Stateful Functions as a Service in Action

Adil Akhter
ING
adil.akhter@ing.com

Marios Fragkoulis
Delft University of Technology
m.fragkoulis@tudelft.nl

Asterios Katsifodimos
Delft University of Technology
a.katsifodimos@tudelft.nl

ABSTRACT

In the *serverless* model, users upload application code to a cloud platform and the cloud provider undertakes the deployment, execution and scaling of the application, relieving users from all operational aspects. Although very popular, current serverless offerings offer poor support for the management of local application *state*, the main reason being that managing state and keeping it consistent at large scale is very challenging. As a result, the serverless model is inadequate for executing stateful, latency-sensitive applications. In this paper we present a high-level programming model for developing stateful functions and deploying them in the cloud. Our programming model allows functions to retain state as well as call other functions. In order to deploy stateful functions in a cloud infrastructure, we translate functions and their data exchanges into a stateful dataflow graph. With this paper we aim at demonstrating that using a modified version of an open-source dataflow engine as a runtime for stateful functions, we can deploy scalable and stateful services in the cloud with surprisingly low latency and high throughput.

PVLDB Reference Format:
Adil Akhter, Marios Fragkoulis, Asterios Katsifodimos. Stateful functions as a service in action. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Serverless computing [12, 7] is a broad term used to describe the cloud service model in which users develop applications and then deploy them in a cloud infrastructure without having to deal with operational aspects such as scaling, recovery, and availability. The most common form of the serverless model is Function-as-a-Service[1] (FaaS). In FaaS users develop functions that read data state from an external data storage/database system, perform a computation and subsequently write data to an external system. Since functions do not encapsulate state, they can be executed in an embarrassingly parallel fashion and they do not require any coordination or data consistency guarantees.

In order to retain some form of state, functions can be initialised with their state retrieved from an external system and store back the new state when they finish. As a result, although it scales extremely well, FaaS is a poor fit for stateful latency-sensitive applications. Furthermore, in FaaS functions are not addressable, i.e., functions cannot call other functions directly; instead, they communicate with each other through cloud storage, which is slow. Finally, since they are not coordinated in any way and because state is external to the application, FaaS makes it virtually impossible to develop correct and consistent distributed applications. For these reasons, the current realisation of serverless falls short of its potential and its original purpose [6, 7].

We believe that remedies to the shortcomings of current FaaS offerings are *i*) the support of *state as a first-class citizen* in order to guarantee some form of state consistency and *ii*) the ability for functions to be composed into more complex applications with some form of transactional guarantees. To the best of our knowledge, the only attempts to this direction are Akka, the Orleans virtual actors [1] and relational actors [11]. We argue that these systems could deal with a large subset of the challenges with respect to scalability and state consistency, by exploiting existing work done in the context of streaming dataflow systems [5, 3, 2, 8].

In this paper we demonstrate this potential by using a high-level programming model allowing users to specify business logic in the form of stateful functions. Functions can call one another through named endpoints and they can be coordinated via distributed transactions, guaranteeing consistent distributed state. In our demonstration we execute such applications on a modified version of Apache Flink [3]. To this end, we translate applications into a stateful streaming dataflow graph in which function code becomes a dataflow operator and function invocations and their return values form events travelling though the dataflow graph. To enable scalability and low latency, state is automatically sharded and function code is given access to local state [4].

In this paper we first present our programming model and translation techniques (Section 2), as well as a demonstration scenario (Section 3). We then present a short discussion of related works (Section 4), concluding with future research and open problems (Section 5).

[1]This term is used by commercial offerings such as AWS Lambda, Azure Functions, and Google Cloud Serverless.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

```
1  class CreateOrderLambda extends LambdaDescriptor[OrderInitiated, OrderProcessed, OrderSummary] {
2   def internalState: ShardedState[OrderSummary] = new PersistedMap[OrderSummary]()
3    def execute(ctx: ExecutionContext, o: OrderInitiated): OrderProcessed = {
4     transaction {
5      val paymentApproved: PaymentApproved =
6        ctx.callLambda[OrderPaymentLambda](PaymentRequested(o.orderID, o.userID, o.amount))
7      internalState.update(OrderSummary(o.orderId, o, paymentApproved))
8      OrderPlaced(o.orderID)
9     }.onFailureReturn(OrderRejected(o.orderId))
10  }
11 }
```

Listing 1: Function definition that includes a state object, as well as a transactional call to another function.

2. STATEFUL FUNCTIONS IN THE CLOUD

We believe that stateful functions can provide the building block for high-throughput low-latency stateful applications with advanced state management requirements. A number of use cases can be supported this way. For instance, stateful microservices require low-latency response times under high load. Moreover, transaction coordination and state consistency has been a big challenge in microservices and people fall back to solutions such as SAGAs and eventual consistency. Distributed systems of stateful actors exchanging messages at scale need to be scalable and resilient to failures, while application state should remain consistent.

2.1 Programming model

Declaring Functions. Our declarative programming model consists of a `Function` and a `State` Scala trait. To specify a stateful function, the users of the FaaS platform extend the `LambdaDescriptor` trait and implement the `execute` method, which receives a set of parameters as input as well as an execution context. For instance, Listing 1 describes such a function namely `CreateOrderLambda` that outlines the creation of a new order in the system. As shown in Line 1, it extends the `LambdaDescriptor` trait, specifying an input (`OrderInitiated`) and an output (`OrderProcessed`). The `execute` method of `CreateOrderLambda` specifies the computation of the function when an `OrderInitiated` event arrives at the system.

Sharding and Paralellization Contracts. Line 2 declares the type of state that the function wants to have managed by the system, namely a `ShardedState` map of `OrderSummary`. It is important to note that the compiler ensures that the author of `OrderSummary` class declares a *key* on which the state will be partitioned in the cluster. The same applies to the input type `OrderInitiated`. This use of explicit keys both in the inputs but also in the state, is used by the underlying runtime engine to consistently route input events of the same key to the same process. This is the only "contract" that the system and the user have to agree on, in order to guarantee parallel and elastic deployment of functions. The actual sharding of state and parallelization of functions is taken over by the backend system.

Remote Function Calls. Our programming model provides primitives to manage and execute computational effects in a scalable and fault-tolerant manner. It facilitates high-level abstractions to declaratively specify control-flows, i.e., the interactions with the other functions and coordination among them as if the code is going to be executed in a local context. For instance, Line 6 calls the function `OrderPaymentLambda` with the `PaymentRequested` event that consists of a specific order id of a specified user and the total amount. This call returns with the status of the payment approval. The subsequent lines update the internal state with a new `OrderSummary` and complete the computation with `OrderPlaced`.

Transactional Workflow Execution. Line 4 begins a separate branch of the code which is meant to execute functions in a transactional manner. The implicit contract between the system and the user is that if one of the external function calls or access to a local state fails for any reason, we rollback all the changes that have resulted from all function calls within the `transaction` branch. For instance, if the local update to the `internalState` has failed in Line 7, the effects of the function call to the `OrderPaymentLambda` in the line above will be rolled back and we complete the execution of `CreateOrderLambda` with `OrderRejected`.

2.2 State Management at Scale

Deploying stateful functions in the cloud brings forward very challenging operational issues such as automatic elasticity and failure recovery. First, state has to be partitioned or replicated across the running instances of a function and input events should be routed to the correct partition in a deterministic manner. Second, scale-out actions require sharding a state partition even further, while scale-in actions entail merging multiple state partitions into one. Third, recovering a function instance with partitioned state entails keeping a snapshot of this state in a way that constitutes a global consistent snapshot of the application's state at a specific point in time when combined with the rest snapshots of partitioned state. When a failure strikes, the latest snapshot of a function's partitioned state together with the function's code can instantiate a new function instance that requests and re-processes all events from the latest checkpoint. These challenges are well known to the stream processing literature. Hence, we can benefit from years of prior research if we map stateful functions into a stream processing topology and deploy it on a streaming dataflow engine.

2.3 From Functions to Dataflows

Our intermediate representation (IR) is a directed graph which can contain cycles. We depict one of these graphs in Figure 2, where we show how a set of three functions and their invocations result into a cyclic dataflow graph with operators, and edges. The diamonds connecting the edges represent the types of events (parameters or returned values) sent from one function to another. The translation from our IR to a streaming dataflow is quite straightforward.

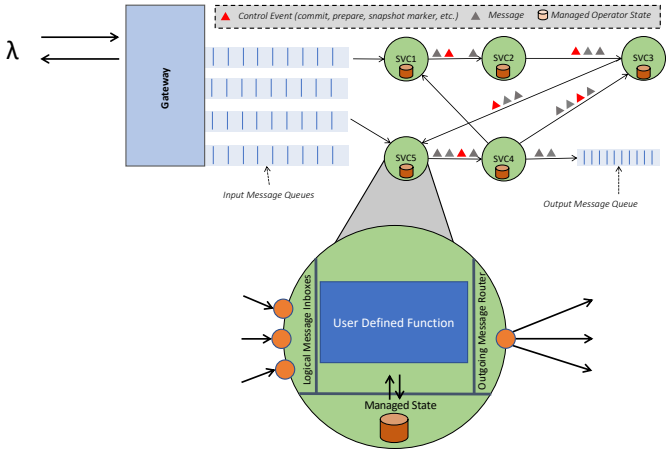

Figure 1: Overview of the overall system architecture. Functions are embedded inside dataflow operators, events become dataflow events. All inputs or outputs are persisted in a event broker/queue to be used for fault tolerance in case of failures.

Functions → Dataflow Operators. Functions with a single input and output event type and destination are simply translated into a dataflow operator.

Function State → Operator State. Each annotated managed state becomes the state of a stateful operator in the dataflow graph. Our macro-based static analyzer makes sure that only functions of a given context can read or write on a given state.

Data Dependencies → Dataflow Edges. Edges in the dataflow represent data dependencies among functions, i.e., for an event to be directed from one function to another and a response to come back, the two dataflow operators are connected with an edge encoding the type of events that this edge carries.

AND-nodes → Transaction Coordination. The AND nodes in our IR represent a transactional constraint: for the execution of `CreateOrderLambda` to complete, it has to receive success events from the other functions pointing to that AND node. If one of them fails, all the involved functions have to rollback their changes. We implement this functionality with transaction processing inside a dataflow as follows. In our prototype we implemented the two-phase commit (2PC) protocol via events that travel through the dataflow graph between operators and a transaction coordinator. More specifically, when a function uses the `transaction` primitive, the operator executing that function hands over the responsibility of executing the 2PC protocol to the transaction coordinator. Any changes to the local state of a given operator is first written to a log and applied to the state only upon successful commit of the 2PC protocol. When any function call fails, the calling operator informs the coordinator to instruct all the transaction participants to rollback their changes.

Function Orchestration. Special care has to be taken when a function depends on multiple input events that possibly come from different functions. In that case, the dataflow operator hosting that function needs to buffer events (e.g., in a `window` operator) and trigger the execution of the function as soon as all events are gathered together.

Handling External Requests. As seen in Figure 1, functions can be invoked via a gateway service. All request events enter a persistent event queue or broker (e.g. Kafka or RabbitMQ) and those requests are then processed by the dataflow operators in the system. Every incoming invocation of a function (via events) goes through a persistent queue for resilience reasons. The underlying streaming dataflow system needs to be able to replay certain parts of a stream in order to be able to perform fault tolerance with consistent state, debugging, and exactly-once processing guarantees [2]. Output events are also directed to a queue where they can be read and sent back to the invoker.

3. DEMONSTRATION SCENARIO

Demonstration Use-case. To showcase stateful functions we plan to use a real-world scenario of a set of stateful services dealing with orders, payments and stock management. The application use-case simply receives a request for filing an order. The order service then needs to reserve the credit of a given user with the amount required for the order, while making sure that it also reserves a certain stock. When both the stock and credit have been secured, the order service replies with a message to the user invoking that order request.

Part 1: Authoring Stateful Functions. In the first part, we give the attendants the opportunity to use our programming model and abstractions to specify several stateful lambdas declaratively. The implementation of our real-world scenario can serve as a basis to modify or extend during the demonstration. The compiled functions forming a logical dataflow graph can be interactively visualised at compile time from our static analyzer, as shown in Figure 2.

Part 2: Executing Functions as Dataflows. In the second part, we focus on the runtime of the system. We show how we can interpret and execute the logical dataflow graph and how it can be executed and debugged in a local environment on a single node. We then show how stateful functions translate into concrete Apache Flink jobs.

Part 3: Debugging Functions. In the third part, we allow users to replay input event queues and debug their functions with time-travel debugging.

Part 4: Scaling and Elasticity. Finally, we demonstrate executing a dataflow graph in a cloud infrastructure. We focus on several key system aspects such as the partitioning of the state, data-parallel execution of the stateful functions, transactional processing, etc. The attendants will have the opportunity to submit massive amounts of requests via a scalable request generator and follow how the system reveals possible bottlenecks and how it responds to pressure by dynamic re-scaling.

4. RELATED WORK

Our work is related to serverless computing and FaaS offerings, high-level programming models for stateful streaming dataflow graphs, and streaming transactions.

Serverless Computing and FaaS. We described FaaS, the current state of serverless computing offerings, in the introduction. Closest to our rationale is another flavour of serverless computing, Microsoft Azure's Service Fabric Mesh. According to the documentation, this is a platform

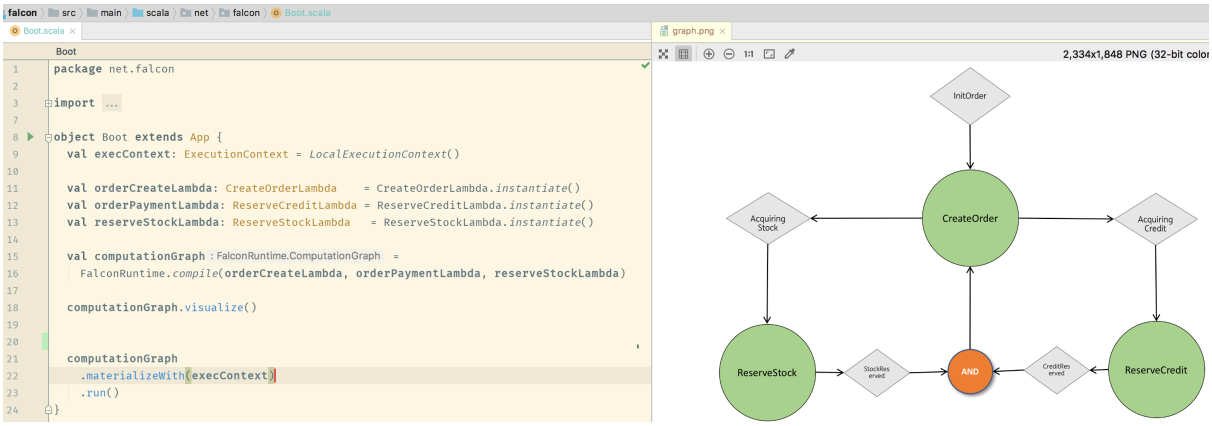

Figure 2: Live visualization of dataflows created from a set of functions that call each other in different ways.

for authoring and executing microservices that provides a managed service which includes state management, automatic scaling, distributed transaction, and failure recovery among others. Although going through the documentation for details is difficult, we believe that the ideas presented in this paper can be used to build a backend for executing service fabric microservices. Our programming model is meant to complement models such as the Azure Functions or Amazon's AWS Lambda.

High-level Programming Models. High-level programming models for stateful streaming dataflow graphs have been considered in big data processing [5] and reactive programming [10]. In [5] a stateful dataflow graph is statically inferred from imperative Java programs and executed in a data-parallel fashion using partitioned state to achieve scalability and checkpoints to provide fault-tolerance. The executing tasks are pipelined with one another to fulfill low latency as in streaming systems. Skitter [10] proposes a domain specific language for writing reactive programs and composing those into distributed workflows that Skitter's runtime deploys in a cluster. Skitter scales stateless reactive components according to available resources, but needs to synchronize state changes across stateful components. Contrary to the above works, this paper focuses on the translation of function definitions with dependencies and interconnections into a standalone dataflow graph.

Streaming transactions. Transactions in streaming systems is scarce. The only complete work is S-Store [9], which provides ACID guarantees on shared mutable state on a single machine, while a closed-source implementation of multi-key transactions exists in Apache Flink [3]. Our work builds on these works to provide an early prototype of a distributed transaction mechanism on a streaming system.

5. CONCLUSIONS & FUTURE WORK

In this paper we make the case for using a streaming dataflow system in order to execute stateful and resilient functions in the cloud in a scalable and elastic fashion. We show how surprisingly well existing dataflow sytems can be used for executing functions in the cloud, and point out some of their inefficiencies. Our next immediate steps are to optimize the throughput of transactions, developing versioning schemes for functions and advanced reconfiguration methods for dataflow graphs. Finally, we plan on working on a model for querying snapshots of live function states to allow users to have a complete view of the state of their applications.

6. REFERENCES

[1] P. A. Bernstein and S. Bykov. Developing cloud services using the orleans virtual actor model. *IEEE Internet Computing*, 20(5), 2016.

[2] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *PVLDB*, 10(12), 2017.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink®: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38, 2015.

[4] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A concurrent key-value store with in-place updates. In *ACM SIGMOD*, 2018.

[5] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *USENIX ATC*, 2014.

[6] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *CIDR '19*, 2019.

[7] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[8] A. Katsifodimos and M. Fragkoulis. Operational stream processing: Towards scalable and consistent event-driven applications. In *EDBT*. ACM, 2019.

[9] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, et al. S-store: streaming meets transaction processing. *PVLDB*, 8(13), 2015.

[10] M. Saey, J. De Koster, and W. De Meuter. Skitter: A dsl for distributed reactive workflows. In *ACM SIGPLAN Workshop on Reactive and Event-Based Languages and Systems*, 2018.

[11] V. Shah and M. A. Vaz Salles. Reactors: A case for predictable, virtualized actor database systems. In *ACM SIGMOD*, 2018.

[12] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *USENIX ATC '18*, 2018.