Snapshot Isolation for Serverless Computing Environments

João Rafael Soares
joao.rafael.pinto.soares@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Function as a Service (FaaS) is a relatively recent paradigm offered by several cloud providers; it allows the development of applications as a composition of functions that can be executed without the prior allocation of servers. Current FaaS implementations offer very weak data consistency guarantees to functions that access shared data. The study of mechanisms that can offer stronger guarantees on FaaS in an efficient manner is still a research topic. In this work, we address to problem of offering strong consistency to FaaS applications. We survey the state-of-the-art Cloud and FaaS systems and their consistency guarantees, analysing their tradeoffs in terms of latency, memory usage and consistency strength. Based on this analysis, we propose a new architecture to support transactional support, offering Snapshot Isolation, in Serverless Computing Environments.

Table of Contents

1 Introduction

Serverless Computing, also known as Function as a Service (FaaS), is a relatively recent paradigm offered by several cloud providers. It allows programmers to run their applications in the form of high level code functions uploaded in the cloud. These functions can be executed without the prior allocation of servers, that are automatically managed by the cloud provider, that dynamically increases or decreases the resources allocated to the execution of each function in response to demand fluctuations. When using this service, clients are billed based on the actually used processing power, contrary to other cloud offerings where resources need to be rented and billing is proportional to the rented time, regardless of the actual usage.

Serverless computing disaggregates the computational and the storage layers, allowing cloud providers to perform elastic scaling for each of these layers independently. The computational layer consists of multiple physical nodes containing executor threads, where each executor handles the execution of a single function. A given functionality of an application may require the execution of multiple functions. Because different functions may be executed by different executor nodes, they may observe inconsistent versions of the same data. In fact, current FaaS implementations are unable to guarantee even the weakest forms of consistency, such as *Read your Writes*[1]. In turn, this may require programmers to code compensating actions that correct the results of observing inconsistent states[2].

In theory, it would be interesting to offer strong consistency to applications using the FaaS paradigm, including the support for running transactions that span multiple functions and can offer ACID properties. The challenge is to offer this support in an efficient manner. First, to ensure consistency, functions are required to coordinate and to exchange information about the snapshot they need to read. Second, the FaaS paradigm requires functions to be stateless and to rely on shared backends and stateful storage services to share state. In many cloud environments, these backends also do not support strong consistency. This may force functions to read multiple time from the storage service, in order to obtain a version that is consistent with the version previously read by other functions in the same transaction.

It is worth noting that storage for serverless computing does not usually offer strong consistency due to the coordination costs associated with the enforcement of strong semantics, instead turning to highly scalable, low latency storage services like Anna and Redis [3, 4]. There is an inherent tradeoff between the consistency level and the performance of the system. Thus, in our work we will be partially concerned with the performance aspects and scalability aspects associated with the implementation of strong consistency. To support our design, this report makes a survey of the current state-of-the-art systems and their approach to enforce consistency levels efficiently, studying their design principles and the tradeoffs they between consistency and performance.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 to 6 we present all the

background related with our work. Section 7 describes the proposed architecture to be implemented and Section 8 describes how we plan to evaluate our results. Finally, Section 9 presents the schedule of future work and Section 10 concludes the report.

2 Goals

This work addresses the problem of supporting transactional consistency support for applications implemented using the Function as a Service paradigm. More precisely:

> *Goals:* We aim at extending the current Function as a Service architectures with an additional layers and services to allow the execution of transactions, that span multiple functions, offering strong consistency criteria, such as snapshot isolation.

In the design of our system we plan to leverage recent research results from works that have augmented the consistency level supported by FaaS, for instance, to offer transactional causal consistency[5]. The results from these works can give us insights on the tradeoffs involved in the development of strong consistency. The project will produce the following expected results:

> *Expected results:* The work will produce i) a survey of state-of-the-art FaaS and cloud systems; ii) an implementation of new consistency layer and services for FaaS, iii) an extensive experimental evaluation of the performance and scalability of the resulting system.

3 Serverless Computing

We start by making an overview of some of the most popular and commercial FaaS systems and on how consistency is applied overall in Serverless Computing environments.

3.1 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[7] and Microsoft Azure Functions[8]. Any of these platforms supports functions written in different programming languages, such Python, Java, or C#, among others, that can invoke other services provided by the cloud provider: Lambda function can use Amazon S3[9] and DynamoDB[10], GCF functions can use the Google Cloud services[11], and Azure functions can use other Azure services[12]. When using these services, the ease of programming comes with some drawbacks: not only the code becomes

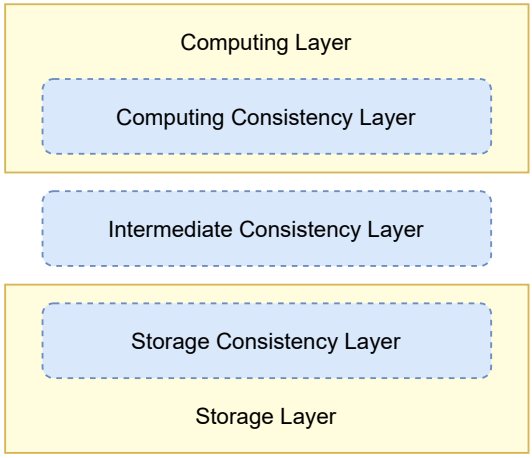

Fig. 1. Consistency layers. Yellow boxes represent the main FaaS layers, blue boxes are the optional. consistency layers

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[5, 13], 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[14] is a FaaS system that aims at supporting the development of stateful FaaS programs that uses a low latency autoscaling key-value store (Anna[3]) and a cache co-located with the executors to reduce latency the latency in the execution of transactions. A more detailed overview of current FaaS services can be found in [15].

3.2 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. The data consistency observed by applications is the results of the interaction between the computing layer and the storage layer, which can be supported by an intermediate helper layer, as illustrated in Figure 1. In the following, we discuss how each of these layers can contribute to enforce data consistency for FaaS applications.

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 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[5].

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. Many of the storage services that can be seamlessly used with FaaS offer weak consistency guarantees, as they favor scalability over consistency. Another strategy to increase the consistency level provide to FaaS applications is to augment the cloud storage service with support to offer stronger guarantees. In some cases, this can be implemented by wrapping an underlying weak consistent storage service with a software layer that intercepts requests in the storage nodes. The wrapper may implement the necessary coordination among storage servers, namely among servers from different partitions, to ensure that only consistent versions of data become visible to clients. The required coordination may assume different forms, such as the execution of a two-phase commit among different storage nodes[16], storing multiple versions of the same item to increase the chances of returning a version consistent with the snapshot observed by the client[17, 18], among others. Note however that, 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.

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[13]. Similar approaches have been used for cloud systems like [19, 20]. 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.

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 [21]. Also, implementing a wrapper for stronger consistency on the storage layer can help the intermediate layer to fetch a consistent version quicker[20].

In summary, each layer implementation brings particular advantages and disadvantages. The computing layers 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.

4 Consistency Concepts

Currently, 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 [13, 5], 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)*[22] 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 ..."[23]. Thus, the practical drawback on 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 [22] and [24].

4.1 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.

4.2 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 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 (due to space constraints, we omit a few variants; the interested reader can find a more thorough discussion in [22, 24]).

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

Monotonic Reads (MR) 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 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.

Read your Writes (RYW) requires that a read by a client always observes a state where all write operations previously performed by that client have already been applied.

Pipelined Random Access Memory (PRAM) 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.

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. Then, 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.

Causal Consistency (CC), derived from Lamport's "happens-before" relation[25], is the combination of Write Follows Read with PRAM. It's known as one of the strongest consistency level that is still highly available.

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

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.

4.3 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 de 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) 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. 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.[22].

Monotonic Atomic View (MAV) [22] 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) [16] 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.

Transactional Causal Consistency (TCC) 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.

Snapshot Isolation (SI) 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) 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) 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.

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 [22, 26].

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 1. Fuzzy Read anomaly

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

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.

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

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

Table 3. Fractured Read anomaly

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

Table 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 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
RR	✓	✗	✗	✗	✗	✗
MAV	✗	✓	✗	✗	✗	✗
RA	✓	✓	✓	✗	✗	✗
TCC	✓	✓	✓	✗	✗	✗
SI	✓	✓	✓	✓	✗	✗
1SR	✓	✓	✓	✓	✓	✗
Strong 1SR	✓	✓	✓	✓	✓	✓

Table 6. Anomaly overview

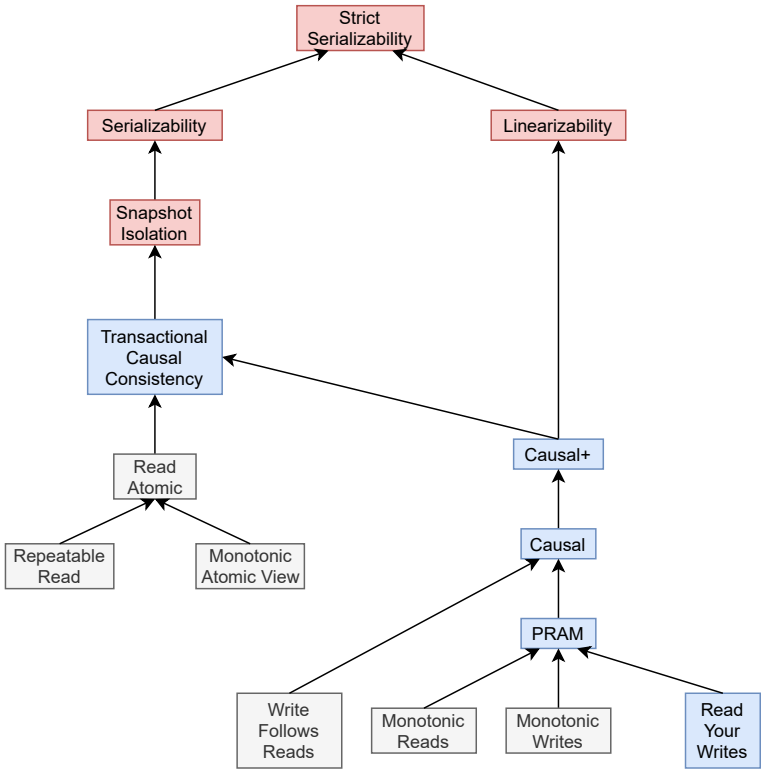

Fig. 2. Consistency Guarantees by strength. Grey box means HAT compliant, blue box means HAT-sticky compliant, red box means non-HAT compliant.

5 Related Work

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.

5.1 Cloud Systems

RAMP RAMP[16] 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 [16], 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 key versions written on its transaction 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.

FastCCS FastCCS[17] 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.

Wren Wren[18] 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.

Clock-SI Clock-SI[27] 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.

Padhye Padhye PhD thesis[20] 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.

CloudTPS CloudTPS[19] provides transactional support to cloud environments on top of a weakly consistent storage by creating an intermediate transactional layer, focusing on providing ACID properties. This layer is composed of several Local Transaction Managers (LTM), partitioning the key space among them. Each LTM holds 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 as they are replicated through IPC calls.

The system itself does not implement any specific consistency level. 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. However, for the sake of comparison, we will consider CloudTPS as a Snapshot Isolation system.

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

have commited or aborted. This timestamp ordering is assigned by an external timestamp manager.

5.2 FaaS Systems

AFT Atomic Fault Tolerant shim[13] 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.

Hydrocache Hydrocache[5] 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, Optimistic 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 it runs the Optimistic first try, worst case it runs both the Optimistic and Conservative approach, averaging better latency than Conservative and tail latency than Optimistic.

FaaSFS FaaSFS[21] is a shared File System built specifically for FaaS, offering POSIX semantics and linearizability 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.

6 Analysis

We now discuss the advantages and disadvantages of the systems surveyed in the previous section. Table 7 presents a comparative analysis of the different features of each system.

Systems	Target	Consistency	Storage Consistency	Metadata Size	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	SI	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*

Table 7. System Comparison. W represents writeset, N represents number of partitions, K is key-space, R is number of replicas, Nodes is the number of tree nodes. * represents that action may include blocking or aborts.

6.1 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.

6.2 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 RTTs 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.

6.3 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).

6.4 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 numebr 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 totallky ordered). Therefore, they only use a single timestamp as metadata; this timestamp is enough to represent a global state of the system.

6.5 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 is an outlier in storage consistency and memory usage tradeoff, as maintaining Strong 1SR efficiently requires heavy memory usage.

6.6 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.

6.7 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. CMost 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 only requires 1 RTT for commit, however it has a much higher abort rate due to its strict serializability.

Note that strong consistency levels have either locking mechanisms or optimistic approaches that may abort on commit time. These extra costs must be taken into account in the system performance. Transactions on SI systems must wait for concurrently running transactions with a lower snapshot to commit, as conflicts may arise from its writeset.

7 Architecture

We plan to develop a middleware layer, named FaaSSI, to offer Snapshot Isolation to transactions executing in the FaaS model. The layer will mainly consist of a set of servers, that work as an intermediate layer, between the executors and the storage. More precisely, we plan to implement FaaSSI as an extension to the Cloudburst system, and our intermediate layer will sit between the Cloudburst cache and the Anna storage.

7.1 System Components

FaaSSI will use two types of servers, as depicted in Figure 3: a set of conflict managers and a timestamp management server.

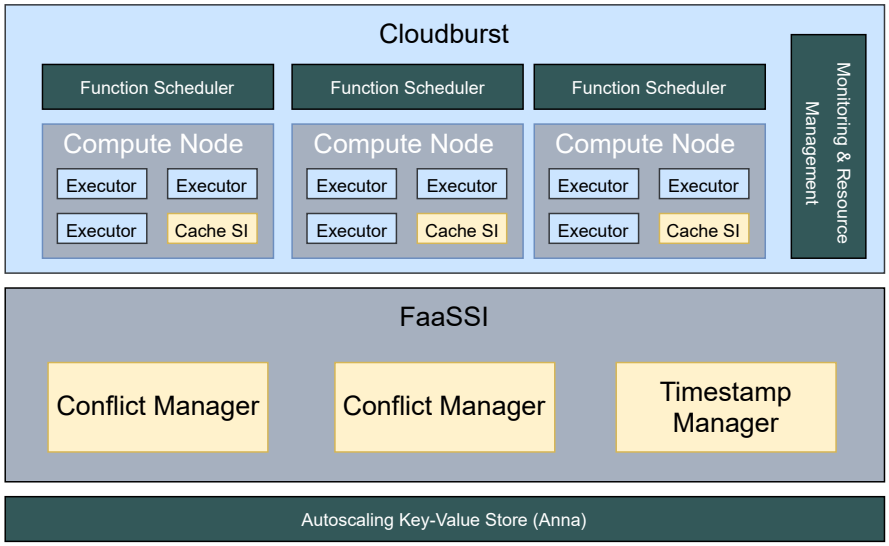

Fig. 3. FaaS-SI transactional layer. Based on the diagram from [5]. Yellow squares represent our additions.

Conflict Managers The conflict managers will be in charge of checking if transactions satisfy SI and commit or abort the transaction accordingly. We follow the approach of CloudTPS [19], where each conflict manager will be in charge of a different data partition, keeping copies of the latest committed values in memory in order to serve clients that need to access the most recent version. The executor that runs the sink function of a transactional DAG sends all the writes to the corresponding conflict managers. Transactions that span multiple partitions will require coordination among the conflict managers for safe committing, relying on a 2-Phase Commit protocol. The commit protocol decides if the transaction can commit or needs to abort. If the transaction commits, writes are written in stable storage by the managers.

The size of the memory pool of each conflict manager will be a system parameter. The conflict manager will discard old values, using some policy (at this point, we plan to use the Least Recently Used policy to release entries in the cache). Because all writes to a given item are always performed by the conflict managers, managers can always retrieve any given version of an object, even if it has already been purged from its memory. This in ensured because managers can always read their own writes, even for the weakest consistency models provided by the cloud storage system.

Conflict managers will also be in charge of telling executioners which is the most recent committed snapshot, to be used as the snapshot that a newly started transaction will read from. Transactions obtain the transaction snapshot during the first read operation. For this purpose, they will contact the conflict manager in charge of the first object that is read by the transaction; this conflict manager returns the timestamp corresponding to its latest committed transactions it is aware of.

Timestamp Management Server The timestamp management server will be responsible for assigning unique, totally ordered timestamps for committing transactions. The timestamp corresponds to a monotonically increasing sequence number, obtained by the conflict manager to represent the system state on commit time. When a transaction spans multiple conflict managers, the system will select deterministically which manager will request the commit timestamp.

Cache SI We will modify the Cloudburst Cache implementation to subscribe to updates from the Conflict Managers in order to increase the probability of serving clients with the most recent version from the executor cache. To maintain consistency when moving execution between physical nodes, the transaction snapshot will be passed on between caches.

7.2 Executing Read, Write, and Commit Operations

Read operations When executing read operations, executors first read optimistically from the cache. In case of cache-miss or requiring an older key version due to the chosen transaction snapshot, the cache contacts simultaneously the

storage service to obtain a copy of the data and the conflict manager to check what is the most recent version the transaction should read from given the desired snapshot. If the storage service serves an non-matching version with the version returned by the conflict manager (this can happen because the cloud storage service only guarantees Read Your Writes eventual consistency), the cache will contact the conflict manager to obtain the correct version.

Write operations For write operations, only on transaction commit, once the transaction SI properties have been verified, will the transaction updates be applied to storage and updated onto each conflict manager. During transaction execution, the writeset will be propagated throughout the transaction and used for conflict detection by the conflict managers.

Commit operations At the end of transaction execution, the sink function of the transaction must send the transaction writeset and readset to the corresponding conflict managers to verify if SI was upheld. The conflict managers will verify if any write conflicts exist with concurrently committing transactions or with committed updated occurred during execution. The readset must also be verified, as optimistic cache reads may have read an older version than the one required by the snapshot timestamp. Concurrency is handled using a locking mechanisms, where conflict managers can only verify once it obtains a lock corresponding to every key in its writeset and readset. Once both sets are verified, the conflict managers inform the coordinator. If no verification has failed, the coordinator sends the commit message to the participant conflict manager, so they may update its corresponding keys in storage and updates its local information, completing the transaction. Otherwise, the transaction must abort and be retried.

7.3 Incremental Implementation

We plan to develop our system in an incremental manner. In the first implementation we will use a single conflict manager, which will avoid the need for distributed coordination during transaction commit. Later, we plan to extend this version to support multiple conflict managers.

8 Evaluation

We plan to implement a prototype of our system and to deploy in a experimental testbed, such as GRID5000[28], that is compatible with the Cloudburst middleware. To ease the deployment of the experiments we plan to use kubernets to deploy the executors. We will assess the feasibility of deploying in the same infrastructure some representative examples of the previous work, offering different guarantees, for both traditional cloud applications and for FaaS applications, such that we can have comparative results. The evaluation will focus on multiple aspects of the system performance, such as transaction latency, throughput, abort rate, scalability, fault tolerance, and memory usage.

8.1 Latency

We will measure the average transaction execution time for different workloads. We plan to vary the read/write ratio, the number of functions that are part of the transcation, the number of objects accessed by each function, and also the type of function DAG used by the transactions (including the serial and parallel execution of functions[5]). We plan to understand how the different components of our architecture contribute to the latency, including the time required to obtain a timestamp to execute the transaction and the time required to perform the commit protocol.

8.2 Throughput

We will measure the maximum number of transactions per second that the system is able to execute, while still ensuring response time under some predefined latency threshold in the 99% percentile, as suggested in [19].

8.3 Abort Rate

We will also measure the abort rate for different workloads. The abort rate depends on several factors, including the characteristics of the workload (in particular the skew in data access), the level of consistency provided, and the latency added to the execution of transactions (the longer the transactions, the more likely are conflicts to occur).

8.4 Scalability

We will measure the impact of system size on the performance of the system. We plan to vary the data-set size, the number of partitions used in the storage system, the number of executors, and the number of clients.

8.5 Fault Tolerance

We will measure the impact of failures in the system. We will consider failures in the servers of the middle layer used to enforce SI and also failures in the executors. In this context, we will also measure the recovery time observed for the different types of faults.

8.6 Memory Usage

We will study the relation between the memory used by the components in the middle layer and the performance of the system. For that purpose, we will execute the system by putting different limits on the memory used by our servers.

9 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15: Deliver the MSc dissertation.

10 Conclusions

Serverless computing brings a new way of application development, allowing for a more fine-grained and cost-efficient scaling and deployment without worrying about prior allocation of infrastructure. Currently there is a hole for application that want to go Serverless but require stronger consistency.

In this report, we surveyed the state-of-the-art FaaS and Cloud systems. We analysed their consistency guarantees, discussing gains and tradeoffs of each consistency level and implementation. We elaborated a solution to support strong transactional consistency in Serverless Computing while maintaining an acceptable latency in this environment. Finally, we defined the evaluation methods and presented the scheduling for future work.

Acknowledgments We are grateful to Taras Lykhenko for the fruitful discussions and comments during the preparation of this report. This work was partially supported by the FCT via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271) and project UIDB/ 50021/ 2020.

References

1. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M.J., Theimer, M.M., Welch, B.B.: Session guarantees for weakly consistent replicated data. In: Proceedings of the Third International Conference on on Parallel and Distributed Information Systems. PDIS '94, Autin, Texas, USA, IEEE Computer Society Press (September 1994) 140–150
2. Santos, N., Rito Silva, A.: A complexity metric for microservices architecture migration. In: Proceedings of the IEEE International Conference on Software Architecture (ICSA), Salvador, Brazil (November 2020) 169–178
3. Wu, C., Faleiro, J., Lin, Y., Hellerstein, J.: Anna: A KVS for any scale. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France (April 2018) 401–412

4. Redis. https://redis.io/ Accessed: 11/12/2020.
5. Wu, C., Sreekanti, V., Hellerstein, J.M.: Transactional causal consistency for serverless computing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. SIGMOD '20, Portland, OR, USA, Association for Computing Machinery (June 2020) 83–97
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. Amazon s3. https://aws.amazon.com/s3/ Accessed: 11/12/2020.
10. Amazon dynamodb. https://aws.amazon.com/dynamodb/ Accessed: 11/12/2020.
11. Google cloud services. https://cloud.google.com/ Accessed: 11/12/2020.
12. Microsoft azure services. https://azure.microsoft.com/en-us/services/ Accessed: 11/12/2020.
13. Sreekanti, V., Wu, C., Chhatrapati, S., Gonzalez, J.E., Hellerstein, J.M., Faleiro, J.M.: A fault-tolerance shim for serverless computing. EuroSys '20, Heraklion, Greece, Association for Computing Machinery (April 2020)
14. Sreekanti, V., Wu, C., Lin, X.C., Schleier-Smith, J., Gonzalez, J.E., Hellerstein, J.M., Tumanov, A.: Cloudburst: Stateful functions-as-a-service. Proc. VLDB Endow. **13**(12) (July 2020) 2438–2452
15. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P. In: Serverless Computing: Current Trends and Open Problems. Springer Singapore, Singapore (December 2017) 1–20
16. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst. **41**(3) (July 2016)
17. Lykhenko, N.: Efficient implementation of causal consistent transactions in the cloud. Master's thesis, Instituto Superior Tecnico, Universidade de Lisboa (November 2019)
18. Spirovska, K., Didona, D., Zwaenepoel, W.: 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 (June 2018) 1–12
19. Zhou, W., Pierre, G., Chi, C.H.: Cloudtps: Scalable transactions for web applications in the cloud. IEEE Trans. Serv. Comput. **5**(4) (January 2012) 525–539
20. Padhye, V.: Transaction and data consistency models for cloud applications. Ph.D., University of Minnesota, Minneapolis, MN, USA (February 2014)
21. Schleier-Smith, J., Holz, L., Pemberton, N., Hellerstein, J.M.: A FaaS file system for serverless computing. CoRR **abs/2009.09845** (September 2020)
22. Bailis, P., Davidson, A., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Highly available transactions: Virtues and limitations. Proc. VLDB Endow. **7**(3) (November 2013) 181–192
23. Gill, P., Jain, N., Nagappan, N.: Understanding network failures in data centers: Measurement, analysis, and implications. SIGCOMM Comput. Commun. Rev. **41**(4) (August 2011) 350–361
24. Jepsen consistency models. https://jepsen.io/consistency Accessed: 11/12/2020.
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565
26. Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. Ph.D., MIT, Cambridge, MA, USA (March 1999) Also as Technical Report MIT/LCS/TR-786.

27. Du, J., Elnikety, S., Zwaenepoel, W.: 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. SRDS '13, Braga, Portugal, IEEE Computer Society (October 2013) 173–184
28. Balouek, D., Amarie, A.C., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Perez, C., Quesnel, F., Rohr, C., Sarzyniec, L.: Adding virtualization capabilities to the Grid'5000 testbed. In Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T., eds.: Cloud Computing and Services Science, Cham, Springer International Publishing (April 2013) 3–20