly-once guarantees. The concept of the event-driven database as described in section 5.1.4 seems promising. However, there are few features currently in Flink StateFun, which may prevent users from using it. Most importantly, features related to access to data and especially access to data across multiple function instances are missing. Section 6.2.1 already mentions a possible improvement; read access to state of an embedded representation of a remote function without requiring to go over the network to the remote function first. This section presents more ideas to make Flink StateFun more usable.

8.4.1 Index Functions

Currently, function instances in StateFun are only reachable through their exact address, including the ID. If the ID is unknown to the developer, the state encapsulated in the function instance can not be reached. *Index functions* may solve access to single function instances. Similar to coordinator functions, index functions are stateful functions with a specified purpose and programming model. Index functions exist based on a *function type*. This index function can be messaged to find the IDs of all function instances of that type. Another purpose of an index function may be to find a function instance's ID based on a secondary index. The developer would have to write the code to find the secondary index based on the Protobuf state of function instances, i.e., a simple map function. The index function may either respond with the ID of the function instance with the desired secondary index or forward an invocation to the correct function instance(s). This approach does require all function instances of a type to send their state to the index function after every write. Multiple index functions may exist per function type for different secondary instances.

8.4.2 Aggregate Functions

Another problem in Flink StateFun is aggregating state encapsulated in multiple function instances. This problem may be solved with pre-defined *agggregate functions*. Developers may write an aggregate function that receives all state changes from the function instances of some function type and combines them to an aggregate value.

8.4.3 Migrate Tasks

Lastly, a problem in Flink StateFun is updating remote functions. Remote functions can be updated independently from StateFun by deploying a new function behind the load balancer's address that StateFun has. However, this does not update the state of the function instances. So if the updated remote function requires a different state type, logic needs to be implemented in the remote function to check the type of the state and possibly migrate the state to the new type before executing the updated function. This form of migration has to be done indefinitely for all function updates that change the state type since the developer is never sure whether the state type of all function instances changed in the StateFun cluster.

Migrate tasks could be introduced to solve this problem. The migrate task requires the developer to write a map function to transform the old state type into a new state type. This migration function would be applied to all function instances of the function type in StateFun. The remote function now only has to support a single state type (or two for a short period when the migrate task is running). This approach requires the migrate task to have access to all IDs of some function type. That could work with the *index function*.

Appendix A

Cloudstate programming model

This appendix shows the lengthy programming model of Cloudstate. It shows an example of an event-sourced entity. This code is directly taken from the documentation of Cloudstate[1].

Firstly, listing A.1 shows the gRPC contract that describes the entity.

Secondly, listing A.2 shows the actual code of the function.

```
// This is the public API offered by the shopping cart entity.
syntax = "proto3";

import "google/protobuf/empty.proto";
import "cloudstate/entity_key.proto";
import "google/api/annotations.proto";
import "google/api/http.proto";

package com.example.shoppingcart;

message AddLineItem {
    string user_id = 1 [(.cloudstate.entity_key) = true];
    string product_id = 2;
    string name = 3;
    int32 quantity = 4;
}

message RemoveLineItem {
    string user_id = 1 [(.cloudstate.entity_key) = true];
    string product_id = 2;
}

message GetShoppingCart {
    string user_id = 1 [(.cloudstate.entity_key) = true];
}

message LineItem {
    string product_id = 1;
    string name = 2;
    int32 quantity = 3;
}

message Cart {
    repeated LineItem items = 1;
}

service ShoppingCart {
    rpc AddItem(AddLineItem) returns (google.protobuf.Empty) {
        option (google.api.http) = {
            post: "/cart/{user_id}/items/add",
            body: "*",
        };
    }

    rpc RemoveItem(RemoveLineItem) returns (google.protobuf.Empty) {
        option (google.api.http).post = "/cart/{user_id}/items/{product_id}/remove";
    }
```

[1] https://pypi.org/project/cloudstate/

```

    rpc GetCart(GetShoppingCart) returns (Cart) {
        option (google.api.http) = {
          get: "/carts/{user_id}",
          additional_bindings: {
            get: "/carts/{user_id}/items",
            response_body: "items"
          }
        };
    }
}
```

LISTING A.1: gRPC describing a sample event-sourced entity

```python
from dataclasses import dataclass, field
from typing import MutableMapping

from google.protobuf.empty_pb2 import Empty

from cloudstate.event_sourced_context import EventSourcedCommandContext
from cloudstate.event_sourced_entity import EventSourcedEntity
from shoppingcart.domain_pb2 import (Cart as DomainCart, LineItem as DomainLineItem, ItemAdded, ItemRemoved)
from shoppingcart.shoppingcart_pb2 import (Cart, LineItem, AddLineItem, RemoveLineItem)
from shoppingcart.shoppingcart_pb2 import (_SHOPPINGCART, DESCRIPTOR as FILE_DESCRIPTOR)


@dataclass
class ShoppingCartState:
    entity_id: str
    cart: MutableMapping[str, LineItem] = field(default_factory=dict)


def init(entity_id: str) -> ShoppingCartState:
    return ShoppingCartState(entity_id)


entity = EventSourcedEntity(_SHOPPINGCART, [FILE_DESCRIPTOR], init)


def to_domain_line_item(item):
    domain_item = DomainLineItem()
    domain_item.productId = item.product_id
    domain_item.name = item.name
    domain_item.quantity = item.quantity
    return domain_item


@entity.snapshot()
def snapshot(state: ShoppingCartState):
    cart = DomainCart()
    cart.items = [to_domain_line_item(item) for item in state.cart.values()]
    return cart


def to_line_item(domain_item):
    item = LineItem()
    item.product_id = domain_item.productId
    item.name = domain_item.name
    item.quantity = domain_item.quantity
    return item


@entity.snapshot_handler()
def handle_snapshot(state: ShoppingCartState, domain_cart: DomainCart):
    state.cart = {domain_item.productId: to_line_item(domain_item) for domain_item in domain_cart.items}


@entity.event_handler(ItemAdded)
def item_added(state: ShoppingCartState, event: ItemAdded):
    cart = state.cart
    if event.item.productId in cart:
        item = cart[event.item.productId]
        item.quantity = item.quantity + event.item.quantity
    else:
        item = to_line_item(event.item)
        cart[item.product_id] = item


@entity.event_handler(ItemRemoved)
def item_removed(state: ShoppingCartState, event: ItemRemoved):
    del state.cart[event.productId]


@entity.command_handler("GetCart")
def get_cart(state: ShoppingCartState):
    cart = Cart()
    cart.items.extend(state.cart.values())
    return cart


@entity.command_handler("AddItem")
def add_item(item: AddLineItem, ctx: EventSourcedCommandContext):
    if item.quantity <= 0:
        ctx.fail("Cannot add negative quantity of to item {}".format(item.productId))
    else:
        item_added_event = ItemAdded()
        item_added_event.item.CopyFrom(to_domain_line_item(item))
        ctx.emit(item_added_event)
    return Empty()


@entity.command_handler("RemoveItem")
def remove_item(state: ShoppingCartState, item: RemoveLineItem, ctx: EventSourcedCommandContext):
    cart = state.cart
    if item.product_id not in cart:
        ctx.fail("Cannot remove item {} because it is not in the cart.".format(item.productId))
    else:
        item_removed_event = ItemRemoved()
        item_removed_event.productId = item.product_id
        ctx.emit(item_removed_event)
    return Empty()
```

LISTING A.2: Python code implementing the business logic of the function

Bibliography

Ahamad, Mustaque et al. (1995). "Causal memory: Definitions, implementation, and programming". In: *Distributed Computing* 9.1, pp. 37–49.

Akhter, Adil, Marios Fragkoulis, and Asterios Katsifodimos (Aug. 2019). "Stateful functions as a service in action". In: *Proceedings of the VLDB Endowment* 12, pp. 1890–1893. DOI: 10.14778/3352063.3352092.

Akkus, Istemi Ekin et al. (2018). "{SAND}: Towards High-Performance Serverless Computing". In: *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 923–935.

Ameloot, Tom J. et al. (Dec. 2015). "Weaker Forms of Monotonicity for Declarative Networking: A More Fine-Grained Answer to the CALM-Conjecture". In: *ACM Trans. Database Syst.* 40.4. ISSN: 0362-5915. DOI: 10.1145/2809784. URL: https://doi.org/10.1145/2809784.

Armbrust et al. (Jan. 2009). *Above the Clouds: A Berkeley View of Cloud Computing*.

Attar, R., P. A. Bernstein, and N. Goodman (1984). "Site Initialization, Recovery, and Backup in a Distributed Database System". In: *IEEE Transactions on Software Engineering* SE-10.6, pp. 645–650. DOI: 10.1109/TSE.1984.5010293.

Bailis, Peter and Kyle Kingsbury (2014). "The network is reliable: An informal survey of real-world communications failures". In: *Queue* 12.7, pp. 20–32.

Bailis, Peter et al. (Nov. 2013). "Highly Available Transactions: Virtues and Limitations". In: *Proc. VLDB Endow.* 7.3, 181–192. ISSN: 2150-8097. DOI: 10.14778/2732232.2732237. URL: https://doi.org/10.14778/2732232.2732237.

Baldini, Ioana et al. (2017a). "Serverless Computing: Current Trends and Open Problems". In: *CoRR* abs/1706.03178. arXiv: 1706.03178. URL: http://arxiv.org/abs/1706.03178.

— (2017b). "Serverless Computing: Current Trends and Open Problems". In: *CoRR* abs/1706.03178. arXiv: 1706.03178. URL: http://arxiv.org/abs/1706.03178.

Barthels, Claude et al. (Sept. 2019). "Strong Consistency is Not Hard to Get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores". In: *Proc. VLDB Endow.* 12.13, 2325–2338. ISSN: 2150-8097. DOI: 10.14778/3358701.3358702. URL: https://doi.org/10.14778/3358701.3358702.

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

Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. ISBN: 0-201-10715-5.

Carbone, Paris et al. (2015). "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4.

Carbone, Paris et al. (Aug. 2017). "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12, 1718–1729. ISSN: 2150-8097. DOI: 10.14778/3137765.3137777. URL: https://doi.org/10.14778/3137765.3137777.

Cattell, Rick (May 2011). “Scalable SQL and NoSQL Data Stores”. In: 39.4, 12–27. ISSN: 0163-5808. DOI: 10.1145/1978915.1978919. URL: https://doi.org/10.1145/1978915.1978919.

Coffman, E. G., M. Elphick, and A. Shoshani (June 1971). “System Deadlocks”. In: *ACM Comput. Surv.* 3.2, 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588. URL: https://doi.org/10.1145/356586.356588.

Cooper, Brian F et al. (2010). “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154.

Corbett, James C et al. (2013). “Spanner: Google’s globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3, pp. 1–22.

Dey, Akon et al. (2014). “YCSB+T: Benchmarking web-scale transactional databases”. In: *2014 IEEE 30th International Conference on Data Engineering Workshops*. IEEE, pp. 223–230.

Ewen, Stephan (2020). *Stateful Functions 2.0 - An Event-driven Database on Apache Flink*. URL: https://flink.apache.org/news/2020/04/07/release-statefun-2.0.0.html.

Eyk, Erwin van et al. (2017). “The SPEC Cloud Group’s Research Vision on FaaS and Serverless Architectures”. In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC ’17. Las Vegas, Nevada: Association for Computing Machinery, 1–4. ISBN: 9781450354349. DOI: 10.1145/3154847.3154848. URL: https://doi.org/10.1145/3154847.3154848.

Freels, Matt (2018). *FaunaDB: An architectural overview*.

Garcia-Molina, Hector and Kenneth Salem (1987). “Sagas”. In: *ACM Sigmod Record* 16.3, pp. 249–259.

Gilbert, Seth and Nancy Lynch (June 2002). “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *SIGACT News* 33.2, 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

Gray, James N (1978). “Notes on data base operating systems”. In: *Operating Systems*. Springer, pp. 393–481.

Gupta, Prashant, Arumugam Seetharaman, and John Rudolph Raj (2013). *The usage and adoption of cloud computing by small and medium businesses*.

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

Hellerstein, Joseph M. and Peter Alvaro (2019). *Keeping CALM: When Distributed Consistency is Easy*. arXiv: 1901.01930 [cs.DC].

Hellerstein, Joseph M. et al. (2018). “Serverless Computing: One Step Forward, Two Steps Back”. In: *CoRR* abs/1812.03651. arXiv: 1812.03651. URL: http://arxiv.org/abs/1812.03651.

Hendrickson, Scott et al. (2016). “Serverless computation with openlambda”. In: *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

Herlihy, Maurice P. and Jeannette M. Wing (July 1990). “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3, 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972.

Hewitt, C., P. B. Bishop, and R. Steiger (1973). “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *IJCAI*.

Jonas, Eric et al. (2019). “Cloud programming simplified: A berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383*.

Katsifodimos, Asterios and Marios Fragkoulis (2019). "Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications." In: *EDBT*, pp. 682–685.

Lamport, Leslie (July 1978). "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7, 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

— (1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9, pp. 690–691. DOI: 10.1109/TC.1979.1675439.

— (1986). "On interprocess communication". In: *Distributed computing* 1.2, pp. 86–101.

— (May 1998). "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2, 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229.

Lu, Xiaoyi et al. (Aug. 2014). "Accelerating Spark with RDMA for Big Data Processing: Early Experiences". In: pp. 9–16. DOI: 10.1109/HOTI.2014.15.

MacArthur, P. and R. D. Russell (2014). "An Efficient Method for Stream Semantics over RDMA". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 841–851. DOI: 10.1109/IPDPS.2014.91.

Ongaro, Diego and John Ousterhout (2014). "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association, 305–320. ISBN: 9781931971102.

Raab, Francois (1993). "TPC-C - The Standard Benchmark for Online transaction Processing (OLTP)." In: *The Benchmark Handbook*. Ed. by Jim Gray. Morgan Kaufmann. ISBN: 1-55860-292-5. URL: http://dblp.uni-trier.de/db/books/collections/gray93.html#Raab93.

Ren, Kun, Alexander Thomson, and Daniel J Abadi (2014). "An evaluation of the advantages and disadvantages of deterministic database systems". In: *Proceedings of the VLDB Endowment* 7.10, pp. 821–832.

Shapiro, Marc et al. (Jan. 2011). "A comprehensive study of Convergent and Commutative Replicated Data Types". In:

Sreekanti, Vikram et al. (2020). "Cloudburst". In: *Proceedings of the VLDB Endowment* 13.12, 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836. URL: http://dx.doi.org/10.14778/3407790.3407836.

Tai, Tzu-Li (Gordon) (2020). *Stateful Functions: Polyglot Event-Driven Functions for Stateful Distributed Applications*. Ververica. URL: https://www.youtube.com/watch?v=tuSylBadNSo.

Terry, Douglas B et al. (1994). "Session guarantees for weakly consistent replicated data". In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, pp. 140–149.

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.

Wu, C. et al. (2018). "Anna: A KVS for Any Scale". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 401–412. DOI: 10.1109/ICDE.2018.00044.

Zhang, Haoran et al. (Nov. 2020). “Fault-tolerant and transactional stateful serverless workflows”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, pp. 1187–1204. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran.

Zhang, Ziyu et al. (2021). “RDMA-Based Apache Storm for High-Performance Stream Data Processing”. In: *International Journal of Parallel Programming*, pp. 1–14.