An Architecture to Offer Transactional Strong Consistency for FaaS Applications

(extended abstract of the MSc dissertation)

Rafael Soares
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

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 higher throughput then alternative protocols, while using only 5% of their resources.

1 Introduction

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.

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 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 of extending a weakly consistent data store with additional services that can offer transactional support and strong consistency (namely, snapshot isolation) to a FaaS application composed of multiple functions that may execute in different workers.

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. We experimentally show that our solution offers 1.4 higher throughput then alternative protocols, while using only 5% of their resources.

2 Related Work

The simplest way to offer transactional support to applications would be to use a storage service that supports it, usually an SQL database. However, as different functions may be executed by different executor nodes, they may read inconsistently from one another, as transactional support is guaranteed in a per-function basis and not on the whole composition. This is the case for most storage services available for FaaS applications, such as DynamoDB[1] and Amazon S3[2].

In this section we will go over the state of the art in regards to systems that offer transactional support over storage services that do not natively support this abstraction. We cover not only FaaS specific systems as well as other cloud-modeled services that use non-transactional storage.

2.1 FaaS Transactional Support

The topic of transactional support for FaaS has recently gained traction in the literature. In this section we will go over three recent systems, namely Hydrocache [15], AFT [12] and Beldi[16].These systems introduce techniques to overcome two of the main challenges when it comes to transactional support in FaaS. The first one comes from the base design of FaaS, where functions must be stateless. The second one comes from using weakly consistent storage systems that only offer Eventual Consistency.

Hydrocache [15] offers transactional support in the form of Transactional Causal Consistency(TCC), a weak form of transactional consistency, guaranteeing that all functions read consistent versions of the objects and writes are done atomically at the end of the function graph. Each object is stored with metadata of all explicit dependencies of the write transaction. Specifically, it includes which objects (and which versions of those objects) are in the causal past of the write transaction. When a function reads an object, it gathers this information, which is passed from each function in the function graph, to guarantee that all functions read from a consistent causal cut. To reduce the number of accesses to the storage service, each server keeps a cache of the objects read and written in the past. This system, while also not supporting strongly consistent transactions (i.e, consistency models that force the ordering of concurrent transactions, like Snapshot Isolation or Strict Serializability), requires the exchange of lengthy metadata between functions, limiting its performance. Transactions may also abort due to the lack of compatible objects with its current read set due to the optimistic reads from the cache.

Atomic Fault Tolerant Shim (AFT) [12] is a system that offers transactional guarantees to FaaS using a relatively weak isolation level in the form of Read Atomic. This system uses an intermediate layer, interposing between the computational and storage layers. This layer is composed of multiple transactional managers. Clients only need to contact one transactional manager to ensure the transaction consistency, avoiding the metadata transfer costs of systems like Hydrocache. Each AFT keeps an index, mapping each object to its most recent version known by the AFT, avoiding coordination between AFTs in exchange of less fresh reads. To guarantee Read Atomic, each AFT keeps metadata for each object about the objects and versions that were written in the same transaction, guaranteeing the transaction atomicity and avoiding fractured reads. Although the lack of coordination between AFTs brings benefits in regards to latency, not only does it make it impossible for this system to offer strong transactional guarantees while it may also force transactions to abort due to lack of compatible versions in an AFT.

Beldi [16] is a system that offers transactional suport with Opacity[7] guarantees, ensuring that transactions never observe inconsistent state, even if the transaction aborts. Although data is kept in a storage layer with strong consistency guarantees, Beldi relies on an additional storage system, shared by all functions, to keep metadata about currently executing transactions. This additional storage layer needs to support multiple atomic insertions in a log, keeping all executed operations of each transaction. The values kept in the log include information about locks, used for concurrency control to guarantee the transactions isolation properties. The entries in this log are totally ordered by relying on a protocol similar to 2-Phase-Locking. This order is then used to order concurrent operations that access the same objects. A disadvantage of this system is that all read and write operations require consulting the log, making the system inefficient.

2.2 Transactional Support for other Cloud-based Applications

CloudTPS[17] is a system that offers strong consistency on top of a storage layer that offers weak consistency similarly to AFT, relying on an intermediate layer composed of multiple Local Transaction Managers (LTM). Each LTM is responsible for a partition of the data set, executing the certification of transactions that interact with its partition. LTMs are also responsible for serving read requests, ensuring the consistency and isolation properties of transactions, relying on the storage layer only for transaction durability. CloudTPS uses a sequencer to ensure total order of transactions, guaranteeing Strict Serializability. Not only is the

usage of this sequencer a possible bottleneck in the system scalability, but most applications do not require an isolation level as strong as Strict Serializability, which adds unnecessary costs to latency. Note that CloudTPS can be adopted to use storage layers that ensure many consistency levels and transactional support. For comparison sake, we consider the system is using the weakest forms of consistency and transactional support.

Padhye PhD thesis[11] introduces a similar architecture to CloudTPS to offer strong transactional support in the form of Snapshot Isolation to cloud environments, using a service composed of multiple replicas of a conflict detection service. Each replica is responsible for ensuring that no conflicts exist between concurrent transactions for their partition of the data set. Padhye uses a strongly consistent storage layer to serve read requests and keep object data yet to be committed, allowing the conflict detection service to only keep information about the most recent versions of each object, using this information to identify write conflicts. Padhye also uses a sequencer that serves the timestamps to be used by transactions to read from a consistent cut. However, this sequencer keeps information about the state of all transactions, allowing it to always serve timestamps corresponding to stable cuts, i.e, cuts where all transactions in the past have finished. Although Padhyes conflict detection service uses less memory than other alternatives (since the service does not keep object data), the usage of a strongly consistent storage layer introduces latency overheads to all read requests. Furthermore, the usage of the sequencer still holds as a possible bottleneck to the system.

2.3 Commit Timestamp

Previous systems relied on a sequencer to obtain commit timestamps. This method not only introduces overheads in the form of extra communication rounds as well and introducing bottlenecks to the system. In an highly scalable environment where low latency's are required as in FaaS, we pretend to minimize these costs.

ClockSI[6] is a protocol that offers Snapshot Isolation to partitioned systems using synchronized physical clocks, allowing the multiple partitions to serve timestamps of consistent cuts to clients without requiring extra communication rounds between partitions. Furthermore, this protocol does not require a sequencer, as commit timestamps are negotiated between partitions using a 2-Phase-Commit protocol. Due to possible clock skews between partitions, read requests may use a timestamp that is in the future of a certain partition, forcing the read request to block until the partition clock reaches the timestamp value. Using common clock synchronization techniques like NTP[3], the maximum blocking time could be in the order of milliseconds, introducing an high latency overhead in FaaS.

Table 1 shows a summary of the related work. As we can see, there are many systems that offer transactional support to various environments and conditions. However, there are no systems that support strong consistency in FaaS in an efficient manner (Beldi offers latency's in the order of seconds when other systems offer them in the milliseconds). While cloud systems show examples of transactional support on top of storage systems with multiple consistency levels in the form of intermediate layers, the excessive use of this layer brings overheads in the form of lack of scalability, which weights heavier in a high scalable environment like FaaS.

	Environment	Strong Transactional Storage	Isolation Level	Read Requests Target
Beldi	FaaS	✓	Opacity	Storage
Padhye	Cloud	✓	Snapshot Isolation	Storage
HydroCache	FaaS	✗	TCC	Storage
AFT	FaaS	✗	Read Atomic	Transactional Manager
CloudTPS	Cloud	✗	Strict Serializability	Transactional Manager
FaaSSI	FaaS	✗	Snapshot Isolation	Transactional Manager + Storage

Table 1. State of the art comparison

3 FaaSSI

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, so functions that would require weaker consistency properties would not have additional overhead imposed.

FaaSSI was designed to be executed in a datacenter, having been developed as an extension to Cloudburst [13]. We used Anna [14] 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 he can contact Anna without any additional service nor penalty in performance. If he requires TCC she can use a system like Hydrocache [15]. Finally, if he requires Snapshot Isolation she can use FaaSSI.

Like most previous work [11, 12, 17], 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.

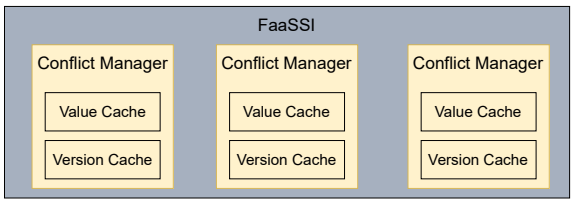


Figure 1. Detailed look of FaaSSI intermediate layer

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 in a fraction of the requests is the intermediate layer required to serve data values to the clients.

3.1 Architecture

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[17], where each conflict manager is responsible for a partition of the space of data. Figure 1 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.

FaaSSI implements a commit protocol similar to ClockSI [6], 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 [9], allows clocks to be synchronized in the order of 100*ns*, making the protocol very efficient.

3.2 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

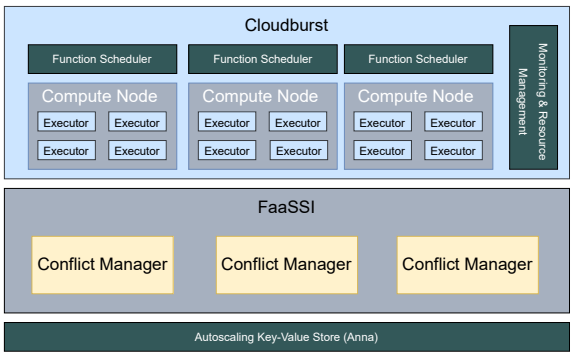


Figure 2. FaaSSI transactional layer. Based on the diagram of [15]. Yellow sections represent FaaSSI additions.

delays due to clock skews and currently committing transactions or he 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. 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 4.2.1.

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 previously, 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 transaction 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 returned 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 very low.

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 ←Read_CM_Versions(keys, snapshot)
15:    kvs_values ←Read_KVS(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 ←Read_CM_Values(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 ←Read_KVS(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
```

3.3 Write Operations

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 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.

3.4 Snapshot Isolation 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 the same object space as the write-set must wait. This is necessary to ensure that only one transaction is able to commit on concurrent writes to the same object space. 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 [6].

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
```

3.5 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[11, 17]. The most straightforward approach to achieve this goal would be to replicate each consistency server using a Paxos group [8], in a manner similar to what is implemented in Spanner [5]. 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.

4 Evaluation

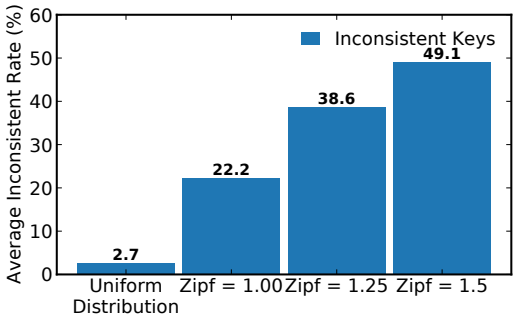
In this chapter we will evaluate the performance of FaaSSI. We compare multiple versions of FaaSSI, implementing different protocols in order to assert the potential advantages and limitations of our proposed techniques.

4.1 Experimental Workbench

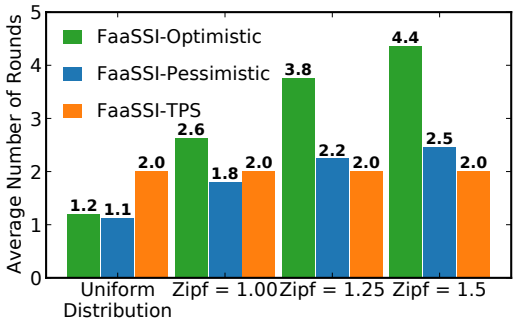
Our evaluation is based on executions of a prototype of our system in the experimental platform Grid'5000 using its dedicated servers. Each server is composed of 1 Intel Xeon Gold 5220 CPU with 18 cores, 96 *GB* RAM and 480 *GB* of SSD storage. The servers were connected by 25 *Gbps* switches. The observed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

For this 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 [15]. 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 2) so each client has always one executor available to execute its graph. We remove the data cache from Cloudburst to not influence the system performance. 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.



(a) Percentage of inconsistent results returned by Anna



(b) Average number of communication rounds per read operation

Figure 3. Skew effect on read protocol

4.2 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 [17]. 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 $mbit/s$.

4.2.1 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.

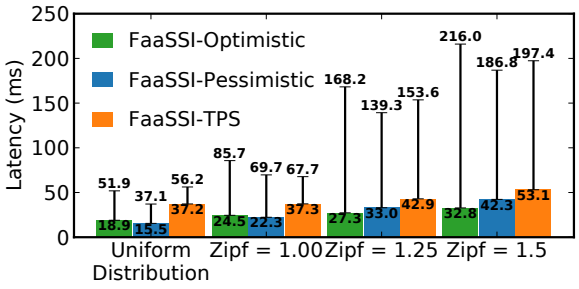
Considering the percentage of times a read returns an inconsistent value, the difference between the performance of FaaSSI-Pessimistic and FaaSSI-Optimistic is less of 1%, thus

we have opted to present a single set of bars in Figure 3a (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.

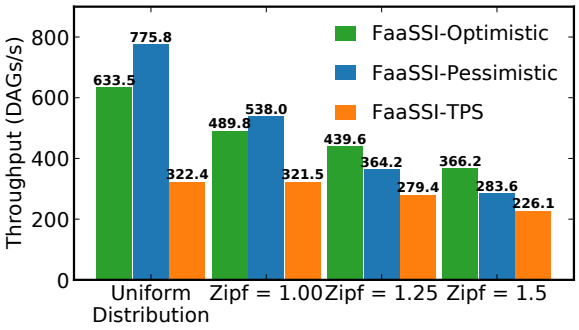
Naturally, the probability of reading a consistent value from Anna impacts the average number of rounds needed to conclude a read operation, as depicted in Figure 3b. Since we are not using the value cache, FaaSSI-TPS will always require 2 communication rounds, as the conflict manager will always need 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. 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.

We now compare the performance of different techniques. As we can see in Figure 4a, 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 4b.

When the access skew increases, we see an increase in transaction latency on all systems, with 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 perform more read rounds on average, given that it allows to distributed the load of read operations among the different replicas. Finally, we can see that for all access distributions, FaaSSI-Optimistic shows a worse



(a) Transactions Median Latency



(b) Transaction Throughput

Figure 4. System performance

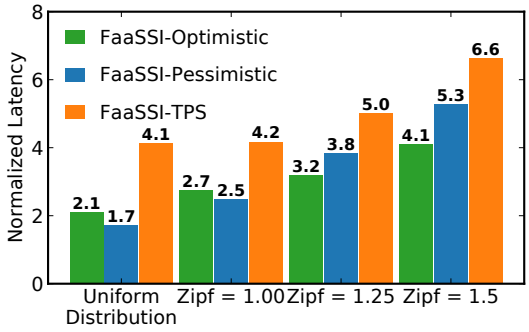


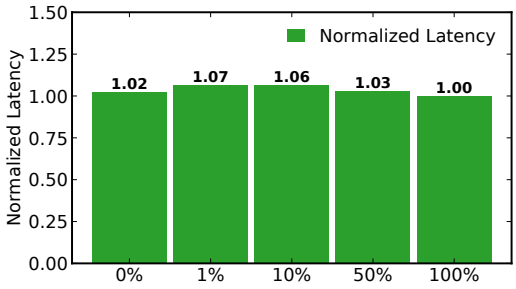
Figure 5. Normalized Latency w.r.t eventual consistency

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.

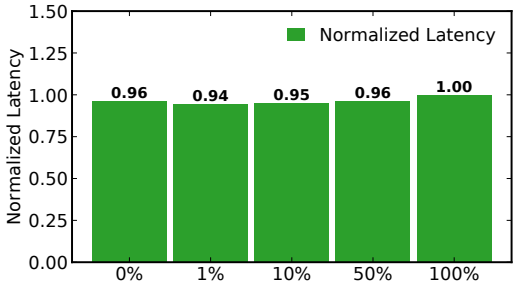
We recall that in our experiments, Anna is executed using “in memory" storage mode, i.e., replicas store data in memory and not on disk. The higher overhead of fetching data from disk could impact the trade-offs of each system.

4.3 Eventual Consistency Comparison

Finally, we 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 4.2.1. We normalize the performance of our multiple techniques with regard to the latency of the eventually consistent system. As shown in Figure 5, our solution incurs a 71% overhead over eventual consistency for



(a) Impact of version cache size



(b) Impact of value cache size

Figure 6. Impact of cache sizes

read transactions on uniform distributions, while FaaSSI-TPS shows up to 4× 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 4.2.1, while the eventual consistency system slightly improves in performance.

4.4 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 4.2.1. 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 6. 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 6a, 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.

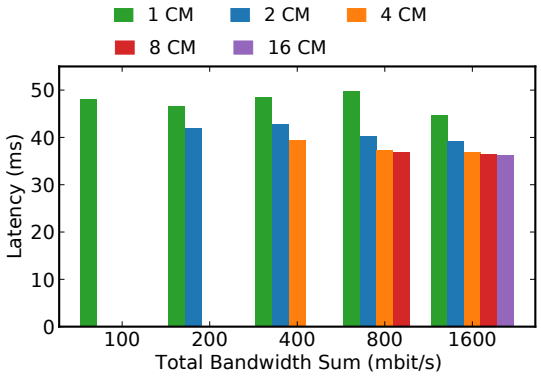


Figure 7. Scalability of the conflict managers

When looking at how the size of the value cache affects the performance, we have observed a similar trend, as depicted in Figure 6b. 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.

4.5 Conflict Manager Scalability

We now 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 4.2.1. 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 7 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 7 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%.

5 Conclusions and Future Work

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. In the future, we plan to study mechanisms that can reduce the average latency of the system, using computational side caching layer, similar to those supported by HydroCache or FaasTCC [10]. We also plan to implement a stateful FaaS-oriented version of the TPC-C [4] benchmark and make a performance study of our system against related work using this benchmark.

Acknowledgements

This work is part of a broader effort of understanding the challenges of offering different consistency levels to FaaS applications. In this context, I have benefited from the interaction of other members of the Distributed Systems Group at INESC-ID working on similar topics and, in particular, from the discussions with Taras Lykhenko. 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).

References

[1] [n.d.]. Amazon DynamoDB. https://aws.amazon.com/dynamodb/. Accessed: 11/12/2020.

[2] [n.d.]. Amazon S3. https://aws.amazon.com/s3/. Accessed: 11/12/2020.

[3] [n.d.]. "The network time procotol". http://www.ntp.org/. Accessed: 07/07/2021.

[4] [n.d.]. TPC-C. http://www.tpc.org/tpcc/. Accessed: 27/10/2021.

[5] 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. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[6] J. Du, S. Elnikety, and W. Zwaenepoel. 2013. 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)*. IEEE Computer Society, Braga, Portugal, 173–184. https://doi.org/10.1109/SRDS.2013.26

[7] R. Guerraoui and M. Kapalka. 2008. On the Correctness of Transactional Memory. In *PPoPP*. Salt Lake City (UT), USA, 175–184.

[8] L. Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 37.

[9] Y. Li, G. Kumar, H. Hariharan, H. Wassel, P. Hochschild, D. Platt, S. Sabato, M. Yu, N. Dukkipati, P. Chandra, and A. Vahdat. 2020. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1171–1186. https://www.usenix.org/conference/osdi20/presentation/li-yuliang

[10] T. Lykhenko, R. Soares, and L. Rodrigues. 2021. FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 22nd ACM/IFIP International Middleware Conference (Middleware '21)*. Association for Computing Machinery, Virtual Event, Canada. https://doi.org/10.1145/3464298.3493392

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

[12] V. Sreekanti, C. Wu, S. Chhatrapati, J.E.Gonzalez, J.M.Hellerstein, and J.M.Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *EuroSys '20: Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Heraklion, Greece, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

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

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

[15] C. Wu, V. Sreekanti, and J.M.Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, Portland, OR, USA, 83–97. https://doi.org/10.1145/3318464.3389710

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

[17] W. Zhou, G. Pierre, and C. Chi. 2012. CloudTPS: Scalable Transactions for Web Applications in the Cloud. *IEEE Trans. Serv. Comput.* 5, 4 (Jan. 2012), 525–539. https://doi.org/10.1109/TSC.2011.18