Object as a Service (OaaS) Serverless Cloud Computing Paradigm

Pawissanutt Lertpongrujikorn, Mohsen Amini Salehi
High Performance Cloud Computing (HPCC) Lab, University of North Texas
pawissanuttlertpongrujikorn@my.unt.edu, mohsen.aminisalehi@unt.edu

I. Introduction

The emergence of cloud technology has drastically transformed the application development process. With cloud infrastructure, provisioning can now be done in a few minutes, as opposed to the weeks or months it used to take. Over the past decade, cloud services have replaced mundane tasks with software automation. The current state-of-the-art, serverless platform utilizes the function-as-a-service (FaaS) paradigm to enable developers to build applications by simply writing code in the form of a function and uploading it to the platform. The system then automates the process of building, deploying, and auto-scaling the application, making the overall development process more effortless and mitigating the burden for programmers and cloud solution architects. Major public cloud providers offer FaaS services (e.g., AWS Lambda, Google Cloud Function, Azure Function), and several open-source platforms for on-premise FaaS deployments are emerging (e.g., OpenFaaS, Knative). In the backend, the serverless platform hides the complexity of resource management and deploys the function seamlessly in a scalable manner. FaaS is proven to reduce development and operation costs via implementing scale-to-zero and charging the user in a truly pay-as-you-go manner. Thus, it aligns with modern software development paradigms, such as CI/CD and DevOps [4].

As the FaaS paradigm is primarily centered around the notion of stateless *functions*, it naturally does not deal with the *data*. However, in practice, most use cases need to maintain some form of (structured or unstructured) state and keep them in the external data store. Thus, often the developers have to intervene and undergo the burden of managing the application data using separate cloud services (e.g., AWS S3 [2]). Even though stateless functions make the system scalable and manageable, the state still exists in the external data store, and the developer must intervene to connect the function to the data store. For instance, in a video streaming application [6], developers must maintain video files, metadata, and access control, in addition to developing functions.

Apart from the lack of data management, current FaaS abstractions do not natively support function workflows. To form a workflow, the developer has to generate an event that triggers another function in each function. However, for large workflows, configuring and managing the chain of events become cumbersome. Although function orchestrator services (e.g., AWS Step Function [1] and Azure Durable Function [5]) can be employed to mitigate this burden, the lack of built-in workflow semantics (see Figure 1) in FaaS forces the developer to intervene and employ other cloud services to manually navigate the data throughout the workflow. In sum, although FaaS makes the resource management details (e.g., auto-scaling) transparent from the developer's perspective, it does not do so for the data, access control, and workflow.

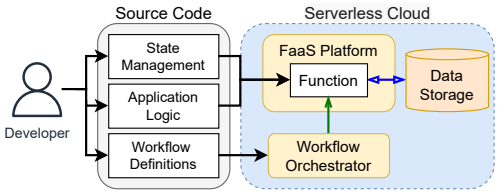
Last but not least, FaaS has limited performance control support. Because the cloud provides separate service abstractions for computing, databases, and other related components (e.g., workflow, messaging, etc.), it prevents the opportunity for the whole application optimization (e.g., data locality, caching, etc.). Moreover, the cloud lacks coordination between the cloud and developers. As a result, cloud service is operated with little knowledge of the application, and developers are less capable of controlling or "hinting" the system to satisfy the QoS requirements.

To overcome these inherent problems of FaaS, we develop a new paradigm on top of the function abstraction that mitigates the burden of resource, data, and workflow management from the developer's perspective. *We borrow the notion of "object" from object-oriented programming (OOP) and develop a new abstraction level within the serverless cloud, called* ***Object as a Service (OaaS)*** paradigm. Incorporating the application data and workflow into the object abstraction unlocks opportunities for built-in optimization features, such as data locality, data reliability, caching, software reusability [7], and data access control. Moreover, objects in OaaS offer developers encapsulation and abstraction benefits and the ability to transparently define workflows of cloud functions (a.k.a. dataflow programming [12]).

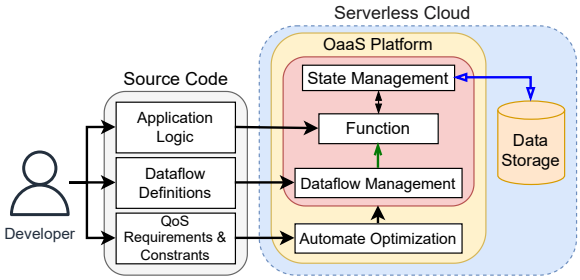
As shown in Figure 1, unlike FaaS, OaaS segregates the state management from the developer's source code and incorporates it into the serverless platform to make it transparent from the developer's perspective. OaaS also incorporates workflow orchestration as the dataflow abstraction with built-in data navigation between functions. Furthermore, developers can provide the QoS requirements to the platform to automate the performance optimization smartly behind the scenes. Such that the application's performance can be guaranteed or promptly rejected if not feasible.

II. Oparaca: OaaS-Based Serverless Platform

To offer the OaaS paradigm, we develop **Oparaca** (**O**bject **Para**digm on Serverless **C**loud **A**bstraction) platform. In this section, we will discuss the noteworthy key features of Oparaca.



(a) Function as a Service (FaaS)



(b) Object as a Service (OaaS)

Fig. 1: A bird-eye view of FaaS vs. OaaS.

A. Modular and platform agnostic designs

First, Oparaca is designed to be modular and platform-agnostic. Oparaca doesn't tightly rely on any FaaS system or underlying platform but instead uses the standardized API/protocol as the abstraction layer. The most important aspect is that it abstracts developers' code from the cloud storage. Because of Oparaca utilizing the schematic of *pure function* that package the object state and input request into the standalone invocation task. This task is offloaded to the code execution runtime and is expected to return with the modified state. Therefore, the code execution runtime is entirely decoupled from the state management. By using an RPC request for offloading a task, any FaaS engine can accept this task to process and return the output and modified state in the response body. Although Oparaca currently only provides comprehensive integration with Knative [8], connecting the other FaaS engine can be done by configuring the URL.

B. Unstructured data support

Other than the *structured data* (e.g., JSON) supported by the previously-mentioned *pure function* schematic, Oparaca allows developers to combine the *unstructured data* (e.g., multimedia file) as a part of an object state. To meet the platform-agnostic objective, Oparaca uses the S3 protocol [2], a standardizing protocol for object storage, for implementing the data access. This approach is not limited to AWS and can be implemented using open-source solutions like MinIO [10] and Ceph [15], which support S3 API. Oparaca employs the presigned URL technique to allow the developer's code access to the file in object storage directly without the need to share the secret key and avoid the leak of sensitive information.

C. Consistency

Oparaca has a mechanism to maintain the data consistency that protects from failure and race conditions. Since Oparaca supports structured and unstructured data, it needs to manage two data stores. Both data stores have to update at the same time successfully. Otherwise, the data becomes inconsistent. If the failure happens, one data store can successfully update the state while the other might not. Oparaca prevents this scenario by employing the *fail-safe state transition*. In another case, when users send multiple requests to modify the same object, it can cause the race condition. Oparaca also avoids this scenario by using a built-in locking mechanism.

D. Performance

To improve the performance, caching is a popular technique used to reduce data movement, and improve the end-to-end response time. Oparaca provides in-memory caching capabilities via implementing distributed hash table (DHT) [9] through integration with the in-memory data grid (IMDG), allowing cache object states at scale. Using *consistent hashing* [13], Oparaca can quickly determine the location of cached data and send requests there in a single hop of the network. As a result, the end-to-end response time is improved without the burden of manually setting up the cache system.

E. Dataflow support

Instead of offering a standard workflow, Oparaca provides the dataflow abstraction that allows the developer to define the invocation steps in the dataflow as a form of *directed acyclic graph (DAG)*. In each step, the developer can declare the output of each invocation as a temporary variable within the workflow. Then, the next invocation can use the previous temporary variable as an input or a target for calling a function, and the system recognizes this step relation as an edge in DAG. Upon dataflow invocation, the system can automatically resolve the order of dataflow steps by topological ordering of DAG. Therefore, developers are relieved from the hurdle of manually navigating the data between multiple functions within the workflow.

F. QoS-driven optimization

Oparaca provides developers with the interface to control the performance of their applications in high-level abstraction. This is achieved by allowing developers to define their Quality of Service (QoS) requirements and constraints. Once the information is received, the system checks for any conflicts or infeasibilities with the current resources available. If any issues are detected, the user's request is promptly rejected. To meet the requirements, Oparaca connects to the monitoring system and reacts to changes in workload or performance by adjusting the allocated resources or system configuration.

G. Dynamic Class Runtime

To fulfill the variety of QoS requirements on different applications or classes, having the *class runtime* (i.e., underlying services for enabling class) shared among them is difficult to manage because of possible conflict in QoS requirements. To address this issue, Oparaca dynamically creates a dedicated class runtime for each class. Using this approach, Oparaca can make the class runtime have specific characteristics based

on the requirement. For instance, Oparaca can disable unimportant components (e.g., in-memory cache store) to reduce the cost for low-budget requirements.

III. Table of Contents

The table of contents of the tutorial will be as follows:

1) Introduction:
   a) Serverless system and our vision of the next-generation cloud computing.
   b) The current state of the practice, FaaS paradigm
2) Motivations:
   a) The shortages of FaaS
   b) The problems of guaranteeing application performance in FaaS development.
3) Our solutions: Object as a Service (OaaS) Paradigm
   a) Unified OaaS Abstraction (Data, function, non-functional requirement encapsulation into the notion of object)
   b) High-level non-functional requirement interface to drive the cloud-developer coordination
4) Oparaca Concepts and Designs:
   a) Our design goals for designing Oparaca
   b) System Architecture
   c) Dynamic Class Runtime
   d) QoS-driven optimization
   e) State Management
   f) Maintaining Consistency with *fail-safe state transition*
   g) Performance improvement with caching and consistent hashing via distributed hash table (DHT)
5) Demonstration of developing services with Oparaca (hands-on)
   a) Installing Oparaca platform inside local Kubernetes
   b) Creating a new function with Python code
   c) Defining a new class in YAML
   d) Using Oparaca CLI to manage the class deployment
   e) Creating a new object and invoking its function
   f) Optimizing by defining the non-functional requirement (QoS and constraints)

IV. Potential Audience of the Tutorial

Embracing the OaaS paradigm can benefit developers of small companies or startups trying to develop and deploy new services without bearing the burden of low-level details. By leveraging this paradigm, developers can focus on high-level application logic while offloading the operational aspects of the service to software automation. This results in a more agile and effective development process, which can help startups quickly deliver new services to the market. The company also benefits from a serverless auto-scaling feature, which reduces resource waste during low workloads and maintains user experience during high traffic by dynamically adjusting allocated resources.

V. Requirements for the Tutorial

For this tutorial, we need a projector and internet access for the presenter and audience. Audiences who want to follow the tutorial should have some basic knowledge of cloud and Kubernetes and have a laptop that can install container runtime.

To make the hands-on tutorial flow smoothly, the audience should have a laptop that already installs the basic software requirements: Docker [11], Java [3] (version 21), and Python [14] (version >= 3.11) runtime. Moreover, the laptop should have at least 16 GB of memory to run all of the software without the out-of-memory error.

VI. Biography of the Instructor(s)

Dr. Mohsen Amini Salehi is an Associate Professor at the Computer Science and Engineering (CSE) department, University of North Texas (UNT), USA. He is the director of High Performance and Cloud Computing (HPCC) Laboratory where several graduate and undergraduate students research on various aspects of Distributed and Cloud computing. Dr. Amini is an NSF CAREER Awardee and, so far, he has had 11 research projects funded by National Science Foundation (NSF) and Board of Regents of Louisiana. He has also received 10 awards and certificates from in recognition of his innovative research, including the "Best Service Award" from IEEE/ACM CCGrid '23 Conference. His research interests are in democratizing cloud-native application development, building smart and trustworthy systems across edge-to-cloud continuum, and heterogeneous computing.

Pawissanutt Lertpongrujikorn is currently a Ph.D. Student in computer science and engineering at the University of North Texas. Pawissanutt works as a research assistant at the High-Performance Cloud Computing (HPCC) Lab in the computer science and engineering department. His research interest includes developing a new paradigm for cloud-native programming and serverless systems. He earned a B.Eng. in computer engineering from Kasetsart University in Thailand.

References

[1] Amazon. AWS Step Functions | Serverless Microservice Orchestration. https://aws.amazon.com/step-functions. Accessed on 23 Jul. 2022.

[2] Amazon. Cloud Object Storage | Amazon S3 – Amazon Web Services. https://aws.amazon.com/s3/. Online; Accessed on 12 Nov. 2023.

[3] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley Professional, 2005.

[4] S. Bangera. *DevOps for Serverless Applications: Design, deploy, and monitor your serverless applications using DevOps practices*. Packt Publishing, 2018.

[5] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8):1591–1604, 2022.

[6] Chavit Denninnart and Mohsen Amini Salehi. SMSE: A Serverless Platform for Multimedia Cloud Systems. *arXiv preprint:220.0194*, 2022.

[7] Chavit Denninnart and Mohsen Amini Salehi. Harnessing the potential of function-reuse in multimedia cloud systems. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):617–629, 2021.

[8] Cloud Native Foundation. Knative. https://knative.dev/. Online; Accessed on 12 Nov. 2023.

[9] Yahya Hassanzadeh-Nazarabadi, Sanaz Taheri-Boshrooyeh, Safa Otoum, Seyhan Ucar, and Öznur Özkasap. Dht-based communications survey: architectures and use cases. *arXiv preprint arXiv:2109.10787*, 2021.

[10] MinIO Inc. MinIO | High Performance, Kubernetes Native Object Storage. https://min.io/. Online; Accessed on 12 Nov. 2023.

[11] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[12] Tiago Boldt Sousa. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, volume 130, 2012.

[13] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[14] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[15] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.