Delft University of Technology

Distributed transactions on serverless stateful functions

de Heus, Martijn ; Psarakis, Kyriakos; Fragkoulis, Marios; Katsifodimos, Asterios

DOI
10.1145/3465480.3466920

Publication date
2021

Document Version
Final published version

Published in
DEBS 2021

Citation (APA)
de Heus, M., Psarakis, K., Fragkoulis, M., & Katsifodimos, A. (2021). Distributed transactions on serverless stateful functions. In A. Margara, & E. Della Valle (Eds.), *DEBS 2021 : Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems* (pp. 31-42). ACM. https://doi.org/10.1145/3465480.3466920

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Distributed Transactions on Serverless Stateful Functions

Martijn de Heus Kyriakos Psarakis Marios Fragkoulis Asterios Katsifodimos
Delft University of Technology
{m.j.deheus,k.psarakis,m.fragkoulis,a.katsifodimos}@tudelft.nl

ABSTRACT

Serverless computing is currently the fastest-growing cloud services segment. The most prominent serverless offering is Function-as-a-Service (FaaS), where users write functions and the cloud automates deployment, maintenance, and scalability. Although FaaS is a good fit for executing stateless functions, it does not adequately support stateful constructs like microservices and scalable, low-latency cloud applications, mainly because it lacks proper state management support and the ability to perform function-to-function calls. Most importantly, executing transactions across stateful functions remains an open problem.

In this paper, we introduce a programming model and implementation for transaction orchestration of stateful serverless functions. Our programming model supports serializable distributed transactions with two-phase commit, as well as relaxed transactional guarantees with Sagas. We design and implement our programming model on Apache Flink StateFun. We choose to build our solution on top of StateFun in order to leverage Flink's exactly-once processing and state management guarantees. We base our evaluation on the YCSB benchmark, which we extended with transactional operations and adapted for the SFaaS programming model. Our experiments show that our transactional orchestration adds 10% overhead to the original system and that Sagas can achieve up to 34% more transactions per second than two-phase commit transactions at a sub-200ms latency.

CCS CONCEPTS

• **Information systems** → *Data streams*; *Middleware for databases*; *Distributed transaction monitors*; *Message queues*.

KEYWORDS

serverless, transactions, FaaS, two-phase commit, Sagas, streaming dataflow

ACM Reference Format:
Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, Asterios Katsifodimos. 2021. Distributed Transactions on Serverless Stateful Functions. In *The 15th ACM International Conference on Distributed and Event-based Systems (DEBS '21), June 28-July 2, 2021, Virtual Event, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3465480.3466920

This work is licensed under a Creative Commons Attribution International 4.0 License.
DEBS '21, June 28-July 2, 2021, Virtual Event, Italy
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8555-8/21/06.
https://doi.org/10.1145/3465480.3466920

1 INTRODUCTION

Serverless computing [22] is a cloud computing execution model promising to simplify the programming, deployment, and operation of scalable cloud applications. In the serverless model, developer teams upload their code written in a high-level API, and the cloud platform takes care of the application's deployment and operations. Serverless computing aims to substantially increase cloud adoption by remedying the status quo in the cloud landscape, where developer teams need to possess skills in distributed systems, data management, and cloud execution model internals to use the cloud effectively.

The most prominent serverless offering is Function-as-a-Service (FaaS), where users write functions and the cloud providers automate deployment and operation. However, FaaS offerings lack the state management support and the ability to perform transactional workflows across multiple functions and state backends [3, 21], which are needed by general-purpose cloud applications. Although there is ongoing work in supporting stateful FaaS (SFaaS) applications, transactions across functions remain an open problem.

The only system addressing distributed transactions in a SFaaS setting is Beldi [35], which provides fault-tolerant ACID transactions on workflows across stateful functions by logging the functions' operations to a serverless cloud database. Cloudburst [31] with HydroCache [34] provides causal consistency on function workflows forming a DAG by leveraging Anna [33], a key-value store with conflict resolution policies in place. Cloudburst does not provide isolation between DAG workflows.

In sharp contrast with the aforementioned approaches, developer teams in the microservices and cloud applications landscape go to extreme lengths when they need to implement transactional workflows across the boundaries of a single service or function. Depending on the requirements of each use case with respect to consistency and performance, two approaches stand out: application-level implementations of the two-phase commit protocol or the Saga pattern. These two approaches serve opposing goals. Two-phase commit (TPC) [20] offers ACID, serializable transactions but imposes blocking across functions participating in a transaction, which penalizes performance in return for strict atomicity.

On the other hand, the Saga pattern [16] separates a transaction into sub-transactions that proceed independently with the benefit of improved performance but at the risk of having to undo or compensate the changes of successful sub-transactions when at least one of the involved sub-transactions fails. In addition, compensating actions can be challenging when concurrent changes are applied to the state because Sagas do not require any means of isolation. For this reason, state consistency needs to be dealt with at the application level.

To alleviate these issues, we propose a programming model and a corresponding implementation, publicly available on GitHub[1],

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

for authoring workflows across stateful functions in FaaS with transactional guarantees. Our programming model provides two alternatives: two-phase commit and Sagas, drawing inspiration from best practices in developing microservices and cloud applications. We implement the two approaches on an open-source stateful FaaS system, Apache Flink's [9] StateFun.[2]

Although different flavors of FaaS platforms exist, we base our work on a stateful dataflow engine for reasons that we have argued in previous work [3, 25]. In short, modern dataflow engines, used by stream processing systems [4, 5, 8, 9, 17] can execute stateful functions as follows: incoming events represent function execution requests routed to continuous stateful operators that execute the corresponding functions. With proper, consistent fault tolerance mechanisms [7, 28], state of the art stream processing systems operate at high-throughput and low-latency. At the same time, they guarantee the correctness of execution even in the presence of failures. As we explain in Section 2 this set of properties is important for supporting transactions managed by the platform with minimal involvement from the application developers.
In summary, our work makes the following contributions:

- we make a case for implementing transactions on a stateful dataflow engine and present the advantages that come with such an approach
- we propose a programming model for transactions across stateful serverless functions
- we implement the two main approaches used by cloud application practitioners to achieve transactional guarantees: two-phase commit and Saga workflows
- we evaluate two transactional schemes using an extended version of the YCSB benchmark on a cloud infrastructure

Section 2 gives the motivation of this work and explains the benefits of running transactions on dataflow graphs, while Section 3 presents the background. Next, Section 4 introduces the concept of coordinator functions, and Section 5 details their implementation and the introduced programming model. The experimental setup is presented in Section 6, while the performance of coordinator functions is evaluated in Section 7. Section 8 presents the related work. Finally, Section 9 summarizes the work and discusses interesting areas for further research.

2 TRANSACTIONS ON STREAMING DATAFLOWS

Serverless platforms come in different flavors. One breed of SFaaS systems (e.g., Apache Flink StateFun and [3]) is built on top of a stateful streaming dataflow engine. This architecture bears important implications for the support of transactions because of how distribution, state management, and fault tolerance work.

Network communication between distributed components in a typical streaming dataflow engine is implemented via FIFO network channels that guarantee exactly-once data delivery and preserve delivery order. In a serverless FaaS system, this characteristic obviates the need for handling lost messages and implementing retry logic with respect to function invocations in transactional workflows. Messaging errors and retries are a significant source of friction and development effort at the application level, and those are offered by the underlying dataflow system.

State management in state-of-the-art streaming systems achieves exactly-once processing guarantees by taking consistent snapshots of the system's distributed state periodically [7]. The snapshots capture a globally consistent state of the system at a specific point in time and are used to recover the system's state upon failure. Exactly-once means that the changes brought by each function execution instance are recorded in the system's state exactly once, even in the face of failures. For transactions, this capability is essential because fault recovery of transactions can piggyback on the underlying fault tolerance mechanism with zero effort and knowledge by the application. Given that a big part of code and effort is spent on failure handling, fault tolerance, and virtual resiliency [18] provided at the system level can play a significant role.

Furthermore, unlike stream processing applications where the computations are fully encapsulated within the system's operators, it is common to have nondeterministic side effects in microservices and cloud applications. One popular way is by interacting with external services located out of the system's boundaries or by having business logic that is nondeterministic. However, the fault tolerance of streaming dataflow systems was not designed to support the many faces of non-determinism that are possible in general-purpose applications. Thus, the consistency of applications and the integrity of transactions are endangered when transactions involve nondeterministic operations. Extending the fault tolerance approach of streaming dataflow systems to support nondeterministic computations [28] is an important step towards opening their adoption for executing general-purpose applications. Finally, recent work [11, 31] also recognizes the dataflow model as a key enabler for the SFaaS systems of the future.

In summary, we believe that stateful streaming dataflows and the associated research that has been proposed so far [1, 14, 25] can alleviate the burden of building rich stateful and transactional applications on top of streaming dataflows. This paper presents a step towards this direction.

3 PRELIMINARIES

3.1 Transaction Model

In the context of this work, a transaction is an atomic execution of a set of stateful-function invocations. More specifically, the transactional model introduced in this paper considers transactions that are defined up-front. This is referred to as *single-shot* [32] or *one-shot* [23] *transactions* in prior works. More specifically, we follow the definition of H-Store [23] *one-shot transactions* assuming that the output of a function (query) cannot be used as input to subsequent functions (queries) in the same transaction. This simplifies coordination of the transaction across the system while still providing a practical model for transactions; for instance, Amazon's DynamoDB [29] implements one-shot transactions [32] and has wide applicability. Implementing one-shot transactions on top of a SFaaS system has a significant advantage: functions can implement arbitrary business logic in a touring complete programming language such as Java or Python. This creates a lot more flexibility in the programming model and allows for complex transaction definitions and business logic.

[2] https://statefun.io

3.2 Apache Flink StateFun

Apache Flink StateFun[3] offers an abstraction and runtime for users to implement stateful cloud functions. A stateful function implemented by user-code is referred to as a function type and describes the state held by this function type. Multiple instances based on the same function type can exist in parallel and are identified by an id. Each of these function instances encapsulates its own state and can be uniquely addressed by the combination of its type and id. Function instances can be invoked from other function instances or through ingresses such as Kafka. Function instances can have four different controlled side effects; (1) state updates, (2) function invocations, (3) delayed function invocations, (4) egress messages (for example, Kafka). StateFun supports end-to-end exactly-once guarantees from ingress to egress, including any state updates.

Embedded vs. Remote Functions. Functions can be deployed both inside the StateFun workers (referred to as embedded functions) and outside the StateFun cluster (co-located and remote functions). Embedded functions are simply an abstraction on top of stateful streaming operators in Flink, therefore providing exactly-once and fault-tolerance guarantees. StateFun allows dynamic communication between these streaming operators by introducing a cycle in the streaming graph. The co-located and remote functions are entirely stateless because the state is persisted within StateFun. This paper focuses on remote functions as these can leverage existing FaaS services such as AWS Lambda to auto-scale the compute layer. Figure 1 shows how remote functions work. Each function instance is represented by an embedded stateful function in the StateFun cluster. This standardized embedded function is responsible for managing the state of the function instance and the communication with the remote function that may be deployed anywhere. The persisted data in the embedded stateful function with the communication pattern for remote functions are shown in Figure 1.

Function Invocations as Dataflow Messages. Invocations that are sent to a function instance arrive in a queue, as shown in step 1 of Figure 1. If the embedded stateful function is ready to process the next invocation, it pulls a message (invocation parameters) from the queue (step 2). When no invocation is being executed at the remote function, the remote function is called. However, if the remote function is busy with a previous function call, the current invocation message is appended to the next batch. Batches are used to avoid multiple remote calls to a given function, with a trade-off in latency (see Section 7.1). Batches are also used to preserve the invocation order and order of state access (the batch has to wait until the state updates caused by the previous batch have been performed), thus ensuring linearizability at the function instance level.

In step 3, the stateless remote function is called through a Protobuf interface that contains both the (keyed) state required for the remote function to operate and the invocation parameters of the function. The stateless remote function can execute the (batch of) invocations and will be ready to return the updated state back to the Flink worker that made the call. In step 4, the response of the stateless remote function is appended to the queue of incoming

[3]https://flink.apache.org/stateful-functions.html

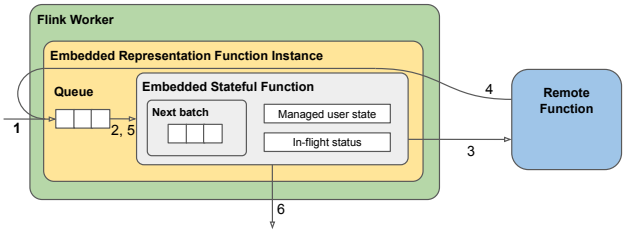


Figure 1: Original communication flow for remote functions

messages to the function. The response includes any side effects caused by the invocation(s), including updates to the user-defined state.

When the response from the stateless function is processed (step 5), the side effects caused by the invocation(s) are applied to the state of the embedded function. This updates the managed state in the embedded stateful function. If any invocations are batched, the next batch of invocations is sent to the remote stateless function, and the batch is truncated. When there are no batched invocations, the in-flight status is cleared. Finally, any outgoing function invocations are sent to the queues of their respective function instances, and egress messages are sent to their respective egresses (step 6).

3.3 Assumptions & Requirements

As we describe in the next section, our coordinator functions rely on an underlying SFaaS system for bookkeeping the state of ongoing transactions and reliable messaging. To allow this, the underlying system should satisfy two requirements.

Exactly-once Processing Guarantees. Firstly, it is required for all communication to be reliable and executed with exactly-once processing guarantees. Thus, we require that the underlying system is fault-tolerant [9] to ensure atomicity in case of a failure in the middle of a transaction. This also means that the state is durable within a snapshot/checkpoint, even in the event of failures. If we can rely on exactly-once processing guarantees, message replay and error handling, which involve a significant part of transaction coordination, are greatly simplified. Exactly-once processing guarantees are supported in Flink StateFun.

Linearizable Operations. The second requirement is that the operations for any specific function instance should be linearizable, which means that there is a given order that operations are performed on the function instance and the state encapsulated in this instance. Accordingly, a function invocation will always have the correct state of the function instance in order to implement transactions. Since Flink StateFun's function instances use a single replica of the state per function instance and a single process is executing function invocations for that function instance in a sequential FIFO manner, this ensures linearizable operations per function instance.

4 APPROACH OVERVIEW

In this section, we introduce the concept of stateful coordinator functions and provide an overview of our approach. Our approach is based on the simple observation that since an underlying SFaaS system provides exactly-once processing and message delivery guarantees, conceptually, it would be much simpler to implement

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

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

Listing 1: Two-phase commit coordinator function.

```
1 def sagas_transfer(context, message: Transfer):
2     subtract_credit = SubtractCreditMessage(amount=message.amount)
3     add_credit = AddCreditMessage(amount=message.amount)
4     context.saga_invocation_pair("account_function",
5                                         message.debtor,
6                                         subtract_credit,
7                                         add_credit)
8     context.saga_invocation_pair("account_function",
9                                         message.creditor,
10                                        add_credit,
11                                        subtract_credit)
```

Listing 2: Saga coordinator function.

a transaction coordinator as a regular, stateful function. With this in mind, we opted for implementing a transaction API on top of stateful functions, which we present in Listing 3. Notably, further work is required to raise the transaction abstractions at a higher level [25] as syntactic sugar.

4.1 API

A stateful coordinator function is a stateful function that preserves state about the execution of a given transaction. Coordinator functions have the ability to force other function instances to abort or compensate for the changes they applied.

API Overview. Our coordinator function implements two transaction coordination patterns: two-phase commit and Sagas [16]. A complete example of a coordinator function for two-phase commit and Saga is shown in Listings 1 and 2 respectively. In short, to coordinate a two-phase commit transaction, the user needs to invoke function instances via `2pc_invocation`, while for a Saga, an invocation pair is expected, which consists of the normal transaction invocation and the corresponding compensation invocation to be sent to the same function instance. A Saga invocation pair can be called with `saga_invocation_pair`. An important difference between the behavior of the two schemes is that a failure in a Saga workflow will incur a compensating function call.

4.2 Two-Phase Commit

The `serializable_transfer` function of Listing 1, receives a context (the underlying context of StateFun as we have extended it to support transactions) and a message. The message is of type `Transfer`, and it contains three fields: the amount of money transfer, a creditor, and a debtor. The amount mentioned in the message must be subtracted from the debtor and transferred to the creditor. To this end, assuming that there is a function type registered in the system as `account_function`, as per the original StateFun API, we need to construct an object containing the parameters for the `account_function` and push that message to the transaction coordinator. This is done in lines 4-6: we give the TPC coordinator the function type to invoke, alongside the id of the debtor to form the address of the function instance and the `SubtractCreditMessage` which is going to be given to that function as a parameter. Subsequently, we do the same for the creditor: we construct an `AddCreditMessage` and we pass it over to the function type `account_function`. In short, the transaction coordinator function instance will make sure that the two function instances are invoked with serializable guarantees. It does this by coordinating a two-phase commit protocol across the function instances with locking to ensure isolation. More details on these aspects are given in Section 5.

4.3 Sagas

Similarly to two-phase commit, our API offers the ability to specify Sagas: as seen in Listing 2, the `saga_invocation_pair` function in line 4 will receive the target function name, the ID of the debtor as well as two messages: the `subtract_credit` and its compensating action `add_credit`. If there is a failure during the execution of `subtract_credit` our Sagas transaction coordinator will execute the compensating action `add_credit` which will put back the original credit to the debtor's account. The details on how Sagas are executed are given in Section 5.

4.4 Extensions to Regular Functions

To allow the execution of a transaction by the two types of coordinator functions across any arbitrary function instances, some extensions to regular functions are required.

First, functions that can partake in a coordinated transaction need to be able to fail explicitly. After a failure is communicated to a coordinator function, it results in a transaction rollback. Currently, there is no notion of failing an invocation in Flink StateFun; the function invocation may simply perform no side effects. To allow explicit failure, a field containing these details is added to the protocol between StateFun and the remotely deployed functions. From the API perspective, a function failure can be triggered by throwing an exception. The failure of a function can be roughly compared to integrity constraint violations based on the state encapsulated in a function instance in traditional database terms.

Second, any batching mechanism needs to be changed. TPC coordinator functions ensure isolated transactions. This means that any function invocation that is part of such a transaction may not be batched between other function invocations. Third, appropriate locking should be implemented on the level of function instances to ensure the isolation of serializable transactions based on two-phase commit coordinator functions.

Finally, the function instances should communicate with the coordinator functions transparently; developers should not be burdened with this task.

5 TRANSACTIONAL WORKFLOWS

In this section, we present our Python API in more detail, and we present the implementation for transactional workflows across stateful serverless functions on Apache Flink StateFun. Our implementation consists of coordinator functions that enforce either a distributed serializable transaction with a two-phase commit or a Saga workflow as a transaction without isolation.

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

Listing 3: Coordinator functions' Python API.

5.1 Coordinator Functions

Coordinator functions instrument transactional workflows across ordinary Stateful functions. To achieve this, coordinator functions encapsulate the state of active transactional workflows that they are in charge of but hold no state of the participating function executions or custom user-defined state. A coordinator function can be invoked simply by its name (uniquely identified by a type internally) and an ID generated randomly at initialization time. Then an input message will arrive at the coordinator's input queue. If the coordinator function is involved in an ongoing transaction, the message will be queued until the workflow that is executing completes. The coordinator functions' Python API is listed in Listing 3.

Figure 2 shows the common communication flow between a coordinator function and regular function instances. Specializations of this communication for two-phase commit and Saga workflows are described in Section 5.2 and Section 5.3 respectively. Messages that are not always sent in both cases are annotated with a *. Figure 2 shows the enriched internal structure for regular function instances compared to Figure 1. These are the extensions that we implement for regular functions so that they can participate in transactional workflows.

5.2 Saga Coordination

The programming model of the Saga coordinator function is shown in Listing 2 through an example. Listing 3 presents the API. In Sagas, the developer is responsible for defining pairs of function invocations so that the invocation of the second function compensates the one of the first function[16]. Besides this, the Saga coordinator function can also define side effects (e.g., outgoing egress messages) based on different completion scenarios of the transaction (success or failure). The function invocations composing a Saga are executed in parallel in the current implementation. [4] In the following, we describe the messages specifically for Sagas seen in Figure 2.

Initialization & remote coordinator function call. First, a message is sent to the coordinator function to initialize a transaction (step 1). The message is taken from the queue to initialize the transaction (step 2). Then, the remote Saga coordinator function is called with the incoming message (step 3). The remote function returns the definition of the Saga workflow to its embedded counterpart (step 4). This includes the function invocations involved in the transaction and their compensating invocations, as well as the side effects to perform on success or failure.

Processing the remote coordinator function's result. When the embedded function processes the result of the remote function (step 5), a random transaction ID is generated, and a map is created holding the addresses of function instances and the result of their execution (at this stage, those are initialized as `null` values). It follows that only one invocation per function instance can be involved in a particular workflow. If multiple invocations of a single function instance are required, this can be solved at the application level by allowing a single message, which combines multiple function invocations to be sent to the function instance.

Invoking regular functions. In step 6, each of the participating regular (non-coordinator) function instances receives a function invocation in its input queue. All the invocations are sent simultaneously, and the function instances can do the work in parallel. These function invocations are distinguishable as function invocations that belong to a Saga workflow. Each Saga function invocation is fetched from the queue, and it is either directly sent to the remote function or batched with other invocations for efficiency (step 7). Because Sagas do not require isolation, a function invocation can

[4] We plan to expose a configuration for the intended behavior in the API, in order to optionally make these sequential.

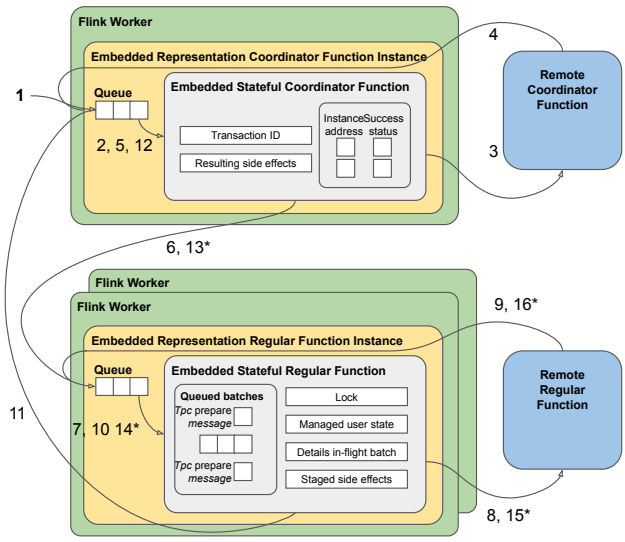


Figure 2: Communication flow for transactions.

be batched with other invocations. Then it is sent to the regular remote function (step 8). After processing it, the function's response is added to the queue of its stateful embedded representation in StateFun (step 9). When the response of the stateless remote function is processed in the embedded stateful function at step 10, the indices in the in-flight function invocation metadata and new list added to the Protobuf interface, i.e., the regular function extensions, are used to identify the result status of the Saga function invocations and the corresponding coordinator's addresses. If the function invocation fails, no side effects of the function are performed. After this, this function can continue processing other function invocations.

Saga success vs. compensation. Based on the success status of the Saga function invocation, a success or failure message is sent to the coordinator function (step 11). When the embedded coordinator function processes the success status of each function invocation, the map is updated with either a success or failure status (step 12). If a function instance failed, any function instances that successfully executed their function invocation are messaged with their respective compensating actions (step 13), and the side effects in case of a failure are performed (steps 14, 15, 16). The coordinator function has to wait until the result of all function invocations is received before it is done. In case any of the function invocations fails, the coordinator function sends the compensating messages to all function instances that successfully processed their invocation. Note that there is no need to send compensating invocations to function instances that failed since those function instances have applied no side effects. The compensating messages are processed as regular messages and are only required when any of the function invocations fail. This means that the performance of a Saga workflow will be worse if it is likely to fail as this will require extra messaging and processing, up to double. As a matter of fact, this is the trade-off offered by optimistic transaction approaches like Sagas.

5.3 Two-phase Commit Coordination

In Listing 1 we presented the programming model for a two-phase commit coordinator function; Listing 3 shows the available functions of the two-phase commit API. Similar to Saga coordinator functions, two-phase commit coordinator functions can also define side effects to execute for any completion scenario. Beyond successful and failed completion, two-phase commit transactions can also complete as "retryable". This occurs when the transaction is aborted due to a deadlock (Section 5.4). In the following, we describe the workflow of the two-phase commit as seen in Figure 2. Note that the initialization of the workflow, i.e., steps 1-5, is the same as in Sagas. Thus, we do not detail it here.

`PREPARE` & Two-phase locking growing phase. Each involved function instance is messaged with its respective function invocation in step 6. This message is identifiable as a `PREPARE` message of the two-phase commit protocol. When a two-phase commit function invocation arrives at the embedded stateful regular function and a batch of invocations for this function is currently in-flight, this two-phase commit function invocation is not batched with other invocations. Instead, the two-phase commit function invocations split up batches and are sent to the remote function in isolation as seen in Figure 2. This practice increases the complexity of the batching mechanism, as it now requires a queue of batches rather than an append-only batch as shown in Figure 1.

Invoking regular remote functions. When the message (and current state) is processed and sent to the remote function in steps 7 and 8, the transaction ID and the address of the two-phase commit coordinator function are stored in the details of the in-flight batch of invocations. The lock on the function instance is also set at this point. The response of the stateless remote function includes the result status of the function invocation and any side effects (step 9). Suppose a `FunctionInvocationException` is thrown at the stateless remote function. In that case, the response of the remote function is discarded, a response to the coordinator function instance is sent to notify it that the invocation failed, and the regular function instance's lock is removed as it knows the transaction will be aborted. If the function invocation is successful, the lock is kept, and a success response is sent to the coordinator function instance. The state effects are then stored as staged side effects in the function instance (step 10). Any other messages that arrive while the function instance is locked are put in the queued batches.

`ABORT` & Two-phase locking shrinking phase upon Failure. The message at step 11 notifies the two-phase commit coordinator function instance whether the function invocation succeeded. If the two-phase commit function instance receives the message that a function invocation failed (step 12), it immediately sends an `ABORT` message to all other function instances and performs the appropriate side effects (step 13), and calls the two-phase lock shrinking phase. After this, the two-phase commit function is done.

`COMMIT` & Two-phase locking shrinking phase. If the two-phase commit function instance receives the message that a function invocation was successful, it updates the map it keeps of all involved function instances. If all function instances succeeded, it sends `COMMIT` messages to all involved function instances and publishes

the appropriate side effects (i.e., applies the changes to the embedded function state).

COMMIT/ABORT & Two-phase locking shrinking phase. When a function instance receives a COMMIT message (step 14), it executes its staged side effects, releases the lock and continues processing the next request. When a function instance receives an ABORT message, it discards its staged changes, releases the lock, and continues processing. Note that a function could also receive the ABORT message while the PREPARE message is still in the queue or in-flight. In this case, the PREPARE message is discarded. Messages 15 and 16 are never sent for two-phase commit transactions.

5.4 Deadlock Detection in Two-phase Commit

Due to the use of locks, the two-phase commit protocol is susceptible to deadlocks. A deadlock can happen when two or more different two-phase commit transactions wait on the locks on function instances that are held by other transactions. To deal with deadlocks, we have implemented a deadlock detection mechanism, which we describe below. All participants in the two-phase commit transaction can be partitioned across different machines, and the state of active transactions is encapsulated in different coordinator function instances. Thus, we do not want transactions to rely on any centralized component for handling deadlocks. We implemented the Chandy-Misra-Haas algorithm [10] that provides a simple way to detect deadlocks in a distributed manner, without dependence on a single global coordinator.

Whenever a deadlock is detected in a transaction, it immediately completes as a retry-able transaction and sends abort messages to all involved function instances. Upon receiving a retry-able result status, a two-phase commit regular function may send itself a delayed invocation with the same initial message (and possibly a counter attached) to perform a retry. This is left to the developer so that the system remains flexible across various use cases.

6 EXPERIMENTAL SETUP

In this section, we describe in detail our experimental evaluation methodology. In the lack of a proper benchmark for SFaaS, we opted for an extension of the *Yahoo! Cloud Serving Benchmark* (YCSB) [12] benchmark.

6.1 Benchmark Workload

In YCSB, the first step is to insert records into the system with a unique ID and several task-specific fields. After the data insertion stage, the benchmark performs operations on the initialized state. YCSB defines read and write operations as part of their core workloads. Because this work's main contribution is distributed transactions across stateful function instances, we added a new operation based on an extension introduced in [13]. This operation is called a transfer, and it atomically subtracts balance from one account and adds this to another, meaning that records also include a numeric balance field. These additions mean that the workloads can consist of the following three operations:

read Reads the state associated with a single key and outputs it to the egress.

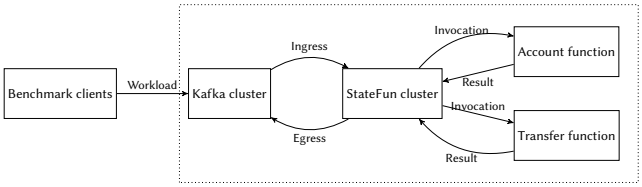


Figure 3: Benchmark application architecture

write Updates a field associated with a single key and outputs a *success* message to the egress.

transfer Requires two keys and a specified amount, and subtracts the amount from the balance of one key and adds it to the other. Depending on the transaction result, the output is either a success or failure message to the egress.

Across experiments, we vary the proportion of each operation in the resulting workloads. In YCSB, the user selects the probability distribution of the operations' record IDs. In this work, we consider solely uniform distributions to make the experiment results explainable. The added benefit is that the number of requests for a single key can be increased transparently by decreasing the system's total number of records. Finally, YCSB allows variations in the number of fields and the size of the values associated with each field. In this evaluation process, all records have ten fields containing a single random string of 128 bits and a single integer field. A StateFun application is implemented with the following two functions to support the operations defined in Section 6.1:

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

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

Figure 3 showcases a diagram of the system under test. The benchmark publishes the workload to a Kafka cluster. StateFun reads from Kafka as ingress, invokes the appropriate functions, and then publishes the result to a Kafka topic as an egress. The system under test is deployed on SurfSara[5], an HPC cloud with instances with up to 80-vCPUs. For our experiments, we used a two VM Kubernetes cluster to simplify deployment and management of the system's separate components with enough vCPUs to support the system's configuration under test. All components shown in Figure 3 can be horizontally scaled as necessary. Additionally, we give the Kafka cluster enough resources to ensure it can handle the load so that when a bottleneck comes up, it would be owed to the StateFun cluster.

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

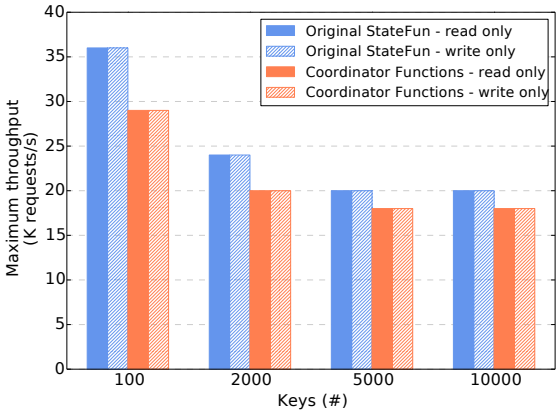


Figure 4: Maximum throughput for the original StateFun vs. StateFun with coordinator functions

6.2 Evaluation metrics

We evaluate the system based on two metrics. The throughput either at max or stable (80%), showing the number of workload operations the system can handle per second, and the latency, showing the time it takes to process an operation.

The maximum throughput of each workload and system configuration is found by steadily increasing the input throughput created by the benchmark clients in Kafka until the StateFun cluster can no longer consistently handle the load, as measured by the system's output throughput in Kafka. At some point, the output throughput starts fluctuating, and we define this value to be the maximum throughput for the configuration.

We use the Kafka event time for the ingress and egress events of correlated operations to measure end-to-end latency. Because latency is always dependent on the throughput, in the experiments, we set the throughput to 80% of the maximum throughput to allow consistent operation of StateFun and measure the latency accurately. When comparing latencies, the different throughput rates at which the latency is measured should be considered.

7 EXPERIMENT RESULTS

In this section, we go through the experimental evaluation of our system that is split into four experiments with the following goals. i) Determine the overhead that function coordination introduced to StateFun (Section 7.1). ii) Compare between the two transaction protocols with/out rollback operations (Section 7.2). iii) Evaluate the system's scalability (Section 7.3). iv) Perform a microbenchmark with a fixed number of machines and a variable number of keys and proportions of `transfer` operations (Section 7.4). In terms of resources used, for (i, ii, iv), we used three 4-CPU StateFun workers, and for (iii), each worker had 2 CPUs.

7.1 Coordination Overhead

In the first experiment, the performance of StateFun with coordinator functions is compared against the original on non-transactional workloads to see how much computational overhead the coordination logic has added. In Figure 4 we show the maximum throughput achieved by the two systems for a varying number of keys. While in Figure 5 we show the different latencies for the systems across `read` and `write` workloads at different throughputs and numbers of keys.

Throughput. The first observation we can make is that there is a 20% decrease in throughput in the case of 100 keys that plateaus to 10% as the number of keys increases. The decrease in performance is due to the changed batching mechanism being more complex than the original append-only approach by enforcing isolated function invocations as part of a two-phase commit transaction. In addition, coordinator functions keep track of transaction progress, which incurs some overhead. Another observation is that there is no noticeable throughput difference between workloads with only `read` or `write` operations. The reason behind this behavior is that, in StateFun, both operations need to access the remote function, making the communication layer the bottleneck.

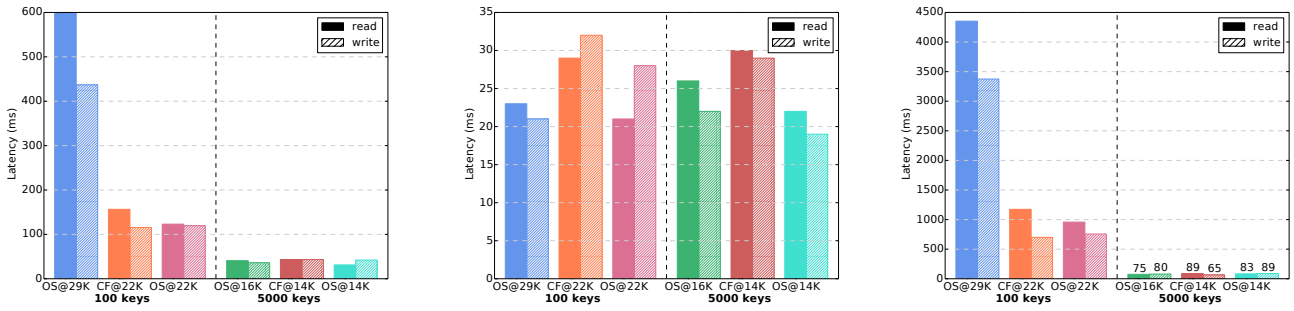
Latency. The latencies in Figure 5 are approximately 20% higher for our version of StateFun for `read` workloads. However, as the number of keys increases, the difference becomes smaller, towards 7%. This decrease in performance is due to the additional logic required for the function coordination. Another interesting observation is the indifference in performance for `write` workloads. It relates to StateFun requiring every read operation in a batch to be serialized, which adds up over time for larger batches. In contrast, for writes, only the last version needs to be serialized. Additionally, this serialization happens at the remote function, which explains why it does not affect throughput, but it does affect latency.

Finally, we consider the introduced overhead as a reasonably low price to pay for having full-fledged transaction execution primitives added to the system.

7.2 Sagas vs Two-Phase Commit

The second experiment shows a performance comparison between the two implemented transaction protocols, their impact on the maximum throughput in perfect conditions (Figure 6), and with failures, measuring the impact of locking for the two-phase commit (Figure 7) and of rollbacks (Figure 8) for the Saga protocols. In these experiments, we set a certain proportion of the workload to be `transfer` operations and the remaining proportion is equally shared between `read` and `write` operations. In our case, each `transfer` operation causes three remote function invocations. We do not include messages sent to detect deadlocks in the total number of invocations when evaluating two-phase commit functions. Therefore, the indicator is a lower bound. Finally, we used a uniform key access distribution for these experiments, while in real-world conditions, this will likely be skewed towards some popular keys.

Figure 6 plots the achieved throughput against the absolute number of `transfer` operations in the workload with varying number of keys given that the benchmark provided the accounts enough balance to ensure all transactions succeeded. It also displays indicators for the absolute amount of total internal function invocations, taking into account additional internal invocations required for transactions and the absolute amount of total remote function invocations. We observe that Sagas perform much better than two-phase commit for few keys (100 and 2000). For two reasons: i) Sagas can still benefit from the batching mechanism of StateFun since they



(a) Mean (b) Median (c) 95th percentile

Figure 5: Graphs comparing latencies of original StateFun (OS) and StateFun with coordinator functions (CF) at different throughputs for read-only and write-only workloads

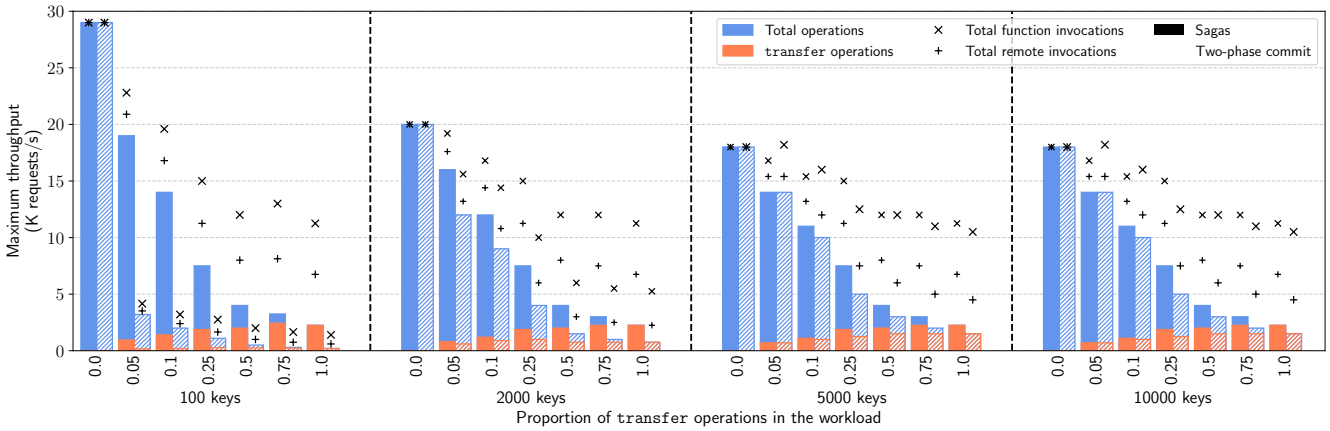
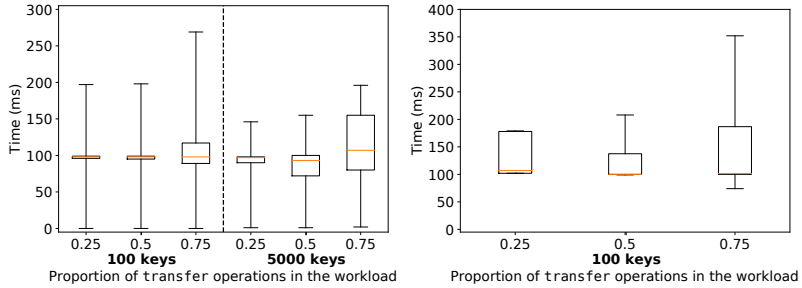


Figure 6: Maximum throughput for workloads with increasing proportions of `transfer` operations in the workload

do not require isolation, and ii) the locking in two-phase commit severely limits the throughput. However, it is also interesting that for a higher number of keys (5000-10000), two-phase commit performs comparably to Sagas even though it provides much stronger guarantees. This is because there is less contention on a single function, decreasing the effect of locking, while batching provides no benefits, as also shown in Figure 4. A second observation from Figure 6 is that the total function invocations still drop when the proportion of transactions increases. The total function invocations account for the additional messaging required to coordinate transactions, leading to the overall throughput of workloads with a high proportion of `transfer` operations being relatively low.

Locking Overhead. In Figure 7 we measure the behavior of locking and deadlocks that accompany the two-phase commit protocol. The lock duration is measured between the point in time where the function instance sends the response to the `prepare` message and when it either receives a `commit` or `abort` message, sending the next batch to the remote function. In Figure 7a, we see little to no difference in the median across the different workloads but, when the proportion of `transfer` operations is higher, the higher percentiles increase significantly. Next, we want to measure the deadlock frequency, and Figure 7c shows the number of deadlocks against the total number of `transfer` operations in the workload. As expected, there are no deadlocks for workloads on 5000 keys since the contention is low. For 100 keys, we observe an increasing number of deadlocks while the proportion of `transfer` operations increase. However, the percentage of deadlocks across all `transfer` operations is still small. Finally, Figure 7b shows the time it takes to detect a deadlock, i.e., perform the Chandy-Misra-Haas algorithm. We observe that the median of the time this takes is similar across all workloads, and it also shows that as the amount of `transfer` operations increase, so do the higher percentile times.

Rollback Overhead. Figure 8 shows the maximum throughput for workloads with 50% and 100% `transfer` operations where different proportions of `transfer` operations fail for Sagas and two-phase commit coordinator functions. As expected, when using two-phase commit, a rollback does not increase the load in the system because the coordinator function needs to send a second message either way. Again, nothing out of the ordinary happened as the proportion of `transfer` operations to be rolled back increased. The throughput decreased as the protocol required additional compensating messages to be sent in the system. However, with 5000



(a) Time regular locks are held

(b) Time to detect deadlocks

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

(c) Frequency of deadlocks

Figure 7: Details of locking behaviour for a workload for 100 and 5000 keys with various proportions of transfers without rollbacks. The boxplots show the 5th and 95th percentiles.

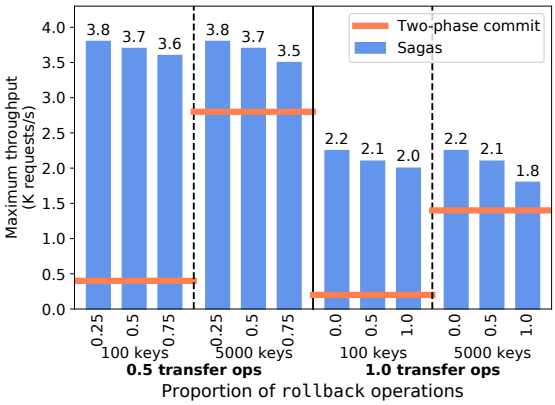


Figure 8: Throughput with different proportions of rolled back `transfer` operations for workloads with 50% and 100% `transfer` operations

keys, the difference is small at 50% `transfer` operations: 8% when going from 25 to 75% rollbacks and increasing to 18% with 100% `transfer` operations. This is larger than the 100 keys case that can still leverage the batching mechanism of StateFun and limit the performance drop to 10% in the worst case. Still, no matter the decrease in performance due to the compensating actions of the Saga protocol, it remains 20% faster than two-phase commit in the worst-case scenario of 5000 keys and 75% rollbacks.

7.3 Scalability Comparison

In the last experiment, we evaluate the scalability of the proposed system with coordinator functions. In Figure 9 we display the maximum throughput for both two-phase commit and Sagas at different amounts of StateFun workers and different transaction proportions in the workload. For Sagas, the scalability from 1 to 5 workers is close to 90% throughout for all workloads. For two-phase commit, the scalability from 1 to 5 workers starts at 87% at 10% `transfer` operations and drops to 75% at 100% `transfer` operations.

The reason for the low decrease in scalability on both protocols is that as workers increase, more traffic needs to go over the network. In the Sagas' case, the efficiency does not decrease across all workloads for the same reasons as expressed in Section 7.2. Namely, the system can still utilize batching, no locking is required, and the number of messages is two times lower than the two-phase commit protocol when all transactions succeed. On the other hand, the 8% decrease in scalability in two-phase commit from 10% to 100% `transfer` operations is due to the protocol's requirements for locks, more messages, and the inability to use batching. Considering all the impeding factors, it still manages to achieve decent efficiency with strong consistency guarantees in fully transactional workloads.

7.4 Micro Benchmark

As a final experiment, we conduct a micro benchmark on the system. At first, we keep the number of resources fixed, and then for every `transfer` proportion and number of keys, we measure the throughput at 80% load and the corresponding latency. By the results presented in Figure 10 we can see that for a use case with a low number of keys, the Sagas beat by a large margin the two-phase commit protocol in both throughput, with more than a 650% increase in performance, and latency that is at least two times lower. For a larger amount of keys, the contention becomes less of a problem. We observe a smaller difference between the two protocols at around 40% on average for throughput and a stable difference in latency around 20%. To conclude, Sagas seems to be the obvious choice for a small number of keys or high contention; if the business logic permits it. In any other case, the choice is mainly about the consistency guarantee requirements since the difference is not that significant.

8 RELATED WORK

SFaaS has been a very active area in both research and the open-source community. From the research community, the most relevant work is Beldi [35] which, like AFT[30], builds on top of Amazon's AWS Lambda to add fault tolerance and transaction support allowing for more complex state management. Their principal difference is that Beldi's execution environment is entirely serverless, while AFT relies on external servers for transaction support. To make that happen, Beldi uses atomic logging, extending Olive [27], to ensure fault tolerance for read and write operations, with garbage collection to manage the logs' growth. Regarding transactions, Beldi supports a variant of the two-phase commit protocol enforcing strong consistency guarantees with wait-die deadlock prevention.

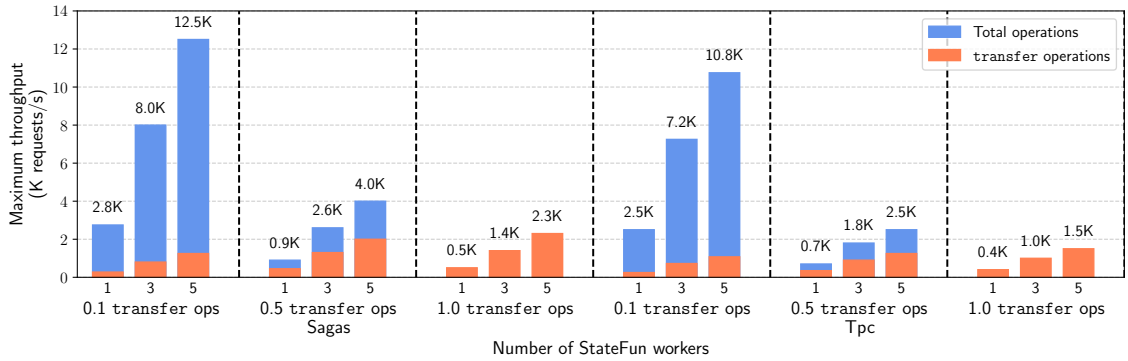
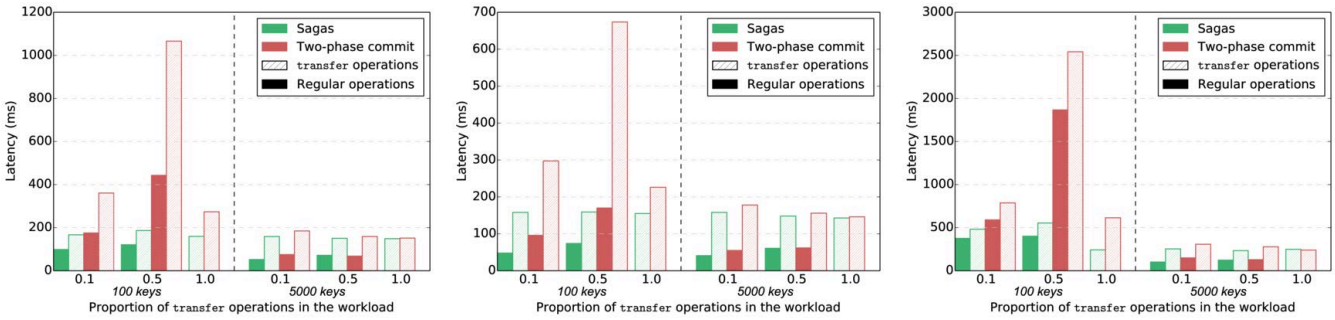


Figure 9: Maximum throughput for the system with 5000 keys for different numbers of StateFun workers for workloads with different proportions of `transfer` operations



(a) Mean **(b) Median** **(c) 95th percentile**

100 keys						5000 keys					
Sagas			Tpc			Sagas			Tpc		
0.1	0.5	1.0	0.1	0.5	1.0	0.1	0.5	1.0	0.1	0.5	1.0
11K	3K	2K	1.5K	0.4K	0.16K	9K	3K	2K	8K	2K	1.2K

(d) Throughputs at which latency was measured

Figure 10: Graph comparing latencies for Sagas and two-phase commit coordinator function for different keys and transaction proportions in the workload at 80% of the respective maximum throughputs

Cloudburst with Hydrocache [34] provides causal consistency guarantees within the same DAG workflow backed by Anna [33], a key-value state backend. The two most prominent open-source SFaaS projects are Cloudstate[6], based on stateful actors, and Apache Flink StateFun, which is presented in detail in Section 3.2. In Cloudstate, communication is allowed between different actors within the same cluster and between user-defined functions over gRPC.

The most notable difference among these systems in terms of programming model is state access. Both StateFun and Cloudstate encapsulate state within a specific function instance. In contrast, Cloudburst and Beldi allow any function access to any state stored in Anna or DynamoDB, respectively. Regarding transactions, only Beldi offers a programming model where the programmer writes two markers (begin/end_tx), and every function invocation in between will execute as part of a transaction. Our contribution is a programming model that supports transactions on StateFun with the choice of strong or relaxed consistency guarantees as shown in Section 4.1.

Furthermore, transactions on top of stream processing systems have received some interest in the literature. In [6] the authors introduce a transactional model over both data streams and traditional tabular data. Following a similar model, in [19] the authors add guarantees for snapshot isolation and consistency across partitioned state. Then TSpoon [2], an extension of FlowDB [1]), proposes a data management system built on top of a stream processor that supports transactions, giving the option of both strong and weak transactional guarantees and, queryable state. Our work focuses

[6]https://cloudstate.io/

on transactional workflows between generic stateful functions executed on a serverless dataflow system.

The large variety of use cases and systems makes them difficult to compare using a standardized benchmark. The related benchmarks that could be used to evaluate SFaaS systems are the *Yahoo! Cloud Serving Benchmark* (YCSB) [12] and the DeathStarBench [15]. Given that StateFun is based on Flink, which is a stream processing system, a stream processing benchmark [24] would be another alternative, but its workloads are not representative of those executed by a SFaaS system. In addition, we did not consider TPC-C [26] because it was created to test relational database management systems, including transactions, and requiring many additional features not present in SFaaS. We ultimately chose to develop and use an extension of YCSB [13] that introduced explainable transactional workloads, allowing for an easier interpretation of the results.

9 CONCLUSIONS

In this paper, we tackle the problem of supporting transactional workflows across cloud applications on a serverless platform. This problem is notorious in the microservices and cloud applications landscape. In this paper, we introduced a programming model and corresponding implementation for authoring workflows across stateful serverless functions with configurable transactional guarantees. Developers can opt for a distributed transaction across functions with strict atomicity and consistency guarantees or a Saga workflow that provides eventual atomicity and consistency. These complementary alternatives faithfully represent the requirements of real-world use cases. We described our implementation on top of Apache Flink StateFun, an open-source production-grade serverless sFaaS platform, and evaluated our implementation on an extended version of the YCSB benchmark that we developed in terms of a) throughput and latency overhead against the original StateFun, b) performance efficiency between distributed transactions and Saga workflows, and c) scalability. We found that our transactional workflows add affordable overhead to the system around 10%, Sagas significantly outperform distributed transactions on a scale of 15% – 34% depending on the amount of ongoing transactional workflows in the system, and scalability manifests a factor of 90% for Sagas compared to 75-87% for two-phase commit.

Acknowledgments. This work has been partially funded by the H2020 project OpertusMundi No. 870228. Experiments were carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

REFERENCES

[1] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2017. Flowdb: Integrating stream processing and consistent state management. In *DEBS*.
[2] Lorenzo Affetti, Alessandro Margara, and Gianpaolo Cugola. 2020. TSpoon: Transactions on a stream processor. In *Journal of Parallel and Distributed Computing*.
[3] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. 2019. Stateful functions as a service in action. In *VLDB*.
[4] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: fault-tolerant stream processing at internet scale. In *VLDB*.
[5] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*.
[6] Irina Botan, Peter M Fischer, Donald Kossmann, and Nesime Tatbul. 2012. Transactional stream processing. In *EDBT*.
[7] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink: Consistent Stateful Distributed Stream Processing. In *VLDB*.
[8] Paris Carbone, Marios Fragkoulis, Vasiliki Kalavri, and Asterios Katsifodimos. 2020. Beyond Analytics: The Evolution of Stream Processing Systems. In *SIGMOD*.
[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink TM : Stream and Batch Processing in a Single Engine. In *IEEE Data Eng. Bull.*
[10] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. 1983. Distributed Deadlock Detection. In *ACM Trans. Comput. Syst.*
[11] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Matthew Milano. 2021. New Directions in Cloud Programming. In *CIDR*.
[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SOCC*.
[13] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *ICDE Workshops*.
[14] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*.
[15] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*.
[16] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *ACM Sigmod Record*.
[17] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. 2021. Hazelcast Jet: Low-latency Stream Processing at the 99.99th Percentile. In *VLDB*.
[18] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, et al. 2020. Ambrosia: Providing performant virtual resiliency for distributed applications. In *VLDB*.
[19] Philipp Götze and Kai-Uwe Sattler. 2019. Snapshot Isolation for Transactional Stream Processing.. In *EDBT*.
[20] J. N. Gray. 1978. *Notes on data base operating systems*. Springer Berlin Heidelberg.
[21] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. In *CoRR*.
[22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. In *arXiv*.
[23] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB*.
[24] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *ICDE*.
[25] Asterios Katsifodimos and Marios Fragkoulis. 2019. Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications. In *EDBT*.
[26] Francois Raab. 1993. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In *The Benchmark Handbook*, Jim Gray (Ed.). Morgan Kaufmann.
[27] Srinath Setty, Chunzhi Su, Jacob R Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the fault-tolerance promise of cloud storage using locks with intent. In *OSDI*.
[28] Pedro Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. 2021. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. In *SIGMOD*.
[29] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *SIGMOD*.
[30] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. 2020. A fault-tolerance shim for serverless computing. In *EuroSys*.
[31] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst. In *VLDB*.
[32] Doug Terry. 2019. Transactions and Scalability in Cloud Databases—Can't We Have Both?. In *USENIX Association*.
[33] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. 2018. Anna: A KVS for Any Scale. In *ICDE*.
[34] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In *SIGMOD*.
[35] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *OSDI*.