2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S) | 979-8-3503-2545-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/DSN-S58398.2023.00053

Multi-Consistency Transactional Support for Function-as-a-Service

Rafael Soares
INESC-ID,
Instituto Superior Tecnico, Universidade de Lisboa
Lisboa, Portugal
joao.rafael.pinto.soares@tecnico.ulisboa.pt

Luís Rodrigues
INESC-ID,
Instituto Superior Tecnico, Universidade de Lisboa
Lisboa, Portugal
ler@tecnico.ulisboa.pt

***Abstract*—We propose to design and implement a framework to support the concurrent execution of transactions with different consistency levels in Function-as-a-Service (FaaS) environments. The goal is to allow functionalities that have weak consistency requirements to execute efficiently, with minimal coordination, while, at the same time, allow functionalities that have strong consistency requirements to access the same data. We wish to explore what techniques may be leveraged to implement such a framework, what consistency levels should be supported, and how they can be supported efficiently in existing FaaS environments.**

***Index Terms*—FaaS, Data-Consitency, Transactions**

I. Introduction

Function-as-a-Service (FaaS), also known as Serverless Computing, has emerged as a key paradigm to support the execution of applications in the cloud. A significant advantage of FaaS is that users are not required to reserve resources explicitly: these are automatically provisioned by the cloud provider as needed. This paradigm requires programmers to develop their applications in the form of compositions of stateless functions that can be organized into workflows, such as directed acyclic execution graphs (DAGs), to implement complex functionalities. In run-time, the provider assigns the computational nodes required to execute these functions: different functions, or even different instances of the same function, can be executed by different nodes.

Functions cannot preserve state across invocations or share state in memory. Applications that require state persistence must execute in *stateful serverless functions* (SSFs) [1], allowing functions to preserve and share state on external storage services. However, even if the external storage service is shared across SSFs [2] and is able to offer strong guarantees to each individual function, it may be hard to offer strong consistency for a composition of SSFs, because functions from the same composition may be executed by different nodes and, therefore, can be observed as different clients (that are not part of the same "session") by the storage service. For instance, it may not be trivial to ensure the atomicity of the writes performed by a composition, if parts of the write-set are persisted by different functions. Also, when reading from the persistent storage, functions from the same instance of a given composition can read versions that belong to different snapshots. In most cases, functions from a composition only have Eventual Consistency (EC) [3] guarantees.

Due to these limitations, a number of recent works have proposed to extend FaaS frameworks with support for consistent access to persistent storage, including different forms of transactional guarantees. Examples of supported models include Transactional Causal Consistency (TCC) [4] and Opacity [1]. To the best of our knowledge, all of these system assume that all compositions run under the same consistency level: the one provided by the middleware. This can be overly restrictive, and impose a performance penalty for compositions that can operate under weak consistency. For example, a social media application may require Strict Serializability (SS) for login, TCC for adding new contacts, and EC for making a post visible, while these functionalities may be using the same functions or key-space in different transactional contexts. Compositions that only require EC should not have their performance penalized by others that require Strict Serializability.

II. Research Objectives and Questions

We propose to design and implement a framework to support the concurrent execution of transactions with different consistency levels in Function-as-a-Service (FaaS) environments. More specifically, we wish to answer the following research questions:

- Is it possible to create a coordination protocol for the concurrent execution of multiple consistency levels without affecting the performance of each consistency level?
- What consistency levels should be supported by this protocol?
- If such protocol exists, is it possible to implement it in an efficient and scalable manner in the FaaS environment?

III. Background and Challenges

Multi-consistency protocols have been proposed in different seetings. [5] supports the execution of different operations with different consistency levels in geo-replicated scenarios, allowing the reordering of weakly consistent operations in different sites while ensuring the total ordering of strongly consistent operations across all sites. However, [5] requires existing operations to be rewritten into at least two sub-operation, a generator operation and a shadow operation,

979-8-3503-2545-4/23/$31.00 ©2023 IEEE
DOI 10.1109/DSN-S58398.2023.00053

requiring developers to rewrite their existing FaaS functionalities. [6] supports multi-consistency in the context of distributed databases with transactional support, by allowing eventually consistent (EC) transactions to concurrently execute with eachother, while ensuring that strictly serializable (SS) transactions execute isolated from all other conflicting transactions (SS and EC). However, [6] requires EC transactions to block until conflicting SS transactions terminate, which may hamper the performance of EC transactions. These systems only consider a limited number of consistency criteria (such as Strict Serializability and Eventual Consistency) and do not support other consistency levels currently used in FaaS, such as Transactional Causal Consistency.

Different strategies can be considered to adopt previous works to FaaS environments. The first relies on the coordination between a caching layer and storage [4]. While this approach is efficient, it requires modifications to the storage system to manage the metadata required for coordination. This would require clients to host, deploy, and maintain their own storage to support such a system, going against the simplicity and pay-per-use nature of FaaS. A second approach would be to rely on an intermediate layer, positioned between the computation and storage layers of FaaS, operating as a transactional manager to functions [7]. While this technique is compatible with current FaaS infrastructures, it still requires clients to deploy a specialized service that executes the intermediate layer. A third approach consists in relying on some existing strongly consistent storage service to maintain a shared log that is used to coordinate the execution of transactions [1]. This approach avoids the deployment of custom services, but forces all functions to read and write the log to perform coordination, which can impair the performance. Finally, one may also implement a shared log in a FaaS environments [8], but its usage requires changes to the FaaS engine, becoming incompatible with existing FaaS solutions.

With this background, we identify two main challenges:

- **Challenge 1:** How do we coordinate transactions of different consistency levels without affecting the performance of weakly consistent transactions (in particular, the latency of weakly consistent transactions should not be impaired by complex coordination procedures).
- **Challenge 2:** How do we efficiently implement our multi-consistency protocol while supporting compatibility with existing FaaS offers.

IV. Research Directions

We now provide some insight on how we aim to address the aforementioned challenges.

Addressing Challenge 1. Contrary to previous works, we focus our attention on the performance of read operations, as most weakly consistent workloads are read-intensive. We want read operations of each transactional level to have only the minimal overhead required to ensure its consistency criteria: SS must read the last committed values at the time the transaction is initiated, TCC should read a set of causally consistent values (not necessarily the last committed), and EC should read directly from storage without any additional coordination. This support should be seamless, without requiring modifications to the provided application code to provide better portability.

We propose a multi-layered approach to provide to each transactional level the required amount of coordination to support its guarantees. As the bottom layer we would have a EC storage layer, allowing EC transactions to directly contact the storage and obtain results in an efficient manner. We then add a TCC protocol layer on top of the storage system, allowing TCC transactions to obtain a causally consistent snapshot. Finally, for SS, we will augment existing TCC protocols to provide a snapshot that respects external consistency. Systems like Cure [9], which rely on physical clocks to causally order transactions, can be leveraged to request the freshest snapshot at the transaction start time.

Regarding write operations, two properties must be maintained to respect the consistency guarantees of each transactional level. First, causal dependencies should be maintained to allow TCC read transactions to obtain causally consistent snapshots. Second, SS transactions must be able to detect write conflicts. Regarding the first property, we propose that all write transactions should follow a TCC commit protocol to ensure causal dependencies are maintained regardless of consistency levels. While it penalizes the performance of EC write transactions, it will allow efficient performance of TCC read transactions. It does not affect the performance of SS transactions, as most TCC write protocols follow a 2PC protocol, which can be included in the SS commit protocol. Regarding the second property, we plan to use optimistic concurrency control (OCC) approach to detect write conflicts. For this purpose, before committing, SS transactions need to check for conflicts. EC and TCC are not required to validate their write-set, as they do not impose any restriction regarding conflicting transactions, but they must ensure that they are serialized after any conflicting SS transaction to not break the consistency of the stronger consistency model.

Consistency is defined at the function composition granularity. Different functions may belong to multiple compositions with different consistency levels. Objects itself do not have any associated consistency level.

Addressing Challenge 2. Implementing these layers in a compatible way with existing FaaS providers is an important step to allow the wide adoption of these techniques. We plan to build on top of a weakly consistent storage systems available from existing FaaS providers. The TCC and SS conflict detection tasks can then be implemented by an intermediate layer, similar to AFT [7], executing the read and write protocols of each transactional level. While clients must incur the costs of maintaining this intermediate layer, we consider a worth while trade-off for the compatibility with existing FaaS providers.

Acknowledgments: The authors are grateful to the anonymous reviewers for their comments on an early version of this paper. This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) under grant UI/BD/153590/2022 and via project UIDB/50021/2020 and DACOMICO (via OE withref. PTDC/CCI-COM/2156/2021).

REFERENCES

[1] H. Zhang, A. Cardoza, P. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *OSDI '20*, Banff, Canada, Nov. 2020.

[2] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *VLDB*, vol. 13, no. 12, p. 2438–2452, Jul. 2020.

[3] T. Ward, R. Mehta, and R. Tewani. (2020) Implement the serverless saga pattern by using AWS Step Functions. [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/implement-the-serverless-saga-pattern-by-using-aws-step-functions.html

[4] T. Lykhenko, R. Soares, and L. Rodrigues, "FaaSTCC: efficient transactional causal consistency for serverless computing," in *Middleware '21*, Virtual Event, Canada, Dec. 2021.

[5] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *OSDI '12*, Hollywood (CA), USA, Oct. 2012.

[6] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "SALT: Combining ACID and BASE in a distributed database," in *OSDI '14*, Broomfield (CO), USA, Oct. 2014.

[7] V. Sreekanti, C. Wu, S. Chhatrapati, J. Gonzalez, J. Hellerstein, and J. Faleiro, "A fault-tolerance shim for serverless computing," in *EuroSys '20*, Heraklion, Greece, Apr. 2020.

[8] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *SOSP '21*, Virtual Event, Germany, Oct. 2021.

[9] D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *ICDCS '16*, Nara, Japan, Jun. 2016.