TÉCNICO LISBOA

An Architecture to Offer Transactional Strong Consistency for FaaS Applications

João Rafael Pinto Soares

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. José Orlando Roque Nascimento Pereira

November 2021

Acknowledgments

I would like to thank my family for all their help, support and guidance throughout these years, without whom this work would not be possible.

I would also like to thank my dissertation supervisor Prof. Luís Rodrigues for his insight, support and sharing of knowledge that has made this Thesis possible. I would also like to thank the members of the committee for their constructive feedback especially to Professor José Orlando Pereira for comments on a previous version of this work. A personal thank you to Taras Lykhenko and Cláudio Correia for their help, support, and discussions throughout this thesis

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

Function-as-a-Service (FaaS) is a relatively recent paradigm supported by many cloud providers that supports the execution of applications without prior allocation of servers. Applications are written as a composition of stateless functions, organized in a graph. Different functions may execute in different servers, that are provisioned automatically by the cloud provider. Functions may read and write from/to stable storage using a storage service of their choice. For cost/efficiency reasons, most FaaS applications use storage services that cannot provide strong consistency to functions executing in different servers. In this thesis we study efficient ways of extending a weakly consistent data store with additional services that can offer transactional support and strong consistency (namely, snapshot isolation) to FaaS applications. Some previous works that aim at achieving the same goals force all storage read-/write requests to be forwarded to one or more consistency servers, that are responsible for ensuring that a consistent version of the data is returned to the functions. In this work we propose and evaluate a different strategy, where functions read optimistically from storage and use the consistency servers to obtain metadata that is used to check if the version returned by the storage system is consistent. This strategy decreases the load on the consistency server, improving the scalability of the system. Our experimental evaluation shows that our solution offers $1.4\times$ higher throughput then alternative protocols, while using only 5% of their resources.

Keywords

FaaS; Consistency; Transactions; Snapshot Isolation; Cloud.

Resumo

Function-as-a-Service (FaaS) é um paradigma relativamente recente, suportado por vários fornecedores de serviços na nuvem, que permite executar aplicações na nuvem sem obrigar a uma reserva prévia de servidores. Neste modelo, as aplicações são escritas na forma de uma composição de funções que não mantêm estado, organizadas num grafo de execução. Diferentes funções podem ser executadas por diferentes servidores, escolhidos de forma automática pelo fornecedor do serviço. As funções podem ler e escrever em memória persistente, usando o serviço de armazenamento que considerem mais adequado. Por razões de custo/benefício, a grande maioria das aplicações FaaS usam serviços de armazenamento que não têm capacidade de oferecer garantias de coerência forte para funções que se executem em servidores distintos. Nesta dissertação estudamos mecanismos eficientes para estender um serviço de armazenamento que oferece coerência fraca de forma a oferecer garantias transaccionais e coerência forte (nomeadamente, isolamento instantâneo) a aplicações FaaS. Alguns trabalhos anteriores requerem que todos os acessos ao serviço de armazenamento sejam encaminhados para um ou mais servidores de coerência. Neste trabalho propomos e avaliamos uma estratégia alternativa, em que os clientes lêm directamente os dados do armazenamento e usam os servidores de coerência para verificar se a versão retornada pelo armazenamento é coerente. Esta estratégia reduz a carga dos servidores de coerência e dá mais capacidade de escala ao sistema. Através de uma avaliação experimental, mostramos que a nossa solução proposta oferece um débito $1.4\times$ superior ao apresentado pelos protocolos alternativos, utilizando 5% dos recursos dos mesmos.

Palavras Chave

FaaS; Coerência; Transações; Isolamento Instantâneo; Nuvem

Contents

List of Figures

List of Tables

Acronyms

CC	Cloud Computing
FaaS	Function-as-a-Service
TCC	Transaction Causal Consistency
GCF	Google Cloud Functions
MW	Monotonic Writes
MR	Monotonic Reads
RYW	Read your Writes
PRAM	Pipelined Random Access Memory
WFR	Write Follows Read
CC	Causal Consistency
CC+	Causal+ Consistency
RR	Repeatable Read
MAV	Monotonic Atomic View
RA	Read Atomic
SI	Snapshot Isolation
1SR	Serializability
Strong 1SR	Strict Serializability
HAT	Highly Available Transactions
DVC	Dependency Vector Clock

CVC	Commit Vector Clock
SVC	Snapshot Vector Clock
MCT	Maximum Committed Timestamp
LST	Local Stable Time
RST	Remote Stable Time
LTM	Local Transaction Managers
TCS	Transaction Commit Set
IC	Intent Colector
RTT	Round Trip Time
CRDT	Conflict-free Replicated Data Type

1

Introduction

Contents

The Serverless Computing paradigm, also known as Function-as-a-Service (FaaS), is a recent paradigm supported by many cloud providers. This paradigm allows programmers to run their applications in the cloud without allocating servers beforehand. To use this paradigm, programmers must code their applications as a composition of stateless functions, organized in an execution graph. Functions are executed in servers chosen automatically by the cloud provider, without any client intervention. Clients are billed based on the computing power effectively used, in opposition to reservation-based models, where clients are billed proportionally to the reserved time, regardless of the resource usage.

In this thesis, we address the problem of offering transactional storage access to Function-as-a-Service applications. For cost/efficiency reasons, most FaaS applications use storage services that cannot provide strong consistency to functions executing in different servers. In this thesis we study efficient ways to extend a weakly consistent data store with additional services that can offer transactional support and strong consistency (namely, snapshot isolation) to a FaaS applications composed of multiple functions that may execute in different workers.

1.1 Motivation

FaaS architectures disaggregates computational and storage layers, allowing for an independent and finer-grained elastic scaling of each layer. FaaS requires functions to be stateless for a better scaling of the computational layer. Thus, functions are required to use a storage layer in order to share state. In abstract, the FaaS paradigm does not restrict the type of storage service to be used by the programmers, which are free to choose a storage service that offers the consistency level required by the application. However, for cost/efficiency reasons, most FaaS applications use storage services that cannot provide strong consistency to functions executing in different servers. In particular, if one wants to obtain transactional guarantees, the different functions need to share a single transactional context, that needs to be exported by the storage service and passed from function to function. As a result, many FaaS applications rely on weakly consistent data access to storage. This may lead to applications observing intermediate and/or inconsistent states, producing undesirable results. In this thesis we address the problem of extending the FaaS architecture with support for strongly consistent transactional access to persistent storage, namely, by supporting consistency levels such as Snapshot Isolation or Strict Serializability.

A significant challenge in offering transactional support to FaaS applications consists in coordinating different workers in an efficient manner. Some previous works that aim at achieving similar goals force all storage read/write requests to be forwarded to one or more consistency servers, that are responsible for ensuring that a consistent version of the data is returned to the functions. This solution decreases the performance and the scalability of the FaaS system. In this work we propose and evaluate a different

strategy, where functions read optimistically from storage and use the consistency servers to obtain metadata that is used to check if the version returned by the storage system is consistent. We then propose two different techniques to handle inconsistent results from storage system, with the ultimate goal of reducing the load these consistency servers. Still, as previous works, the access to consistency servers is still required to commit updates. Our strategy decreases the load on the consistency servers, improving the scalability of the system.

1.2 Contributions

The thesis makes the following contribution:

- We propose an architecture, named FaaSSI, that supports strongly consistent transactions to graph function execution in FaaS architecture. FaaSSI guarantees the ACID properties of transactions, providing isolation properties in the form of Snapshot Isolation. The architecture supports a protocol to read consistent data versions from storage that is able to offer performance gains when compared to previous alternatives.

1.3 Results

This thesis has produced the following results:

- An implementation of FaaSSI, integrated using the state-of-the-art Stateful FaaS platform Cloudburst and an highly scalable weakly consistent storage Anna;
- An experimental evaluation of FaaSSI implementation, regarding its performance and scalability. We compare the performance of our protocols with the performance of other systems that aim at achieving similar goals.

1.4 Research History

This work is part of a broader effort of understanding the challenges of offering different consistency levels to FaaS applications. In particular, it extends previous work on offering Transaction Causal Consistency (TCC) to distributed storage systems [5]. In fact, a substantial part of the work performed in the context of this thesis involved the consolidation and evaluation of a system to offer TCC in FaaS, named *FaaSTCC*, initially designed by Taras Lykhenko, that was subsequently used as a framework to incorporate the ideas proposed in this thesis.

My contribution to the evaluation of *FaaSTCC* is reflected in the following paper:

- T. Lykhenko, R. Soares and L. Rodrigues. FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 22nd ACM/IFIP International Middleware Conference*, Online, December 2021.

Parts of the work described in this thesis have been published as:

- R. Soares and L. Rodrigues. Uma Arquitectura para Oferecer Garantias de Coerência forte a Aplicações FaaS. In *Actas do décimo segundo Simpósio de Informática (Inforum)*, Lisboa, Portugal, September 2021.

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) as part of the projects with references UID/CEC/50021/2019 and COSMOS (financed by the OE with ref. PTDC/EEICOM/29271/2017 and by Programa Operacional Regional de Lisboa in its FEDER component with ref. Lisbon-01-0145-FEDER-029271). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

1.5 Structure of the Document

The rest of the thesis is organized as follows: Chapter 2 introduces the key concepts relevant to our work and provides an overview of the related work; Chapter 3 describes the architecture of FaaSSI; Chapter 4 describes the results from our experimental evaluation, where we compare the performance of FaaSSI against previous systems that aim at similar goals; finally, Chapter 5 concludes the thesis and provides directions for future work.

2

Related Work

Contents

In this chapter we go over the basis of FaaS and previous work that has faced similar challenges. Section 2.1 starts by introducing the main FaaS platforms, with Section 2.2 presenting the main problems and challenges in providing strong consistency guarantees to FaaS applications. We then make an overview of the main consistency models that have been proposed to augment the efficiency and availability of large-scale distributed storage systems in Section 2.3. Finally, Section 2.4 presents a survey of the most relevant system that support some of these consistency models and discuss the challenges of adapting these solutions to the FaaS environment.

2.1 Serverless Computing

We start by making an introduction to FaaS systems and to the consistency guarantees offered to programmers in Serverless Computing environments.

2.1.1 Function-as-a-Service

Function as a Service is a model that is now supported by the main cloud providers, such as Amazon, Google and Microsoft, each with its own platform, namely AWS Lambda [6], Google Cloud Functions (GCF) [7] and Microsoft Azure Functions [8]. These platforms support functions written in different programming languages, such Python, Java, or C#, among others, that can invoke other services provided by the cloud provider [9], such as Amazon S3 [10] and DynamoDB [11] for storage and Amazon Step Functions [12] for function composition and execution, GCF functions can use the Google Cloud services [13], and Azure functions can use other Azure services [14]. When using these services, the ease of programming comes with some drawbacks: not only the code becomes locked to a particular platform but also the global performance of an application becomes constrained by the functional and non-functional properties of the underlying services. For instance, S3 is known to perform well with large items but incurs big latency overheads [3, 15], and can offer poor performance when executing transactions that read and write small objects. Due to this reason, there is still on-going research that attempts to developed services that can be more adequate for the support of FaaS applications. For instance, Cloudburst [16] is a FaaS system that aims at supporting the development of stateful FaaS programs that uses a low latency autoscaling key-value store (Anna [4]) and a cache co-located with the executors to reduce the latency in the execution of transactions. A more detailed overview of current FaaS services can be found in [17].

2.1.2 Data Consistency in Serverless Computing

The Serverless Computing model requires functions to be stateless in order to be fully disaggreated from the storage level. Applications that are required to maintain or access state must do so by interaction with a storage layer.

In theory, the programmer can chose any storage service that offers the consistency guarantees required by the application. Unfortunately, most storage services for the cloud offer little to no support to execute transactions that span multiple functions executing at different workers. In fact, even if the storage supports ACID properties, to offer these properties to FaaS applications requires support for distributed transactions, with a transactional context to be passed from function to function. Also, if ACID properties are implemented using pessimistic concurrency control, a FaaS function composition that aborts at any given function may leave items locked in the database, requiring complex mechanisms to monitor the execution of the function compositions or timeouts that leave some items blocked for potentially long periods.

Due to these challenges, most FaaS environments do not offer support for transactions that span multiple functions. Instead, they encourage users to let each function to execute separate sub-transactions and then resort to programming patterns that can offer some global consistency properties even if sub-transactions are executed independently from each other. For example, one way to offer transactional support to AWS Step Functions [12] is to use the Saga pattern [18], as suggested in [19]. This pattern avoids partial transactions by orchestrating a set of compensation actions in case of function failure. However, as these functions are not running as a whole ACID transaction but, instead, as a single ACID transaction per function, this sub-transactions may read inconsistently from one another, which may lead the application to undesirable states.

The data consistency observed by applications is the result of the interaction between the computing layer and the storage layer, which can be supported by an intermediate helper layer, as illustrated in Figure 2.1. In the following, we discuss how each of these layers can contribute to enforce data consistency for FaaS applications.

2.1.2.1 Computational Layer

The computational layer is mostly comprised of computing nodes running Executor processes, that are in charge of executing a requested function. One strategy to offer data consistency to functions consists in augmenting the computing layer with a caching service. When a function needs to access the storage layer, the call is intercepted by the caching service that, in turn, ensures that a consistent version is returned to the function. The caching service not only allows to provide data consistency but also has the potential for improving the performance of the system, as in many cases clients can be directly served by the cache. Note however, that when there is a cache miss, the caching service may need to

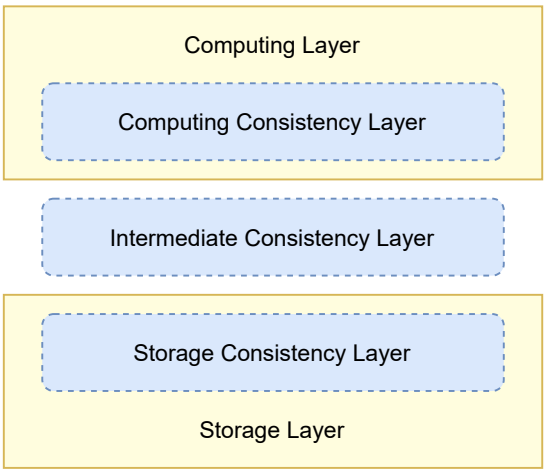


Figure 2.1: Consistency layers. Yellow boxes represent the main FaaS layers, blue boxes are the optional. consistency layers

contact the storage service multiple times in order to obtain a consistent version of the data. A significant challenge in this approach is that functions that execute on behalf of the same transaction must carry metadata to support the coordination of multiple caches. Because different functions are executed in different compute nodes, intra-cache consistency is not enough to ensure the consistency of the entire transaction: inter-cache consistency needs also to be ensured. An architecture that uses this approach is Hydrocache [3].

2.1.2.2 Storage Layer

The storage layer is comprised of multiple storage nodes. Typically, the data set is partitioned and different sets of nodes are responsible for storing different partitions. Furthermore, each partition is replicated. It is important to note that for most cases, it is not enough to use a storage system that implements the desired consistency level to obtain said level. This is because in many systems, data consistency is guaranteed to each client in isolation, while in FaaS data consistency needs to be ensured across multiple functions that execute on behalf of the same transaction. For example, Amazon S3 [10] offers linearizable writes but it does not support transactional support, and while DynamoDB [11] offers read and write transactions, these are not supported across multiple functions, leading to different transactions executing on each function possibly reading inconsistent versions from each other.

Another strategy to increase the consistency level provided to FaaS applications is to augment the cloud storage service with helper services that allow multiple clients to share session guarantees. For example, systems like Wren [20] that utilize a snapshot identifier to read consistent values can be extended to expose these snapshot identifiers to clients, that can be subsequently passed from one client to another, allowing multiple clients to read from a consistent snapshot, as presented in FaaSTCC [21].

2.1.2.3 Intermediate Layer

It is also possible to enforce data consistency by using an intermediate layer that sits between the Computational and Storage layer. This approach has the advantage of avoiding changes to the cloud layers, that can be treated as a black-box. The layer intercepts request from the computing nodes and serves them after obtaining a consistent version. For this purpose, the intermediate layer may need to keep a copy of the application data or, at least, metadata regarding the versions kept by the storage layer [15]. Similar approaches have been used for cloud systems like [22,23]. The intermediate layer can be partitioned and/or replicated to allow an higher scalability; in this case, replicas of the intermediate layer need to execute coordination protocols to ensure the desired consistency level. A disadvantage of this approach is that it consumes additional computation and memory resources.

2.1.2.4 Combining Multiple Layers

Multiple layers can be combined to efficiently guarantee stronger consistency levels. For instance, caching services on the computational layer can be combined with services provided by the intermediate or storage layer to allow for faster data access, as done in [24]. Another example is leveraging a library on the computational layer with extensive logging to the storage layer to obtain and maintain locks [25], possible as long as the storage layer allows to execute atomic operations on sufficiently large data items (the data item may include object values and identifiers, information regarding locks hold by the application, transactional metadata, and information of past writes). Also, implementing a wrapper for stronger consistency on the storage layer can help the intermediate layer to fetch a consistent version quicker when using a weaker consistent storage [22].

In summary, each layer implementation brings particular advantages and disadvantages. The computing layer supports fast access times but requires large amounts of metadata to be transferred between nodes and suffers from large latency overheads when there is a cache miss. The storage layer enforces stronger consistency but requires coordination among storage nodes and may not be enough to serve consistent data to function executing on different compute nodes. Finally, the intermediate layer enforces stronger consistency, independently of other layers, which allows for more portable solutions at the cost of additional resource consumption.

2.2 Consistency in FaaS

As noted before, current FaaS platforms provide little or no support for the execution of transactions that span multiple functions. Due to this limitation, recent research work has addressed the implementation of

transactional support in FaaS environments [3, 15, 25], wrapping function composition on a transactional layer providing ACID properties. The challenge is to design schemes that can support transactions without imposing a large overhead.

Bailis et al. have used the term *Highly Available Transactions (HAT)* [1] to characterize transactional implementations that allow client to make progress in face of network partitions and/or other slow clients. Strongly consistent transactions are not HAT, because they require the execution of consensus and may block in the presence of network partitions. This, however, may not be a significant limitation when transactions are executed in a single datacenter. In fact, studies on data center network partition show that "... the data center network exhibits high reliability with more than four 9's of availability for about 80% of the links ..." [26]. Thus, the practical drawback of non-HAT transactions can be small in a FaaS environment, while the benefits that result from the stronger semantics may be large for programmers.

We will now analyse some of the existing consistency levels following the analysis made in [1] and [2]. We have summarized the most relevant consistency levels defined in the previous works into Figure 2.2. We will follow our description by increasing strength consistency level, describing first consistency levels with regards to single objects and then to multiple objects.

We separate the consistency levels into three categories:

- HAT compliant, meaning that the transaction can be executed without any form of coordination;
- HAT-sticky compliant, meaning that the transaction can be executed without any form of coordination as long as it sticks to a server/set of servers;
- Non-HAT compliant, meaning that they require a form of coordination between servers and a transaction may have to block in face of a network partition.

The interested reader can find a more thorough discussion in [1, 2].

2.2.1 Eventual Consistency

Eventual Consistency, also known as Convergence, states that, even under arbitrarily long delays, different replicas of the same item will eventually converge to the same value. Most weakly consistent databases ensure at least this property, often in conjunction with other consistency levels.

2.2.2 Single-Object Consistency Levels

Single-object consistency levels address how each process observes the updates performed to a given object, regardless of the updates performed to other objects. Typically, the guarantees provided to clients are in the context of a *session*, a sequence of operations on individual objects performed by the client in a context where the client is able to maintain some state regarding its past operations. A session

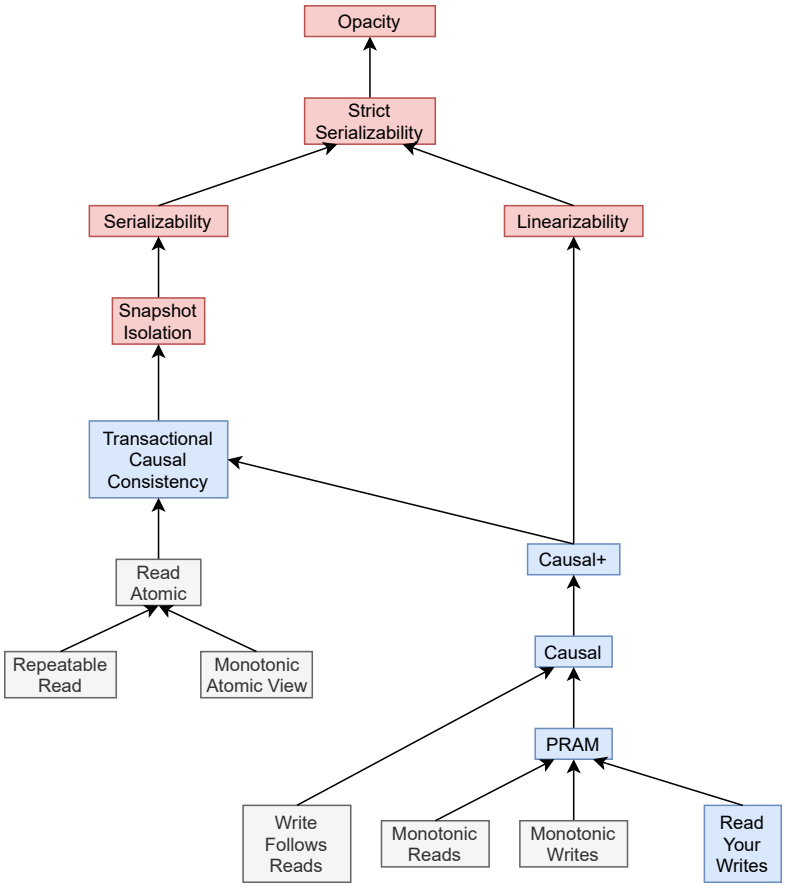


Figure 2.2: Consistency Guarantees by strength. Grey box meansHAT compliant, blue box means HAT-sticky compliant, red box means non-HAT compliant. This figure was adapted from [1, 2]

.

can be the execution of a single application, from the moment it starts until it ends, or a sequence of applications that share some context, for instance a user session from login to logout. The most relevant consistency criteria are the following;

Monotonic Writes (MW): Monotonic Writes requires that write operations become visible in the order they were submitted.

Monotonic Reads (MR) Monotonic Reads requires that a read by a client always observes at least the writes that have been already observed by any previous read of that client. I.e., a read operation never returns an older version than the ones that have been observed in the past, even if the client contacts multiple replicas in the same session.

Both MW and MR are HAT compliant, as they only require servers to wait for confirmation that all previous updates were installed on all servers to install newer updates in the correct order.

Read your Writes (RYW) Read your Writes requires that a read by a client always observes a state where all write operations previously performed by that client have already been applied. This level is HAT-Sticky, as it requires the client to stick to the same server to obtain its applied updates.

Pipelined Random Access Memory (PRAM) Pipelined Random Access Memory is the combination of the three previously defined consistency levels. It ensures that a client always observes a state that is consistent with some serialization of all operations that respects the serial order of the operations in its own session. Since PRAM extends from RYW, it is also HAT-Sticky.

Write Follows Read (WFR) Let R_i be a read operation and W_j be a write operation executed by a client after R_i. Let $\mathcal{W}_i$ be the set of write operations that have been applied to originate the state returned by R_i. Write Follows Read is ensured when the write operation W_j is guaranteed to be performed in a state where at least all writes in $\mathcal{W}_i$ have already been applied. Just like MW and MR, this level is HAT compliant, as it can be easily achieved by waiting for all servers to apply the required updates.

Causal Consistency (CC) Causal Consistency, derived from Lamport's "happens-before" relation [27], is the combination of Write Follows Read with PRAM. It's known as one of the strongest consistency level that is still highly available, as long as it is sticky to a set of servers.

Causal+ Consistency (CC+) Causal+ Consistency is an extension of Causal Consistency, ensuring that replicas converge to the same value in face of concurrent updates.

Linearizability Linearizability is the strongest single-object consistency model, requiring that operations take place atomically and in an ordering consistent with the real-time ordering they were submitted. This level is not HAT compliant, as it requires coordination with other servers to guarantee the real-time ordering of objects.

2.2.3 Multi-Object Consistency Levels

We now address consistency levels that can be defined to *transactions*, i.e., sequences of operations that read and write multiple objects. In this case, the consistency model defines the degree of isolation that is ensured among transactions. In the stronger consistency models, transactions are fully isolated from each other: even if they execute concurrently, the result is equivalent to some serial execution of the transactions, where each transaction runs alone.

Repeatable Read (RR) Repeatable Reads requires a transaction to execute all read operations from the same snapshot. As a result, it enforces multiple read operations on the same object to return the same value. This level is HAT compliant, as it is easily achievable by having the transaction save all the values read. Different properties have been labeled Repeatable Read with different isolation strengths associated with them. In this report we follow its commonly used definition in related work, corresponding to the *Item Cut Isolation* level defined in Bailis et. al. [1].

Monotonic Atomic View (MAV) Monotonic Atomic View [1] enforces the atomic visibility of updates. Once a transaction T_j observes the effects of a transaction T_i, all future read operations of T_j must reflect the updates of T_i.

Read Atomic (RA) Read Atomic [28] builds upon MAV, enforcing stronger atomicity semantics. Let T_0 be a committed transaction that has written X_0 and Y_0. Let T_1 be a running transactions that has read X_0 and T_2 be a concurrent transaction committing the updates X_2 and Y_2. If T_1 reads Y, MAV allows T_1 to read Y_2. Read Atomic disallows this execution as it would break the atomicity of T_2 updates, as Y_2 was cowritten with X_2 and X has already been read by T_1 with value X_0. RA requires T_1 to read Y_0 or a more recent version of Y which was not cowritten with more recent versions of any previously read updates from T_1. A readset that follow these constraints is known as an Atomic Readset.

Both MAV and RA can be HAT compliant by having a lower bound of all replicas for each key, ensuring that updates are made visible atomically.

Transactional Causal Consistency (TCC) Transactional Causal Consistency extends causal consistency to sets of objects, requiring transaction readsets to form a causal snapshot and updates to be atomically visible. To form a causal snapshot, the readset must maintain all its object version dependencies. Let T_0 be a committed transaction that created X_0 which depends on a object version Y_0. Let T_1 be a running transaction that has read X_0. When reading a object version of Y, it must read a object version that has not occurred before Y_0, as it would break the dependency requirements of X_0. T_1 can read either Y_0, a concurrent version to it or a more recent version of Y_0. However, it can not read a version Y_1 that is dependent on a more recent object version X_1, as it already has X_0 in its snapshot. TCC also requires the atomic visibility of updates. By adding read dependencies between a transaction writeset, it is transparently obtained by the causal snapshot property. We can consider TCC a merge of the requirements of RA and Causal+. TCC is HAT-Sticky by following both HAT requirements of the Causal+ and RA consistency levels.

Snapshot Isolation (SI) Snapshot Isolation requires users to read from a stable snapshot, ensuring all committed operations prior to the chosen snapshot are visible. It also requires that no executed operations are visible to concurrent transactions during execution and that no two concurrently committing transactions writesets intersect. The chosen snapshot can either be the starting point of the transaction or another past point in time. It is a stronger isolation than TCC, as reading from a stable snapshot is equivalent to having all prior updates and, hence, all prior dependencies.

Serializability (1SR) Serializability requires transactions to appear in a total order in all processes. This however does not necessarily correspond to the transaction real-time execution, so a transaction T_0 might have occurred before a transaction T_1 but in the total order appear after T_1.

Strict Serializability (Strong 1SR) Strict Serializability is the strongest possible isolation level. It is the union of Serializability with Linearizability, ensuring transactions are totally ordered and the real time ordering of transactions are respected.

Opacity Opacity [29], originally designed for Transactional Memory, is an extension of Strict Serializability, with the additional requirement that even non-committed transactions do not observe inconsistent state. This property is relevant when considering fault-tolerance and recovering, as seen in [25, 30].

The four previous levels are not HAT compliant, as they require coordination between partitions and/or replicas when committing to guarantee serializability of operations.

2.2.4 Anomalies

Anomalies are defined as execution orderings that break the isolation properties of a transaction. In literature, the strength of a multi-object consistency criteria is often defined by which anomalies that level prevents. We make a short overview of some of the anomalies our previously defined consistency levels prevent. The various examples were taken from [1, 31].

Fuzzy Reads occur when multiple reads on the same object return different values.

T1	T2
R(X_0)	
	W(X_1)
	Commit
R(X_1)	

Table 2.1: Fuzzy Read anomaly

Dirty Reads occurs when transactions read either uncommitted, aborted or intermediate state of a concurrently running transaction.

T1	T2
W(X_1)	
	R(X_1)
Commit	

T1	T2
W(X_1)	
Abort	
	R(X_1)

T1	T2
W(X_1)	
W(X_2)	
Commit	
	R(X_1)

Table 2.2: Dirty Reads. From left to right: Read uncommitted state, Read aborted state, Read intermediate state.

Fractured Reads occurs when a transaction T_1 writes object versions X_1 and Y_1 and a transaction T_2 reads X_1 and a Y object version older than Y_1.

T0	T1	T2
W(Y_0)		
Commit		
	W(X_1)	
	W(Y_1)	
	Commit	
		R(X_1)
		R(Y_0)

Table 2.3: Fractured Read anomaly

Lost Update occurs when a transaction T_1 reads a object value X_0, a transaction T_2 updates X and T_1 updates X based on the initial read value X_0.

T1	T2
R(X_0)	
	W(X_2)
W(X_0 + 10)	

Table 2.4: Lost Update anomaly

Write Skew occurs when multiple transactions concurrently update different objects present on each other readset. If there were a constraint between these objects, it would be impossible to serialize the execution.

T1	T2
R(Y_0)	R(X_0)
W(X_1)	W(Y_2)

Table 2.5: Write Skew anomaly

We finish with a small table representing which anomalies are prevented by which consistency level and an overall view of the relation between consistency and isolation levels.

	Fuzzy Reads	Dirty Reads	Fractured Reads	Lost Update	Write Skew	Real Time Constraint	Consistent Aborted Transactions
RR	✓	✗	✗	✗	✗	✗	✗
MAV	✗	✓	✗	✗	✗	✗	✗
RA	✓	✓	✓	✗	✗	✗	✗
TCC	✓	✓	✓	✗	✗	✗	✗
SI	✓	✓	✓	✓	✗	✗	✗
1SR	✓	✓	✓	✓	✓	✗	✗
Strong 1SR	✓	✓	✓	✓	✓	✓	✗
Opacity	✓	✓	✓	✓	✓	✓	✓

Table 2.6: Anomaly overview

2.3 Systems with Transactional Support

In this section we survey a number of systems that offer transactional semantics in both Cloud and FaaS settings, and provide an overview of the challenges and limitations of implementing each consistency level. We make a brief description of each system, discussing the isolation level it provides, the target environment, and the proposed solution. Unless otherwise stated, all systems are built on top of strongly consistent storage.

2.3.1 Cloud Systems

2.3.1.1 RAMP

RAMP [28] offers the Read Atomic isolation level, ensuring atomic visibility of transaction updates on a partitioned environment while maintaining scalability and high availability properties. It maintains atomicity by attaching metadata do each key version, detecting when readset atomicity is broken.

Three different variants of RAMP are introduced in [28], each bringing different tradeoffs between metadata size and the required number of communication round trips for the operations to terminate. Note that transactions are either read-only or write-only. Support for read-write transactions is possible but only when the read-set is known a priory (i.e, when transactions are *static*).

The first variant is named RAMP-Fast. In this variant, metadata scales linearly with the number of write operations in the transaction, with each key version attaching the transaction co-written key versions as metadata. Write transactions require the execution of a 2-Phase commit protocol. It first writes the keys into their respecting partitions marked as in "*Prepared*" state, waiting to be committed. Once all writes finish, a commit message is sent to make the prepared updates visible. On read transactions, it first obtains all required keys. Due to message asynchrony, some updates from read transactions may yet to become visible. To detect this, each key metadata is compared against the transaction readset to ensure that obtained key versions respect the atomicity. In the best case, it can return to the client in a single read. However, if an obtained version is mismatched, an extra communication round is used to

fetch a consistent version that may still be in the prepared state.

RAMP-Small takes the opposite tradeoff from RAMP-Fast, using constant sized metadata, in the form of the transaction number, but always requiring two communication rounds for reads. Write transactions execute as in RAMP-Fast. To execute read transactions, the first communication round obtains the highest transaction number that each partition has committed for the requested keys. Then, with the second round, the protocol sends the list of transaction numbers obtained and asks for the key version with the highest number presented in the list from both committed and prepared keys.

The paper also presents a third variant, named RAMP-Hybrid, which we will not go in detail has the two previous versions already present the inherent tradeoffs of implementing RA.

2.3.1.2 FastCCS

FastCCS [5] is an algorithm that offers TCC on top of a partitioned storage systems in an efficient manner, providing non-blocking read and write transactions. Transactions read from a fresh causally consistent snapshot, with a read latency of at most two rounds even in face of skewed workload, and an average latency of approximately one round. By using more precise metadata and vector clocks, it avoids the common pitfall of false dependencies, where the system does not have enough information to decide whether the obtained values are consistent.

The client holds a Dependency Vector Clock (DVC), representing the last seen system state by the client to ensure causality of future requests. On write-only transactions, a partition is chosen as coordinator, requesting the highest timestamp of each partition to create a Commit Vector Clock (CVC), representing the system state when that transaction committed. The coordinator commits the transaction by sending the CVC to every partition and the client, updating its DVC with the obtained CVC. Key versions go through three stages: Prepared, Committed and Visible. When the key version is created, it is on the Prepared state. After getting the CVC from the coordinator, it passes to the Committed state. Only when the version has been committed at all partitions, detected by a Snapshot Vector Clock (SVC) that each partition holds with the latest gossiped timestamp from each partition, can it pass to the Visible state.

For reads, the client sends its DVC to the coordinator to ensure the partition has at least the most recent state seen by the client, possibly updating its SVC with the system state. Partitions return the key versions whose clocks are dominated by the SVC, meaning that they have been committed in all partitions. When obtaining the requested keys, the partition obtains the maximum commit timestamp of each partition, creating a Maximum Committed Timestamp (MCT) vector clock. The MCT is used to check if the obtained keys are the most recent key version given the snapshot, as partitions may still have keys on the Committed phase while others are already in the Visible one. Since we know those transactions have already been committed in some partitions, we can obtain the keys directly from the

prepared or committed state. This ensures the minimal progression of key updates.

2.3.1.3 Wren

Wren [20] was the first system that implemented TCC with non-blocking reads on a partitioned, geo-replicated datacenter environment. It does so by calculating a Local Stable Time (LST), representing which transactions have been installed in all local partitions of a datacenter, forming a stable causal snapshot. To maintain Read Your Writes, a client-side cache is implemented to keep the client updates that are yet to be installed in all partitions.

Wren differentiates local and remote items to avoid coordination with remote entities to determine which updates can become visible locally. It does so by calculating the LST and a Remote Stable Time (RST), representing the remote transactions that have been installed in all local partitions. The LST and RST are calculated by having each partition gossip their last installed local and remote update to all local partitions in the datacenter, with the LST and RST being the the minimum of the local and remote updates respectively. Updates are represented by Hybrid Logical Clocks, allowing for clocks to move forward in case of clock skews, removing delays from waiting for clocks to catch up.

When a transaction begins, a partition is chosen as the coordinator, exchanging the client last seen LST and RST to maintain causality. Updates are kept in a writeset and sent on commit time. When reading, the client first goes trough its writeset (guaranteeing Read Your Writes), readset (guaranteeing Repeatable Reads) and client cache (updates not yet visible) before fetching from storage. When committing, the client last committed clock is sent to the coordinator to ensure causality. The coordinator sends the updates kept on the client writeset to their respective local partitions and asks for the partitions proposed commit timestamp, picking the highest proposed value and sending it to the partitions and client for committing. Updates are only installed in a partition when no proposed clock from the partition is lower than the committed transaction clock, ensuring that no update with smaller timestamp is installed after the update installation.

2.3.1.4 Clock-SI

Clock-SI [32] is a distributed protocol that implements Snapshot Isolation on partitioned data stores by relying on loosely synchronized physical clocks. While most systems that implement Snapshot Isolation require a global variable or centralized timestamp authority to reliably obtain the current snapshot of the database, Clock-SI diverges by having partition clocks loosely synchronized, requiring no coordination with a separate module or between partitions to obtain a snapshot.

When a transaction begins, the client contacts a partition, hereby named originating partition, and obtains the transaction snapshot timestamp by reading the physical clock of the partition. It uses this timestamp to obtain the desired keys from the snapshot. When reading, there are two cases where the

transaction must temporarily block before returning the requested keys. First, a concurrent transaction may be in the middle of committing, with its values possibly belonging to the transaction snapshot. Partitions must wait for the transaction to commit to avoid returning inconsistent results. Second, the partition may have its clock delayed due to clock skew. In this case, it must wait for the partition clock to catch up, as it could commit a transaction between this interval. To avoid these delays, Clock-SI allows for clients to pick an older snapshot. However, by extending the physical time between snapshot and commit time, it raises the chances of occurring a concurrent write on the same key space, leading to a transaction aborting. This delay is most useful for read-only transactions, sacrificing freshness for performance.

On commit time, if a transaction only accessed a single partition, it ensures no local write conflicts exist and uses the current partition clock as commit timestamp. For multiple partition accesses, it uses a 2-Phase Commit protocol with the originating partition serving as coordinator. Each partition checks for write conflicts, returning a proposed commit timestamp if none are found or aborting otherwise. The coordinator picks the highest timestamp as commit timestamp and commits to the partitions and the client. Committing to partitions with delayed clocks does not impose an issue, as it will only become visible once the clock reaches the transaction commit timestamp.

2.3.1.5 Padhye

Padhye PhD thesis [23] studies the challenges of transactional support in cloud environment using Snapshot Isolation. The author presents two solutions for transaction management, a decentralized model and a centralized one. We will focus our analysis on the centralized approach. As the author does not specify a name for its models, we will call them Padhye models.

The centralized approach is based on a service-base model, using a centralized lock management system to coordinate transaction writes on commit time, being partitioned into several processes, each in charge of the locks for a disjoint set of keys. Each partitions keeps records of the currently obtained locks and latest commit timestamp of each key. A first-updater-wins approach was chosen for the write conflict detection, where transactions try to obtain a lock of the key, with the first one to obtain the lock allowed to commit.

A timestamp management service is used to serve snapshot and commit timestamps. To serve a stable snapshot, it chooses the highest timestamp with which all transactions have either committed or aborted. To serve commit timestamps, it keeps information about the last commit timestamp served to return an higher value.

Updates are sent to the storage during transaction execution to avoid long commit phases, requiring extra effort into maintaining uncommitted data and mapping key version to the respecting timestamp on commit time. For read operations, transactions request the necessary keys from storage using the

obtained timestamp from the timestamp manager. On commit time, a partition of the lock management service is chosen as coordinator, acquiring the necessary locks from each partition. It checks the latest commit timestamp of each key to be updated to check for write conflicts. In case a lock is being used, the coordinator waits for the transaction to finish. To avoid deadlocks, if the waiting transaction has an higher Transaction ID than the one currently holding the lock, it aborts. If the transaction commits, then it must also abort due to write conflict, else it can obtain the lock. Once all locks are acquired, it requests a commit timestamp from the timestamp management service, updating the corresponding uncommitted keys on storage and committing to the user.

2.3.1.6 CloudTPS

CloudTPS [22] is a system that offers strong consistency on top of a storage layer that offers weak consistency, focusing on providing ACID properties and guaranteeing Strict Serializability to transactions. It relies on an intermediate layer composed of multiple Local Transaction Managers (LTM), with each LTM responsible for a partition of the data set, executing the certification of transactions that interact with its partition. LTMs hold the uncommitted data of transactions as well as a full copy of application data of its key space, relying on the storage layer for Durability only.

To guarantee Atomicity, writes are stored as uncommitted data in their respective LTM. On commit time, a 2-Phase Commit protocol is used, choosing an LTM to act as coordinator and checking with each partition for write-conflicts. If no conflicts are found, the coordinator sends the commit requests to participating LTMs and returns to the client. Updates are not sent to storage during the 2-Phase Commit to reduce commit time, being periodically sent in the form of checkpointing. To tolerate faults between checkpoints, transaction state and data items are replicated in multiple LTMs before returning to the client, which is faster than sending to storage.

Consistency rules are applied and maintained within transaction logic, which is checked on commit time. As such, as long as the transaction is able to commit and is executed properly, the consistency properties will be maintained.

In order to ensure Isolation properties, transactions are split into multiple sub-transactions which are globally ordered by timestamps assigned by a Sequencer. Sub-transactions can only be execute once all conflicting sub-transactions with lower timestamps have commited or aborted.

2.3.2 FaaS Systems

2.3.2.1 AFT

Atomic Fault Tolerant shim [15] is an intermediate layer interposing the FaaS platform and the Cloud Storage, providing Read Atomic isolation guarantees on top of a weakly consistent storage. The AFT

shim serves as a transaction manager, tracking each key version read during transaction execution while maintaining an Atomic Readset throughout the transaction. Writes are buffered throughout the transaction in the AFT, only writing to storage on commit time to ensure the atomicity of updates.

On commit time, the AFT first writes the transaction update to storage for durability. Once persisted, it writes the transaction ID and writeset to a Transaction Commit Set (TCS) in storage. Only when persisted in the TCS can the AFT write to a local Metadata Cache, which holds the latest version of each key, and successfully commit the transaction. This method allows for multiple AFTs to concurrently commit without any form of coordination due to multi-versioning of keys, with AFTs periodically sharing their committed transactions to each other. The write to TCS ensures that even in case of AFT failure before sharing the committed transaction, it will end up being discovered by a Fault Manager module, which periodically checks for transaction updates in the TCS that were not shared between AFTs.

To maintain the atomicity of updates, read operations can only read from the Metadata Cache present in the AFT. To form an Atomic Readset, each key version has attached the key versions written in the same transaction. When reading a key K that has been cowritten with a previously read key L, it must ensure that the version read for K is as equally recent or newer than the one cowritten with L. Furthermore, it cannot read a version of K that has been cowritten with a newer version of L, as it would break atomicity. AFT guarantees Read your Writes consistency by first reading from the Atomic Buffer and Repeatable Reads by default due to the restrains imposed by the Atomic Readset.

2.3.2.2 Hydrocache

Hydrocache [3] is a distributed cache layer algorithm that provides low latency while guaranteeing TCC in serverless computing environments without relying on membership. This membershipless approach is achieved by implementing dependencies at the key level instead of the partition level. Since functions of the same transaction can be executed at multiple physical nodes, it must also guarantee Multisite TCC.

To achieve TCC, each cache keeps a Strict Causal Cut, a stronger implementation of a causal snapshot. A cut differentiates from a causal snapshot by requiring the snapshot to include all the key version dependencies in the snapshot and not allowing concurrent updates to fulfil dependency requirements: it is either the exact key version of the dependency or a newer version. The cut is upheld by periodic key updates from a weakly consistent storage and by fetching all dependencies of a key when fetching it for the first time. To ensure update atomicity, keys updated in the same transaction have read dependencies set on each other, ensuring they are obtained together on the cut. Updates are buffered until the end of the transaction last function, where dependencies are added and the transaction commits by writing the updates to storage. These methods form the Centralized approach of Hydrocache, where all functions of a transaction execute in a single physical node. It brings the advantage of having a maintained cut

by the cache, but imposes a bottleneck on parallel function execution and does not take advantage of the scalability properties of FaaS systems. As such, it is desirable to execute a transaction throughout multiple physical nodes, requiring TCC to be achieved on multiple nodes.

To maintain TCC on a multisite environment, Hydrocache implements three approaches:

- An *optimistic* approach runs the functions without prior preparation, checking for snapshot violations between function executions. Functions send the readset and writeset metadata from the transaction when passing the execution to other functions. Before a function starts, it verifies if it has a compatible snapshot with the previous function. If not, it tries to form one by obtaining the missing key versions from the previous function executor, having to abort if unable to form one. It also checks for compatible snapshots when parallel functions join their outputs. If parallel functions executed on incompatible snapshots, the transaction must abort as there is no possible way to recover. Note these causes for aborts can happen multiple times for the same transaction. As such, the optimistic approach brings low averages response time but high tail response latency.
- A *conservative* approach first builds a snapshot across all caches before executing, making sure the transaction will never abort. It allows for only one execution of the functions in exchange for an higher average response time due to the coordination costs between caches.
- A *hybrid* approach takes the advantages of both implementations. It starts by running the optimistic approach while at the same time running a simulation of the Optimistic dependency checks. This simulation checks for any possible version conflicts without having to execute any functions, since Hydrocache assumes static transactions. With this simulation, Hybrid is able to know in advance if the Optimistic approach will abort. If it will, it aborts it right away and starts running the Conservative approach to avoid multiple aborts.

For the best case scenario, Hydrocache runs the Optimistic first try, in worst case it runs both the optimistic and conservative approach, averaging better latency than conservative and tail latency than optimistic.

2.3.2.3 FaaSFS

FaaSFS [24] is a shared File System built specifically for FaaS, offering POSIX semantics and linearizability to single functions while maintaining scalability and good performance when compared with other shared file systems. POSIX semantics allows for an higher level of portability of programs to serverless computing environment, as it does not require modifications to adapt to specific APIs like other stateful storage services.

POSIX semantics are achieved by using multiple optimist execution mechanisms. On transaction start, the client obtains a file-system wide read timestamp, corresponding to the most recent commit of

the file system. It uses it for Optimistic lock elision, assuming it always holds read and write locks, and for snapshot reads, making an optimistic use of cache state in conjunction with a multiversioned backend to obtain older updates if needed. On commit time it verifies if no concurrent transaction has modified the items it read or wrote.

In order to avoid conflicts, writes and reads are represented at the block level instead of the usual file level, avoiding aborts for concurrent writes that occur on the same file but at different sections. It also allows for cache updates to be more efficient, sending less information in the form of blocks instead of whole files.

Cache updates occur using a lazy approach. On transaction start, the cache is informed of which blocks have become invalid but are not updated, as it would incur a significant overhead. Instead, when a read operation requires the block it requests it from storage, updating its cache.

2.3.2.4 Beldi

Beldi [25] is a system that offers transactional support with Opacity [29] guarantees, ensuring that transactions never observe inconsistent state, even if the transaction aborts.

Beldi main goal is to support fault tolerance and transactional support to function compositions in a portable way, utilizing a library for function composition construction. Beldi relies on multiple log files, present in the atomicity scope of the storage layer, to guarantee exactly-once semantics to applications. These logs keep track of various information about the function composition, such as the state of each function execution, read and written objects as well as transactional metadata and locks. To guarantee fault tolerance and exactly-once semantic, Beldi utilizes a serverless function in the form of an *Intent Colector (IC)*, which periodically reads the logs and re-execute functions that might have failed, utilizing the logs to retrace execution such as which object values were read from storage.

Since most traditional storage services only allow small atomicity scopes, for example on table rows, Beldi created a new technique called Distributed Atomic Affinity Logging, or DAAL, that allows all the required information to be in the same atomicity scope, even it it has a large size.

For transactional support, Beldi utilizes a 2-Phase-Locking variation with wait-die as its deadlock prevention mechanisms. Functions explicitly call begin transaction and end transaction, with transactional context being passed from function to function. Before any read or write operation during a transaction, a lock operation occurs to obtain the object lock, aborting the transaction if unable to do so. During a transaction, Beldi also keeps a shadow table, a local copy of the original logs where writes and reads are first routed through, ensuring the ACID properties and ensuring read your writes semantics.

Systems	Target	Consistency	Storage Consistency	Metadata Size	Read Freshness	Memory Usage	Read RTT	Commit RTT
RAMP-Fast	Cloud	RA	Strong	$O(W)$	✓	-	$\leq$2	2
RAMP-Small	Cloud	RA	Strong	$O(1)$	✓	-	2	2
Wren	Cloud	TCC	Strong	$O(1)$	✗	$O(N * R)$	2	3
FastCCS	Cloud	TCC	Strong	$O(N)$	✗	$O(N)$	$\leq$2	2
Clock-SI	Cloud	SI	Strong	$O(1)$	✓/✗	$O(1)$	2*	3*
Padhye Cent.	Cloud	SI	Strong	$O(1)$	✗	$O(W)$	2	4*
CloudTPS	Cloud	Strong 1SR	Weak	$O(1)$	✓	$O(K)$	2	4*
AFT	FaaS	RA	Weak	$O(W)$	✗	$O(K)$	1*	3
Hydrocache	FaaS	TCC	Weak	$O(K)$	✗	$O(K)$	1*	2
FaaSFS	FaaS	Strong 1SR	Strong	$O(1)$	✓	$O(K)$	2*	1*
Beldi	FaaS	Opacity	Strong	$O(1)$	✓	$O(K)$	2	1*

Table 2.7: System comparison (W represents writeset, N represents number of partitions, K is key-space, R is number of replicas, * represents that action may include blocking or aborts).

2.4 Analysis

We now discuss the advantages and disadvantages of the systems surveyed in the previous section. Table 2.7 presents a comparative analysis of the different features of each system.

2.4.1 Target Environment

We can classify the systems into two main categories, according to the type of environment they target, namely into Cloud systems and FaaS systems. Some systems do not explicitly state that they have been designed for Cloud environments, but we include them in this broad category as they solve similar challenges. The distinction is relevant due to the constraints imposed by each environment. Cloud systems mostly operate with a almost static number of partitions and replicas. This simplifies the use of techniques that maintain information for each partition, for instance, the use of vector clocks to keep track of the current snapshot. On the other hand, FaaS systems are designed to ease auto-scaling, and we can expect the number of servers to change often. To avoid frequent transformation to data structures, every time there is a membership change, many system use metadata that is independent from the number of servers, for instance, per-key metadata.

2.4.2 Offered Consistency Guarantees

In the table, we order the surveyed systems by ascending consistency strength. Performance degrades as the consistency strength increases, due to the required additional coordination. This coordination involves the exchange of messages, to execute a 2-Phase Commit protocol (which can add one or more Round Trip Time (RTT) to the latency) and the use of locks (that can also block transactions, adding additional latency). It is interesting to observe that systems that offer stronger consistency criteria usually

require less metadata than system that offer weaker guarantees. This happens because weaker models allow for more concurrency in the system, which requires additional metadata to capture accurately.

2.4.3 Storage Consistency

It is relevant to distinguish systems that require a strongly consistent storage layer from system that can operate on top of weakly consistent stores. Weakly consistent storage systems are orders of magnitude faster than strongly consistent storage systems, but when using the former one must rely on additional layers or additional metadata to enforce strong consistency to the application (as in AFT and Hydrocache).

2.4.4 Metadata Size

Metadata size defines how much information must be transferred between requests to maintain consistency. Cloud environments are able to minimize metadata usage, as they can make assumptions about the number of partitions and use membership techniques as the ones proposed by FastCSS and Wren. RAMP supports a choice between constant or linear metadata usage. In FaaS systems it is harder to reduce the size of metadata, because the number of servers may change often. These systems typically keep metadata for each object read or written in a transaction, and may be required to exchange large volumes of metadata, as it happens in AFT and Hydrocache. Systems enforcing SI or stronger consistency require less metadata, because there is less concurrency in the system (in fact, all write transactions are totally ordered). Therefore, they only use a single timestamp as metadata; this timestamp is enough to represent a global state of the system.

2.4.5 Read Freshness

Due to consistency constraints or design choices, some systems sacrifice the freshness of obtained data for performance gains. Systems that try to read the freshest data may require additional read rounds or have to block. For example, Clock-SI may choose a very recent snapshot to read fresh, but may have to block due to clock skews between partitions and concurrently committing transactions that may yet belong to the desired snapshot. However, by choosing a snapshot in the past it avoids these issues. Apart from RAMP, that makes an effort to read the freshest version of each object from storage at the cost of extra reading rounds, only the strongest transactional supports offer read freshness, as Strict Serializability and Opacity must respect real time constraints.

2.4.6 Memory Usage

Memory usage captures the amount of information that needs to be maintained in memory to enforce the desired consistency. Memory usage has a direct correlation with consistency of the storage layer, as systems with weaker storage consistency cannot rely on storage properties to obtain specific key versions, requiring data or metadata to be kept on memory. AFT, Hydrocache and CloudTPS all use a weakly consistent storage and require the whole key space in memory, with stronger storage systems only requiring general system state information such as vector clocks for TCC systems and write-sets of running transactions in Padhye. FaaSFS and Beldi are outliers in the storage consistency and memory usage tradeoff, as they require much more information to maintain strong consistency efficiently, such as the logs that Beldi maintains.

2.4.7 Cost of Read Operations

We capture the cost of a read operation, by counting how many network round-trips need to be performed to terminate the operation. FaaS systems average one RTT for read operations, as they require low latency. FaaSFS is an exception due to its stronger consistency requirements, using an extra RTT to obtain a read snapshot. Note that due to most FaaS weakly consistent storage, a key version may be requested multiple times until it obtains a desired version or aborts. Cloud systems take on average more RTTs, but "fast" approaches like RAMP-Fast and FastCCS can still achieve averages of one RTT. Wren and Cloud Snapshot Isolation systems also take one RTT to read, however they include a first communication round on transaction start to establish a snapshot, which is accounted for in the read cost. Clock-SI may also have to block for currently committing transactions, as the transaction updates may be included in its snapshot. Although technically Beldi only reads once from storage, we consider a RTT of 2 to its read operations, as they first require a read operation to obtain the logs and then a write operation to update the log with the obtained result.

2.4.8 Cost of the Commit Operation

We also capture the cost of a commit operation, by counting how many network round-trips need to be performed, in worst-case, to terminate the operation. Most systems take an average of 3 RTTs for committing, representing the commit request from the client and a 2-Phase Commit protocol required for partitioned environments. RAMP and FastCCS avoid a communication round by having the client act as coordinator. Hydrocache does not require a 2-Phase Commit, transparently committing by sending the writes to storage. The Cloud SI protocols require an extra communication round for requesting a commit timestamp from an external Timestamp Management system. FaaSFS and Beldi only require 1 RTT for commit, however they may exhibit a much higher abort rate given that the consistency guarantees

provided are stricter.

Note that in order to enforce strong consistency levels one needs to implement concurrency control mechanisms that can be pessimistic (based on locking) or optimistic and that, in either case, can force transaction to abort. The cost of implementing these concurrency control mechanisms must be taken into account in the system performance.

Summary

In this chapter we have addressed the challenges in offering different consistency levels in distributed storage system and, in particular, FaaS systems. We have seen that recent works have proposed to augment existing FaaS systems with support for weak transactions but, to the best of our knowledge, no previous work has addressed the support for strong consistency in this context in an efficient manner. We have surveyed systems that offer strong consistency over weakly consistent storage that have been designed for non-FaaS environments, and have discussed how these could be adapted to support serverless computing.

3

FaaSSI

Contents

In this chapter we present FaaSSI, a system that offers transaction support with strong consistency guarantees in FaaS. Section 3.1 describes the goals our system aims to accomplish. Section 3.2 describes the base design ideas and features of FaaSSI. Section 3.3 describes in detail components used in our implementation and the adaptations that we have performed to these components to fit our architecture. Section 3.4 describes the components of the intermediate layer in detail, with its architecture and libraries. Section 3.5 describes the protocols used by FaaSSI to offer snapshot-isolation to applications. Section 3.6 describes the multiple factors involved in the stability of objects in storage and, finally, Section 3.7 briefly discusses fault tolerance of FaaSSI.

3.1 Goals

The goal of FaaSSI was to design a system that would offer strong transactional support in FaaS while keeping all the advantages of this environment. As such, we designed a system that would be scalable with the increase of executor nodes while imposing small cost overheads over the base FaaS implementation. We also designed our system with consistency interoperability in mind, such that functions that require only weak consistency properties do not suffer any additional overhead.

3.2 Design

FaaSSI was designed to be executed in a datacenter, having been developed as an extension to Cloudburst [16]. We used Anna [4] as storage layer, which only supports eventual consistency. By using a weakly consistent store we do not penalize the performance of applications that only require weak consistency guarantees. This choice allows users to, in a single system, choose the appropriate consistency model for their applications. If the user only requires eventual consistency, they can contact Anna without any additional service nor penalty in performance. If the user requires TCC they can use a system like Hydrocache [3]. Finally, if the user requires Snapshot Isolation they can use FaaSSI.

Like most previous works [15, 22, 23], our system relies on an intermediate layer interposing between the computational and storage layers. This layer is responsible for implementing concurrency control, applying writes, and helping clients to read consistent values from storage. This architecture may be used to support multiple variants of strong consistency but, for this work, we focus on Snapshot Isolation.

Contrary to previous work however, not all read requests need to be served by the intermediate layer. This is possible because FaaSSI leverages the fact that the storage layer can return consistent results for most requests. As such, the intermediate layer is mostly used for consistency checking, and only for a fraction of the requests, the intermediate layer may be required to serve data values to the clients.

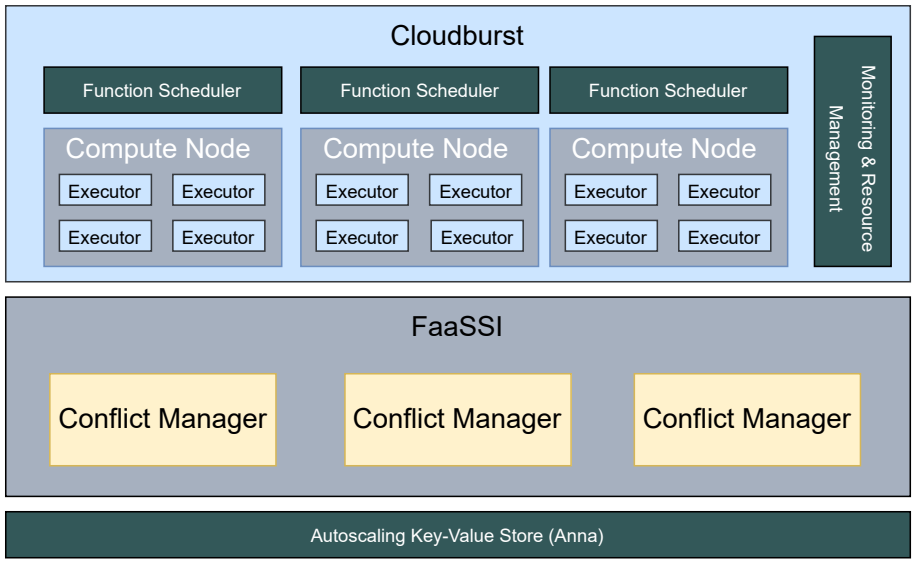


Figure 3.1: FaaSSI transactional layer. Based on the diagram of [3]. Yellow sections represent FaaSSI additions.

3.3 Architectural Details

Figure 3.1 shows the integration of FaaSSI with the Cloudburst FaaS platform and the Anna storage layer. In this section we will take a closer look at each of this components and discuss some of the adaptations that were required to integrate them in FaaSSI.

3.3.1 Cloudburst

Cloudburst [16] is a FaaS platform designed for stateful applications. The system is composed by a set of compute nodes, function schedulers and a monitoring and resource management component.

The compute nodes are composed of multiple function executor threads, each responsible for executing a single function, having that function pinned to the thread. When a function requires an object as a function argument, it contacts the storage layer to obtain the object. We have modified this logic to implement the read protocol defined in Section 3.5.1. When a function finishes, it passes on the function results (if it has any) to the next function in the function composition (if it is not the sink function). We further adapted the metadata passed among functions to include the transaction write-set, as described later in Section 3.5.2. The sink function of the composition writes the results to storage and returns the results to the client if it so requested. We removed this logic and instead added our commit protocol, further described in Section 3.5.3.

In Cloudburst, each node has an attached cache for lower latency. We have removed all caches from the nodes for correctness sake, as reading stale information from cache could lead to inconsistent results. We further discuss caching in Section 5.2.

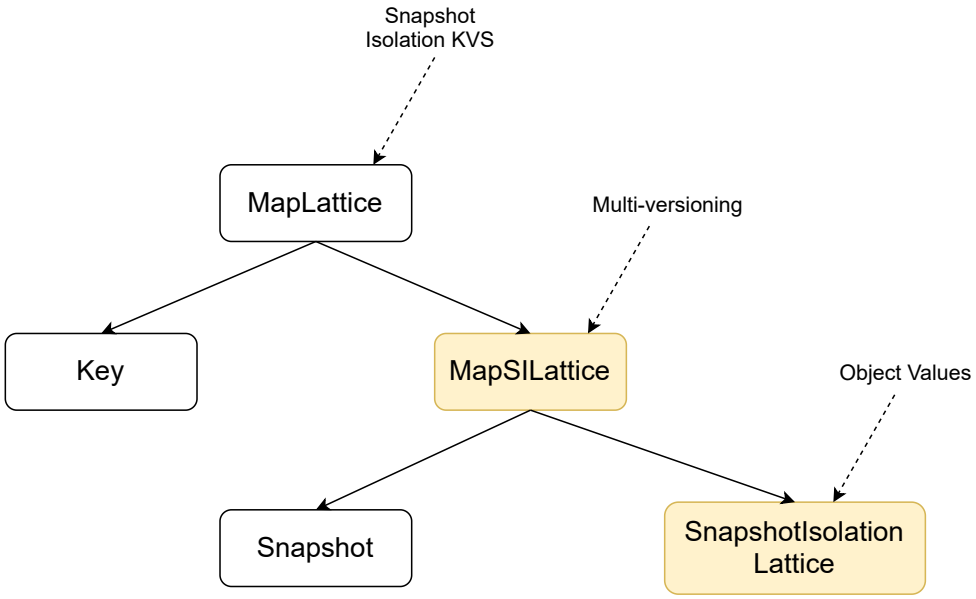


Figure 3.2: Snapshot Isolation KVS implementation in terms of lattices. Based on the diagram of [4]. Yellow sections represent the new Lattice additions.

The function schedulers are responsible for registering and calling both individual functions and function compositions. When scheduling a function composition, it selects the executors to be contacted. The selection process uses different criteria, such pinning specific functions to specific executors or using a round-robin assignment. We further added metadata about the snapshot to be used by the transaction, which is defined when the application is invoked.

Finally, the monitoring and resource management is responsible for scaling the compute nodes and function replicas based on usage statistics. To simplify the analysis of the results, we have disabled the autoscalling feature and, in all experiments, we have manually provisioned Cloudburst with enough resources to serve all clients in parallel.

3.3.2 Anna

Anna [4] is a weakly consistent Key-Value Store without any transactional support, offering only eventual consistency guarantees. Anna was designed to be a highly scalable storage system, a desired feature for a FaaS architecture. It uses lattices [33], a form of Conflict-free Replicated Data Type (CRDT) [34], making the system highly scalable and performant, relying on wait-free and coordination-free protocols. Anna consists of three different modules: the KVS module, a routing module and a monitoring module.

The KVS module consists of multiple partitions, each responsible for a part of the keyset. Data is partitioned using consistent hashing, relying on a hash-ring. Keys can also be replicated among multiple partitions. When replication is active, replicas periodically gossip their recent updates to their corresponding replicas. Programmers can create their own lattices for their desired consistency level and

use case. We will now make a brief description of the used lattices to create the multi-versioning and keeping of metadata for each version in the form of a commit timestamp. More detail on this timestamp will be discussed on Section 3.5.3.

The KVS is implemented as a MapLattice, a lattice acting as a hash map, matching each key to a multi-versioned store in the form of a newly created MapSILattice. MapSILattice is a descending ordered hash map, using the commit timestamps to order versions from most recent to older, reducing access times as the transaction snapshots slowly increase. Finally, Anna has a SnapshotIsolationLattice, which holds the object data and commit timestamp for faster access. When inserting a new item, each lattice uses its merge operation in case the key already exists. The merge process for the MapSILattice involves inserting all values from the new lattice into the existing one. Since we are executing on Snapshot Isolation, there wont be concurrent versions with the same snapshot, and as such there is no need for a merge operator in the SnapshotIsolationLattice. We can see a visual representation of the lattice in Figure 3.2.

The routing module is responsible for routing Anna requests to the correct partitions. It holds a cache of partition addresses for fast response times. When a cache-miss occurs, the routing service traverses the hash-ring to obtain the responsible threads.

Finally, Anna also has a monitoring module, which obtains statistics and information on key access for elastic replication and scalability. We removed the scalability features and instead use a fix value for key replication factor.

3.4 Intermediate Layer

The intermediate layer is composed by a set of *Conflict Managers*. Each conflict manager is responsible for certifying transactions on commit time and answering read requests from clients. We follow an approach similar to CloudTPS [22], where each conflict manager is responsible for a partition of the space of data. Figure 3.3 shows a more detailed view of the intermediate layer.

Each conflict manager keeps a version cache and a data cache of the last written objects to help serve clients the most recent copy of popular items. The size of the conflict managers caches is a system parameter. When the memory runs out, the items are discarded according to a substitution policy that may be configured. On our current version, we exclusively use LRU (*Least Recently Used*) as substitution policy.

The conflict managers are also responsible for persisting the writes to Anna, sticking themselves to an Anna replica for each object, guaranteeing that they are always able to obtain the most recent object values. This is required to ensure that a cache can safely flush cached values and/or versions when needed with the guarantee that it will be able to obtain the value when necessary.

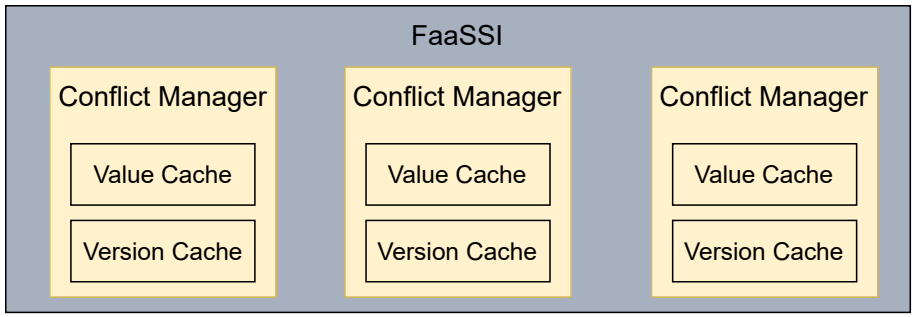


Figure 3.3: Detailed look of FaaSSI intermediate layer

FaaSSI implements a commit protocol similar to ClockSI [32], relying on synchronized clocks to order concurrent transactions. This protocol may introduce delays to read requests proportional do the precision of the clock synchronization. The evolution of clock synchronization algorithms for datacenters, like Sundial [35], allows clocks to be synchronized in the order of $100ns$, making the protocol very efficient.

We implemented our intermediate layer using Cloudburst cache as a code basis, being written in C++. We use ZeroMQ [36] with Protocol Buffers [37] for communication.

3.4.1 FaaSSI Client Library

We now describe the client library that interacts with the intermediate layer. The client library is responsible for communicating with the conflict managers. The library knows every existing conflict manager in the system and routes the requests to the corresponding conflict managers, splitting the requests if necessary. It was written in Python, used by the executor nodes to communicate with the conflict managers. The client library exposes three operations: *get_key*, *get_key_version* and *commit*. Table 3.1 presents the interface of FaaSSI client library.

Interface of the Conflict Manager client library	
get_Key	**input:** set of *keys*, *tx_snapshot*, *async* **output:** set of set of $\langle k, v_k, t_k^v \rangle$
get_Key_Version	**input:** set of *keys*, *tx_snapshot*, *async* **output:** set of set of $\langle k, t_k^v \rangle$
commit	**input:** set of $\langle$*key*, *value*$\rangle$, *tx_snapshot* **output:** *commit_timestamp*

Table 3.1: Interface of the FaaSSI Layers

The *get_key* request obtains the key values of objects. It receives as input one or more keys and the desired read snapshot.

The *get_key_version* request obtains the key versions of objects. It receives as input one or more

keys and the desired read snapshot. It may also return the object data in some cases, further described in Section 3.5.1.

Both functions have an additional optional input for asynchronous or synchronous execution. In case asynchronous mode is chosen, the function instead returns a list of request ids, used to identify the called requests for obtaining the results later. When the requested keys belong to different partitions, the client library splits the read requests and routes the request to the responsible conflict managers. It later merges the responses to return to the client.

The *commit* request starts the commit process for a transaction. It gets as input one or more keys and the corresponding values to be written and committed. It additionally may get a read snapshot as input in case any read operations were made. If it is a write-only transaction, the commit operation is always successful. Further reasoning on this aspect is present in Section 3.5.2. The client library determines the responsible conflict managers for the set of keys to be committed and randomly picks one to be the coordinator of the commit protocol. Further discussion on the commit protocol is present in Section 3.5.3.

3.5 Protocols

3.5.1 Read Operations

A transaction reads the consistent state of the system on a timestamp defined when the transaction starts executing. This timestamp is defined based on the properties the client requires. A client can request an older timestamp to avoid delays due to clock skews and currently committing transactions or they can obtain one by calculating the maximum between the commit timestamp of the clients last write transaction and the clock value of the function graph scheduler on graph function start, ensuring that the client reads their own writes.

When a Cloudburst computational node must read an object, it sends a read request to Anna and, simultaneously, to the corresponding conflict manager to obtain the most recent version of the object (Alg. 1, Lines 14 and 15). This is necessary since Anna only supports eventual consistency and may not return the most recent version of the object, as depicted in Figure 3.4. If the version returned by Anna is consistent with the version indicated by the conflict manager, the read operation finishes. Otherwise, we present two distinct protocols for obtaining a consistent version: FaaSSI-Pessimistic and FaaSSI-Optimistic. We analyse the performance of each of these techniques in Section 4.3.1.

- **FaaSSI-Pessimistic:** In FaaSSI-Pessimistic, the computational node contacts the conflict manager once again to obtain the content of the most recent version (Alg. 1, Line 3). As noted pre-

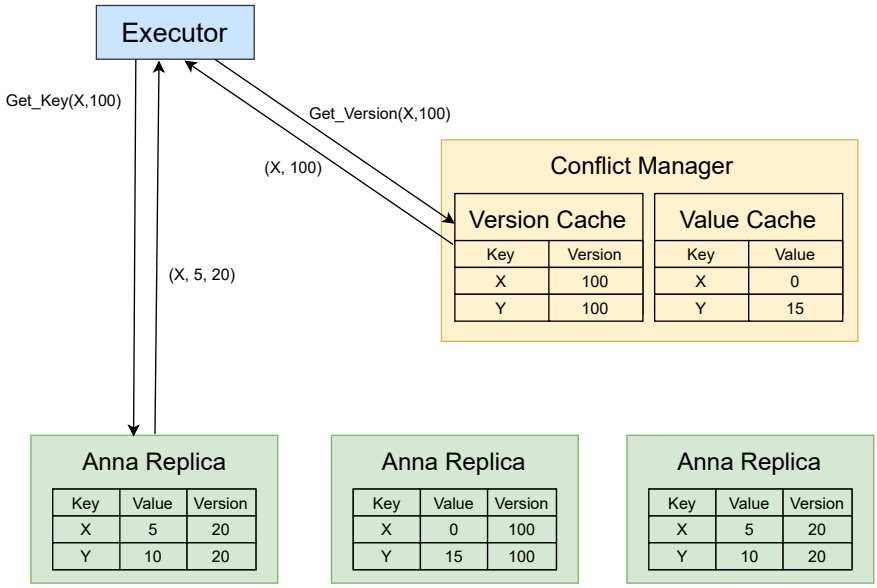


Figure 3.4: FaaSSI base read protocol

viously, the conflict manager can always obtain this value, even if it is not present in the cache. In cases where the version returned by Anna is inconsistent, the method adopted by FaaSSI-Pessimistic introduces an overhead with regard to methods where all versions are routed through the intermediate layer. However, as we will see, in most cases, that version returned by the storage system is consistent, and this second step is not needed. Thus, the method adopted by FaaSSI-Pessimistic significantly reduces the load imposed on the intermediate layer while also reducing latency. In fact, in the case where the object is not in the cache of the intermediate layer, it is more efficient to read directly from Anna than having the intermediate layer performing the read and forwarding the result back to the worker.

- **FaaSSI-Optimistic:** In FaaSSI-Optimistic, the computational node keeps contacting the storage layer until a consistent version is obtained. The advantage of this strategy is that it avoids to overload the conflict manager with additional requests. The disadvantage is that there is no guarantee that the desired version is returned in just one additional read operation. If the storage layer takes time to propagate versions among replicas of the storage nodes, multiple read request may need to be performed before the desired version is returned.

Just like in ClockSI, a read request may also block due to the clock skew between partitions, introducing an overhead in latency (Alg. 4, Lines 9). However, as described in Clock-SI, a client may opt to read from a snapshot in the past, reducing the chances of being blocked, waiting for the most recent version, while reading. Note however that reading from the past increases the chances of a read-write transaction to abort, as we are working with an older snapshot of the system. Also, if a concurrent trans-

action is committing a value that might belong to a cut, the conflict manager must wait for the committing transaction to finish before returning the correct object version to the client (Alg. 4, Lines 14). If the committing transaction commits successfully, the conflict manager returns to the client the object data instead of the respective versions. This optimization is due to the fact that in this scenario, where the transaction has just created the values to be returned, the probability of Anna returning the consistent version is lower.

Algorithm 1 Read Protocol

```
1: function READ_KEYS(keys, snapshot, write_set)
2:     retry_keys ← ∅
3:     key_value_set ← ∅
4:
5:     ▷ It first checks the write-set to ensure Read-Your-Writes
6:     for all k ∈ keys do
7:         if ∃⟨k, v⟩ ∈ write_set then
8:             key_value_set ← key_value_set ∪ {⟨k, v⟩}
9:             keys ← keys \ k
10:        end if
11:    end for
12:
13:    ▷ Request simultaneously the key versions from the conflict managers and values from KVS
14:    cm_versions ←CM.get_Key_Version(keys, snapshot)
15:    kvs_values ←KVS.get_Key(keys, snapshot)
16:    for all ⟨k, v, s_kvs⟩ ∈ kvs_values do
17:        ▷ If the versions obtained from the KVS and the conflict manager do not match, the key must be requested again
18:        if ∃⟨k, s_cm⟩ ∈ cm_versions ∧ s_kvs ≠ s_cm then
19:            retry_keys ← retry_keys ∪k
20:        else
21:            key_value_set ← key_value_set ∪⟨k, v⟩
22:        end if
23:    end for
24:
25:    if retry_keys ≠ ∅ then
26:        ▷ Either executes the FaaSSI-Pessimistic or FaaSSI-Optimistic
27:        key_value_set ← key_value_set ∪ FaaSSI-Pessimistic (retry_keys, snapshot) or FaaSSI-Optimistic (retry_keys, snapshot,
   cm_versions)
28:    end if
29:
30:    return key_value_set
31: end function
```

Algorithm 2 FaaSSI-Pessimistic

```
1: function FAASSI-PESSIMISTIC(keys, snapshot)
2:     key_value_set ← ∅
3:     cm_values ←CM.get_Key(retry_keys, snapshot)
4:     for all ⟨k, v⟩ ∈ cm_values do
5:         key_value_set ← key_value_set ∪⟨k, v⟩
6:     end for
7:
8:     return key_value_set
9: end function
```

Algorithm 3 FaaSSI-Optimistic

```
1: function FAASSI-OPTIMISTIC(keys, snapshot, cm_versions)
2:     key_value_set ← ∅
3:     retry_keys ← keys
4:     while retry_keys ≠ ∅ do
5:         kvs_values ←KVS.get_Key(retry_keys, snapshot)
6:         for all ⟨k, v, s_kvs⟩ ∈ kvs_values do
7:             if ∃⟨k, s_cm⟩ ∈ cm_versions ∧ s_kvs ≠ s_cm then
8:                 retry_keys ← retry_keys ∪k
9:             else
10:                key_value_set ← key_value_set ∪⟨k, v⟩
11:            end if
12:        end for
13:    end while
14:
15:    return key_value_set
16: end function
```

Algorithm 4 Read Request Conflict Manager

```
1: ▷ State kept by the conflict manager
2: cm.versions ← ∅
3: cm.commit_prepared ← ∅
4:
5: upon RECEIVE ⟨read, keys, tx_snapshot⟩ from client do
6:     return_values ← ∅
7:     storage_requests ← ∅
8:     ▷ We first check if the conflict manager clock is up to date with the snapshot
9:     if tx_snapshot > Clock() then
10:        WAIT tx_snapshot ≤ Clock()
11:    end if
12:    for all k ∈ keys do
13:        ▷ Check if any concurrent commit may belong to this snapshot
14:        if ∃⟨k, prepare_timestamp⟩ ∈ cm.versions ∧prepare_timestamp ≤ tx_snapshot then
15:            WAIT ∄⟨k, prepare_timestamp⟩ in cm.commit_prepared
16:        end if
17:        if ∃⟨k, v, commit_timestamp⟩ in cm.cache ∧commit_timestamp≤ tx_snapshot then
18:            return_values ← return_values ∪⟨k, v, commit_timestamp⟩
19:        else
20:            storage_requests ← storage_requests ∪⟨k⟩
21:        end if
22:    end for
23:
24:    if ∃⟨k⟩ in storage_requests then
25:        kvs_results ← kvs.get(storage_requests, tx_snapshot)
26:        return_values ← return_values ∪ kvs_results
27:        cm.cache ← cm.cache ∪ kvs_results
28:    end if
29:
30:    SEND ⟨read_response, return_values⟩ to client
31: end upon
```

3.5.2 Write Operations

FaaSSI implements a commit protocol similar to ClockSI [32], relying on synchronized clocks to order concurrent transactions. This protocol may introduce delays when processing read requests; these delays are proportional do the precision of the clock synchronization. The evolution of clock synchronization algorithms for datacenters, like Sundial [35], allows clocks to be synchronized in the order of $100ns$, making the protocol very efficient.

When a function graph executes with Snapshot Isolation guarantees, all writes are stored in a buffer, passed between functions. The sink function of the graph (this is, the last function of the graph) sends the data to be written, in conjunction with the timestamp used for the transaction read operations, to the corresponding conflict managers. If the data space of the write set belongs to multiple conflict managers, one of them is elected as coordinator for the commit protocol, using a 2-Phase-Commit protocol. If the transaction is committed successfully, the conflict managers write the updates to the storage layers. Note that if a transaction is a write-only transaction, it will never find any conflicts. This is because the transaction is not dependant of any snapshot and, as such, may be serialized between any two transactions.

3.5.3 Commit Protocol

When a transaction is about to commit, it first checks if the write set belongs to multiple conflict managers or to a single one. If it only belongs to a single conflict manager, then the certification occurs locally, with no need to communicate with other conflict managers, as its own synchronized clock is enough to safely obtain a commit timestamp for the transaction. Otherwise, the transaction must undergo a 2-Phase-Commit process.

The participating conflict managers must first certify the write-set of the objects belonging to its partition. If any write conflict surfaces, the transaction must abort, with an abort message sent to the transaction coordinator. Otherwise, it sends a provisional commit timestamp to the coordinator, symbolizing that the transaction may proceed.

The coordinator of the transaction, chosen randomly among the participating conflict managers, must wait for all provisional commit timestamps. If any abort message is received, the coordinator sends an abort message to all participants and the client, deleting all temporary versions created. Otherwise, when the coordinator obtains all provisional timestamps, the coordinator can calculate the final commit timestamp. The final commit timestamp is calculated by the coordinator using the maximum between all provisional timestamps.

When a transaction finishes, each conflict manager gives a provisional timestamp to the transaction, corresponding to the value of its synchronized clock. This timestamp is used to verify if any write conflicts

exist between objects of each conflict manager. If the transaction passes the local certification process, the provisional timestamp is sent to the coordinator. This process allows the transaction to commit if the written values are still valid in the instance corresponding to the final timestamp. When calculated, the coordinator sends this value to the participating conflict managers with the order to persist the state of the new objects in Anna.

During the 2-Phase-Commit protocol, the transaction write-set is blocked, and other concurrent transactions that try to write on any of these objects must wait. This is necessary to ensure that transactions that access one or more objects in common are serialized. To avoid deadlocks between concurrent transactions, we use a wait-die strategy where the transaction with the most recent timestamp aborts and transactions with older timestamps than the one currently holding the lock wait.

For correctness proof of the commit protocol, we point to [32].

3.6 Number of Rounds to Read from Storage

Our read protocol is based on the idea that the eventual consistent storage layer tends to return consistent values to read requests. The percentage rate of consistent values returned by storage is dependent on multiple factors such as:

- Periodic gossip update interval;
- Number of keys updated on each write transaction;
- Key access skew;
- Transaction length;
- Replication factor of keys in storage;

The time it takes for a write operation to become visible in all replicas depends mainly on the interval between gossips of replicas. The more frequent the gossiping, the lower the percentage of inconsistent results. However, lowering the gossiping interval increases the load on the storage layer, so a balance must be achieved.

If the size of the write-set grows, the number of objects whose replicas become temporarily inconsistent also grows, and the more likely it is that a read transaction obtains an inconsistent value from storage. This effect is amplified if the workload is heavily skewed. On the other end, longer lived transactions have a higher chance of reading stable values, given that their read timestamp gets “older” as the transaction executes, giving more time to the storage layer to stabilize the value.

Finally, the replication factor also affects the odds of reading a consistent value when it is not yet stable in storage. Given a replication factor of x, before the stabilization process is initiated, a read

Algorithm 5 Commit Protocol

```
1: ▷ State kept by the conflict manager
2: cm.versions ← ∅
3: cm.commit_prepared ← ∅
4: cm.commit_pending ← ∅
5: cm.prepare_timestamps ← ∅
6:
7: upon RECEIVE ⟨begin, key_values: set of ⟨k, v⟩, tx_snapshot⟩ from client do
8:     commit_groups ← ∅
9:     ▷ Check which partition is responsible for which key
10:    for all k ∈ key_values do
11:        partition ← hash(k)
12:        commit_groups ←commit_groups ∪{partition, ⟨k, v⟩}
13:    end for
14:    ▷ Send the prepare requests to the responsible partitions
15:    for all partition ∈ commit_groups do
16:        SEND ⟨commit_prepare, commit_groups[partition],tx_snapshot⟩ to partition
17:    end for
18: end upon
19:
20: upon RECEIVE ⟨commit_prepare, key_values: set of ⟨k, v⟩, tx_snapshot⟩ from coordinator do
21:    ▷ Transactions without a snapshot can always commit, so we skip testing
22:    if tx_snapshot ≠ MAX_TIMESTAMP then
23:        for all k ∈ key_values do
24:            ▷ If exists a version more recent than tx_snapshot, transaction must abort
25:            if ∃⟨k, s_cm⟩ ∈ cm.versions ∧ s_cm > tx_snapshot then
26:                SEND ⟨abort⟩ to coordinator
27:            end if
28:            ▷ Transaction may have to wait for a concurrently committing transaction
29:            if ∃⟨k, s_prepared⟩ ∈ cm.commit_pending ∧ s_prepared ≤ tx_snapshot then
30:                cm.commit_pending ← cm.commit_pending ∪⟨key_values, prepare_timestamp⟩
31:            end if
32:        end for
33:    end if
34:    ▷ If no conflict detected, we can read the local clock and send the prepare timestamp
35:    prepare_timestamp ←Clock()
36:    cm.commit_prepared ← cm.commit_prepared ∪⟨key_values, prepare_timestamp⟩
37:
38:    SEND ⟨prepare_response, prepare_timestamp⟩ to coordinator
39: end upon
40:
41: upon RECEIVE ⟨prepare_response, prepare_timestamp⟩ from partition do
42:    cm.prepare_timestamp ← cm.prepare_timestamps ∪ prepare_timestamp
43:    ▷ Wait for all prepare timestamps to commit
44:    if cm.prepare_timestamps = commit_groups then
45:        commit_timestamp ← max(cm.prepare_timestamps)
46:        for all partition ∈ commit_groups do
47:            SEND ⟨commit, commit_timestamp⟩ to partition
48:        end for
49:    end if
50: end upon
51: upon RECEIVE ⟨commit, commit_timestamp⟩ from coordinator do
52:    kvs.write(cm.commit_pending, commit_timestamp)
53:    cm.versions ← cm.versions ∪ cm.commit_pending
54:    SEND ⟨commit, commit_timestamp⟩ to client
55: end upon
56: upon RECEIVE ⟨abort⟩ from p do
57:    for all partition ∈ commit_groups do
58:        SEND ⟨abort⟩ to partition
59:    end for
60:    SEND ⟨abort⟩ to client
61: end upon
```

to a Anna replica has a $1/x$ change of returning the desired value; this probability increases as the stabilization procedure executes, becoming 1 when the object becomes quiescent.

3.7 Fault Tolerance

In this thesis we have not addressed the problem of ensuring that the consistency managers are fault-tolerant. This aspect is orthogonal to our contributions and can be solved with replication techniques that are well described in the literature [22, 23]. The most straightforward approach to achieve this goal would be to replicate each consistency server using a Paxos group [38], in a manner similar to what is implemented in Spanner [39]. It should be noted however that, when there is a leader change, the new leader is not guaranteed to read immediately the writes performed by the previous leader (because Read-Your-Writes consistency may not be provided to different clients); thus, the new leader would need to use the metadata obtained from the previous leader to check the consistency of the objects read from storage.

Summary

FaaSSI offers strong transactional support to applications in FaaS by relying on a set of consistency servers, responsible for serving consistent results and upholding transactional ACID properties. It uses a read protocol that imposes less load on these servers, hence it has the potential to offer better scalability than solutions that require all read operations to be performed via the consistency servers.

4

Evaluation

Contents

In this chapter we will evaluate the performance of FaaSSI. Section 4.1 describes our experimental goals and main questions we wish to answer. Section 4.2 describes the workload and the test bench used to evaluate our system. Section 4.3 shows the performance gains of our read protocol in comparison with the state of the art and how the different features of the workload and of the deployment affect the overhead required to achieve consistency. Section 4.5 presents the impact of multiple cache sizes to the system performance. Section 4.6 studies the system performance with multiple conflict managers. Finally, in Section 4.4, we measure the overhead of our system when compared to a weakly consistent baseline, namely an eventual consistent version of Cloudburst.

4.1 Experimental Goals

In our evaluation we wish to answer several questions, specifically about the gains and disadvantages of our implemented protocols when comparing with the state of the art and the impact of stronger consistency when compared with previous, weakly consistent solutions. More precisely, we address the following questions:

- What is the impact of our read protocol?
- How does the workload affect our read protocol?
- How does Conflict Manager cache size impact our performance?
- How scalable is our system?
- What is the overhead imposed by Snapshot Isolation?

4.2 Experimental Workbench

Our evaluation is based on executions of a prototype of our system in the experimental platform Grid'5000 using its dedicated servers. Each server is composed of 1 Intel Xeon Gold 5220 CPU with 18 cores, $96\,GB$ RAM and $480\,GB$ of SSD storage. The servers were connected by $25\,Gbps$ switches. The observed latency between clusters was approximately $0.15\,ms$. Clocks were synchronized using NTP.

For these experiments we were able to allocate 21 physical machines in a single cluster, used to run multiple virtual machines with the following physical machine distribution: 3 machines for client execution, 4 for Cloudburst function executors, 8 for Anna storage system, 4 for our conflict manager, 1 for the system managers (like the scheduler and monitoring systems of Cloudburst and others) and 1 machine for kubernetes control plane's components.

The experimental load used was based on the test benches used in [3]. The experiments were executed with 12 concurrent clients. Each client executes $2\,000$ function graph requests sequentially, with each graph composed of 6 functions, each executing 2 read requests for a total of 12 read operations per transaction. Apart from these requests, each client has a probability to execute a write function graph, writing 10 objects. We use a read/write ratio of 33%, similar value used in benchmarks such as TPC-C. The data set is composed of $100\,000$ keys, each with $2\,048$ bytes of data. The data set was split between 32 partitions, with a replication factor of 4 (i.e. each key has 4 replicas) and a gossip interval of 3 seconds (i.e. each partition propagates the latest updates of its replicas every 3 seconds). We use 21 executing nodes, each one with 4 threads (as pictured in Figure 3.1) so each client has always one executor available to execute its graph. Due to limitations of our workbench, we are only capable of scaling Cloudburst and the function executors to a limited number of clients. To stress-test the performance, we use a bandwidth limiter on the conflict manager machines.

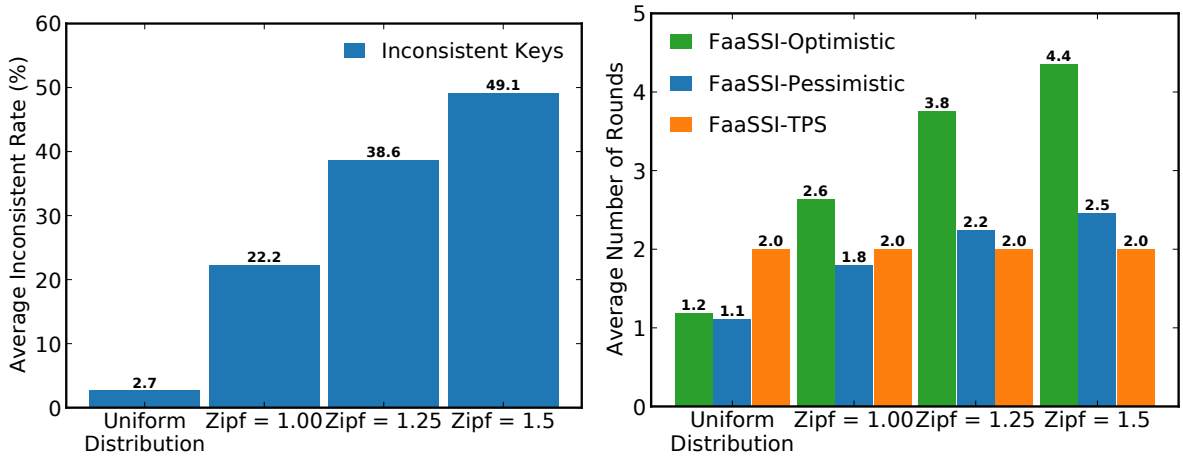
4.3 Read Protocol

We start by analysing the performance of our read protocol when compared with previous systems. More specifically, we compare our techniques with systems that impose all of the reading load in the intermediate layer like CloudTPS [22]. We implemented a version of CloudTPS as a variant of FaaSSI, which we have named FaaSSI-TPS. We measure the median and P99 latency of each system as well as the throughput. We use only one Conflict Manager and removed the value cache, with the version cache having unbounded memory. We vary the key access using an uniform and zipfian distribution. Each client has a unique distribution of the set of objects, and no transaction requests the same object twice to guarantee a larger fan-out of object access. We limited the bandwidth of the conflict manager machine to $100\,mbit/s$.

4.3.1 Effect of the Access Skew

We begin by studying the impact of the access skew on the percentage of reads that return inconsistent key values and on the number of average read rounds.

Considering the percentage of times a read returns an inconsistent value, the difference between the performance of FaaSSI-Pessimistic and FaaSSI-Optimistic is less of 1%, thus we have opted to present a single set of bars in Figure 4.1a (FaaSSI-TPS always returns consistent values). As shown, for uniform distributions, approximately 2.5% of of read operations return inconsistent results. For a skewed workload, using a Zipfian of 1, we observed that less than 25% of read operations return inconsistent values. However, this value can grow up to 50% with higher skews. The key observation is that, for lower skews, Anna returns consistent results in a large fraction of read requests.



(a) Percentage of inconsistent results returned by Anna

(b) Average number of communication rounds per read operation

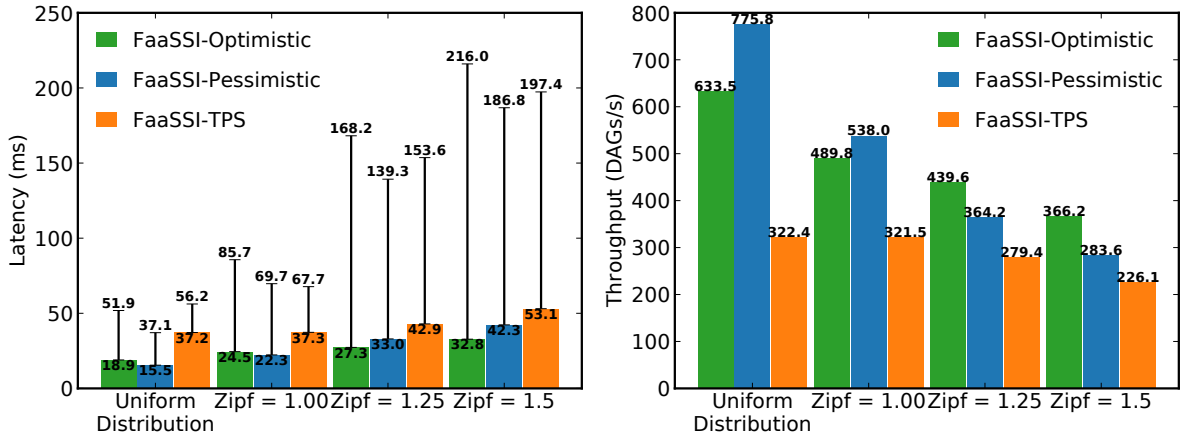
Figure 4.1: Skew effect on read protocol

Naturally, the probability of reading a consistent value from Anna impacts the average number of rounds needed to conclude a read operation, as depicted in Figure 4.1b. Since we are not using the value cache, FaaSSI-TPS will always require 2 communication rounds, as the conflict manager will always need to contact the Anna replica to obtain the value. If the key is consistent in storage, both FaaSSI-Pessimistic and FaaSSI-Optimistic will require 1 round. However, if the value is inconsistent, FaaSSI-Pessimistic may suffer a penalty of 2 extra rounds and FaaSSI-Optimistic may suffer an even larger round penalty (in our experiments, where we use 4 replicas, this can approximate 3 extra rounds, as the number of requests to storage is statistically correlated with the number of replicas present in Anna). The figure also shows that, for the uniform distribution, both FaaSSI-Pessimistic and FaaSSI-Optimistic are very close to only one communication round, as Anna returns mostly consistent results. In fact, FaaSSI-Pessimistic only contacts the conflict manager on 5% of the read operations to obtain values, corresponding to the 0.1 increase in average number of rounds. With the increase of access skew, the number of rounds slowly increase for FaaSSI-Pessimistic (because more requests require the value from the conflict manager), while FaaSSI-Optimistic quickly increases to its average of 4 rounds.

Note that the number of additional rounds does not matches directly the percentage of inconsistent read operations. This phenomenon is discussed more in depth in Section 4.3.2.

We now compare the performance of different techniques. As we can see in Figure 4.2a, both FaaSSI-Pessimistic and FaaSSI-Optimistic show less then half the median latency of transactions then FaaSSI-TPS, reducing the latency by 58% and 49% respectively when compared to FaaSSI-TPS. This translates in FaaSSI-Pessimistic and FaaSSI-Optimistic offering approximately twice the throughput of FaaSSI-TPS, as depicted in Figure 4.2b.

When the access skew increases, we see an increase in transaction latency on all systems, with



(a) Transactions Median Latency

(b) Transaction Throughput

Figure 4.2: System performance

FaaSSI-Pessimistic and FaaSSI-Optimistic still outperforming FaaSSI-TPS by 20% and 38% respectively. One could expect that FaaSSI-TPS would depict the same performance for different skews, given that the protocol always reads consistent versions from storage. However, with the increase of skew, we can observe an increase of load in the data replicas that serve the conflict managers. This phenomenon also affects FaaSSI-Pessimistic, that relies on the conflict managers to perform the second round when the first round fails. In fact, for higher skews, we even see FaaSSI-Optimistic outperforming FaaSSI-Pessimistic, even if it performs more read rounds on average, given that it allows to distribute the load of read operations among the different replicas. Finally, we can see that for all access distributions, FaaSSI-Optimistic shows a worse tail latency than FaaSSI-Pessimistic and, for skewed workloads, also worse than FaaSSI-TPS, as it may require multiple accesses to storage to obtain a consistent value.

We recall that in our experiments, Anna is executed using "in memory" storage mode, i.e., replicas store data in memory and not on disk. The higher overhead of fetching data from disk could impact the trade-offs of each system. We expand this topic on Section 5.2.

4.3.2 Impact of the Workload Skew

We now experimentally analyse how each factor introduced in Section 3.6 affects the amount of consistent keys returned by the storage layer and number of additional rounds. We use FaaSSI-Pessimistic to test these multiple factors, as the technique to obtain a consistent value is orthogonal to the factors we present. We use the workload presented in Section 4.2 and Section 4.3.1. Note that we do not discuss the effect of the data access skew, as it is already captured by Figure 4.1a. We use a zipfian distribution of 1 to test these factors and use no bandwidth limiter. We only evaluate for the case of FaaSSI-Pessimistic, as the number of inconsistent reads in the first round is roughly the same for both

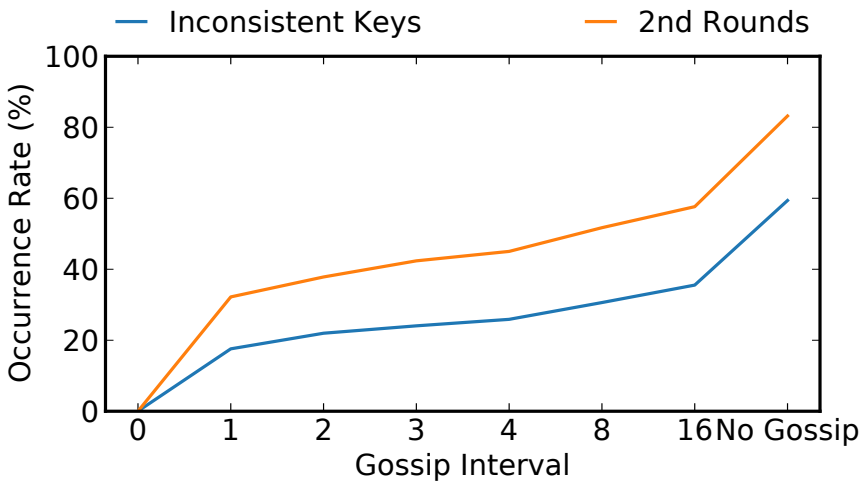


Figure 4.3: Gossip interval impact on read consistency

conservative and optimistic approaches.

As depicted in Figure 4.3, the gossip interval set in Anna affects significantly the number of inconsistent reads, as it takes longer for the storage layer to stabilize. When the value is set to infinite (i.e no gossips are ever sent), we would expect to observed an average 75% reads returning inconsistent values, as it would correspond to the average 25% chance of hitting the consistent replica in storage. The observed value in the experiments was close to this value, albeit smaller (60% of the reads return inconsistent values). The difference is due to the fact that not every key is written during the execution of the experiments, and read operations on keys that have not been written always return consistent values. Still, this indicates that the number of replicas has direct impact on the probability of reading inconsistent values. On the other hand, when the gossiping interval is set to 0 (i.e every update is gossiped immediately after applied) we observe a faction of inconsistent reads of only 0.01%. We also see an approximate increase of 18% in the number of inconsistent reads whenever the gossip interval is doubled. The number of second rounds grows consistently with the number of inconsistent reads. Note that although lower gossip interval values offer lower numbers of inconsistent reads (and, in consequence, fewer second rounds), this mechanism also increases the load on the replicas. Thus, a balance must be achieved between the speed of convergence and the load imposed on the system.

We now discuss the effect of the size of the write-set on the performance of FaaSSI. As shown in Figure 4.4, the number of inconsistent reads increases with the size of the write-set, showing an increment of approximately 20% every time the size of the write-set is doubled. Note however that, when large sizes of the write-set, the number of inconsistent reads start to stabilize. This is because the latency of the commit protocol also increases, giving more time to the storage to stabilize previously written values. In our experiment, the increase in the number of inconsistent reads stops when the size of the write-set approximates 64.

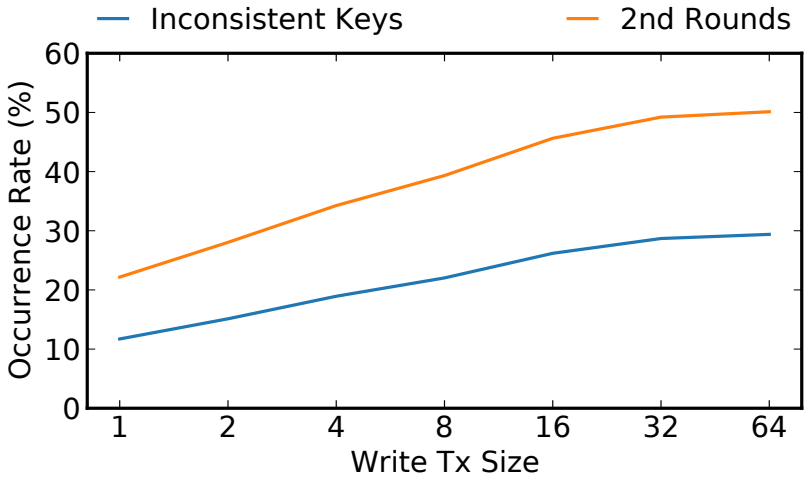


Figure 4.4: Impact of write-set size on the number of consistent reads

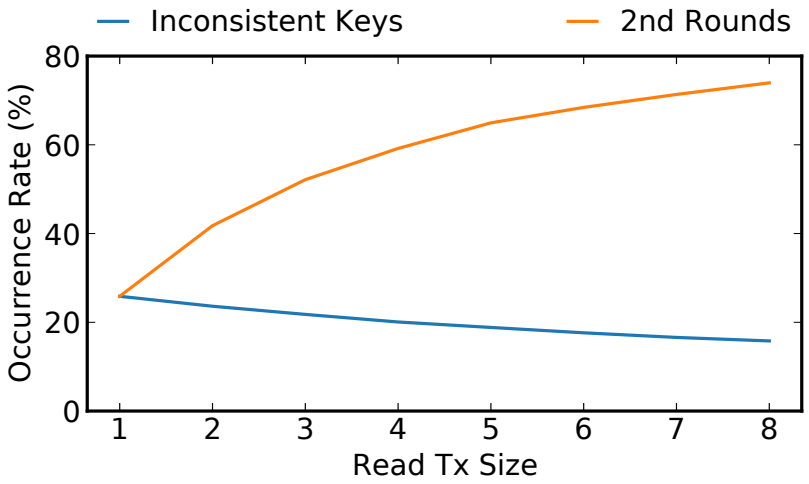


Figure 4.5: Impact of the number of reads per function on the number of consistent reads

Next we discuss the impact of the transaction length on the performance. We analyze the effect of the number of read operations per function and also the effect of the number of function in the DAG.

Figure 4.5 was obtained with a DAG of 6 functions where the number of read per function was varied. We can observe that, when the number of reads per function increases, the number of inconsistent reads decreases but the number of times a second round is needed increases. This appears to be a contradiction but can be explained as follows: because reads are performed in parallel, a single inconsistent read requires a function to perform a second round, even if all the other reads are consistent. Furthermore, given that each function takes more time when a second round is required, this also gives more time for the snapshot to stabilize, and decreases the probability of getting inconsistent reads in subsequent functions.

Figure 4.6 was obtained by keeping two reads per function and increasing the number of functions in the DAG. In this case, as the DAG size increase, we can observe a decrease in the number of

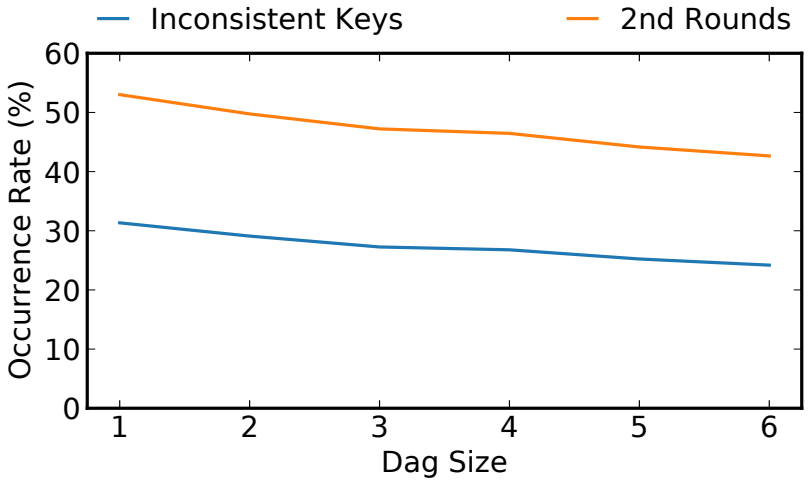


Figure 4.6: Impact of DAG size on read consistency

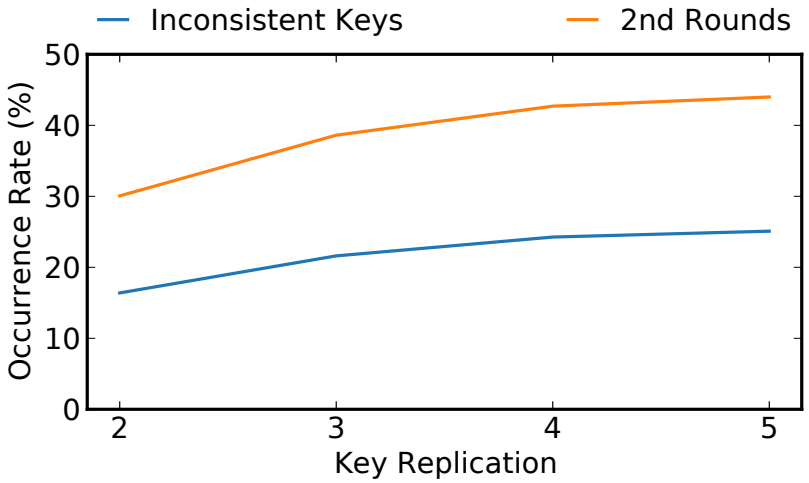


Figure 4.7: Impact of the number of storage replicas on read consistency

inconsistent reads and in the number of times a second round is needed. This happens because, with longer DAGs, Cloudburst suffers from a higher overhead when passing the transaction context from function to function, which increases the latency and gives more time to stabilize the values in the storage.

Finally, we study how the replication factor of keys affect the number of consistent reads. Figure 4.7 was obtained by varying the number of replicas used in Anna. When using a replication factor of 1, all values returned are consistent as there is only one copy of the value, and as such has a 0% rate of retries. When the number of replicas increases, it becomes more likely that a function reads from a replica that is not updates, and obtains an inconsistent value. It can be observed that, when using a key replication of 2, we get a rate of inconsistent reads of approximately 16%. As the number of replicas increase, the value starts to stabilize with a value of approximately 25% inconsistent reads. This behaviour is due to the number of writes in the transaction and gossip interval, showing that key

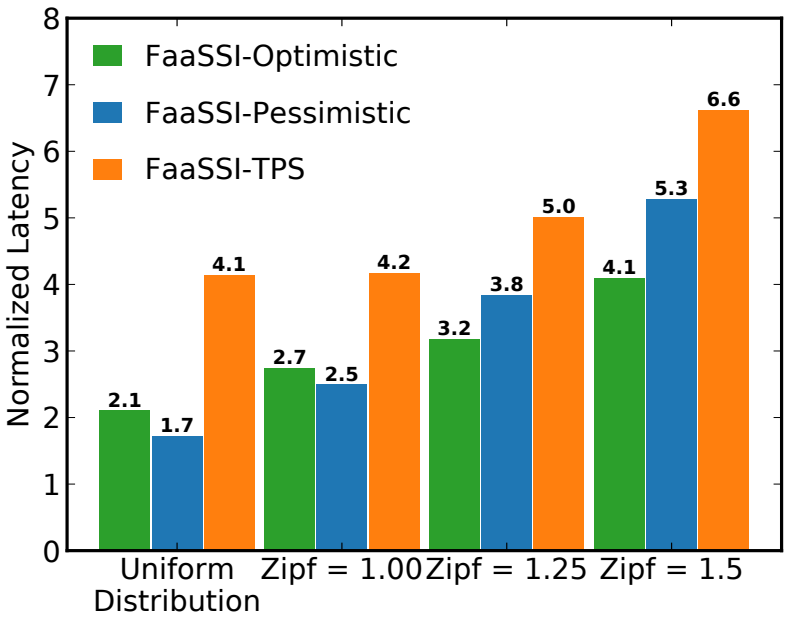


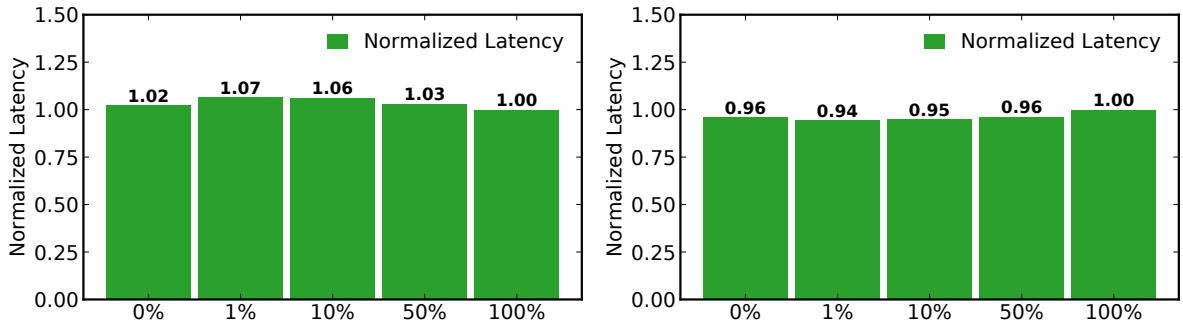
Figure 4.8: Normalized Latency w.r.t eventual consistency

replication does not drastically affects the probability of obtaining an inconsistent value when reading.

4.4 Eventual Consistency Comparison

We now analyse the impact of enforcing snapshot isolation when compared with an eventual consistency baseline that uses only Cloudburst and Anna. We use the same test bench as Section 4.3.1. We normalize the performance of our multiple techniques with regard to the latency of the eventually consistent system. As shown in Figure 4.8, our solution incurs a 71% overhead over eventual consistency for read transactions on uniform distributions, while FaaSSI-TPS shows up to $4\times$ the latency of the eventual consistent solution. This overhead is mainly due to the contact with the conflict manager, which even though for an uniform distribution FaaSSI-Pessimistic and FaaSSI-Optimistic reads only require one round, the scalability of the conflict manager is still lower than that of Anna, leading to a latency increase. With the increase in skew, FaaSSI performance degrades, as already depicted in Section 4.3.1, while the eventual consistency system slightly improves in performance.

As such, we show that our system is able to significantly reduce the overhead imposed by snapshot isolation in comparison with the state of the art.



(a) Version Cache Size Impact

(b) Value Cache Size Impact

Figure 4.9: Impact of the cache size on performance

4.5 Cache Size

We now analyse the impact on the performance of the size of the version and value cache maintained by the conflict managers. We aim at analyzing how different cache sizes affect the system performance. We use FaaSSI-Pessimistic to test both caches. We use a key granularity for the cache replacement policy, with each key holding a maximum of 3 versions in cache. We use the same workload as in Section 4.3. We removed the bandwidth limiter and use a zipfian distribution of 1. We vary the cache sizes as a percentage of the total number of keys in the system, using a total of $100\,000$ keys. We present our results in Figure 4.9. We normalize the results w.r.t to the maximum cache size latency.

We first analyse the impact of the version cache, using a value cache size of 0. As depicted in Figure 4.9a, the cache size appears to have little impact on the performance: the values obtained with different cache sizes are within 10% of the value obtained when all objects can be cached. Even the system with the cache disabled exhibits a performance that is only 2% worse than the performance of the system that can cache all objects.

When looking at how the size of the value cache affects the performance, we have observed a similar trend, as depicted in Figure 4.9b. In fact, the observed results are somehow unexpected: maintaining a cache of key values at the conflict managers has a negative impact on the performance. We speculate that, since all machines are running in the same cluster connected by a high-speed network, the latency added from remote reads has a lower impact than the computational overhead of managing the cache. In the future, we plan on further investigating these results and optimizing the caching system for improved performance.

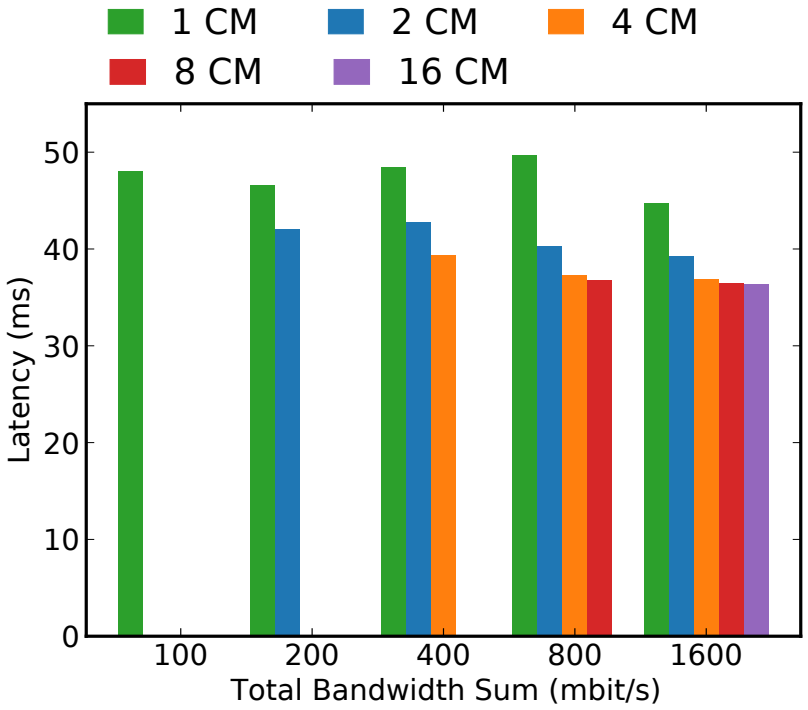


Figure 4.10: Scalability of the conflict managers

4.6 Conflict Manager Scalability

Finally, we analyse how the use of multiple conflict managers affects the system performance. We first evaluate the read performance with increasing number of conflict managers. Once again we use the base workload of Section 4.3. We use a zipfian distribution of 1.5 to evaluate the worst-case scenario in terms of conflict manager load using FaaSSI-Pessimistic. As we increase the number of conflict managers in the system, we proportionally decrease the bandwidth limiter of each machine, maintaining a constant bandwidth to the set of conflict managers.

Note that, in this experiment, the workload is the same for all configurations. Thus, the latency observe when using a single conflict manager (left green bars) should be approximately the same, regardless of the bandwidth, as the bottleneck is the serialization of requests at the manager. The plots depicted in Figure 4.10 show some fluctuation around $48\,ms$, which appears to be noise from the experiments. A similar reasoning could be applied to the results with two conflict managers (blue bars), etc.

For any fixed value of the aggregate bandwidth, the Figure 4.10 shows that the latency decreases as we increase the number of conflict managers, given that different requests can be processed in parallel by different conflict managers. However, the performance gains in terms of response parallelism appears to be capped at 4 conflict managers, as we see the diminishing returns as more conflict managers are added. For the configuration with total bandwidth of $1\,600\,mbit/s$, the latency decrease when using 16

conflict managers is approximately 25%.

Summary

In this chapter we have addressed the multiple evaluation goals that we have defined to answer. We have shown that both FaaSSI-Pessimistic and FaaSSI-Optimistic outperform the current state of the art by nearly double the latency and throughput values. In our experiments, Anna was able to return consistent values in most cases, allowing our protocols to be efficient and respond to most requests in one round trip. We showed that cache size offer little gains in a datacenter environment. We show that our system is able to scale with additional conflict managers and show the impact of providing snapshot isolation in comparison with an eventually consistent baseline.

5

Conclusion

Contents

5.1 Conclusions

In this thesis, we have addressed the problem of offering transactional storage access to Function-as-a-Service applications. In particular, we have studied mechanisms that can augment a weakly storage service with support to snapshot isolation. The benefits of using a weakly storage service in this context is that applications that require strong consistency may coexist with applications with weaker requirements without imposing any performance penalty on the latter. We have designed and evaluated a system, that we have named FaaSSI, to achieve this goal. Like previous works that aims at achieving similar goals, FaaSSI uses a set of consistency servers that are used to validate transactions when they attempt to commit. However, unlike previous work, we do not require all read operation to be routed via the consistency servers. We combine an optimistic access to the storage service with a parallel request for metadata to the consistency servers. We propose multiple protocols for maintaining consistency even in cases where the updates are still being propagated among replicas. For lightly skewed workloads, this allows the workers to read consistent versions directly from storage in a single round, alleviating the load imposed on the consistency servers. This increases the scalability of the system.

5.2 Future Work

In the future, we plan to study mechanisms that can reduce the average latency of the system. In particular, we plan on investigating the effects of a implementing caching layer, to be maintained by the workers. Note that to support snapshot isolation, the caches at different workers must return values from the same snapshot to all functions that execute on behalf of the same application, something that can be achieve using techniques similar to those supported by HydroCache or FaaSTCC. We also plan on adding an adaptive system to the worker nodes, which depending on the skew can either utilize FaaSSI-Pessimistic or FaaSSI-Optimistic, and test it using Anna “in disk” storage mode. Finally, we plan to implement a stateful FaaS-oriented version of the TPC-C [40] benchmark and make a performance study of our system against related work using this benchmark.

Bibliography

[1] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endow.*, vol. 7, no. 3, p. 181–192, Nov. 2013. [Online]. Available: https://doi.org/10.14778/2732232.2732237

[2] "Jepsen consistency models," https://jepsen.io/consistency, accessed: 11/12/2020.

[3] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, Jun. 2020, p. 83–97. [Online]. Available: https://doi.org/10.1145/3318464.3389710

[4] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein, "Anna: A KVS for any scale," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, Paris, France, Apr. 2018, pp. 401–412.

[5] T. Lykhenko, "Efficient implementation of causal consistent transactions in the cloud," Master's thesis, Instituto Superior Tecnico, Universidade de Lisboa, Nov. 2019.

[6] "Lambda," https://aws.amazon.com/lambda/, accessed: 11/12/2020.

[7] "Google cloud function," https://cloud.google.com/functions, accessed: 11/12/2020.

[8] "Microsoft azure functions," https://azure.microsoft.com/en-us/services/functions/, accessed: 11/12/2020.

[9] "AWS lambda customer case studies," https://aws.amazon.com/lambda/resources/customer-case-studies/, accessed: 7/10/2021.

[10] "Amazon s3," https://aws.amazon.com/s3/, accessed: 11/12/2020.

[11] "Amazon dynamodb," https://aws.amazon.com/dynamodb/, accessed: 11/12/2020.

[12] "AWS step functions," https://aws.amazon.com/step-functions/, accessed: 7/10/2021.

[13] "Google cloud services," https://cloud.google.com/, accessed: 11/12/2020.

[14] "Microsoft azure services," https://azure.microsoft.com/en-us/services/, accessed: 11/12/2020.

[15] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," ser. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, Apr. 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387535

[16] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, Jul. 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407836

[17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*. Singapore: Springer Singapore, Dec. 2017, pp. 1–20. [Online]. Available: https://doi.org/10.1007/978-981-10-5026-8_1

[18] H. Garcia-Molina and K. Salem, "Sagas," *SIGMOD Rec.*, vol. 16, no. 3, p. 249–259, Dec. 1987. [Online]. Available: https://doi.org/10.1145/38714.38742

[19] "AWS step function transaction saga pattern," https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/implement-the-serverless-saga-pattern-by-using-aws-step-functions.html, accessed: 7/10/2021.

[20] K. Spirovska, D. Didona, and W. Zwaenepoel, "Wren: Nonblocking reads in a partitioned transactional causally consistent data store," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Luxembourg City, Luxembourg, Jun. 2018, pp. 1–12.

[21] T. Lykhenko, R. Soares, and L. Rodrigues, "Faastcc: Efficient transactional causal consistency for serverless computing," in *Proceedings of the 22nd ACM/IFIP International Middleware Conference*, ser. Middleware '21. Virtual Event, Canada: Association for Computing Machinery, Dec. 2021. [Online]. Available: https://doi.org/10.1145/3464298.3493392

[22] W. Zhou, G. Pierre, and C.-H. Chi, "Cloudtps: Scalable transactions for web applications in the cloud," *IEEE Trans. Serv. Comput.*, vol. 5, no. 4, p. 525–539, Jan. 2012. [Online]. Available: https://doi.org/10.1109/TSC.2011.18

[23] V. Padhye, "Transaction and data consistency models for cloud applications," Ph.D., University of Minnesota, Minneapolis, MN, USA, Feb. 2014.

[24] J. Schleier-Smith, L. Holz, N. Pemberton, and J. M. Hellerstein, "A FaaS file system for serverless computing," *CoRR*, vol. abs/2009.09845, Sep. 2020. [Online]. Available: https://arxiv.org/abs/2009.09845

[25] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, “Fault-tolerant and transactional stateful serverless workflows,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1187–1204. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[26] P. Gill, N. Jain, and N. Nagappan, “Understanding network failures in data centers: Measurement, analysis, and implications,” *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, p. 350–361, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2043164.2018477

[27] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[28] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, “Scalable atomic visibility with RAMP transactions,” *ACM Trans. Database Syst.*, vol. 41, no. 3, Jul. 2016. [Online]. Available: https://doi.org/10.1145/2909870

[29] R. Guerraoui and M. Kapalka, “On the correctness of transactional memory,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’08. New York, NY, USA: Association for Computing Machinery, 2008, p. 175–184. [Online]. Available: https://doi.org/10.1145/1345206.1345233

[30] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro, “Fast general distributed transactions with opacity,” ser. SIGMOD ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 433–448. [Online]. Available: https://doi.org/10.1145/3299869.3300069

[31] A. Adya, “Weak consistency: A generalized theory and optimistic implementations for distributed transactions,” Ph.D., MIT, Cambridge, MA, USA, Mar. 1999, also as Technical Report MIT/LCS/TR-786.

[32] J. Du, S. Elnikety, and W. Zwaenepoel, “Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks,” in *Proceedings of the 2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, ser. SRDS ’13. Braga, Portugal: IEEE Computer Society, Oct. 2013, p. 173–184. [Online]. Available: https://doi.org/10.1109/SRDS.2013.26

[33] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier, “Logic and lattices for distributed programming,” in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2391229.2391230

[34] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS’11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 386–400.

[35] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkipati, P. Chandra, and A. Vahdat, “Sundial: Fault-tolerant clock synchronization for datacenters,” in *OSDI*, Nov. 2020, pp. 1171–1186.

[36] “ZeroMQ,” http://zeromq.org/, accessed: 10/2/2021.

[37] “Protocol buffers,” https://developers.google.com/protocol-buffers, accessed: 10/2/2021.

[38] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, May 1998.

[39] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, “Spanner: Google’s globally distributed database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2491245

[40] “Tpc-c,” http://www.tpc.org/tpcc/, accessed: 22/9/2021.