TU WIEN Informatics

CRDT-based Serverless Middleware for Stateful Objects in the Edge-Cloud Continuum

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Valentin Goronjic, BSc
Matrikelnummer 51900253

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr. Stefan Nastic, BSc

Wien, 6. Mai 2024

Valentin Goronjic	Stefan Nastic

TU WIEN Informatics

CRDT-based Serverless Middleware for Stateful Objects in the Edge-Cloud Continuum

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Valentin Goronjic, BSc
Registration Number 51900253

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr. Stefan Nastic, BSc

Vienna, May 6, 2024

Valentin Goronjic	Stefan Nastic

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Valentin Goronjic, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Mai 2024

Valentin Goronjic

Danksagung

Zu Beginn möchte ich mich bei meinem Betreuer Assistant Prof. Dr. Stefan Nastic für die Möglichkeit, diese Arbeit zu verfassen, bedanken. Das kontinuierliche und wertvolle Feedback und die zahlreichen Meetings haben maßgeblich zum Erfolg der Arbeit beigetragen. Außerdem möchte ich mich bei der gesamten Distributed Systems Group für die freundliche Aufnahme im Büro bedanken.

Außerdem möchte ich mich bei meiner Partnerin, meinen Freunden und meiner Familie bedanken, die mich allesamt während meines Studiums und dieser Arbeit unglaublich unterstützt haben. Vielen Dank für eure Motivation, Ermutigung, Ratschläge und die emotionale Unterstützung!

Acknowledgements

First and foremost, I would like to thank my advisor, Assistant Prof. Dr. Stefan Nastic, for giving me the opportunity to conduct this thesis and continuously providing valuable feedback that significantly contributed to its outcome. I'd also like to thank the whole Distributed Systems Group for the warm welcome in the office.

Furthermore, I'd like to express my deepest gratitude to my girlfriend, friends, and family, who have all been incredibly supportive during my studies and this thesis. Thank you for your motivation, encouragement, advice, and emotional support!

Kurzfassung

Die Beibehaltung von Zuständen ist herausfordernd für serverlose Funktionen am Netzwerkrand, da diese üblicherweise auf entfernten Speicherdiensten liegen und dies somit eine potenziell hohe Latenz mit sich bringt. Andere Lösungen halten die Daten entweder nicht lokal, beinhalten Komponenten, die von einer zentralen Komponente abhängig sind, oder sind für Cloud-basierte serverlose Plattformen konzipiert. All diese Lösungen sind daher nicht für den Edge-Bereich geeignet. Es benötigt daher einer Lösung für dieses Problem, die die oben genannten Anforderungen des Edge-Cloud-Kontinuums erfüllt und gleichzeitig in bestehende serverlose Plattformen integrierbar ist.

In dieser Arbeit präsentieren wir MISO, eine CRDT-basierte serverlose Middleware für das Edge-Cloud-Kontinuum. Sie stellt sogenannte MISO Stateful Objects für serverlose Funktionen bereit, die mehrere dezentralisierte, konfliktfreie replizierte Datentypen in einem einzigen Objekt bündeln. Die Beiträge dieser Arbeit umfassen: i) das konzeptionelle Modell der MISO Stateful Objects, ii) die MISO-Middleware, bestehend aus dem Architekturmodell und einem Open-Source Software-Prototyp, iii) ein SDK für serverlose Funktionen, welches die Verwendung von MISO Stateful Objects in serverlosen Funktionen ermöglicht und iv) die asynchrone Replikation von MISO Stateful Objects mittels einem Overlay Netzwerk, optimiert in Hinblick auf die Datenmenge und den Ressourcenverbrauch.

In unserer Performanceevaluation verwenden wir eine AllReduce-Operation in einer serverlosen Funktion, welche in OpenFaaS läuft. Unsere Lösung benötigte 26.7% weniger Zeit als Redis Enterprise und war fast 2.5-mal schneller als MinIO. Wir evaluieren unseren Replikationsmechanismus mittels einem Open-Source gRPC Benchmark Tool. Wir senden 5 Millionen Anfragen an die Middleware, welche gleichzeitig ein MISO Stateful Object auf mehreren Nodes in unterschiedlich großen Clustern verändern. Unsere Ergebnisse zeigen, dass der Replikationsmechanismus skalierbar ist und Zehntausende von gleichzeitigen Anfragen bewältigen kann. Außerdem beschreiben wir die Auswirkung des Replikationsintervalls auf die RPS, die Latenz, die Experimentdauer und die gesendete Datenmenge. Wir demonstrieren praktisch, dass sich die Middleware nahtlos in eine bestehende serverlose Plattform (OpenFaaS) integrieren lässt. Schließlich zeigen wir, dass unser SDK im oben genannten AllReduce-Experiment weniger Codezeilen benötigt und eine geringere oder ähnliche kognitive Komplexität aufweist wie die anderen beiden verwendeten Lösungen.

Abstract

Maintaining the application state is challenging for serverless functions that run at the edge of the network. This is because the predominant way to maintain the application state using serverless functions is to access remote storage services, which are not suited for the edge due to potential high latency. Other proposed solutions for stateful serverless functions either do not promote data locality, contain components that depend on a centralized authority, or are built for cloud-based serverless platforms, all of which are not suited at the edge. There is a need for a solution to this problem that satisfies the aforementioned requirements of the edge-cloud continuum and is generalizable to multiple serverless platforms.

This thesis introduces MISO, a CRDT-based serverless middleware for the edge-cloud continuum. It provides MISO Stateful Objects for serverless functions, which bundle multiple decentralized and conflict-free replicated data types into a single object. The list of contributions of this thesis includes: i) the conceptual model of MISO Stateful Objects, ii) the MISO middleware consisting of the architectural model and an open-source software prototype, iii) an SDK for serverless functions that enables the usage of MISO Stateful Objects within serverless functions, and iv) the asynchronous replication of MISO Stateful Objects using an overlay network optimized towards data transfer and resource consumption.

For our performance evaluation, we utilize an AllReduce operation in a serverless function running in OpenFaaS. In this experiment, our solution took 26.7% less time than Redis Enterprise and was almost 2.5 times faster than MinIO. We evaluate the middleware's replication mechanism with an open-source gRPC benchmark tool to perform 5 million requests concurrently modifying a MISO Stateful Object on clusters of different sizes. Our results indicate that the middleware's replication is scalable and can handle tens of thousands of concurrent requests. Furthermore, we clearly demonstrate the impact of the replication interval on the RPS, latency, experiment time, and replication traffic. Additionally, we demonstrate that the middleware can be seamlessly integrated into an existing serverless platform (OpenFaas). Finally, we show that our SDK requires fewer lines of code and has less or similar cognitive complexity than the other solutions we utilized in the aforementioned AllReduce experiment.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Contributions . . . 3
1.4 Research Questions . . . 4
1.5 Methodology . . . 5
1.6 Structure . . . 7

2 Background **9**
2.1 Serverless Computing . . . 9
2.2 Conflict-free Replicated Data Types . . . 12
2.3 gRPC . . . 14

3 Related Work **17**
3.1 Stateful Functions for Existing Serverless Platforms . . . 17
3.2 New Approaches for Stateful Serverless Serverless Functions . . . 19

4 MISO Middleware **23**
4.1 Conceptual Model . . . 23
4.2 Middleware Architecture . . . 24
4.3 Middleware Core Modules . . . 26
4.4 SDK for Serverless Functions . . . 34

5 Prototype Implementation **39**
5.1 Middleware Package . . . 40
5.2 SDK Package . . . 45
5.3 CRDT Package . . . 47
5.4 Common Package . . . 49

6 Evaluation **53**
6.1 Performance Overhead 53
6.2 Replication Algorithm 60
6.3 Qualitative Evaluation 66
6.4 Limitations 72

7 Conclusion **73**
7.1 Research Questions 74
7.2 Future Work 76

A GitHub Repository **77**
A.1 Implementation Packages 77
A.2 Evaluation 77

List of Figures **79**

List of Tables **81**

List of Algorithms **83**

List of Listings **85**

Glossary **87**

Acronyms **89**

Bibliography **91**

CHAPTER 1

Introduction

1.1 Motivation

Modern software development increasingly relies on serverless computing. Function as a Service (FaaS) platforms abstract away the complexity of provisioning computational resources from developers [1], [2] by running custom functions in response to events or Application Programming Interface (API) calls [2]. Furthermore, the platforms manage the provisioning and execution of these functions and scale them accordingly on demand [1], [3]. Serverless functions can be written in a broad range of programming languages and are uploaded to the platform along with their dependencies [2]. Serverless functions are stateless, meaning they do not retain any memory about previous executions [2]. This means that applications requiring a globally shared state at fine granularity cannot be built easily with the serverless paradigm due to the fact that serverless functions are stateless [3]. Serverless platforms typically do not provide a way to manage state [2]. The predominant way to handle mutable state, hence, is to access remote storage services in the function code, which may have high access latency [2], [4]–[6] as well as limited or costly I/O performance [2], [4], [5].

The adoption of the serverless paradigm at the edge of the network is rising [1], [7], [8]. Edge computing aims to build a bridge between clouds and applications by distributing computing and storage resources over multiple geographically dispersed regions toward the edge of a network. Edge nodes generally have limited computational resources [9]. The reason why serverless is becoming increasingly relevant at the Edge is the increase of real-time and data-intensive applications powered by Internet of Things (IoT) devices requiring low latency [1], [9]. Network latency from the edge device to the cloud can be high, which poses significant strains on the concept of centralized data centers. Transferring and processing exponentially increasing data volumes can also lead to low throughput [9]. Therefore, the problem of managing the state of serverless functions is particularly challenging at the edge of the network [1].

1.2 Problem Statement

In the literature, multiple proposals for solutions to implement stateful serverless computing can be found. However, to the best of our knowledge, no current solution is explicitly designed for the *distributed and heterogeneous characteristics of the edge-cloud continuum* and the *architectures of existing serverless platforms*. We describe those requirements below in more detail.

Dependence on Centralized Authority The edge-cloud computing continuum is dynamic and consists of many different devices [1], [7]. The principle of edge computing is subject to failure if the applications and networks are improperly designed, implemented, or deployed [1]. Edge devices may fail unexpectedly, have limited computational resources and energy, and are particularly vulnerable Aslanpour *et al.* [1]. Furthermore, complete sites containing multiple devices can fail at any time [10]. Applications at the edge should not depend on a central node due to the potential unreliability [1]. Some authors have proposed solutions for stateful serverless functions that rely on a centralized queue to sequentially perform writes regarding the same entity [11] and are thus not suited for the edge. Other ways to achieve stateful serverless functions often depend on a component that requires the election of primary nodes with consensus protocols such as *Raft* or *Paxos*. Examples of this are MongoDB [12], [13] and Redis [13], [14], where writes are only possible against primary nodes [14], [15]. This is challenging for the volatile environment of edge computing, as such protocols typically assume that at most half of the nodes fail simultaneously [16], which might not be guaranteed in all cases at the edge. Given this information, it is evident that *solutions for stateful serverless functions that rely on any central authority are not ideal for the edge-cloud continuum.*

Data Locality Many existing solutions for stateful serverless functions *separate the data and the serverless platform* into two distinct layers running on separate machines. For example, the *Crucial* framework proposed by Barcelona-Pons *et al.* [2], the *Object as a Service* paradigm introduced by Lertpongrujikorn *et al.* [17], and Apache Flink Stateful Functions [18] manage the state physically separated from the node running the serverless function. Use cases like IoT that run on the Edge of the network often require low latency [1], which is a potential challenge for architectures where the data is physically separated from the serverless functions. Baresi *et al.* [9] propose to add a caching layer to serverless platforms to minimize latency when retrieving states from remote services, which works for retrieving the state concurrently but does not solve the issue of concurrent data modifications and potential data conflicts when serverless functions are called by multiple nodes at the same time. Puliafito *et al.* [8] propose that all requests accessing a certain stateful object are routed to the same container instance. While routing requests to the same container preserves data locality, relying on a single container for the whole session is not ideal, as outlined in the paragraph above. Shahidi *et al.* [6] highlight the necessity for an intermediate layer that is positioned between the

serverless functions and the storage infrastructure that places the application state in close proximity to the nodes executing serverless functions to boost performance.

Lack of Generalizability Some related work like Cloudburst [19] promotes data locality but introduces *entirely new serverless platforms* that do not seamlessly integrate with existing ones. Similarly, Lertpongrujikorn *et al.* [17] have introduced an entirely new paradigm to manage state for serverless functions. [8] propose a system model of a new stateful FaaS platform tailored to the edge but do not provide a concrete implementation. Other solutions, such as Durable Functions [11], are tailored to a specific serverless platform and not generalizable to multiple serverless platforms.

While entirely new serverless platforms might solve the issue of stateful serverless functions, the proposals we encountered in our literature review [19] [17] potentially force developers to adapt the workflows in which they write, deploy, and execute the serverless functions to make use of the stateful objects. There is a need for more research about generalizable solutions that can be seamlessly integrated into existing open-source serverless platforms, such as OpenFaaS [1] or Knative[2]. Such a solution should work with multiple serverless platforms, independent of the programming language that they are developed in. In this thesis, we aim to make a contribution to this gap.

Conflict-free Replicated Data Types (CRDTs) Recent research has introduced *CRDTs*, which are replicated abstract data types that are designed to be modified concurrently. They offer *strong eventual consistency*, which means replicas *eventually converge to the same state* if it is ensured that all replicas receive the exact same updates. It is very promising to use CRDTs to establish a solution to the aforementioned problems, as they are decentralized and do not rely on a central authority for synchronization. The principle of those data types is that state modifications are propagated asynchronously to other replicas, and updates can be merged without the need to resolve data conflicts manually [20].

This thesis presents MISO, a CRDT-based serverless middleware that provides stateful objects to serverless functions in the edge-cloud continuum. The goal of the middleware is to solve the aforementioned challenges.

1.3 Contributions

We summarize the contributions of this work as follows:

1. **Introduction of the conceptual model of MISO Stateful Objects**, which are used to manage state in serverless functions. From the developer's perspective, they are the most important building block of the middleware, as they create and

[1] https://github.com/openfaas/faas
[2] https://github.com/knative/serving

use them in the code of serverless functions. MISO Stateful Objects belong to a single serverless function and combine multiple CRDT-based data types into a single object.

2. **Definition of the MISO middleware's architecture and implementation of a software prototype**. The middleware is architecturally designed to be integrated into existing serverless platforms. Furthermore, it requires different core mechanisms to operate, which are clearly described in this thesis. We additionally provide a practical software prototype implementation of the middleware, following the proposed architecture and core mechanisms.

3. **Definition and implementation of a Software Development Kit (SDK) for serverless functions**. This is a key component to use MISO Stateful Objects in serverless functions. It provides proxy versions of the data types within MISO Stateful Objects. Those proxy versions delegate operations to the middleware transparently whenever operations are invoked on them.

4. **Replication of the data using an overlay network**. The replication algorithm optimizes the state replication in terms of data transfer and resource consumption by utilizing information provided by the overlay network. It is designed to replicate data asynchronously and debounced with a configurable delay to adapt to different latency requirements.

5. **Implementation of various state-based CRDTs**. This is required by the middleware to provide the functionality of the MISO Stateful Objects. The data types are implemented in TypeScript as a separate package/library and can also be used by other developers for purposes unrelated to MISO.

1.4 Research Questions

We formulate the following **research questions** and their expected results:

- **RQ1: What is an appropriate architecture for a CRDT-based middleware that provides Stateful Objects for serverless functions in the edge-cloud continuum, so that it is scalable and integrable into existing serverless platforms?**
  This is a software design/implementation question and is to be determined during the development of the solution. The expected results of this question include a concept and architectural description of the middleware components and modules and a prototype implementation that integrates with an existing serverless platform (e.g., OpenFaaS).

- **RQ2: What is an appropriate way to efficiently replicate Stateful Objects between different nodes running the middleware to avoid unnecessary network traffic?**

Replication of the MISO Stateful Objects is a core mechanism of the middleware. To make this process efficient, we expect that we need to leverage information provided by serverless platforms and/or the underlying container orchestrator. It is expected that simply replicating the state changes between all nodes of the serverless platform (i.e., via gossip protocols) will not scale.

- **RQ3: What is the overhead of using CRDT-based Stateful Objects in the edge-cloud continuum in terms of performance and resource consumption, compared to other state-of-the-art solutions for stateful serverless functions?**
  This is a question that can only be answered after evaluating the results of this thesis.

1.5 Methodology

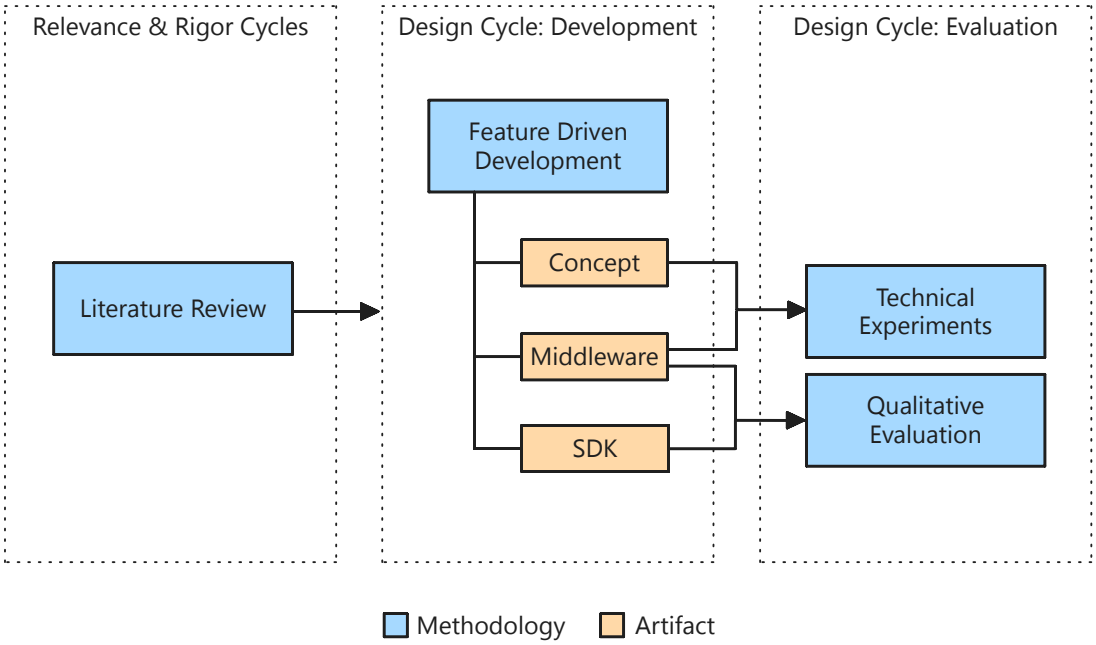

Figure 1.1: Methodology followed in this Thesis

Figure 1.1 provides an overview of the methodology followed in this thesis. The methodology of this thesis aligns with the paradigm of **Design Science Research (DSR)** [21], which is a research approach that aims to develop and evaluate artifacts as solutions to real-world problems. The core of DSR is the *Design Cycle*, in which an artifact is developed and evaluated in an iterative process. This cycle has inputs from existing knowledge (*Rigor Cycle*) and its environment (*Relevance Cycle*). The finished research then contributes back to the knowledge base.

For the **Relevance and Rigor Cycles** of this thesis, a literature review is conducted to highlight existing knowledge, software artifacts, and problems in the field of serverless functions. This literature review focuses on CRDTs, serverless functions, and existing approaches for stateful serverless functions. By including existing software artifacts in the literature review, this thesis aims to ensure that the proposed solution is relevant and builds upon existing knowledge in the field.

In the **Design Cycle**, a concept and implementation of a CRDT-based serverless middleware that enables MISO Stateful Objects for serverless functions in the edge-cloud continuum is derived in an iterative process. *This thesis produces two artifacts: (1) the concept and architectural overview of the middleware, and (2) a practical instantiation of the concept in the form of a software prototype.*

The **Feature Driven Development (FDD)** approach is used to create the artifacts in the *Design Cycle*, which is a popular agile development method [22] that promotes short incremental iterations [23]. In FDD, the development process is organized around the *features* that are to be implemented in the software, aligning with the business requirements. The method is adaptive and capable of accommodating late changes in the requirements. The overall objective of FDD is to consistently deliver high-quality software across every stage in the development cycle [23].

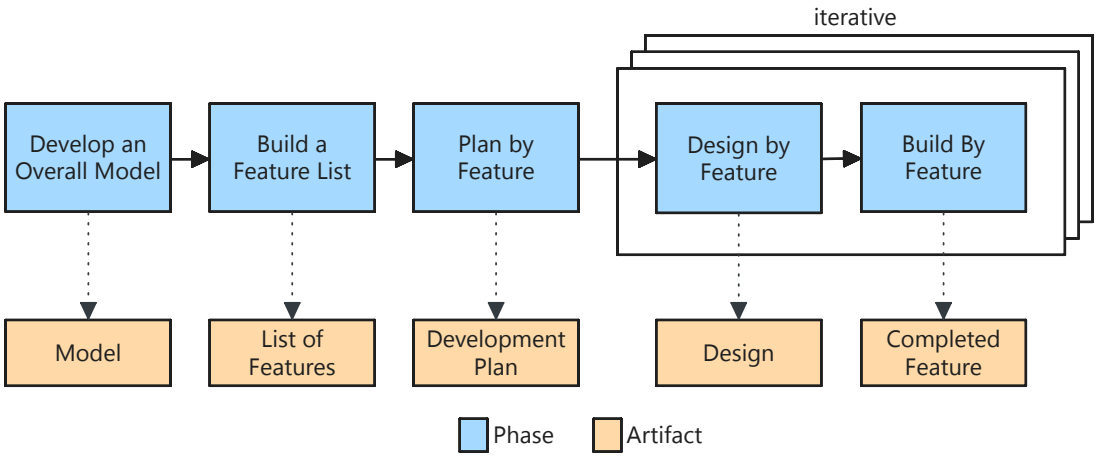

Figure 1.2: Feature Driven Development Process (own work based on [22], [23])

The FDD process is visible in Figure 1.2 consists of five phases [22], [23]. The first phase is to *"Develop an Overall Model"*, where the overall project scope is defined, and domain experts create multiple object models. The most optimal model is then selected after a thorough review. The second phase is to *"Build a Feature List"*, where all features of the system are identified on the basis of the previously selected object model from the first phase. A feature is one that provides business value. The third phase is to *"Plan by Feature"*, where each feature is thoroughly planned to produce a development plan. This is then followed by an iterative phase called *"Design by Feature"*, where the design

phase of the implementation is done (e.g., creating UML diagrams). The last step is implementing and testing the previously created designs in the *"Build by Feature"* phase. This phase is also an iterative one. In this thesis, Chapter 4 describes artifacts of the first three phases, and Chapter 5 describes the outcomes of the last two phases.

In the **Evaluation** part of the *Design Cycle*, the concept and implementation of the middleware is **evaluated** against the state of the art. The software prototype has two parts: a middleware and a SDK. Technical experiments are performed to measure the performance while modifying MISO Stateful Objects. This includes the modification of CRDTs and the replication process. We further assess the *integration complexity* of our middleware to evaluate how well it integrates with existing serverless platforms. The SDK is evaluated qualitatively to measure its usability by analyzing the *understandability*. This is an important characteristic for software systems, as it ensures that developers can quickly understand if a component or system fits specific tasks and helps them pick the right components for their needs [24].

1.6 Structure

This thesis is structured as follows. Chapter 2 provides an overview of relevant background information about serverless functions, gRPC, and CRDTs. Chapter 3 gives an overview of relevant related work. We present the conceptual model and architecture of the MISO middleware in Chapter 4. This concept serves as the basis for implementing the software prototype, and Chapter 5 provides detailed information about the implementation process. This software prototype is then evaluated using technical experiments and a qualitative evaluation, and Chapter 6 describes the results of the evaluation. Finally, we conclude the results of this thesis in Chapter 7, where we reflect on the research questions, summarise our findings, and identify opportunities for future work. The source code for the software prototype and evaluation is available on GitHub[3] and can be found in Appendix A.

[3] https://github.com/

CHAPTER 2

Background

2.1 Serverless Computing

Modern software development offers developers a wide range of possibilities for building software. FaaS providers provide developers with an abstraction of the infrastructure layer so that they can focus more on developing the application logic itself. This is achieved through so-called serverless functions, which are called in response to events or API calls and execute the desired application code. The FaaS platform manages the provisioning and execution of the function and the necessary infrastructure in the background. The application developers can choose from a broad range of programming languages to develop serverless functions. The code is then bundled with the dependencies and uploaded to the FaaS platform [2].

Serverless functions do not retain any state from previous executions. This means that certain applications requiring state modifications and synchronization in a fine-grained matter, such as in machine learning, are not trivial to build with serverless functions [2]. Research shows that the importance of stateful functions is rising [7].

2.1.1 Serverless Edge Computing

The principle of FaaS is well-known in the cloud (e.g., AWS Lambda) but is increasingly used at the edge of the network as well, abstracting away the infrastructure in the whole edge-cloud continuum [1], [7]. This potentially saves costs, as tasks such as scaling applications are accomplished for developers [7]. Existing serverless platforms have started to adapt themselves for limited resources at the edge (such as IoT devices), such as OpenFaaS with the *faasd* provider or AWS Greengrass with *Lambda@Edge*. Requirements for the edge include low latency and real-time execution, which highlights that cloud computing might not be suitable for all use cases due to latency. Edge computing hence allows a continuum ranging from cloud to edge [1].

2.1.2 Kubernetes

Kubernetes is the predominant way to orchestrate and manage containers. Containers run software systems isolated from the host in a lightweight way. Every container is bundled with all binary files, dependencies, and configurations required to run the software [25]. Serverless functions are also run as containers [26]. One of the benefits of using containers in combination with Kubernetes is that the number of replicas that run the software can be changed during runtime, which supports elasticity in modern software. Kubernetes runs containers on multiple compute nodes. The containers are, for example, executed by Docker[1] [25].

One method to ensure containers are deployed on each node in a Kubernetes cluster is to use a *DaemonSet*. This is useful for system-level capabilities and allows Pods deployed on this node to share the local network interface and the local file system. This results in reduced latency [27]. We make use of this in our OpenFaaS integration and discuss this in Chapter 6.

Serverless platforms typically contain an ingress controller or API gateway that clients can use to invoke serverless functions [9], [28]. This component is the basis for a URL-based routing to invoke serverless functions [28]. Additionally, serverless platforms contain a component that manages the containers on which the serverless functions are run [28], [29]. The name of this component varies depending on the serverless platform, and in this thesis, we will call it *provider* in accordance with how the serverless platform *OpenFaaS* names it [30].

Helm Charts Manually defining all the necessary configuration files in large software systems is inconvenient. This is the reason why the Cloud Native Computing Foundation (CNCF) released Helm, a package manager for applications running in Kubernetes. Helm works by bundling applications into so-called *Helm Charts* that are stored in a local or remote repository. They bundle all resources that are required to run the application in Kubernetes and can easily be (un-)installed and upgraded [25].

Every chart has the following structure [25]:

- **chart/**
  - **Chart.yaml**
    Contains metadata about the chart
  - **values.yaml**
    Contains configuration values for the chart (e.g., image repository and tag)
- **templates/**
  Contains the Kubernetes manifest files to be deployed, which are populated with values from the values.yaml file

[1] https://www.docker.com/

- **charts/**
  Folder that contains dependencies of the chart

Helm charts are important for the middleware developed as part of this thesis for the following reasons:

- Serverless platforms often also provide their software as a Helm chart, as they often use Kubernetes as a container orchestrator. Our middleware can seamlessly be integrated into those platforms by modifying the Helm chart of the serverless platform.
- Helm charts provide a way to bundle the middleware so other developers can easily run it in a Kubernetes cluster. It can certainly also be run without Kubernetes, so using Kubernetes is not mandatory.
- We use Helm charts in our evaluation (Chapter 6) to simplify the deployment of our test scripts.

Prometheus Prometheus[2] is an open-source monitoring tool. It collects metrics from different sources and stores them ordered by time. The list of supported metric types includes Counters, Gauges, Histograms, and Summaries. The software also offers a powerful query language and real-time alerting functionalities [31].

Grafana Grafana[3] is an open-source tool that can visualize time-series data. Manually loading the data into the software is not required as it can directly work with various data sources. Grafana provides flexible dashboards to visualize and manipulate the data [32].

In this thesis, we combine Prometheus and Grafana to monitor metrics during the evaluation. More details on this can be found in Section 6.2.

2.1.3 OpenFaaS

OpenFaas is an open-source FaaS platform [33]. It supports different *providers* using which serverless functions are deployed and executed. A popular provider is *faas-netes* [30], a provider that deploys and executes functions using Kubernetes. when a function is deployed to OpenFaaS, the Kubernetes provider creates a Kubernetes service and Deployment with a defined number of replicas. Scaling the function to multiple pods works via Kubernetes mechanisms. The provider is preceded by the OpenFaas gateway, via which the function can be called from outside (e.g., via HTTP)[34].

The *faas-netes* Kubernetes provider can conveniently be deployed via a Helm chart [35].

[2] https://prometheus.io/
[3] https://grafana.com/

2.2 Conflict-free Replicated Data Types

CRDTs are specific abstract data types built to be replicated optimistically in such a way that automatic reconciliation is possible despite potentially retrieving intermediate updates in arbitrary order. CRDTs offer strong eventual consistency, which is a consistency model that guarantees that two replicas return the same value for a CRDT when both have received the same updates. Common fields where CRDTs are used include key-value databases, collaborative editing tools, blockchains, and others [36].

Every CRDT consists of three main parts: *the state, interface, and the update propagation mechanism*. The state is an internal data structure that represents the data type. The interface defines the set of operations that can be executed against the CRDT. The update propagation mechanism describes how updates (i.e., operations that modify the state) are sent to other replicas and how they process the updates [36].

CRDTs are divided into state-based (i.e., Convergent Replicated Data Types) and operation-based (i.e., Commutative Replicated Data Types), depending on the type of update propagation mechanism used [20], [36].

In state-based CRDTs, also called Convergent Replicated Data Types (CvRDTs), the complete state of the source replica is propagated for synchronization after every applied update. All other replicas then merge the received state with their current state, producing a converged state. There are three requirements for the merge function: it needs to be associative, commutative, and idempotent (ACI). Furthermore, the states must monotonically increase with every update so that a join-semilattice can be formed with the set of all possible states [36].

In operation-based CRDTs, also called Commutative Replicated Data Types (CmRDTs), an encoded version of the update is propagated to other replicas as soon as the update is executed on the source replica through a process called *prepare-update* [36]. This works by splitting the update into a pair (t, u) where t is the *prepare-update* method that does not change the state and u is the *effect-update* method that changes the state [20]. The *prepare-update* is only executed at the replica that triggered the update, and *effect-update* is executed on all replicas [20]. The update operation must, therefore, be commutative so that all replicas end up in the same state, regardless of the order in which the updates are delivered. This, however, also requires causally reliable delivery of the updates [36].

Figure 2.1 compares state- and operation-based CRDTs. In state-based CRDTs, as depicted in the left half of Figure 2.1, whenever an update modifies the internal state, it is sent to other replicas, which then merge the incoming state with its state. Eventually, every update reaches every other replica directly or indirectly. Alternatively, in operation-based CRDTs, every update triggers the *prepare-update* method on the source replica. After this, the *effect-update* method is executed on every other replica [20].

Examples of CRDTs include Counters, Sets, and Registers [37].

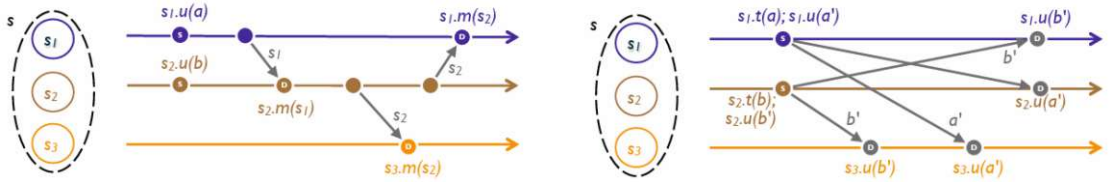

Figure 2.1: State-based vs. Operation-based CRDTs [20]

Counters A Counter is a data type that supports incrementing and decrementing an integer-based value and retrieving the current numeric value. A counter could, for example, store the number of registered or logged-in users [37].

One version of a counter that only supports increments is a **Grow-only Counter (GCounter)**. To allow concurrent modifications in the counter in a state-based CRDT, it is required to use a vector clock where every source replica has an entry. Every replica modifies only the value in its entry in the vector clock, and the value of the counter equals the sum of all entries in the vector clock [37].

A **Positive-Negative Counter (PNCounter)** is a counter that allows both incrementing and decrementing. A solution to implement such a counter in a state-based way is to use two separate GCounters for increments and decrements. The value of the counter then is the difference between those counters [37].

Registers A Register is a data structure that works like a memory cell. Values can be *assigned* to it, and the *value* can also be queried. Concurrent updates do normally not compute. Potential solutions for this are only to retain the last write (Last-Write-Wins Register) or to retain all conflicting values (Multi-Value Register). [37].

A **Multi-Value Register (MVRegister)** retains multiple values in the case of conflicting writes. Clients are in charge of manually reducing the values into a single one, if required, by assigning the reduced value. Using only a scalar timestamp, such as in a Last-Write Wins Register, is insufficient for a state-based MVRegister. A set of $(X, versionVector)$ pairs is required, where X is the data type value to be assigned. When assigning new values, a new version vector is computed that supersedes previous writes [37].

Sets Sets offer the operations `add` and `remove`. Implementations of CRDT-based Sets differ in the semantics of concurrent add and remove operations. For example, a 2P-Set prefers removes, and an Observed-Remove Set (ORSet) retains elements in case of conflicting addition and removal of the same element [37].

An **ORSet** supports both additions and removals of elements, and the causal history of the sequence defines the outcome of a combination of additions/removals. When an element is both added and removed at the same time, the element will stay in the set [37]. The initial version of an ORSet used tombstones to mark elements that have

been removed from the set, which accumulate over time, which means that the allocated memory grows as elements are removed [38]. Bieniusa *et al.* [38] proposed a new design of an *Optimized ORSet* where every replica i stores a vector v of unique identifiers that have already been part of the set. Each replica has a local counter at the i-th entry of the vector $v[i]$ with a default value of 0. When an element is added, a unique identifier is created by the next local counter value of this replica as well as the replica ID. When a ORSet is merged on a downstream replica, an element is only contained in the final result if contained in both payloads or if it is present in the local payload and not recently removed in the upstream replica. Elements count as removed when they are not in the payload, but their identifier is stored in the vector v [38].

Flags Flags behave like boolean switches. An **Enable-Wins Flag (EWFlag)** retains true in case of conflicting true/false values, and the opposite is true for the Disable-Wins Flag (DWFlag). This works by incrementing the value of a counter in every replica whenever the flag value is set to true or false for EWFlags and DWFlags, respectively [39].

Considering the characteristics of Edge Computing, it is evident that we need to use state-based CRDTs for this thesis. Causally reliable delivery of all updates cannot be ensured with devices at the edge of the network, as these are potentially unreliable. This is why we use state-based CRDTs for our solution, as lost updates are no problem as every update contains the entire state of the data type. The downside of this is that the whole state needs to be transmitted in every update. However, the middleware only replicates to nodes that run the same serverless function. Furthermore, using state-based CRDTs simplifies state restoring in the case of restarted nodes.

2.3 gRPC

gRPC is an open-source and cross-platform Remote Procedure Call (RPC) framework released by Google [40]. RPC is a form of Inter-Process Communication (IPC), which is a crucial part of distributed software systems [41]. Initially released in 2015, gRPC is now part of the CNCF, and gRPC has become very popular [40].

The list of advantages of gRPC includes:

- **Efficiency**
  gRPC transmits data using a protocol buffer-based binary protocol on top of HTTP/2 [40], offering high performance [41].
- **Contract-first approach**
  Services and their interfaces are defined before implementation [40], which are then used to generate client/server code [41].
- **Strongly typed**
  Helps to reduce errors caused by wrong data types [40].

- **Cross-Platform compatible**
  gRPC service definitions are language-agnostic, hence clients and servers can be built with different programming languages [40], [41].
- **Bidirectional stream support**
  gRPC offers client- and server-side streaming, allowing developers to use both request-response and streaming-like communication [40].
- **Mature and stable**
  gRPC is mature and used by various tech companies such as Google, Netflix, Docker, Cisco, and CoreOS [40], [41].

Some disadvantages of gRPC are [40]:

- **Potentially not suitable for external-facing services**
  The contract-first approach might negatively influence the flexibility of consumers
- **Changes in the contract might require regeneration of client/server**
  gRPC is usually interoperable with different schema versions on the client/server. However, breaking changes might require the regeneration of the client and server code.
- **No mature support in browsers/mobile applications**
  Low maturity level, compared to Representational State Transfer (REST).

Figure 2.2 gives an overview of how gRPC works. In this example, a gRPC *Service Definition* called `ProductInfo` is defined in the `ProductInfo.proto` file. This file is then used to generate both client and server code. It can be seen that, in this case, the client and server are implemented in different programming languages (Java and Go). The communication between the gRPC stub on the client and the gRPC server happens via HTTP/2 [40].

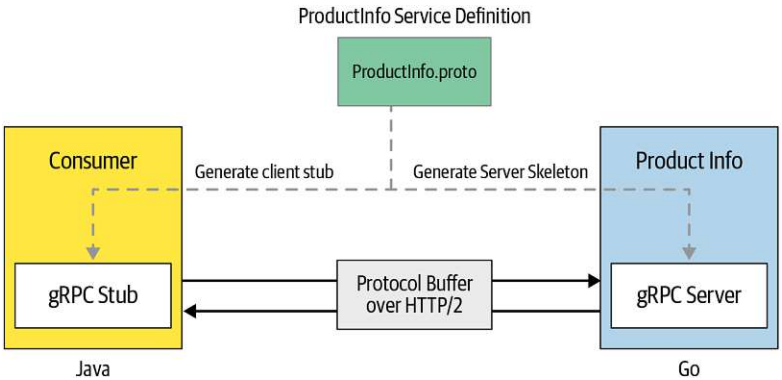

Figure 2.2: gRPC Overview [40]

Each gRPC service is defined in protocol buffers in files with a `.proto` extension. This file defines all gRPC services with their methods, parameters, and return types [40].

For the prototype developed as part of this thesis, we will leverage gRPC for communication between instances of the middleware as well as between serverless functions and the middleware. In the case of this thesis, the SDK acts as the gRPC client and the middleware as the gRPC server. The efficiency of gRPC contributes to fast response times. Cross-platform compatibility is a requirement, as serverless functions can be written in different programming languages. As part of this thesis, an SDK to be used in serverless functions is developed, so the strongly defined types/schemas are helpful and do not hinder the flexibility of developers. For the replication logic, the replication module makes use of bidirectional streams to avoid opening/closing network connections too often with very frequent replication intervals.

The disadvantages of gRPC mentioned above are not significant in our case, as the middleware-related gRPC endpoints are usually only used by our SDK, where we take care of the connection management and the gRPC types. This means there is no negative impact on the flexibility of developers who write serverless functions. Furthermore, MISO does not run on browsers or mobile platforms, and as we control the code for both SDK and middleware, regeneration for clients/servers is feasible.

CHAPTER 3

Related Work

This Chapter presents related work. The problem of stateful serverless computing has already been addressed multiple times in the literature. The existing approaches that provide stateful objects to serverless functions can be categorized into two distinct categories. Some authors propose a solution for state management that works in tandem or enhances existing serverless platforms. Other authors propose entirely new approaches for stateful serverless functions, such as serverless platforms that treat the state as a first-class citizen or entirely new paradigms that advance the regular FaaS model. To the best of our knowledge, we could not find a solution for stateful serverless functions in the literature that satisfies all of our requirements for the edge-cloud continuum. Those are: i) data locality, ii) generalizable to work with multiple different serverless platforms, and iii) no dependency on a central authority.

3.1 Stateful Functions for Existing Serverless Platforms

This section highlights the most relevant existing approaches that either work in tandem or enhance existing serverless platforms and provide stateful objects for serverless functions. This thesis also contributes to this category.

Crucial Framework Barcelona-Pons *et al.* [2] proposed the *Crucial* framework that allows developers to implement stateful distributed applications using the serverless paradigm. Their implementation works by mapping local threads to the invocation of cloud functions (called cloud threads). To manage the issue around shared state, they developed a distributed shared objects (DSO) layer using state machine replication with strong consistency guarantees. The framework offers the possibility to make DSOs durable by replicating them across multiple nodes and then storing them on stable storage. The DSO layer is built on top of a disaggregated in-memory data store (Infinispan) and deployed together with the serverless function. The authors claim to have achieved higher

throughput than comparable in-memory storage solutions such as Redis with their DSO implementation. Crucial works with FaaS platforms that offer a Java runtime [2].

Similar to Crucial, our work provides an SDK for developers to instantiate MISO Stateful Objects that behave like a proxy. However, in contrast to the data types of Crucial, our MISO Stateful Objects are based on CRDTs. Furthermore, Crucial runs next to the serverless platform (e.g., on EC2 instances connected via a Virtual Private Cloud to AWS Lambda functions). Furthermore, developers have to supply the code of the serverless functions to *CloudThreads*, which then execute the code on the respective serverless platform. In our work, we want to extend existing serverless platforms and run our middleware on nodes of the serverless platform that run serverless functions. This means that we join the infrastructure with the middleware, which could run in the same container orchestrator as the serverless platform. This means that the data of our MISO Stateful Objects does not have to travel to different machines, which is especially relevant for the area of Edge Computing. Furthermore, if our work is integrated into the serverless platform, developers do not have to provision additional services next to the serverless platform. Our middleware does not modify the process of deploying serverless functions to the platform, so there is no abstraction like Crucial's CloudThreads. Unlike Crucial, our middleware does not support arbitrary classes as MISO Stateful Objects because it is a limitation of CRDTs. However, we support a broad set of data types that developers can add to our MISO Stateful Objects to maintain state.

Azure Durable Functions Durable Functions (DFs) [11] is a component of Microsoft's Azure Functions FaaS platform. In DF, there are three types of functions: (1) Activities (stateless FaaS functions), (2) Entities (actors that encapsulate application state), and (3) Orchestrations (coordinate activities and entities). The DF model implicitly saves the progress of orchestrations and entity states to storage, which can then be restored after failures. For the aim of this thesis, DF entities are most relevant. They allow developers to store durable objects and the operations that can be executed against them. Entities are identified by a unique ID. DF orchestrations can either call operations on Entities and retrieve the result or signal an operation in a "fire-and-forget"-like way. Entities can also signal other entities, however, calls between entities are not supported in order to prevent deadlocks. However, DF do not allow parallel operations on the same entity. Each operation request is stored in a queue and is thus executed one by one.

Our work is different in various ways. First, we propose a solution that works for multiple serverless platforms and not only one platform, such as what is proposed with DF. Furthermore, because we use CRDTs, we allow the concurrent modification of the same data structure instead of storing those operations in a queue and executing them sequentially. Unlike DF, our middleware does not support arbitrary classes to work as MISO Stateful Object because it is a limitation of CRDTs. However, we support a broad set of data types that developers can use to maintain state.

3.2 New Approaches for Stateful Serverless Serverless Functions

Some authors have proposed new approaches for stateful serverless functions, which we describe in this Section. This includes entirely new serverless platforms that have the management of the state built in and entirely new paradigms for stateful serverless functions, different from regular FaaS.

Cloudburst Sreekanti *et al.* [19] proposed Cloudburst, a serverless platform written in Python that aims to solve the challenges of shared state while maintaining the advantages of serverless computing. To achieve this, Cloudburst uses an autoscaling key-value store for low-latency access to shared state combined with mutable caches located alongside function executors to keep frequently used data locally available [19]. MISO Stateful Objects can either be passed as remote references (which are resolved before function invocation using local caches), local objects, or dynamically via the key-value store's API. Updates to the shared state are propagated asynchronously to the storage [19].

Our work is different from Cloudburst in multiple ways. Cloudburst is a whole FaaS platform, while we present a middleware that extends existing serverless platforms. This means that developers have to learn the programming model of Cloudburst to develop, register, and execute serverless functions. Our solution aims to retain the workflows of existing serverless platforms in terms of the development of serverless functions. The programming model of Cloudburst includes operations to set and retrieve objects identified by a key. Our solution does not directly work in the same way that arbitrary classes can be used to store the state, but we provide CRDT-based data types that behave like a proxy. Unlike Cloudburst, our middleware does not support arbitrary classes as MISO Stateful Object because it is a limitation of CRDTs. However, we support a broad set of data types that developers can use to maintain state.

Object as a Service (OaaS) Lertpongrujikorn *et al.* [17] propose a new paradigm to manage application state in serverless functions called *Object as a Service (OaaS)*. In OaaS, objects are immutable entities that have functions defined. Each function may perform an operation on the state of the objects. However, a new object is instantiated by OaaS instead of modifying the existing one, which means no synchronization is required to modify an object. OaaS stores object metadata in a key-value database and caches those entries in memory, while the object state is persisted in remote object storage (e.g., S3). Developers have to use a REST API to declare new classes. For each class, developers must specify which states exist and where the state is stored (e.g., S3). The authors [17] mention that scalability is a challenge in their approach, especially with regard to concurrent access to objects. OaaS follows a similar direction to our thesis in that it integrates the application state with the serverless platform. Our work differs in various ways: i) we do not introduce an additional management layer where objects have to be declared externally to the serverless platform via a REST API, ii) we use data types built on CRDTs and therefore provide mutable instead of immutable objects, and

iii) contrary to our solution, OaaS requires deployment of various additional services (e.g., Apache Kafka, Zookeeper, ArangoDB, and Infinispan) next to the serverless platform itself.

Serverless Platforms for the Edge Puliafito *et al.* [8] classify serverless functions as either remote-state functions which typically run in the cloud (i.e., the state is stored on an external storage service and function containers are shared between applications) or local-state functions which typically run at the edge (i.e., the state is stored locally). However, in their proposal, containers executing local-state function code are dedicated to specific application instances/sessions. This means that all requests belonging to the same session are routed to the same local-state container.

Similarly, Baresi *et al.* [9] propose to enhance Serverless Edge Platforms with *Stateful Compute Services*. This works by initializing container instances (where functions are executed) with unique session tokens that are shared by users within the same session. Each request contains the token, and the request is thus delegated to a specific container instance so that the state does not have to be synchronized or replicated between instances. This also means that such platforms are limited to vertical scaling of the container instances, which is similar to what has been proposed by Puliafito *et al.* [8].

Our work is different from what has been proposed by [8] and [9] as in our solution, it is not required to route all traffic of an instance to the same container. Because we rely on CRDTs, every replica of a particular serverless function can modify the state simultaneously. This means that when a particular serverless function is scaled to a different node, the state can be modified locally at the container running on the node executing the corresponding replica of the serverless function.

Apache Flink Stateful Functions The Apache Flink Stateful Functions (*StateFun*) [42] project combines stream processing and serverless functions. It offers developers a runtime on top of Apache Flink that is based on the paradigm of serverless computing. Every function has a local persisted state, so whenever a function runs and performs some computation, it uses a local state in the form of local variables. There are different types of functions: embedded functions, co-located functions, and remote functions. Embedded functions run directly in the JVM. In co-located functions, a Flink *TaskManager* communicates with a function that is located close to it. This is often achieved by using Kubernetes, where a Flink container and the function side-car container are deployed as pods that are then able to communicate via a local pod network. Co-located functions cannot scale the state and compute parts independently of each other. Remote functions are physically separated from the Flink StateFun Cluster, so they can be deployed and scaled independently. Remote functions can invoke any remote endpoint via HTTP/gRPC, such as an AWS request gateway for AWS Lambda. Invocations contain all necessary state variables and messages, so the state is updated in the cluster after the response of the remote function has been received [18].

The Stateful Functions runtime enforces serial invocations per entity (e.g., a shopping cart for a specific person), so only one invocation per function instance can be running at the same time, so concurrent modifications of the same state are avoided [43]. The StateFun project, therefore, aims towards a different use case and technology than our work. Our middleware is aimed towards existing serverless platforms that are not based on a stream processor.

CHAPTER 4

MISO Middleware

4.1 Conceptual Model

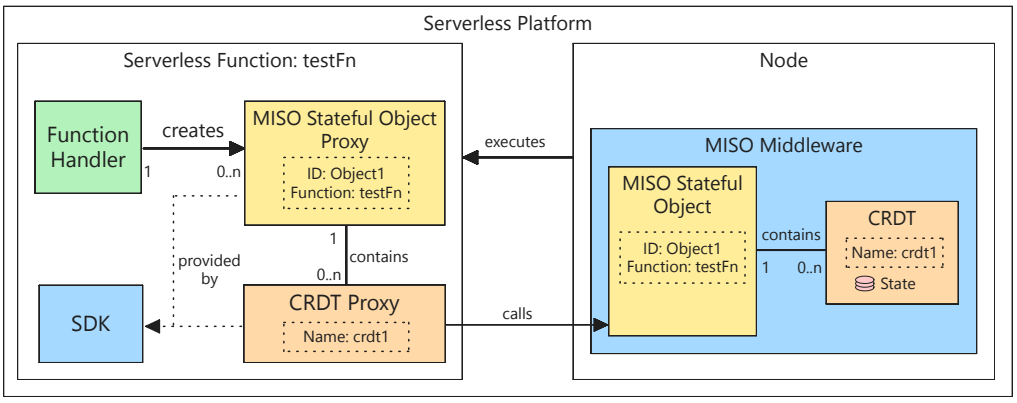

Figure 4.1: Conceptual Model of MISO Stateful Objects

Figure 4.1 depicts the conceptual model of the MISO middleware. It is based around *MISO Stateful Objects*, which are stateful objects that are accessed and modified from the code of serverless function handlers. One serverless function can create multiple such MISO Stateful Objects in the code of the serverless function handler. Every MISO Stateful Object is identified by an ID and contains zero, one, or more data types. The ID can be set manually or auto-generated from the function name. As MISO Stateful Objects are linked to a serverless function, there is no chance of conflicting IDs for different serverless functions, as the ID only needs to be unique for each serverless function. Every data type within a MISO Stateful Object has a name. A certain name can only exist once per object. However, the same data type can be present multiple times within a single MISO Stateful Object with different names. Using a simple name to identify the

data type contributes to the developer experience, as no ID needs to be memorized to access the individual data types.

Every MISO Stateful Object is bound to one particular serverless function, identified by its name. Multiple replicas of the same serverless function can share the same MISO Stateful Object, even if they are executed on different nodes. MISO Stateful Objects bundle multiple instances of CRDTs into a single object. This means they do not carry the state directly in the objects, but the state is contained in the individual data types. Each CRDT within a MISO Stateful Object is identified by a name.

The lifecycle of MISO Stateful Objects is managed by the middleware. This includes creating, retrieving, modifying, and replicating the data types within the objects and managing their state. For this reason, the SDK provides proxy objects of the CRDTs. The proxies can be retrieved from the MISO Stateful Object proxy object with the name of the data type. The CRDT proxies behave like regular data types but internally call the middleware whenever an operation is executed against them. The serverless function does not store the data of the CRDTs. It is only stored in memory on the middleware. The middleware ensures that the data of the MISO Stateful Objects is available on every node of the serverless platform that runs this serverless function. In Figure 4.1, it is visible that in this particular case, a single MISO Stateful Objects was created for the function *testFn* containing one CRDT with the name *crdt1*.

4.2 Middleware Architecture

Figure 4.2 provides an overview of MISO's architecture. The system is designed to be modular and flexible enough to be integrated into several different serverless platforms and is divided into two main components (blue): the *middleware* and a *SDK*. A major characteristic of the middleware is that it is distributed across multiple nodes. More precisely, the middleware runs on every node of the serverless platform that executes serverless functions. This is necessary so that the middleware and thus the data of the MISO Stateful Objects are located in close proximity to the serverless functions. The middleware is responsible for managing MISO Stateful Objects, which includes the management of their lifecycle and the states they contain. It also provides an API for serverless functions using which they can modify the MISO Stateful Objects. Due to the fact that serverless functions might run on different nodes due to load balancing, the middleware needs to replicate data between different nodes. For Kubernetes-based serverless platforms, a *DaemonSet* as introduced in Section 2.1.2 can be used to run the middleware on every node of the serverless platform in the form of a container. For other platforms, the middleware could be added to the deployment script of the serverless platform or similar to ensure that it runs on every node, either as a container or running on the host directly.

The middleware is combined with an SDK for serverless functions. It enables serverless functions to use the middleware and provides proxy versions of MISO Stateful Objects and the data types they contain. Developers can use the data types as if they are

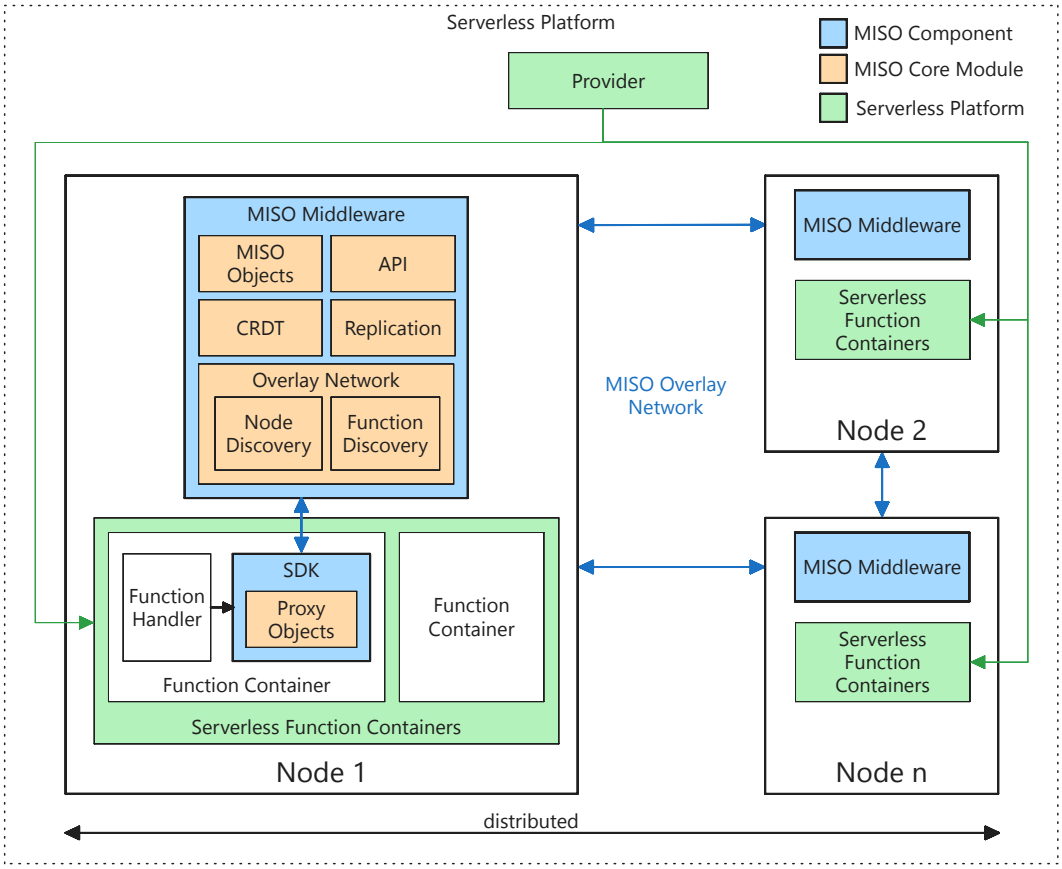

Figure 4.2: Architecture of the Middleware

regular local data types. However, in the background, the operations are delegated to the middleware over the network. This is transparent to the developers of the serverless functions. The state of the data types themselves are not stored in the proxy versions but only on the middleware instance that executes this particular serverless function. The details of this component are described below in Section 4.4. The SDK is dependent on the programming language used to write the serverless function, so there might be multiple such SDKs for various programming languages in the future.

All middleware instances are interconnected via an *Overlay Network*. This is depicted by the blue arrows in Figure 4.2. This network is mainly used for the replication of updates whenever a MISO Stateful Object is modified. To make the process of sending updates more efficient, the overlay network needs to provide certain information, such as the information on which nodes a particular function is currently being executed. The details of this core module are described below.

Figure 4.2 also shows how MISO integrates with existing serverless platforms. Certain components of the serverless platforms, such as the ingress controller, are deliberately

omitted from the figure as they are irrelevant to the middleware. The serverless platform provider manages the containers that run the serverless function containers. To integrate the middleware with the serverless platform, it is necessary to extend the provider in such a way that it provides certain environment variables to the containers of the serverless function. The list of required information includes the hostname and IP address of the node that executes the function so that the SDK can communicate and register with the middleware correctly. In case the serverless platform already provides such information, the provider does not have to be modified to integrate the middleware. To integrate the SDK, it has to be added to the dependencies of the serverless function. We show a practical integration of the middleware with OpenFaaS, an existing serverless platform, in Section 6.3.1.

4.3 Middleware Core Modules

The middleware contains multiple core modules. They are illustrated in Figure 4.2 inside the middleware (orange). The following paragraphs will explain the details of the middleware's core modules.

MISO Stateful Objects Module Covers the functionalities related to the creation, retrieval, and deletion of MISO Stateful Objects. They are bound to one particular serverless function, denoted by its name. Every serverless function can also access multiple such objects. For every mentioned operation (create, retrieve, delete), the object's identifier and the serverless function's name need to be supplied.

CRDT Module Provides an implementation of various CRDTs that can be used in MISO Stateful Objects. The middleware uses state-based CRDTs, as they offer a more resilient and straightforward approach for reconciling data in case of missed updates than operation-based CRDTs. This is because they propagate their whole state in every update, mitigating potential complexities that could arise due to partial synchronization. Furthermore, they do not require a causally reliant delivery of updates, which would be challenging for edge devices. The data types are implemented like a library so that they can be developed independently of other core modules of the middleware. This library is then used in the middleware, but other applications could also use the implemented CRDTs for use cases unrelated to ours.

API Module Exposes several core functionalities of the middleware via an API exposed over the network. This includes the following functionalities:

- Create, retrieve, and remove MISO Stateful Objects
- Add and remove CRDTs to MISO Stateful Objects
- Retrieve the states of CRDTs (e.g., value of a counter)

- Modify CRDTs (e.g., increase a counter)

Serverless functions can invoke those functionalities via the SDK.

Overlay Network Module All instances that run MISO create an internal overlay network. It is required for the replication module and has two main tasks. The first one is to discover other nodes that run the middleware, and the second is to discover which replicas of a serverless function are executed on a particular node. This information is necessary so the Replication Module can efficiently replicate MISO Stateful Objects to nodes that also run replicas of the same serverless function. This increases the efficiency of the replication process, saving bandwidth and processing power compared to replicating updates to all discovered nodes. This effectively allows the middleware to only send updates to nodes that are relevant to them.

Algorithm 4.1: Node Discovery Service

```
1  discoveredNodes = new Map()
2  startDiscovery()
3  while discovery is running do
4      for n: discovered node name do
5          if n ∉ discoveredNodes then
6              details = retrieveNodeDetails(n)
7              discoveredNodes.put(n, details)
8              startHeartbeatTimer(n)
9              startTimeoutTimer(n)
10     end
11     for every expired heartbeat timer do
12         response = sendHeartbeat(n)
13         clearHeartbeatTimer(n)
14         if ∃ response then
15             clearTimeoutTimer(n)
16         end
17     end
18     for every expired timeout timer do
19         discoveredNodes.remove(n)
20         clearAllTimers(n)
21     end
22 end
```

The *node discovery* service is responsible for discovering other nodes that run the MISO middleware. It can work with multiple different mechanisms to discover nodes as long as they adhere to the required interface. Some strategies might only discover nodes that are directly reachable over the network (e.g., mDNS). Other strategies might solve this task

differently. This is subject to the implementation and discussed in Section 5.1, where we discuss how we use mDNS for this purpose. The algorithm of the node discovery service is depicted in Algorithm 4.1. Before the discovery is started, every node instantiates a `Map` that stores node names (i.e., hostnames) alongside the `Set` of IP addresses this node has. Whenever a new node is discovered, the details about this node are added to the map. This process is performed regularly so that new middleware instances are identified. A heartbeat mechanism removes unresponsive nodes from the list of discovered nodes when no connection can be established over a longer time, which is realized using two timers. The heartbeat timer is responsible for triggering a heartbeat request to the other node, which is cleared whenever a response to the heartbeat is received. The timeout timer fires if no heartbeat response has been received in a long time and triggers the removal of the node from the list of discovered nodes so that the replication module does not unnecessarily try to send the updates to this node.

The function discovery service is responsible for providing the information on which serverless functions are executed by the previously identified nodes. To achieve this, serverless functions explicitly need to register and unregister with the overlay network. This can be done via the SDK, which is introduced in Section 4.4, and if our solution is integrated, this is transparent to developers of serverless functions, as discussed in Section 6.3.

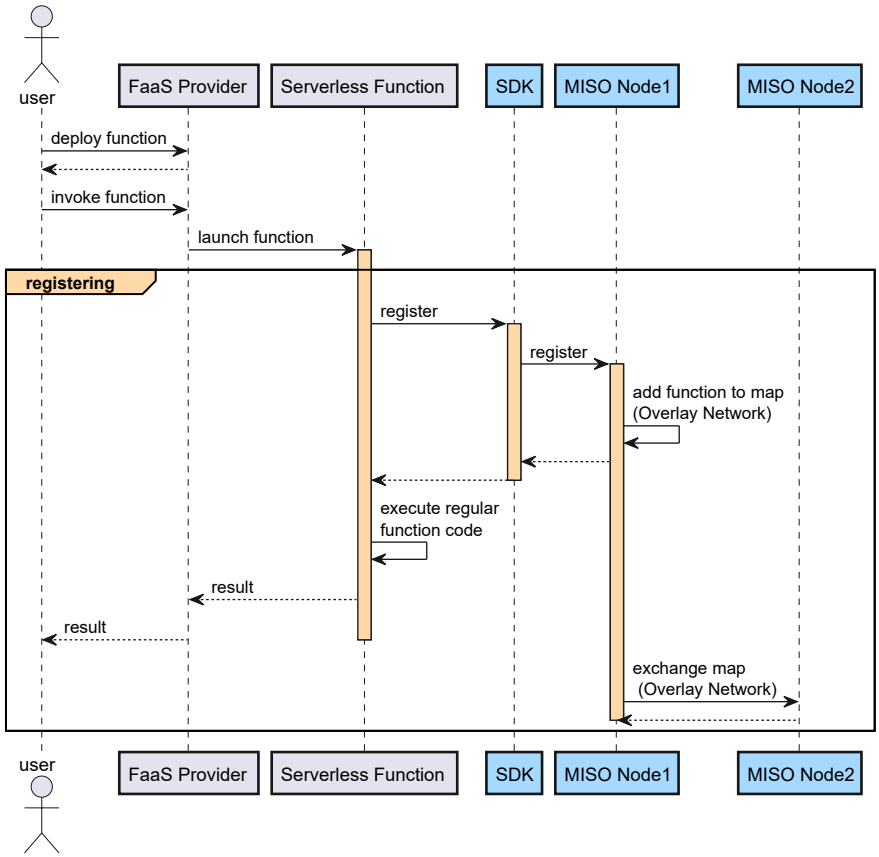

Figure 4.3: Function Discovery Service Registering

Algorithm 4.2 describes how the function discovery process works. It is visible that every node stores a `Map` in the function discovery service that stores the details about every node that a particular serverless function runs. This is because every serverless function can be scaled to multiple nodes that execute them. Whenever a function has (un-)registered, the updated map is sent to other middleware instances via the aforementioned API module. The sequence diagram in Figure 4.3 depicts a simplified version of the algorithm. It can be seen that whenever a serverless function is invoked, it first registers with the function discovery service of the overlay network. It then adds the function to its local data and informs other nodes about the new function asynchronously. The regular code of the serverless function is executed after the function has registered.

Replication Module Disseminates modifications to CRDTs to all nodes executing the relevant serverless functions. It employs a debounced/delayed transmission of updates to other nodes, subject to a configurable time interval. This assures that multiple updates in a short time are batched and replicated in a single update only.

The pseudocode for the replication algorithm is visible in Algorithm 4.3. Whenever the

Algorithm 4.2: Function Discovery Service

```
   input  : nds // NodeDiscoveryService from Overlay Network
 1 functionNodeMap ← new Map()
 2 startDiscovery()
 3 while discovery is running do
 4     for fnInfo: serverless function that registers do
 5         nodes = nds.getDiscoveredNodes(fnInfo.name)
 6         nodesRunningFn = functionNodeMap.get(fnInfo.name)
 7         if fnInfo.node ∉ nodesRunningFn then
 8             nodesRunningFn.add(fnInfo.node)
 9             functionNodeMap.put(fnInfo.name, nodesRunningFn)
10             for node in nodes do
11                 send(functionNodeMap, n)
12             end
13     end
14     for fnInfo: serverless function that unregisters do
15         nodes = nds.getDiscoveredNodes(fnInfo.name)
16         nodesRunningFn = functionNodeMap.get(fnInfo.name)
17         if fnInfo.node ∈ nodesRunningFn then
18             nodesRunningFn = nodesRunningFn \ fnInfo.node
19             functionNodeMap.put(fnInfo.name, nodesRunningFn)
20             for node in nodes do
21                 send(functionNodeMap, n)
22             end
23     end
24 end
```

state of a CRDT within a MISO Stateful Object is modified by calling the corresponding API endpoints, this data eventually needs to be replicated. To achieve this, an update is generated for every such request, which is then wrapped in a *ReplicationTask*. This task is then queued in the *ReplicationService* and replicated asynchronously after a configurable delay. The most important part of the algorithm is the batched/debounced transmission of replication tasks. This is because the replication might be triggered continuously for the same CRDT in frequent intervals, depending on how often the API is called. The replication module must ensure that exactly one replication happens in the configured replication interval. In our algorithm, we always use the latest of all queued tasks for a given CRDT within a MISO Stateful Object. The reason why replicating the latest task of a CRDT is sufficient is that MISO uses state-based CRDTs, where every state update carries the whole state. Subsequent tasks, therefore, already contain the state of previous tasks. Therefore, they can be ignored in the replication process. After the latest task has been retrieved, the aforementioned overlay network is utilized to determine which nodes need the data. The function discovery service provides the information which

other discovered nodes run the same serverless function. The replication module does not replicate the update to nodes that currently do not run any replica of this serverless function, avoiding unnecessary network requests and processing power. All nodes that need the update then receive the update via a stream. In case the network transmission fails, the request is retried on the network level. In case a node does not successfully receive an update, it will eventually receive one of the next updates, given that the node is back online. Whenever the target node receives an update, it sets the received state to the local CRDT by merging it with an empty instance of the CRDT.

Every update that is disseminated contains the following data:

1. Information Regarding the MISO Stateful Object and affected data type (ID and CRDT name)
2. Information regarding the serverless function that is affected (serverless function name)
3. The whole state of the source CRDT after it has been modified. This field's data depends on the particular CRDT that is transmitted.

The updates are sent over the network in a stream. Streaming mitigates the need for constant re-openings of network connections and is especially useful for frequent replication intervals to reduce the replication time. The network connection is specific to another node and is shared between replication calls of different CRDTs. In case there is an error on the stream (e.g., node disconnects), the stream is closed. Whenever the next update arrives, the stream is then re-opened to this node unless it is no longer present in the overlay network. The merging of the states itself is implemented in the CRDT data types. Every state-based CRDT has a `merge` method that has one argument, which is another instance of the same CRDT. The replication algorithm described in Algorithm 4.3, therefore, has no information about the actual merge logic but only needs to make sure that the correct method is called with the right data.

CRDT State Restoring

Another important functionality is restoring states from other middleware instances. This is important when one of the nodes restarts, or the serverless function is scaled to nodes that did not previously execute this particular serverless function. The restoring of state works on a CRDT level. This means that whenever the middleware receives a request to modify a certain CRDT within a MISO Stateful Object, it is checked if the data is present locally. If the data is present locally, it is returned immediately, otherwise the restore process is started.

Algorithm 4.4 describes the price of restoring state on a CRDT level. This is also depicted in the sequence diagram in Figure 4.4. The process starts when the user invokes a serverless function. When an operation on the CRDT is called the code of the serverless

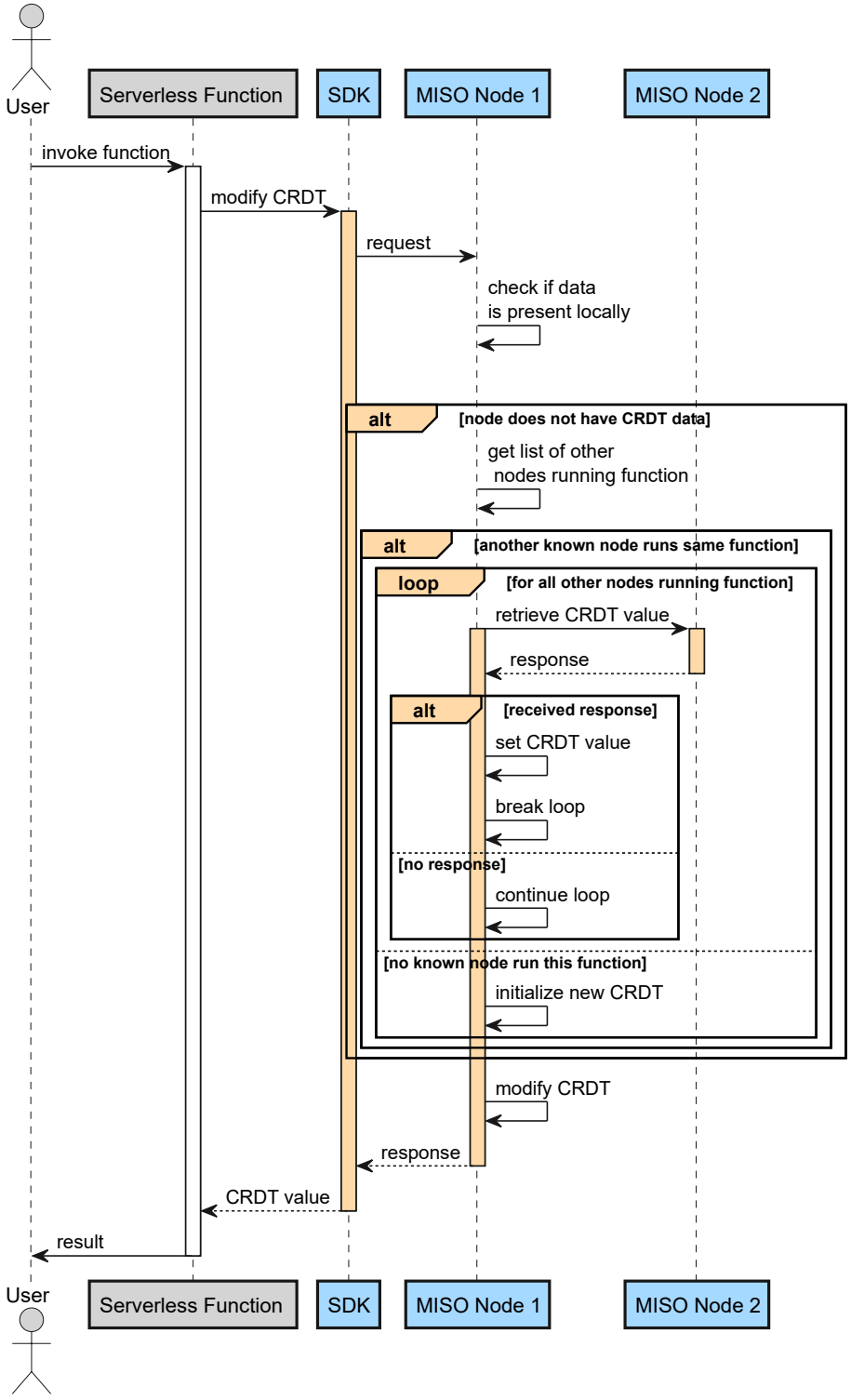

Figure 4.4: CRDT State Restoring Sequence Diagram

Algorithm 4.3: Replication Algorithm

```
input : fds // FunctionDiscoveryService from Overlay Network
input : object // MISO Stateful Object
input : crdt // CRDT that is modified
input : fnName // serverless function name
1  crdtTasks = new Observable(object.id, crdt.name)
2  while state of crdt is modified do
3      update = crdt.createUpdateMessage()
4      task = new ReplicationTask(crdt.name, object.id, fnName, update)
5      crdtTasks.next(task)
6      debounce replication tasks in crdtTasks
7      targetNodes = fds.getReplicationTargets(fnName)
8      task = crdtTasks.getLatestTask()
       // on source node
9      for node in targetNodes do
10         stream = getStream(node)
11         result = stream.sendUpdate(task)
12         if result.status == FAILED then
13             retry sending
14         end
           // on target node
15         object = getStatefulObject(task)
16         crdt = object.getCrdt(task.crdtName)
17         crdt.merge(task.update)
18     end
19 end
```

function, this operation is delegated to the middleware node that runs this particular serverless function. The middleware then checks if the corresponding CRDT with the requested name is present in a MISO Stateful Object with the given identifier is present on this node. If this is true, the requested operation is executed, and the response is returned to the serverless function via the proxies of the SDK. The process of how the SDK works is described in Section 4.4. If the data is not present locally, the aforementioned function discovery service provides a list of nodes running the same serverless function. If there are such known nodes, then every one of them is contacted via the network to send the current state of the CRDT. If there was a response containing a state, it is then to the local CRDT. Otherwise, the next node in the list is contacted. The overlay network does not know if the other nodes actually have the state or which versions they might have. Therefore, the process runs in a loop over all known nodes that run the same serverless function. The requested CRDT is automatically initialized with a default value if the state cannot be retrieved from other nodes. The whole initialization process is

transparent to developers of serverless functions when they modify MISO Stateful Object.

Algorithm 4.4: CRDT State Restoring

```
   input  : fds // FunctionDiscoveryService from Overlay Network
   input  : object // MISO Stateful Object
   input  : crdtType // Type of CRDT that is modified
   input  : crdtName // Name of CRDT that is modified
   input  : fnName // serverless function name
 1 for r: request modifying CRDT do
 2     intercept r
 3     if crdtName ∉ object then
 4         targetNodes = fds.getReplicationTargets(fnName)
 5         for node in targetNodes do
 6             stream = getStream(node)
 7             payload = getPayload(crdtType, crdtName, object.id, fnName)
 8             result = stream.retrieveCrdt(payload)
 9             if ∄ result then
10                 continue
11             end
12             crdt = new CRDT(object.id, crdtName)
13             crdt.merge(result)
14             object.addCrdt(crdtName, crdt)
15             break
16         end
17         continue regular execution of r
18     end
19 end
```

4.4 SDK for Serverless Functions

The *SDK* facilitates the communication between serverless functions and the middleware. It serves as an intermediary layer that invokes the API endpoints of the middleware over the network, abstracting away the complexity of connection management from developers of serverless functions. This architectural separation between middleware and the proxies contributes to the maintainability of both the middleware and the code of serverless functions, as the SDK can be developed independently of the serverless function code. The SDK is created for a particular programming language, so there can be multiple such SDKs for different programming languages in the future. The SDK is not dependent on a serverless platform and can be used in multiple such platforms that support a runtime for serverless functions that the SDK is written in.

Table 4.1: SDK Programming Abstractions

Abstraction	Description
`StatefulObjectProxy`	Main abstraction that reflects a proxy-based version to interact with MISO Stateful Objects from the code of serverless functions.
`CRDT Proxies`	Proxy objects that behave like regular data types but delegate the operations to the middleware, which is transparent for developers. The abstraction for each proxy contains the same set of methods as the underlying CRDT.

4.4.1 API and Programming Model

Table 4.1 summarizes the key abstractions that are exposed to developers of serverless functions, which are `StatefulObjectProxy` and `CRDT Proxies`

`StatefulObjectProxy` is the most important programming abstraction. This is the central component that is exposed to developers. They can instantiate them in the code of serverless functions. Instances of the *StatefulObjectProxy* are directly linked to a MISO Stateful Objects (introduced in Section 4.1) and are bound to a particular serverless function. They represent a proxy version of one specific MISO Stateful Object and the CRDT data types it includes. The proxy objects delegate all operations to the middleware, where the actual state and lifecycle of MISO Stateful Objects is managed. The SDK is responsible for configuring the proxy objects correctly to communicate with the middleware.

The `StatefulObjectProxy` is responsible for the following tasks:

1. Abstracting away the complexity of communicating with the middleware so developers of serverless functions do not need to perform such steps.
2. Providing mechanisms to register and unregister serverless functions. This is necessary so the overlay network can properly provide the information required in the replication process.
3. Providing instances of proxies of CRDT-based data types. They behave like regular data types but proxy the operations to the middleware and do not locally save the state.

The API of the `StatefulObjectProxy` is summarized in Table 4.2. It can be seen that the list of methods is rather compact. There are methods to register and unregister the serverless function instance. This is necessary so that the overlay network of the middleware can provide the necessary details to the replication module. Furthermore,

Table 4.2: StatefulObjectProxy API

Method	Description
`registerServerlessFunction()`	Registers the current serverless function replica with the middleware.
`unregisterServerlessFunction()`	Removes the current serverless function replica from the middleware.
`get<CRDT>(name)`	Returns a proxy instance of a given CRDT identified by its name, where `<CRDT>` must be a supported data type. This list of currently supported data types includes `EWFlag`, `GCounter`, `PNCounter`, `GSet`, `MVRegister`, `ORSet`.
`deleteCrdt(name)`	Removes a data type with a given name from the MISO Stateful Object.

the API exposes methods to get or delete certain data types from the MISO Stateful Object.

`CRDT Proxies` provide a proxy instance of a given CRDT. The functionality that is exposed to developers closely mirrors the underlying CRDT, but internally the operations are delegated to the middleware. The proxies can purposely not be instantiated without the `StatefulObjectProxy`, as otherwise, developers would have to manually configure the details on how these data types can communicate with the middleware. We discuss the list of supported operations in the implementation details of the CRDTs in Section 5.3. The SDK does not modify the interface of the data types to make it understandable for developers familiar with CRDTs. It needs to be noted that due to the nature of CRDTs, there is no one-to-one mapping of existing familiar data types of programming languages (List, Array, ...) to a corresponding CRDT. For CRDTs that are closely related to familiar data types such as Sets, the operations reflect the ones of the regular data types as closely as possible. For example, a EWFlag is similar to a *Boolean* data type, and a ORSet behaves similarly to a regular `Set`. However, a exact match to familiar data types is not always possible. For example, a Grow-only Sets (GSets) is a `Set` that does not allow the removal of elements, so such an operation can not be provided by the SDK, and it is not possible to use the interface of the programming language. Similarly, a MVRegister is a data structure that typically does not have a direct counterpart in all programming languages. This implies that developers that use the SDK need to learn new data types, but we are confident that future research will introduce further CRDT-based data types and that the awareness of CRDTs among developers will rise.

4.4.2 Usage in Serverless Functions

In this Section, we describe an example of how we expect the SDK to be used in the code of serverless functions.

```
1 // function handler of the serverless function (OpenFaaS)
2 module.exports = async (event, context) => {
3     const so = context.statefulObject;
4     const counter = so.getGCounter('countExecutions');
5     await counter.add(1);
6     return {
7         data: await counter.getValue() // value is 1
8     };
9 };
```

Listing 1: Using the SDK in Serverless Functions

The SDK can be used by creating a new instance of the MISO Stateful Object proxy. This instance can then be used to create one or multiple CRDTs. Listing 1 gives an example of creating and using a GCounter proxy from a MISO Stateful Object from the code of a serverless function for OpenFaaS which is written in TypeScript. It can be seen that, in this case, a MISO Stateful Object is injected into the *context* argument of the serverless function, which developers can directly use. This is possible when our work is integrated into the serverless platform. The code of the function uses the default MISO Stateful Object to create a GCounter, then increments the counter by 1 and returns the current value. We describe the necessary steps to integrate the SDK, including the usage of the MISO Stateful Object to register and unregister from the middleware in Section 6.3.1. We expect that the usage of the SDK is similar in other programming languages and open-source serverless platforms.

CHAPTER 5

Prototype Implementation

The implementation of MISO is open-source and available online. The link can be found in Appendix A. The source code includes the code for the middleware, SDK, and evaluation. The repository also contains scripts to easily deploy and run MISO in a Kubernetes in Docker (KinD) cluster.

The prototype implementation of the middleware runs under Node.js 18 LTS, utilizing TypeScript for enhanced type safety. It employs npm as a package manager, facilitating streamlined dependency management for the middleware. Additionally, all middleware packages are located in the same repository. The advantages of monolithic repositories include code reuse and simpler dependency management [44]. This organizational approach also promotes modularity and eases maintenance across the various components of the middleware. We use npm workspaces [45] to accomplish this, which is a collection of functionalities within the npm CLI that supports the management of multiple packages from the root package. It simplifies the development, as the local packages are seamlessly linked together when installing dependencies. This configuration can be found in the `package.json` file in the repository's root folder.

An illustration of the prototype package hierarchy can be seen in Figure 5.1. There are four packages in total: the middleware package, the CRDT package, the SDK package, and a common package. The common package is used by the SDK and middleware, and the CRDT package is used by the middleware. During development, we self-hosted an instance of a private package registry called Verdaccio[1] where the built files were published. This is necessary as we build MISO as a Docker container and run it inside Kubernetes for our evaluation in Chapter 6. Manually linking all packages/dependencies would be very cumbersome. MISO is currently not available on the public npm registry.

[1] https://verdaccio.org/

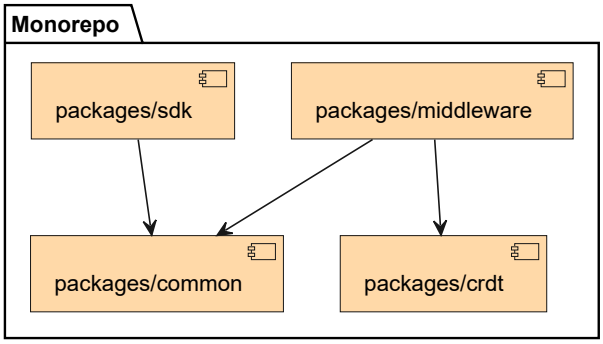

Figure 5.1: Monolithic Repository Package Structure

Table 5.1: Middleware Folders and Descriptions

Folder Name	Description
controllers	API endpoints to modify MISO Stateful Objects and CRDTs
interceptor	Request interceptors, e.g., for logging and CRDT initialization
objects	Code regarding MISO Stateful Object creation and modification
overlay-network	Code for the overlay network with node and function discovery
replication	Code to replicate the state of MISO Stateful Objects to other instances of MISO
utils	Helper code

5.1 Middleware Package

The *@miso/middleware* package contains the code for the middleware. The link can be found in Appendix Section A.1.1. It is developed using the open-source NestJS [46] framework due to its robust set of features, including excellent support for a modular architecture, powerful dependency injection, and built-in support for gRPC. These attributes make NestJS an ideal choice, ensuring maintainability and efficient development for our middleware. The middleware package is structured as described in Table 5.1:

5.1.1 API

The prototype application exposes the operations to modify MISO Stateful Objects and CRDTs via gRPC. The definition of the API can be found in the protobuf files of the common package, as discussed in Section 5.4. The middleware uses the protobuf files to start a gRPC server on port 5001. There is one gRPC service for each CRDT.

Due to the complexity involved in sending arbitrary data types over gRPC and limited reflection support in TypeScript, as types are erased during runtime, we have limited the possible value types for all Sets and Registers to string, number, and JavaScript objects in this prototype.

5.1.2 Overlay Network

Our implementation is designed to allow multiple implementations of function and node discovery. To enable this, we have defined interfaces for node and function discovery so that different strategies can implement the interface and provide the functionality. The overlay network mechanism allows switching between node and function discovery types. We currently provide a single implementation for node discovery (mDNS) and function discovery (gRPC).

Node Discovery Strategies

Figure 5.2 shows how the *mDNS* discovery strategy works. When the middleware starts, the discovery process starts. It then repeatedly sends mDNS queries to a certain *service name* (default: `miso-middleware -instance.local`, but this is configurable). Other middleware instances then respond to this query with their local IP addresses via the network. For the mDNS responses, a DNS reverse lookup is performed to find the hostnames of the IP addresses so that every node stores a map of IP addresses for every hostname. This process is performed regularly to identify new middleware nodes continuously. The interval in which mDNS queries are sent is configurable and defaults to a random time between 1 and 2 minutes. This ensures that the mDNS queries do not all happen simultaneously in a cluster with multiple MISO instances.

Function Discovery Strategies

When using the *gRPC* function discovery strategy, every serverless function registers itself via gRPC at the node where the container of the serverless function runs every time the function handler is executed. The middleware exposes the corresponding API endpoints to register and unregister serverless function instances via gRPC. Those endpoints must then be manually called by the serverless functions or by calling the respective method in our SDK. When the template of the serverless function is adapted to integrate the MISO SDK, the process of registering and unregistering is completely transparent to the developers of serverless functions. We demonstrate this in our evaluation of the SDK in Section 6.3.

Whenever a function registers or unregisters, an update message is sent to other discovered nodes that run MISO. This is necessary so that the replication module knows which nodes run which serverless functions, avoiding unnecessary replication traffic. This is implemented as previously described in Section 4.3.

For Kubernetes-based serverless platforms, it would also be possible to build node- and function discovery strategies that use the Kubernetes API to identify which serverless

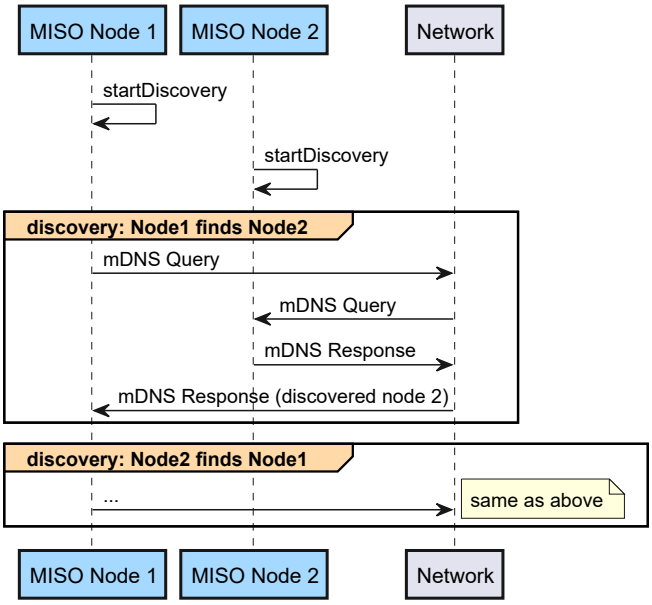

Figure 5.2: mDNS Node Discovery

function pods are running on which nodes. While technically possible, this adds a dependence on a centralized authority which we generally want to avoid in this work. However, we still designed the implementation to be flexible and, therefore, do not force developers to stick with our mDNS/gRPC-based solutions; instead, our software is designed so that multiple different solutions for node- and function discovery can be implemented in the future.

5.1.3 Replication

Replication is a core mechanism of the middleware and, therefore, makes up a significant part of the software prototype. Figure 5.3 gives an overview of the replication process. The client (e.g., the SDK) sends the request to the middleware, which passes through the controller to the service handling the logic (Steps 1 and 2). The service triggers the replication task whenever a state is modified (Step 3). The client receives the response (Steps 4 and 5), and the replication process starts asynchronously (Step 6).

The replication service internally uses the RxJS library [47] to implement the Algorithm 4.3 as specified in Section 4.2. It is the official implementation of the "Reactive Extensions" (ReactiveX) specification for JavaScript [48]. The core of ReactiveX is an `Observable`, which can be used to develop asynchronous event-driven software that composes multiple sequences of data/events together [49]. ReactiveX is a combination of the *Observer* and *Iterator* patterns as well as functional programming [50].

It is possible to *subscribe* to observables to get notified whenever an event is emitted on the source observable by calling a callback function. Observables can be transformed

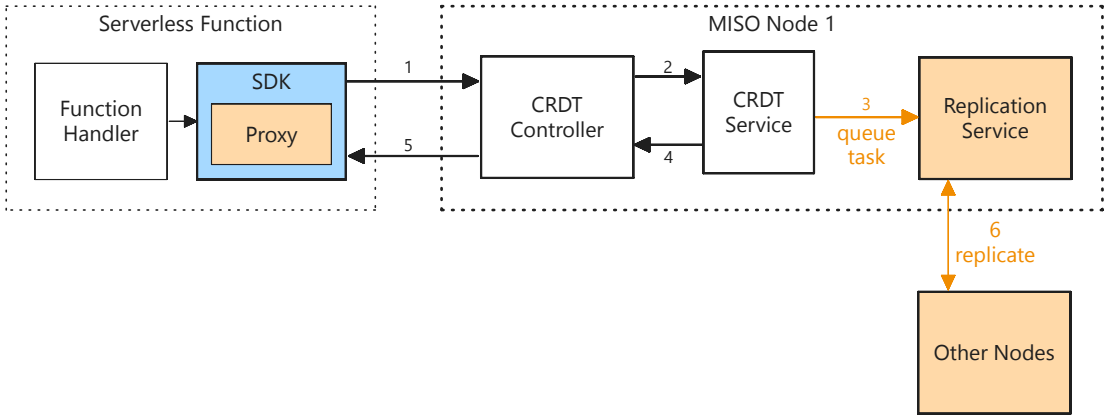

Figure 5.3: High-Level Overview of Replication Process

and composed together with other observables using a set of operators [48] that follow a functional programming style [50]. Observables can handle the emission of single or multiple values as well as infinite streams [49].

The ReactiveX specification is a great fit for MISO's replication service. A stream of replication tasks emits tasks as long as the CRDT is modified. The stream of incoming events must be transformed to ensure that they are debounced and handled at fixed intervals.

Figure 5.4 shows how the RxJS `auditTime` operator works. When a source value is emitted on the source observable (e.g., a), a timer is enabled, and further emissions on the observable are ignored. The timer becomes disabled at a configurable delay (i.e., the replication interval). In this example, the delay is set to 50ms. After this time, the most recent source value (e.g., y) is emitted on the output observable [51]. This process is then repeated. When no events are emitted on the source observable (i.e., no replication tasks are required), no events are emitted on the output observable.

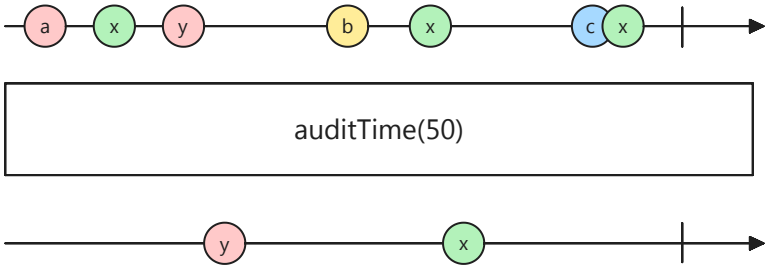

Figure 5.4: RxJS auditTime Operator (own work based on [51])

We utilize the `auditTime` operator to filter incoming events in the replication service. This means that only the latest replication task is handled, which makes sense as we

use state-based CRDTs where every update contains the whole state. This effectively limits the replication process for each particular CRDT to at most once in the configured replication interval. There is a unique observable for every combination of MISO Stateful Object ID and CRDT name.

5.1.4 State Recovery

The middleware supports state recovery when nodes restart, or functions are scaled to nodes that have not yet run this precise serverless function. This works by intercepting every request of the API that retrieves or modifies a CRDT state. Every CRDT has its own interceptor, located in the `controller` folder of the middleware.

For each request, it is checked if the stateful object and CRDT are found locally. If the stateful object and/or CRDT were not found locally, it is checked if the function discovery service from the overlay network knows which other nodes run this function. For each of those nodes, the state is then requested, and the state of the first reply is saved locally. This is transparent to developers of serverless functions. The original request is only answered after the initialization of CRDT and MISO Stateful Object.

This is demonstrated in Figure 5.5. It is visible that each CRDT controller is preceded by a CRDT initializer. After trying to initialize the current value, the request is forwarded to the controller. The client, e.g. the SDK, does not notice the initialization process.

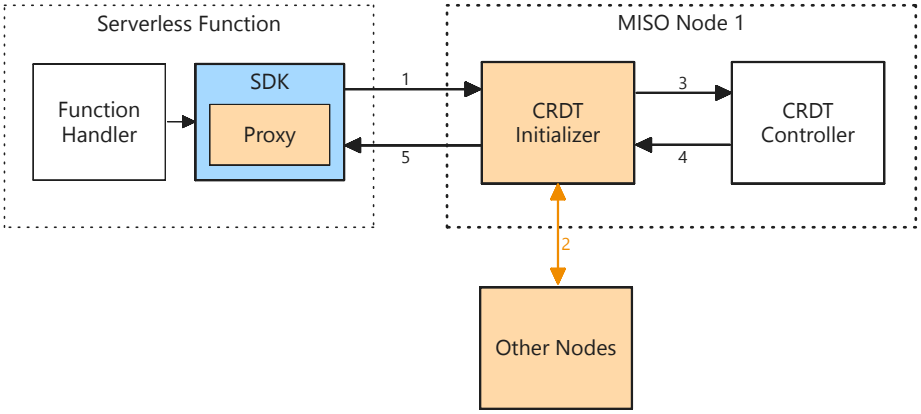

Figure 5.5: CRDT Initializer Interceptor

5.1.5 Configuration

The list of configurable environment variables for the middleware includes:

- **MISO_NODE_NAME (required)**:
  The (host-) name of this MISO instance.

- **MISO_MIDDLEWARE_REPLICATION_DELAY_MS (optional):**
  The interval in which CRDT data is replicated. Defaults to 200ms.
- **MISO_MIDDLEWARE_DISCOVERY_MDNS_SVC_NAME (optional):**
  The service name used for mDNS node discovery. Defaults to `miso-middleware-instance.local`.
- **NODE_ENV (optional):**
  Enables development logging if set to *development*.

5.1.6 Integration into Serverless Platforms

We demonstrate the seamless integration of our middleware and propose an innovative approach to incorporating MISO Stateful Objects into Kubernetes-based serverless functions through the inclusion of our middleware within the Helm chart deployment of serverless platforms. This underscores the ease of the integration and successfully bridges the gap between middleware and serverless functions.

We have developed a dedicated template file in the `yaml` format that can be included in the Helm chart of Kubernetes-based serverless platforms. The link can be found in Appendix Section A.2.2. This enables our middleware to operate in tandem with the platform as if it were originally part of the core architecture. In our Helm chart, we have defined that our middleware runs as a Kubernetes `DaemonSet`, i.e., one middleware instance runs on every node in the cluster. Additionally, environment variables to configure the middleware are set, e.g., the name of the node. This innovative approach simplifies the integration process of MISO Stateful Objects into serverless platforms in the edge-cloud continuum.

5.2 SDK Package

The `@miso/sdk` package implements the SDK for serverless functions and is also written for Node.js in TypeScript, similar to the middleware. The link to the code can be found in Appendix Section A.1.2. The idea behind the package is that developers can add it as a dependency to their Node.js project to interact with MISO Stateful Objects.

The most important exported class of the package is `StatefulObjectProxy`, which allows developers to create proxy-based MISO Stateful Objects. Developers do not need to know how the SDK internally handles the communication with the middleware. By calling the respective method on the proxy objects, developers can create or delete CRDTs. The `StatefulObjectProxy` class handles the gRPC connection to the middleware. To this end, the SDK creates a gRPC channel to the node that runs this serverless function. This channel is then used to register and unregister the serverless function with the overlay network and by the proxy versions of the CRDTs. To accomplish this, the SDK package imports the `@miso/common` package (described in Section 5.4) so that the shared protobuf definitions can be used.

Table 5.2: SDK Environment Variables

Environment Variable	Required	Description
MISO_HOST_IP	Yes	IP address of the middleware instance running this function
MISO_FUNCTION_NAME	Yes	Name of the serverless function that calls the SDK.
MISO_NODE_NAME	Yes	Hostname of the node that runs this serverless function.
MISO_HOST_PORT	No	Port of the middleware instance running this function. Defaults to 5001.
MISO_REPLICA_ID	No	Replica ID to be used when modifying CRDTs. Defaults to the hostname of the container running this function.
NODE_ENV	No	Changes the amount of data that is logged. Defaults to 'development'.

The SDK automatically generates an ID of the MISO Stateful Object if no one is provided by hashing the serverless function name with the SHA256 hash function. This means there is always a default MISO Stateful Object for every serverless function if no name is provided. In subsequent invocations of the same serverless function, the same MISO Stateful Object will be used as the name will be the same. This is convenient for developers that do not require multiple different MISO Stateful Objects. In case more such objects are required, developers can create more instances but must provide an ID for the MISO Stateful Object that needs to be unique for this particular serverless function. This could, for example, be a customer ID or similar.

Configuration The MISO Stateful Object proxies require a few configuration values to be able to communicate with the middleware. The SDK tries to load the configuration values from environment variables. Alternatively, it is also possible to manually pass a configuration object to the constructor. Table 5.2 shows which environment variables can be set and which of them are required. We propose that these variables be set as part of the function deployment. When the serverless platform uses Kubernetes for container orchestration, the environment variables can be set automatically when utilizing the Kubernetes Downward API [52].

Due to the fact that the operations against CRDTs run via the middleware using gRPC, the methods are asynchronous and return a `Promise`.

5.3 CRDT Package

The `@miso/crdt` package provides TypeScript-based implementations of various state-based CRDTs, including

1. **Counters** (GCounter, PNCounter),
2. **Flags** (EnableWinsFlag),
3. **Registers** (MVRegister), and
4. **Sets** (GSet, ORSet).

The link can be found in Appendix Section A.1.3.

In the following paragraphs, we will outline the details about each implemented CRDT, including its operations and implementation details.

5.3.1 Base Classes

All implemented CRDTs extend one of the package's multiple abstract base classes, described in the paragraphs below.

VectorClock Our VectorClock internally uses a map with string-based keys and numeric values. When the clock is incremented, the map value with the given string-based replica ID is incremented. The clock can also be merged with another clock, where the maximum version of all replica IDs of the given two vector clocks is taken as the final value. Furthermore, it is possible to compare two vector clocks to find out whether they are equal, whether one of them has a newer version, or whether there was a concurrent modification.

StateBasedCRDT This is an abstract class that defines that the constructor of each data type requires a stateful object ID and a CRDT name and that there must exist a `merge` method that takes another state-based CRDT as input. This class is extended by the `GCounter`, `PNCounter`, and `GSet` data types.

CausalCRDT This base class combines the `StateBasedCRDT` class with a vector clock. This class is extended by the `EWFlag`, `MVRegister`, and `ORSet` CRDTs.

5.3.2 CRDT Implementation Details

GCounter Our GCounter offers the following methods:

- State modifications: `add()`

- Queries: `getValue()`

Our implementation works by maintaining a map with string-based keys and numeric values. When the counter is incremented, the entry in the map with the corresponding replica ID is incremented. The map is then reduced to a single numeric number to get the counter value.

PNCounter Our PNCounter offers the following operations:

- State modifications: `add(), subtract()`
- Queries: `getValue()`

We re-use the GCounter CRDT mentioned above to accomplish the behavior of a PNCounter. More precisely, we use a GCounter for the additions and one for the subtractions. To get the value of the counter, the sum of the negative counter is subtracted from the sum of the positive counter.

EWFlag Our EWFlag offers the following operations:

- State modifications: `assign(value)`
- Queries: `getValue()`

Our `EWFlag` makes use of a vector clock. The vector clock is incremented whenever a boolean value is assigned to the flag. When two such flags are merged, there can be two cases. If one of the vector clocks is newer than the other, we take the flag value of the newer vector clock. The flag is set to true when there is a concurrent modification, and one of the flags sets the value to true. If both flags concurrently set the value to false, then this is the new value of the flag.

Other developers can easily build a similar flag where disabling has precedence.

MVRegister Our MVRegister offers the following operations:

- State modifications: `assign(value)`
- Queries: `getValue()`

The `MVRegister` works by incrementing the vector clock whenever a value is assigned. Before a value is assigned, the values are cleared. When two instances are merged, the following cases must be considered:

- If there is a concurrent write with the other instance, the value from the other CRDT is added to this instance.
- If the other CRDT has newer writes, the value of the other instance is set to this instance.

GSet Our GSet offers the following operations:

- State modifications: `add(value)`
- Queries: `has(value)`, `getValue()`, `keys()`, `values()`, `entries()`

No vector clock is required in our `GSet`, as there can only be additions. Whenever two instances are merged, the union of all values is taken as the new value. Internally, we make use of the `Set` provided by TypeScript.

ORSet The list of supported operations for our ORSet includes:

- State modifications: `add(value)`, `remove(value)`, `clear()`
- Queries: `has(value)`, `getValue()`, `keys()`, `values()`, `entries()`

This is the most complex CRDT in our package. The vector clock is incremented whenever a value is added to the set. For each value to be added, the value is added to a Map where the key is the value to be added to the Set, and the value is a pair consisting of a replica ID and vector clock. When merging two `ORSet` instances, the map of the incoming entry is iterated. Every entry that is present in the incoming instance but missing in the current instance is added to the current instance, except when the vector clock of the incoming instance is smaller than the one of the current instance. This is because the current instance might have deleted the entry.

Furthermore, every element present in the current instance but missing in the incoming instance is removed when the incoming vector clock is greater than the current one. This would mean that the incoming instance has deleted the entry.

5.4 Common Package

The `@miso/common` package is responsible for reducing code duplication by storing artifacts that are required by multiple packages in this package. The link can be found in Appendix Section A.1.4. The main reason for this package currently is to store all protobuf files in the common subpackage, as they are required by both the SDK and middleware packages.

The structure of the common package is the following:

Table 5.3: Protobuf Files and Services

File Name	gRPC services and methods
middleware.proto	**GCounterService, PNCounterService, MVRegisterService, SetService, EWFlagService.** Methods described in Section 5.3

replication.proto	**ReplicationService**
	For each CRDT:
	`Retrieve<CRDT>, Merge<CRDT>`
	`<CRDT> = Type of CRDT`

discovery-function.proto	**OverlayFunctionDiscoveryService** • `(un-)registerServerlessFunction` • `exchangeMiddlewareServerless-FunctionPodInfo`
discovery-node.proto	**OverlayNodeDiscoveryService** • `init, heartbeat`

- **`middleware.proto`:** Protobuf definitions for core middleware operations (i.e., modify CRDTs)
- **`replication.proto`:** Protobuf definitions for replication logic
- **`discovery-function.proto`:** Protobuf definitions for overlay network
- **`discovery-node.proto`:** Protobuf definitions for overlay network
- **`common.proto`:** Shared protobuf definitions
- **`server/`**
  Contains generated types for the middleware (i.e., gRPC server)
- **`client/`**
  Contains generated types for the SDK (i.e., gRPC client)

Table 5.3 shows which gRPC services each protobuf file defines (in bold) and which methods they offer. The `common.proto` file does not expose any gRPC services but only stores messages that are used in the other protobuf files. The client and server types must be regenerated whenever the protobuf files are modified. This can be done

by running the `npm run build` command from the root of the common package. The generation process is only tested on Unix-based systems.

Listing 2 shows an example protocol buffer definition of MISO. In this example, a gRPC Service with the name `GCounterService` is defined with two methods. The `Add` and `GetValue` methods return a `CounterResponse` message, which defines that the response contains two fields, a field of type `StatefulObjectBaseInformation` and the value of the counter as a signed 64-bit integer value.

```
1  syntax = "proto3";
2  package miso.middleware;
3  import "common.proto";
4
5  service GCounterService {
6    rpc Add(CounterAddOrSubtractValueRequest) returns (
7      miso.common.CounterResponse
8    ) {}
9    rpc GetValue(CounterGetValueRequest) returns (
10     miso.common.CounterResponse
11   ) {}
12 }
13
14 message CounterResponse {
15   StatefulObjectBaseInformation statefulObjectBase = 1;
16   sint64 value = 2;
17 }
18 // ... rest omitted
```

Listing 2: gRPC Protobuf Example from MISO

CHAPTER 6

Evaluation

In this section, we present a multifaceted evaluation of our middleware that encompasses both technical experiments and qualitative aspects. This provides a holistic perspective of the performance, scalability, and usability of our software prototype. The evaluation starts with an evaluation of the performance overhead of the middleware (Section 6.1). This is followed by an in-depth analysis of the replication algorithm (Section 6.2). Finally, we conclude the evaluation with a qualitative evaluation of the middleware's integrability with existing serverless platforms and the usability of our SDK (Section 6.3).

6.1 Performance Overhead

In this evaluation, we focus on assessing the performance of the core middleware operations using technical experiments, specifically the modification of MISO Stateful Objects via serverless functions. For this experiment, we have chosen to utilize an *AllReduce* operation.

6.1.1 Experiment Definition

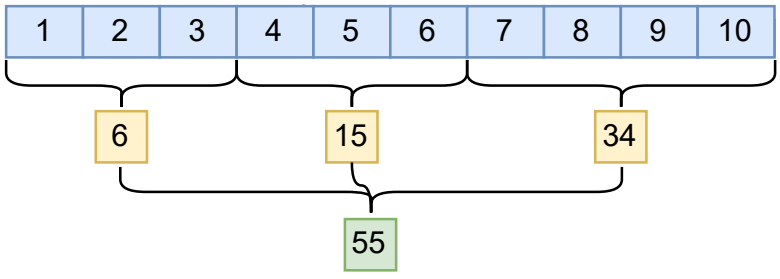

Figure 6.1: AllReduce Operation

Table 6.1: AllReduce Experiment Configurations

Type	Nodes in Cluster	Replication	Description
MISO	1	No	Baseline measure where a single node is used for writes
MISO	5	Yes	Replication to all nodes, delay set to 0ms
Redis Enterprise	5	Yes	Writes to primary node, replication to secondary node
MinIO	5	No	One cluster
MinIO	5+5	Yes	Two clusters, Site Replication enabled

Figure 6.1 shows how the AllReduce operation works. The objective of this experiment is to reduce a numeric array to a single number as fast as possible. At the beginning of the experiment, an array of integer numbers is generated (e.g., numbers from 1-10). This array is then split into multiple chunks. Then, a serverless function is invoked with the chunk of numbers, which the function then sums up to an intermediate sum (e.g., 6, 15, and 34 in Figure 6.1). The function then stores this partial sum. The intermediate sums are then reduced to a single number (e.g., 55 in Figure 6.1) by invoking the serverless function again with a different parameter. The serverless function is called $n+1$ times (n times for writing partial results, 1 time to retrieve the overall result).

This means the serverless function is invoked 1 000 times per test run with high concurrency. For each test run, the following metrics are measured:

- Average total time (i.e., the time it takes from the start of the test run until the result has been retrieved, including the time it takes to invoke the serverless functions)
- Average write time (i.e., the time it takes to calculate all intermediate results in the serverless function, including the time it takes to invoke the serverless functions)
- Average read time (i.e., the time it takes to retrieve the final reduced sum, including potential retries due to replication delay, including the time it takes to invoke the serverless functions)

6.1.2 Experiment Setup

The experiment was executed 500 times using three different solutions to store the partial/total results: MISO, MinIO (S3-compatible Object Store), and Redis (Key-Value Store). Table 6.1 shows the different configurations in which the configurations were run.

The experiment has been executed on a VM that runs Ubuntu 22.04.3 LTS, powered by an AMD EPYC 7742 processor with 64 cores and 128 GB of RAM reserved for the

VM. On this VM we run a KinD cluster (KinD version 0.20.0) with 5 nodes, where we have installed our middleware, Redis Enterprise (version 7.2.4-92), and MinIO (version 2024-01-18T22-51-28Z) using Helm charts.

Because MISO uses CRDTs, the result of the AllReduce operation can only be retrieved after all nodes have replicated their changes. For this reason, our test script repeats the retrieval of the result in case the replication has not yet been completed.

MISO Setup

We have created a setup shell script that installs MISO in a KinD cluster, optionally with OpenFaaS. The link can be found in Appendix Section A.2.3. It performs the following tasks:

1. Start a local image registry container
2. Build middleware docker image
3. Create KinD cluster with 5 nodes, with local registry configured
4. Connect registry with KinD network
5. If `-m` flag is supplied, our middleware is started on every node (standalone setup).
6. If `-m` flag is not supplied, OpenFaaS is started with MISO integrated (integrated setup)
7. Metrics server is added to the Kubernetes cluster
8. Monitoring is started (Prometheus + Grafana + our own test script)

We have tested the setup script under Ubuntu 22.04 LTS. `Docker`, `faas-cli`, `kind`, and `Helm` need to be installed beforehand.

The details of the integration of MISO with OpenFaaS are explained in Section 6.3.

```
{
  "testRun": 1,
  "type": "miso",
  "timeStart": 17042883074174,
  "timeWrite": 872.423559,
  "timeRead": 10.817261,
  "timeTotal": 883.24082,
  "isCorrect": true
}
```

Listing 3: Test Script Output

Redis Enterprise Setup

For Redis Enterprise, we followed the official guide to deploy it into Kubernetes [53]. Our deployment file requested 4 CPUs and 4 Gi of memory, with a limit of 10 CPUs and 10 Gi.

Our Redis Enterprise cluster comprises 5 nodes. This cluster runs a Redis Enterprise database with 10 GB of RAM and replication enabled. All write operations performed by the serverless functions are routed to the same Redis Enterprise master node automatically by Redis. This is because we use the Kubernetes Service for the Redis Enterprise database, which Redis Enterprise creates. These writes are then asynchronously propagated to a replica node by Redis Enterprise. It is important to emphasize that we are using a regular Redis Enterprise database in a single cluster and not an Active-Active geo-replicated Redis Enterprise database.

MinIO Setup

For MinIO, the experiment was conducted in two different configurations. The first configuration involved a single MinIO cluster comprising 5 nodes with one drive per server, 10 GB total capacity, and replication and versioning turned off. The second configuration comprised two clusters with 1 drive per server, 10 GB total capacity, and site replication/versioning turned on. To set up the clusters, we have first deployed the MinIO operator [54]. The MinIO tenants were then created using the operator in accordance with the official documentation [55].

Test script

To perform the AllReduce operation, a test script has been created that can be deployed as a Helm chart. This is because OpenFaaS and the serverless functions run inside Kubernetes. In order to avoid additional latency caused by port-forwarding into the cluster, the test script is also run inside the same Kubernetes cluster. It can then be triggered via HTTP after creating a port forward to the pod that is created. However, it is important to state that the actual test runs completely inside Kubernetes. The test script can be called with a POST request to `curl -X POST http://localhost:3000/allreduce/:type/:repetitions/:arraySize/:chunkSize`, where `type` can be `miso, minio, redis`. For the remaining parameters, we utilized 500 repetitions, an array size of one million, and a chunk size of 1 000. The script generates a JSON file in the `/data/json` directory, which contains data about each repetition. An excerpt of an example entry of the results can be found in Listing 3. It is visible that the script stores detailed information about the write and read times and verifies that the test run returned the correct sum. For the sake of brevity, certain less important fields have been omitted from this Listing.

6.1.3 Experiment Results

Figure 6.2 provides a comprehensive summary of the experiment results. It depicts the average total, read, and write times, as well as the 99th percentile of the total average time for different technologies and configurations. In the outlined AllReduce use case, the mean response times (encompassing total, read, and write times) of MISO consistently outperform those recorded for Redis Enterprise and MinIO. A more detailed analysis of the results is presented in the subsequent sections.

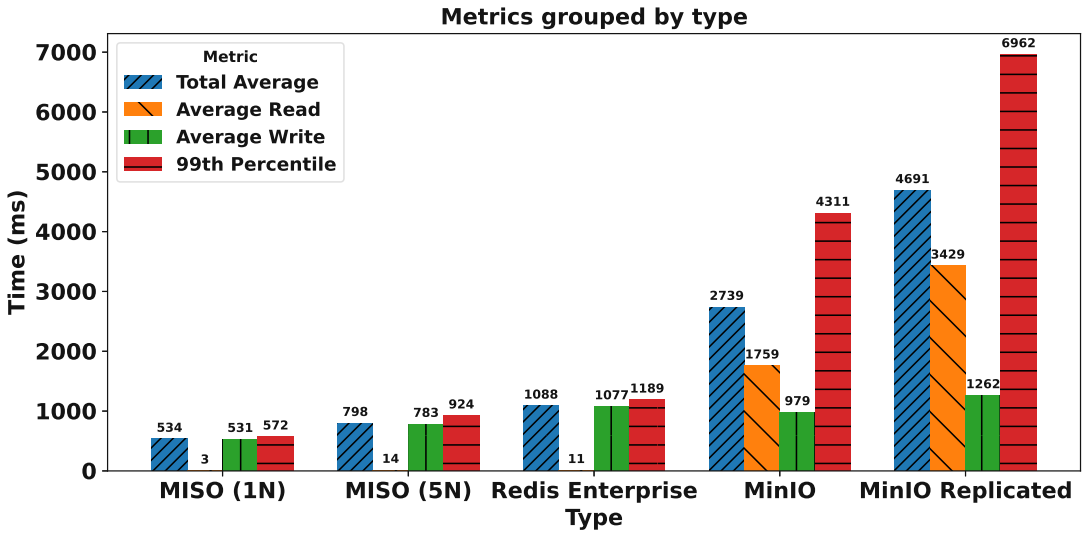

Figure 6.2: AllReduce Results - Grouped Bar Chart

Figure 6.3 shows the moving average read times of all types observed in our experiment, rolling over a window of 10 experiment runs.

Figure 6.4 illustrates the moving average write times of all types observed in our experiment, rolling over a window of 10 experiment runs.

MISO

In Figures 6.3 and 6.4, we observe that CRDT-based counters, like those implemented in MISO, are well-suited for this AllReduce use case. This is because each intermediate sum calculated from each chunk of the input array can simply be added to a counter. This counter is shared between all instances of the serverless function. To retrieve the final reduced sum, the serverless function needs to retrieve the current value of the counter after every chunk has been processed. The serverless function does not have to perform a manual aggregation of the intermediate sums, as this is essentially built into the data type itself when retrieving the current value of the counter. It is, however, important to note that the final result, when using MISO, is only available to the serverless function

Figure 6.3: AllReduce Results - Read Times

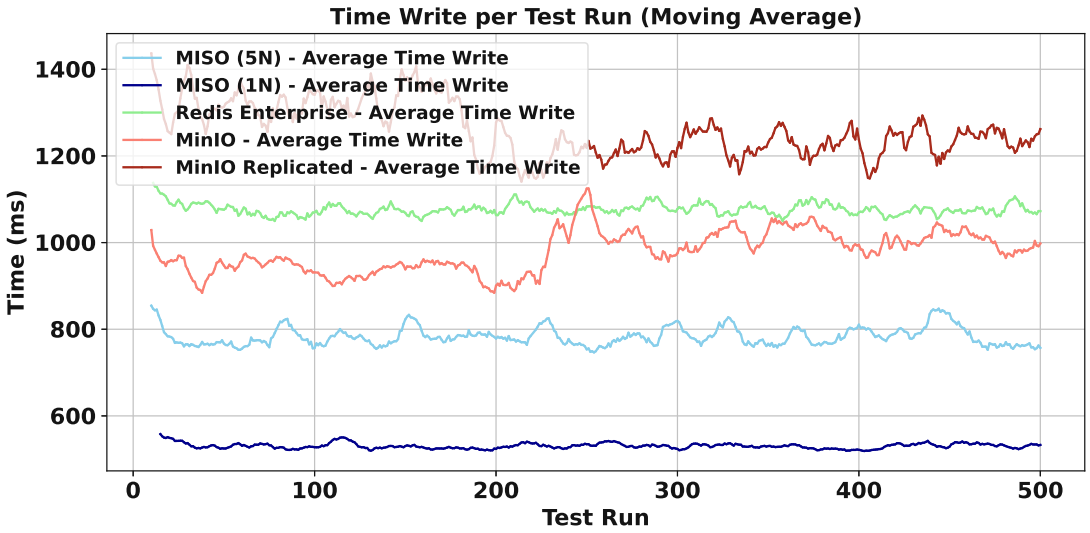

Figure 6.4: AllReduce Results - Write Times

after all affected nodes have replicated their changes due to the principle of eventual consistency.

As visible in Figure 6.3, the read times for MISO remain consistently low with an average of 14ms and a standard deviation of 5ms when using a cluster of 5 nodes. Furthermore, as visible in Figure 6.4, the write times are also stable, averaging at 783ms with a standard deviation of 44ms. The write time in this experiment is consistently lower than that of Redis Enterprise and MinIO. The read time for MISO is minimally higher than that of Redis Enterprise, which can be attributed to how CRDT-based counters, as employed by our middleware, are designed. They operate by storing the sum of each replica within a map. To obtain the current total sum of the counter, the partial sums from all replicas need to be aggregated, which contributes to the overall computational overhead. The difference in read times between our solution and Redis Enterprise in this setting is marginal, with MISO at 14ms and Redis at 11ms. This negligible difference underscores the efficiency of our middleware in this use case despite the additional computational steps involved in its operation.

Redis Enterprise

Our implementation of the AllReduce experiment in Redis Enterprise works by utilizing a unique *hash* for each test run. This hash is then populated with the intermediate sums of the chunks of the input array, using a unique key for each interim result. By using unique keys, we avoid synchronizing or locking the whole object, as the same replica ID (i.e., the name of the pod that runs the serverless function instance) might be accessing the object concurrently. To compute the total reduced sum, the serverless function has to retrieve all values from the hash to aggregate all values into a single value.

As visible in Figure 6.3, the read times for Redis Enterprise in this use case are similar to the one of our solution, with an average of 11ms. This is marginally lower than what we have observed for MISO. The write time average, visible in Figure 6.4, is 1077ms, which is almost 300ms higher than MISO, with a standard deviation of 48ms. However, the read times for Redis Enterprise have a smaller standard deviation than both MISO (with 5 nodes) and MinIO.

MinIO

To accomplish this AllReduce use case with an object store such as MinIO, every intermediate sum of each chunk is stored as a file in a certain bucket. We create an empty bucket for each test run and remove the files afterward. The times to create the bucket in the beginning and remove the files and bucket are not taken into account when measuring the times of each test run. After all intermediate results have been stored in the bucket, the serverless function retrieves all objects in the bucket. It performs the final reduction to get a single number by aggregating all file numbers.

Figures 6.3 and 6.4 show the read and write results of MinIO in this use case using both a single cluster and two replicated clusters. The average read and write times for MinIO

are substantially higher than those of MISO and Redis Enterprise. The read-time average is 1759ms in a single cluster and 3429ms in a replicated cluster. The write-time average is 979ms in a single cluster and 1262ms in a replicated cluster. The standard deviations for a non-replicated cluster are 571ms (read) and 69ms (write). For a replicated cluster, the deviation is 1078ms (read) and 118ms (write). This behavior, which is that the read time is higher than the write time, differs from what we have experienced in both MinIO and Redis Enterprise. This can be explained by the fact that every intermediate result is stored as a separate file in the bucket, and retrieving all files is an expensive operation. The standard deviations are significantly higher for MinIO than for MISO and Redis Enterprise, which implies that the observed values have more variance.

Comparison

Our results show an improvement over Redis Enterprise by 26.7% for the total average time. Compared to MinIO, our solution was 243.2% faster in a non-replicated cluster and 487.9% faster in a cluster with site replication turned on. To compare our numbers, we used the total average time of MISO in a cluster of 5 nodes, so all solutions utilized a cluster of 5 nodes. This implies that CRDT-based data structures offer great performance for use cases that can utilize their potential, as our presented AllReduce use case.

6.2 Replication Algorithm

In this section, the replication algorithm of MISO is evaluated by performing load tests. The following metrics are measured:

1. Replication time (i.e., the time it takes to replicate to all relevant nodes for one replication run without waiting for acknowledgement),
2. Total Requests per Second (RPS) (i.e., sum of all MISO-related RPS on all participating nodes),
3. Replication Data Volume,
4. Process Memory Usage.

6.2.1 Experiment Definition

We perform a stress test on the middleware to evaluate the replication algorithm. As the middleware exposes its operations through gRPC, we use an open-source gRPC benchmarking tool called *ghz* [56]. We did not modify it and do not distribute it as part of this work. The benchmark works by repeatedly calling a gRPC endpoint with multiple threads. We use ghz to increase a PNCounter concurrently and then study the metrics of the replication algorithm with varying nodes. Our test script distributes the total number of requests that should be made over all nodes and thus produces a distributed load among the nodes, simulating concurrent data modification on multiple nodes.

6.2.2 Experiment Setup

Our test script runs as a Kubernetes pod that wraps an HTTP server. It can then be started via a POST request. We have used 10 concurrent threads and performed 5 million increases to a PNCounter. We have deployed MISO to the same Kubernetes cluster, running on KinD with 5, 10, 20, and 30 nodes. The replication interval was set to 5ms, and the cluster ran on the same virtual machine as discussed in the evaluation of the performance overhead (see Section 6.1). The `concurrency` flag of `ghz` was set to 10.

To collect the metrics, we deploy the `kube-prometheus-stack` Helm chart [57] to the KinD cluster. This means that Prometheus and Grafana are deployed to collect and visualize metrics, respectively. The test script, Prometheus, and Grafana are bundled in the same Helm chart as previously described in Section 6.1.2. The link can be found in Appendix Section A.2.1.

6.2.3 Experiment Results

Average Replication Time

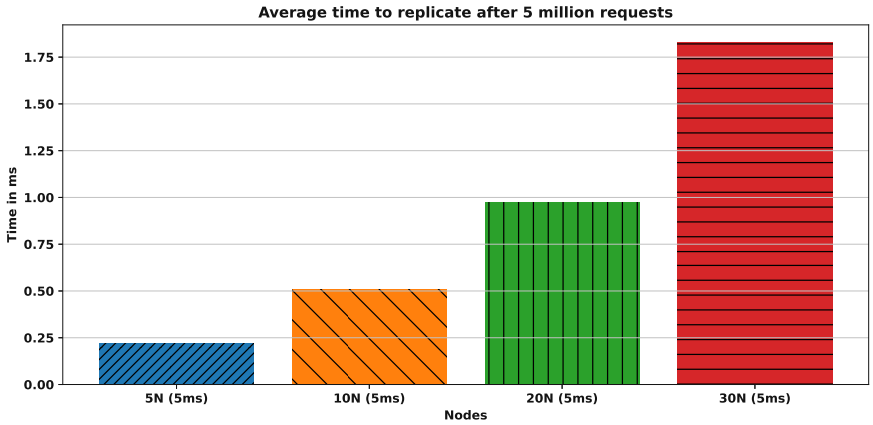

Figure 6.5: Average Replication Time

Figure 6.5 shows the average time it takes to replicate for each replication run after performing 5 million requests. This is the time between the start of the replication run and when the update was sent to all participating nodes without waiting for the acknowledgements of the receiving nodes. We do not wait for the reply of the receiving nodes as we simulate the nodes on a single machine, where there is no artificial network delay between the nodes. It can be seen that there is an increase in time as more nodes are added to the cluster. With 5 nodes, the average replication time was 0.22ms. This increased to 0.51ms with 10, 0.97ms with 20 nodes, and 1.83ms with 30 nodes. Between 5 and 20 nodes, the addition of time is approximately linear (approximately 341% increase in time with a 300% increase in cluster size). The increase in time (approximately 732%) significantly exceeds the increase in cluster size (500%) when we increased the

cluster size from 5 to 30 and is thus higher than linear. The increase in time between the different cluster sizes generally is expected, as MISO needs to send more updates with increasing cluster sizes where every node runs the affected serverless function. In our experiment, we simulate that replication to every node is required. Therefore, as all nodes replicate to all others, this leads to a quadratic increase in total connections between the nodes and replicated data in a fully connected network. Furthermore, in our experiments, all participating nodes in the cluster modify the same CRDT concurrently with a high load, putting enormous stress on the system. This means that we demonstrate a worst-case scenario where the exact same data type is modified on all nodes in a cluster simultaneously with very high concurrency and a very frequent replication interval of 5ms. Given these circumstances, we figure that the replication time increases accordingly and within reasonable limits.

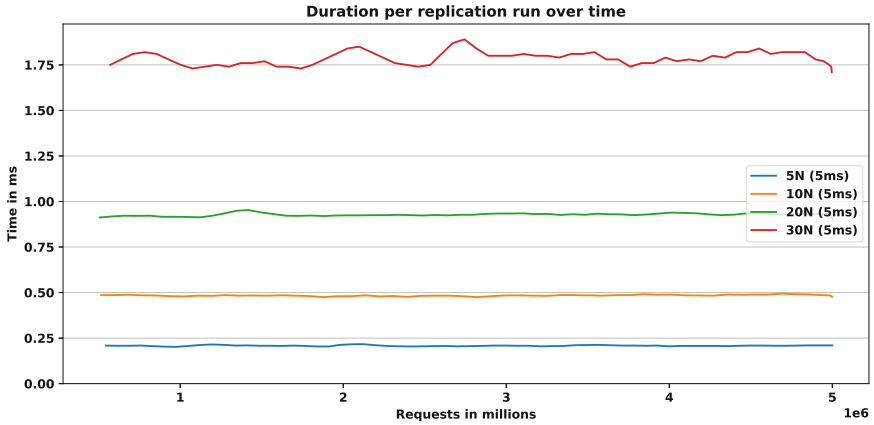

Figure 6.6: Replication Time over Time

Figure 6.6 depicts how the replication time changed over time during the experiment. The average value stays consistent during the experiment run for all cluster sizes. It is visible that the replication takes more time as more nodes are added to the system, which is expected. In the cluster with 30 nodes, some minimal spikes in time are visible after around 2 and 2.75 million requests. However, such minimal differences are expected in a distributed system. Furthermore, our replication interval was set to a low value of 5ms, placing a high stress on the replication part of the middleware. Because of how our experiment works, we leave out the initial 500 000 requests to show results with equal load and without initial connection setup.

Requests per Second

Figure 6.7 shows the average total RPS rate after performing 5 million requests accumulated across all nodes. The total RPS rate is composed of the core requests, i.e., the requests to modify the MISO Stateful Objects and the replication requests. For 5, 10, and 20 nodes, the average total RPS rate was between 13 800 and 16 300 core requests

per second. When utilizing 30 nodes, the core RPS rate dropped slightly to around 13 800 RPS, compared to 16 300 with 10 nodes. However, it is visible that the replication requests rise significantly as more nodes are added to the system that concurrently modify the same MISO Stateful Object. Given the increase in replication requests, the core RPS rate stays stable. The replication requests rose from approximately 3 100 to 43 300 RPS with 5 and 30 nodes, respectively. This is an increase in requests by approximately 1297% while the cluster size increased by 600%. Similarly to what we have described for the average replication time, this behavior can be explained by the quadratic increase in network connections between the nodes in a fully connected network when the same data structure is concurrently modified on all nodes.

Therefore, it is apparent that with such a high replication interval as we used in this experiment (5ms), an infinite number of nodes cannot be added where the same data structure is changed simultaneously on all nodes. In such circumstances, a slower replication interval should be used. Nevertheless, with a total average of approximately 57 100 RPS (core + replication) in a setting with 30 nodes and a replication delay of 5ms, our results indicate that MISO can handle a high number of concurrent modifications without problems.

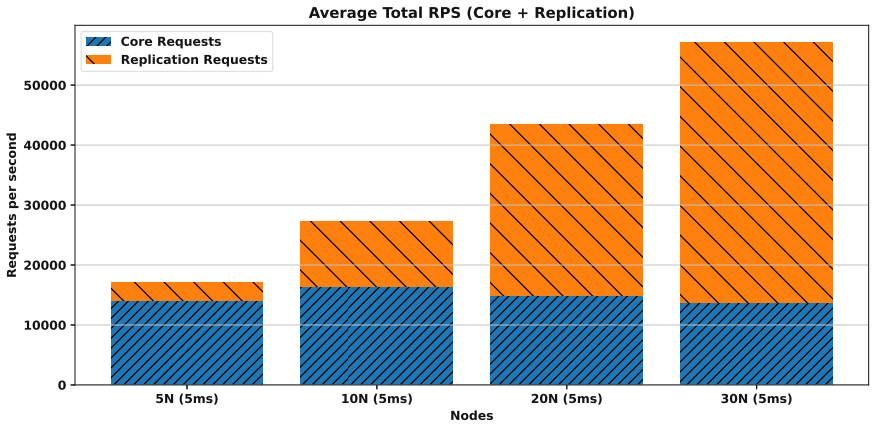

Figure 6.7: Average Requests per Second

Figure 6.8 demonstrates how the RPS rate of the core requests changed during the experiment execution time without taking into account the replication requests. The reason why the core RPS rate is lower in a cluster with 5 nodes than, for instance, 10 nodes can be explained by the way our test works. The total number of requests is distributed evenly across all nodes in a cluster. Even though we have set a very frequent replication interval of 5ms in our experiments, 10 nodes can still process more requests per second than 5 nodes in this scenario. However, starting with a cluster size of 30 nodes, we have experienced a drop in the overall RPS rate. This result is expected, as MISO needs to replicate tens of thousands of requests at the same time between all nodes of the cluster. As more nodes are added to the cluster, more time is spent replicating data to more nodes, and fewer resources are available for the core middleware operations.

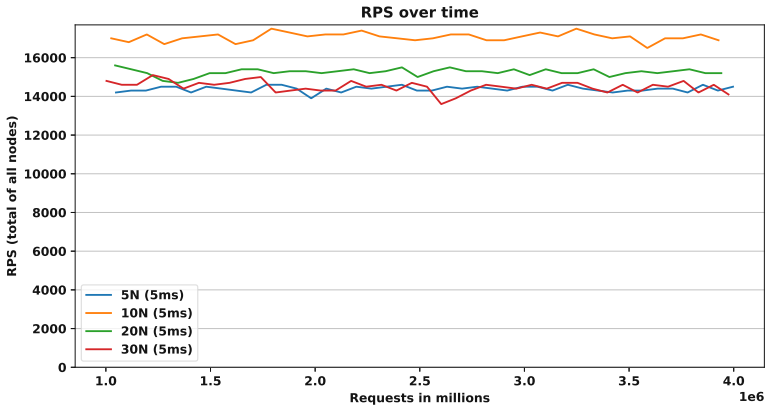

Figure 6.8: Requests per Second over Time

However, it is visible that the RPS rate stays consistent over time. Because of how our experiment works, we leave out the initial and last 1 million requests to show results with equal load and without initial connection setup.

Replication Data Volume

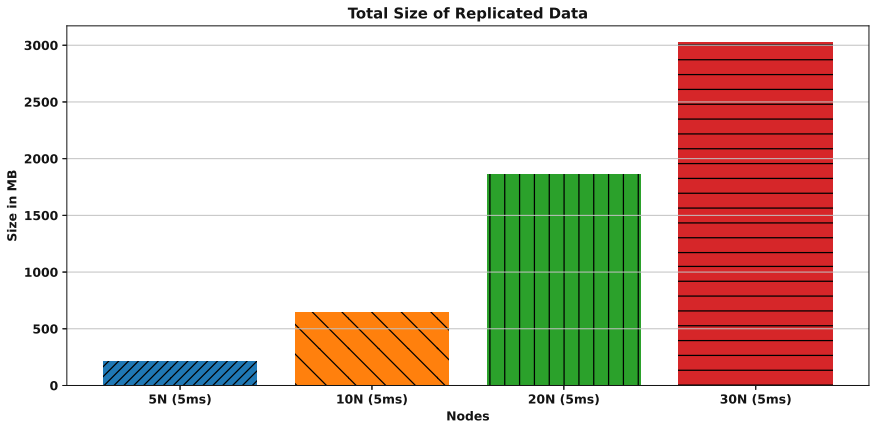

Figure 6.9: Replication Data Volume

Figure 6.9 depicts the total replication data volume. It is visible that the amount of data increased from 211 MB with 5 nodes to 3.02GB with a cluster of 30 nodes. This non-linear increase was expected, as in our experiment all nodes replicate to all others, leading to a quadratic increase in connections and replicated data in a fully connected network. We have deliberately chosen a low replication interval where replication happens almost in real-time to simulate as high a load as possible and demonstrate the limits. Depending on the use case, the amount of data sent over the network can be drastically

reduced with a lower replication interval.

Process Memory Usage

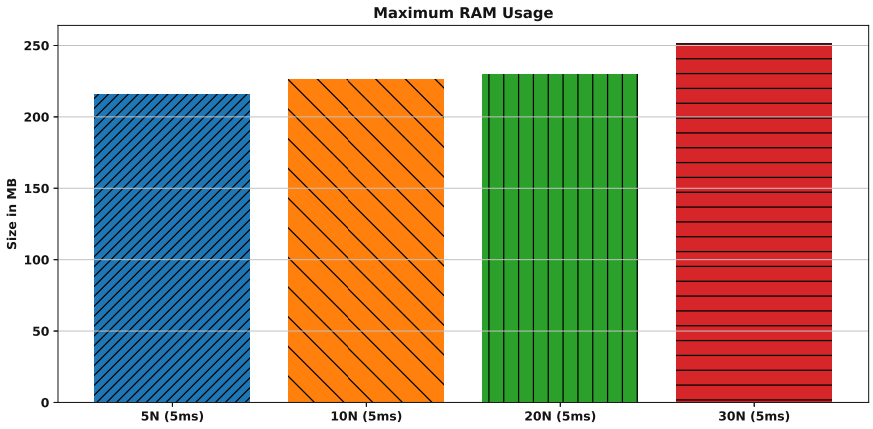

Figure 6.10: Maximum Process Memory Usage

Figure 6.10 shows the maximum observed process memory usage. This is the maximum amount of memory a single middleware process took during the experiment run time on a single node, including all middleware components. The maximum memory usage for 5 nodes was lowest, with 216 MB. With 10 and 20 nodes, we observed a higher value of around 230 MB. For 30 nodes, we have observed a maximum of 251 MB. This shows that the amount of process memory required tends to rise as more nodes are added when data is concurrently written and replicated to every node in the cluster. This does not imply that the same is true as more nodes are added to the cluster, and concurrent replication to all participating MISO instances is not required. The observed increase in memory from 5 to 30 nodes was 16.2%, while the cluster size increased by 500%. This means that the increase in memory is significantly smaller than the increase in node size. This suggests that our work can be applied in resource-limited environments.

Impact of Replication Interval

Another aspect that was investigated is the impact of the replication interval on the middleware performance. We have performed a gRPC throughput test using ghz with 10 concurrent connections over a single connection to increment a PNCounter 500 000 times. Different from the experiment described above, we have now directed all load to a single MISO node in a cluster of 20 nodes, and this single node replicated the data to the other 19 nodes using a varying replication interval. This enables us to better measure the impact on a single MISO node. The results are visible in Table 6.2.

Our results show that the configured replication interval heavily influences the performance of the middleware and the replication. The higher the replication interval is set (i.e., less frequent updates), the higher the RPS scores are, and the request latency for core

Table 6.2: Impact of Replication Interval on Performance

Replication Interval	RPS	Total Time	Avg. Latency	Repl. Data	Repl. Requests
0ms	1 502	333s	6.41ms	244 MB	1 726 131
5ms	1 909	262s	4.99ms	124 MB	640 490
10ms	2 742	182s	3.41 ms	52.3 MB	269 762
50ms	3 884	129s	2.36ms	8.91 MB	45 904
200ms	4 168	120s	2.19ms	2.19 MB	11 153
1000ms	4 339	115s	2.10ms	449 KB	2 185
5000ms	4 501	111s	2.01ms	111 KB	437

middleware operations decreases. For example, with a replication interval of 0ms (i.e., immediate replication), we have achieved an average of 1 502 RPS over 500 000 requests, which increased to 4 144 RPS with an interval of 200ms. Similarly, a higher replication interval decreases the total amount of replicated data and the number of replication network requests. Starting from a replication interval of around 200ms, further increases in the replication delay had a negligible impact on the overall performance. When we decreased the replication interval by 500% from 200ms to 1000ms, the RPS rate increased from 4 168 to 4 339, which is an improvement of approximately 4%. This suggests that tuning the replication interval to the requirements of each use case is essential and has a big impact on the overall system performance.

6.3 Qualitative Evaluation

In this section, we evaluate whether it is possible to integrate MISO with an existing open-source serverless platform and the usability of the SDK. The goal is to show that our middleware is easy to integrate into existing serverless platforms and that the SDK is easy to use and understandable.

6.3.1 Integrability

This section demonstrates the process of integrating both the middleware and SDK with OpenFaaS, as depicted in Figure 6.11. The process consists of two steps. The first is to integrate the middleware into the serverless platform provider, and the second is the integration of the SDK into the serverless function template.

Integrating the Middleware

The necessary steps to integrate the middleware with OpenFaaS can be summarized as follows:

1. Add MISO middleware to the Helm chart template of the OpenFaaS provider.

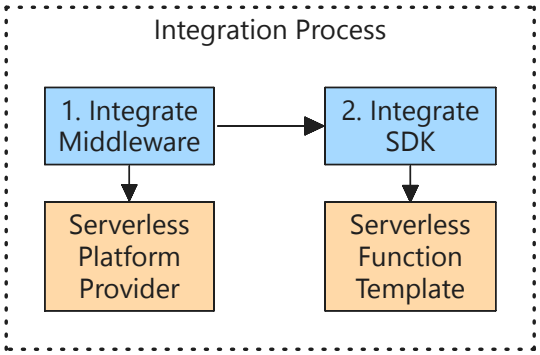

Figure 6.11: Overview of Serverless Platform Integration Process

2. Set environment variables in the function deployment handler of the provider.
3. Build the provider image locally and change the image pull policy.

The following paragraphs will outline the details of each of the three steps.

Step 1: Add MISO to Helm chart As described in Section 5.1.6, we included our Helm chart in the deployment of OpenFaaS by modifying the Helm chart of the `faas-netes` provider. More precisely, we added our template as found in Appendix Section A.2.2 to the `faas-netes/chart/openfaas/templates` folder so that it is included in the deployment. In the future, if our Helm chart is hosted in a publicly available repository, it could be added to the serverless platform's Helm chart dependencies without manually copying the `yaml` file.

Step 2: Set environment variables To simplify the configuration for the MISO Stateful Object proxy in the SDK, we have also adapted the function deployment handler of `faas-netes`. More precisely, in the `faas-netes/pkg/handlers/deploy.go` file, we have modified the `buildEnvVars` function so that the environment variables mentioned in Section 5.2 are set by utilizing the Kubernetes Downward API. This step is optional if the serverless platform already provides similar information out of the box or if the SDK is manually configured in the code of the serverless function handler.

Listing 4 shows a part of the modified `buildEnvVars` function code. It is visible that we have added the environment variable `MISO_HOST_IP` where Kubernetes automatically sets the value of the field `status.hostIP`. Other environment variables are set in the same way.

Step 3: Build provider image The locally modified `faas-netes` provider then needs to be built with the `make` command from the `faas-netes` root folder. We changed the image server in the `Makefile` of the provider to `localhost:4000` so that we can run the image with a local image registry by specifying this image in

```
func buildEnvVars(request *types.FunctionDeployment) []corev1.EnvVar {
    envVars := []corev1.EnvVar{}

    envVars = append(envVars, corev1.EnvVar{
        Name: "MISO_HOST_IP",
        ValueFrom: &corev1.EnvVarSource{
            FieldRef: &corev1.ObjectFieldSelector{
                FieldPath: "status.hostIP",
            },
        },
    })
    // rest omitted
    return envVars
}
```

Listing 4: Modified faas-netes Function Deployment Handler

the `faas-netes/chart/openfaas/values.yaml` file with an image pull policy of `Always`. This ensures that the modified provider is used instead of pulling the official image from Docker Hub.

Integrating the SDK

This section demonstrates how the SDK can be integrated with OpenFaaS to use it within the code of serverless functions. It is currently available for Node.js projects and can, therefore, be added as an npm dependency in every such project. Developers have two choices for the integration of the SDK:

1. Integrate it in the serverless platform template for Node.js
2. Add it to the dependencies of the serverless function only

The first option has the benefit that it is more convenient for developers. Many serverless platforms provide templates for serverless functions in different programming languages, which can be customized. The second option implies that the creating the MISO Stateful Objects and registering/unregistering the function must take place in the handler of the serverless function itself.

In our evaluation, we demonstrate Option 1. We have modified the *node18* template of OpenFaaS in such a way that it adds the SDK as an npm dependency. Additionally, the function template can take over the responsibility of registering and unregistering the function with MISO at startup or termination, respectively. A simplified excerpt of the corresponding code fragment is shown in Listing 5. In the adapted template, the function first registers itself in the main entry point using our SDK and then executes

the function handler as usual. Whenever a *SIGTERM* event is registered, the function is unregistered from MISO.

```
const sdk = require('@miso/sdk');
const so = new sdk.StatefulObjectProxy();
so.registerServerlessFunction().then(() => {
  process.once('SIGTERM', function (code) {
    so.unregisterServerlessFunction();
  });

  class FunctionContext {
    constructor(cb) {
        // set stateful object in fn context
        this.statefulObject = so;
        // ...
    }
    // .. details omitted
});
```

Listing 5: Example of Extended Node18 Template

Summary

We observed a straightforward integration of MISO into OpenFaaS and expect that such an integration will look similar for other Kubernetes-based serverless platforms. The principle of how we integrated the SDK into serverless functions also applies to other programming languages and/or serverless platforms. We have demonstrated that integrating our SDK requires only minimal code changes. More precisely, in the above example, we needed to modify less than 10 lines of code in the serverless function template to integrate our SDK and to create a default MISO Stateful Object in the context of every request. In our example, the required configuration is automatically loaded by the SDK. This is because we have integrated it into OpenFaaS for our experiments and set the required environment variables when deploying the function pod. Alternatively, the configuration could also be set in the constructor of the MISO Stateful Object proxy class.

6.3.2 Usability

A major part of software development cost is poor code understandability [58]. This is because inspecting and maintaining poorly understood code is hard, and a lot of time is spent there. Refactoring code sections that are hard to understand improves maintainability and saves time and effort.

In this section, we use two metrics to measure the **understandability** of MISO's SDK. The first one is *Lines of Code*, a widely used traditional code measure. The second one

is *Cognitive Complexity*, which is a newer metric introduced in 2018 [58]. According to Campbell [59], Cognitive Complexity is based on three principles:

- "Ignore structures that allow multiple statements to be readably shorthanded into one"
- "Increment for each break in the linear flow of the code"
- "Increment when flow-breaking structures are nested"

To measure the cognitive complexity of the code, we use an open-source tool available on GitHub [60]. We did not modify it and do not distribute it as part of this work.

We recall the experiment previously mentioned in Section 6.1. In this experiment, we have utilized three different solutions to accomplish an AllReduce use case: MISO, Redis Enterprise, and MinIO. In this section, we now compare the implementations of the three different SDKs to measure the *understandability* of the solutions to see whether our own SDK is understandable.

MISO

Storing the intermediate sums is straightforward with MISO. All values can simply be added to the same counter instance. The final reduced value can be retrieved by accessing the current value of the counter after all intermediate sums have been processed, as the final reduction is built into the Counter data type itself. Listing 6 shows the code to set the intermediate values and retrieve the final sum. The interaction with MISO effectively consists of only two method calls: *counter.add()* and *counter.getValue()*.

Our minimal code example for MISO, without error management, consists of 44 lines of code.

Redis Enterprise

Storing the intermediate values of AllReduce is simple in Redis Enterprise. We set all intermediate values as unique keys in the same Redis Hash. To Retrieve the overall result, we retrieve all values of this Redis Hash and manually perform the reduction in the serverless function afterward.

Our minimal code example for Redis Enterprise, without error management, consists of 53 lines of code. This is an increase of 20.5% compared to the minimal sample of MISO.

MinIO

Implementing the AllReduce experiment equally with MinIO was more complex than for MISO and Redis Enterprise. First, we create a new bucket for each test run. Afterward, we save each intermediate result as a new file in this bucket. To retrieve the final result,

```
async function performAllReduce(
  body: any,
  statefulObject: StatefulObjectProxy
) {
  const counter = statefulObject.getPNCounter(
    body.crdtName
  );
  const sum = body.values.reduce(
    (a: number, b: number) => a + b, 0
  );
  counter.add(sum);
}
async function getResult(
  body: any,
  statefulObject: StatefulObjectProxy
) {
  const counter = statefulObject.getPNCounter(
    body.crdtName
  );
  return await counter.getValue();
}
```

Listing 6: Code for AllReduce with MISO

we retrieve all files of the bucket and then manually compute the sum by reducing the values in all files to a single number.

Our minimal code example for MinIO, without error management, consists of 110 lines of code. This is an increase of 150% compared to MISO. Our MinIO code contains more than twice as many lines of code than Redis Enterprise and MISO. Also, the MinIO SDK returns the list of objects and the objects themselves of a bucket as a stream, which requires developers to concatenate the data manually. This is useful for objects that are very large. For our experiment, however, this is not as relevant and thus not optimal, as we do not have very large objects but rather a high number of relatively small objects containing the intermediate sums only.

Comparison

Figure 6.12 shows a comparison of the lines of code and cognitive complexity of our experiments. It can be seen that MISO required the least amount of code. Redis Enterprise required approximately 20.5% more and MinIO 150% more lines of code than our solution. Similarly, our minimal code samples for both MISO and Redis Enterprise have a cognitive complexity score of *2*, while our sample for MinIO has a score of *8*. This means that the cognitive complexity of MinIO was four times as high as what we have

measured for MISO and Redis Enterprise.

These results of the lines of code and cognitive complexity, in combination with the simplicity we have shown in Listing 6, underscore that our SDK is easy to understand. This positively affects the maintainability and positions it as a valuable addition when it comes to providing MISO Stateful Objects to serverless functions.

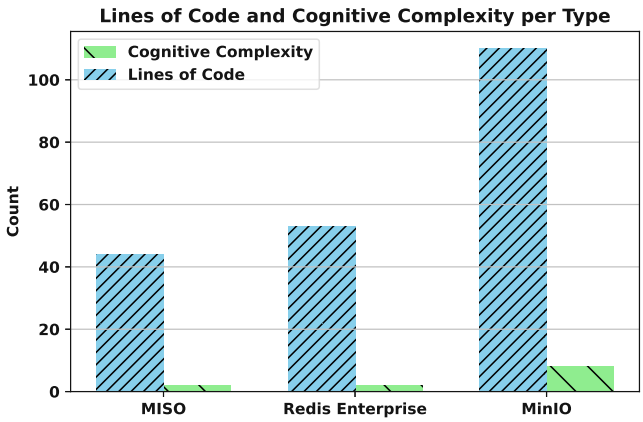

Figure 6.12: Comparison of Lines of Code and Cognitive Complexity

6.4 Limitations

One limitation of our evaluation is that due to constraints related to resources and time, we only had access to a single virtual machine. This is because for a diploma thesis, creating 30 or more real virtual machines in different cluster combinations is beyond practical limits. Our single virtual machine has many resources (AMD EPYC 7742 processor, 64 cores, 128GB RAM), and thus, we simulated multiple nodes using KinD. This simulation of nodes provided us with valuable insights and enabled us to evaluate MISO in different settings. Still, this setting does not fully replicate the same conditions as if we used separate virtual machines. Starting with cluster sizes bigger than 30 nodes, we have experienced various operational challenges such as unresponsive Kubernetes clusters/APIs, termination of containers, and dropped network connections. We believe this is an inherent limitation of our evaluation environment, which consists of only a single virtual machine. Given this fact, there naturally exists an upper bound until which we can simulate the nodes. This upper bound not only encompasses CPU/RAM limits but also network delay and I/O capabilities of the host. Lastly, in our evaluation, we do not take into account the network delay between different nodes, as all network traffic effectively runs on the same host.

CHAPTER 7

Conclusion

In this thesis we introduce MISO, a CRDT-based serverless middleware that provides MISO Stateful Objects for serverless functions in the edge-cloud continuum. MISO Stateful Objects are the conceptual foundation of the middleware and combine multiple CRDT-based data types into a single object. They are accessed from serverless function handlers using proxies, and the middleware running on the nodes executing the serverless functions manages their lifecycle. This includes retrieving, modifying, and replicating the state of the MISO Stateful Objects. Our middleware is designed in such a way that it can be integrated into multiple different open-source serverless platforms. Furthermore, it provides data locality and does not depend on a central authority for data synchronization, which is essential to make our solution fit the edge-cloud continuum.

We introduce the concept of MISO Stateful Objects and the architecture and modules of the middleware in Chapter 4. Architecturally, the middleware contains two large components: a middleware that runs on each node of the serverless platform and an SDK for serverless functions that provides proxy versions of the MISO Stateful Objects. Those proxies internally call the API of the middleware, and the developers do not need to know the internal characteristics of the communication with the middleware. The middleware contains multiple modules, such as an API or replication module. The middleware instances create an overlay network of all interconnected nodes to leverage information about which nodes currently execute which serverless functions. This is used to make the replication process more efficient and avoid sending unnecessary updates. We have implemented the concept of MISO Stateful Object and the proposed architecture practically in an open-source software prototype, which we describe in Chapter 5. The software prototype, consisting of the middleware, SDK and various state-based CRDTs, are written in TypeScript for Node.js.

We have performed a multifaceted evaluation of our software prototype and describe the results in Chapter 6. Our analysis of the middleware's performance in Section 6.1 using technical experiments has shown that in a real-world use case involving an

AllReduce operation, our middleware has outperformed Redis Enterprise and MinIO. In this experiment, our solution took 26.7% less time than Redis Enterprise and was almost 2.5 times faster than MinIO. This shows that our CRDT-based data types provide solid performance for use cases that can utilize their characteristics. In the case of AllReduce, one of the reasons why our middleware performs so well is because the final reduction of the total sum is built into the data type itself. We have also performed an in-depth assessment of the replication algorithm in Section 6.2, where we clearly demonstrate that the replication process is scalable and that our middleware handles thousands of concurrent requests with frequent replication intervals without issues. Furthermore, we clearly demonstrate the impact of the replication interval on the RPS, latency, experiment time, and replication traffic. Lastly, in Section 6.3, we also evaluate the possibility of integrating MISO into an existing serverless platform and the usability of our SDK. The integration with OpenFaaS, a popular serverless platform, was seamless, and we clearly described the necessary steps. Finally, we show that our SDK requires fewer lines of code and has less or similar cognitive complexity than the other solutions we utilized in the aforementioned AllReduce experiment. The cognitive complexity of our code for the AllReduce experiment was on par with Redis Enterprise and significantly lower than MinIO.

The list of main contributions of this work includes: i) the conceptual model of MISO Stateful Objects, ii) the MISO middleware consisting of the architectural model and an open-source software prototype, iii) an SDK for serverless functions that enables the usage of MISO Stateful Objects within serverless functions, and iv) the asynchronous replication of MISO Stateful Objects using an overlay network optimized towards data transfer and resource consumption.

7.1 Research Questions

In this section, we reflect on the research questions of this thesis and answer them accordingly.

- **RQ1: What is an appropriate architecture for a CRDT-based middleware that provides Stateful Objects for serverless functions in the edge-cloud continuum, so that it is scalable and integrable into existing serverless platforms?**
  We answer this research question in multiple steps. We describe the concept of MISO Stateful Objects as well as the architecture and modules of the middleware in Chapter 4. The architectural model of the middleware contains two large components: a SDK for serverless functions and a middleware that runs on every node of the serverless platform.

  After defining the middleware, we implemented it practically in a software prototype. The implementation details are discussed in Chapter 5.

In the evaluation part of this thesis in Chapter 6, we have demonstrated that our middleware is scalable and that it can seamlessly be integrated with existing serverless platforms. We clearly describe the steps that are required to integrate our work with OpenFaaS. Furthermore, we demonstrate the usability of the SDK using the metrics of lines of code and cognitive complexity. From these practical demonstrations, we conclude that our concept for the middleware is appropriate.

- **RQ2: What is an appropriate way to efficiently replicate Stateful Objects between different nodes running the middleware to avoid unnecessary network traffic?**
  Scalability is an important topic for serverless functions, as the serverless platform can scale to new nodes anytime. This means that a particular serverless function is not necessarily running on all nodes simultaneously. This is important for the middleware's replication module, as we expect that simply replicating the data to all nodes would not scale, especially with increasing utilization of the middleware.

  This is the reason why our middleware creates an overlay network between all interconnected nodes in the cluster. This overlay network is then used to discover all nodes and serverless function instances running on these. This information, particularly which node runs which function, is used by the replication module so that only nodes currently running the serverless function receive the corresponding updates. This drastically reduces the amount of data that is transferred over the network and avoids unnecessary CRDT state merges. Therefore, the efficiency of the overall replication process is improved. Our software prototype uses gRPC to replicate data, which is an efficient binary protocol running on top of HTTP/2.

  We have further conducted technical experiments to evaluate the replication algorithm and describe the results in Section 6.2. The outcome of the evaluation clearly indicates that the replication process scales and that it handles thousands of concurrent requests without issues. Furthermore, we provide concrete information about how different replication intervals influence the replication process and performance of the core middleware operations (i.e., modification of CRDTs). While it is possible to use an almost immediate replication in our prototype, it significantly affects the performance. Our results indicate that beginning with a replication interval of 50ms, the performance of the middleware is not significantly affected anymore if the replication interval is set to a higher number (i.e., less frequent replication). Ultimately, the decision of which replication interval to use depends on the particular use case and requirements, and our middleware is flexible enough to adapt to them.

- **RQ3: What is the overhead of using CRDT-based Stateful Objects in the edge-cloud continuum in terms of performance and resource consumption, compared to other state-of-the-art solutions for stateful serverless functions?**
  It's important to show that our middleware provides competitive performance

in real-world scenarios. For this reason, we have developed an experiment in Section 6.1 that performs an *AllReduce* operation in a serverless function. We have implemented the same function three times, using MISO, Redis Enterprise, and MinIO to store the state, respectively.

Our work has outperformed Redis Enterprise (by 26.7%) and MinIO (by 243.2% in a regular cluster and 487.9% in a replicated cluster) in this particular experiment. This shows that our middleware is beneficial for serverless functions, especially for use cases that can utilize the potential of CRDT-based data structures. Furthermore, our middleware operated well below the official minimum memory requirements of the other two solutions, which highlights that our work can be used in resource-constrained environments.

7.2 Future Work

This thesis provides a solid foundation for a CRDT-based serverless middleware for MISO Stateful Objects in the edge-cloud continuum. Our implementation is open-source and can already be used by other developers.

One aspect that remains unexplored in this thesis is the permanent storage of MISO Stateful Objects on disk. Such storage could be added to the middleware so that restarting all nodes running a particular serverless function does not result in data loss.

Future work could extend the list of supported CRDT, for example, by adding a Map. Furthermore, our implementation of set-like data structures currently has a limitation in that they only support string, number, and JavaScript object values due to the complexity of sending arbitrary data types over gRPC and the limited reflection support in TypeScript. Future work could explore the possibility of supporting a wider range of data types as the values for our CRDTs.

Our SDK is currently only available in TypeScript. Future work could develop similar SDKs for other programming languages. Furthermore, it would be interesting to integrate MISO with further open-source serverless platforms.

Finally, our evaluation has limitations in terms of cluster setup, as we simulated multiple nodes using KinD on a single virtual machine. It would be interesting to see future work using MISO in a large-scale cluster with dedicated (virtual) machines for each node to explore its potential further.

APPENDIX A

GitHub Repository

The link to the GitHub repository is: https://github.com/valentingc/miso-serverless-middleware

A.1 Implementation Packages

A.1.1 Package @miso/middleware

https://github.com/valentingc/miso-serverless-middleware/tree/main/packages/ middleware

A.1.2 Package @miso/sdk

https://github.com/valentingc/miso-serverless-middleware/tree/main/packages/sdk

A.1.3 Package @miso/crdt

https://github.com/valentingc/miso-serverless-middleware/tree/main/packages/crdt

A.1.4 Package @miso/common

https://github.com/valentingc/miso-serverless-middleware/tree/main/packages/common

A.2 Evaluation

A.2.1 Complete Helm Chart

https://github.com/valentingc/miso-serverless-middleware/tree/main/setup/benchmark/evaluation/chart

A.2.2 MISO Standalone Helm Chart

https://github.com/valentingc/miso-serverless-middleware/tree/main/setup/miso

A.2.3 MISO Kind Cluster Setup Script

https://github.com/valentingc/miso-serverless-middleware/blob/main/setup/configure_kind_cluster.sh

List of Figures

1.1 Methodology followed in this Thesis 5
1.2 Feature Driven Development Process (own work based on [22], [23]) . . . 6

2.1 State-based vs. Operation-based CRDTs [20] 13
2.2 gRPC Overview [40] 15

4.1 Conceptual Model of MISO Stateful Objects 23
4.2 Architecture of the Middleware 25
4.3 Function Discovery Service Registering 29
4.4 CRDT State Restoring Sequence Diagram 32

5.1 Monolithic Repository Package Structure 40
5.2 mDNS Node Discovery 42
5.3 High-Level Overview of Replication Process 43
5.4 RxJS auditTime Operator (own work based on [51]) 43
5.5 CRDT Initializer Interceptor 44

6.1 AllReduce Operation 53
6.2 AllReduce Results - Grouped Bar Chart 57
6.3 AllReduce Results - Read Times 58
6.4 AllReduce Results - Write Times 58
6.5 Average Replication Time 61
6.6 Replication Time over Time 62
6.7 Average Requests per Second 63
6.8 Requests per Second over Time 64
6.9 Replication Data Volume 64
6.10 Maximum Process Memory Usage 65
6.11 Overview of Serverless Platform Integration Process 67
6.12 Comparison of Lines of Code and Cognitive Complexity 72

List of Tables

4.1 SDK Programming Abstractions 35
4.2 StatefulObjectProxy API 36

5.1 Middleware Folders and Descriptions 40
5.2 SDK Environment Variables 46
5.3 Protobuf Files and Services 50

6.1 AllReduce Experiment Configurations 54
6.2 Impact of Replication Interval on Performance 66

List of Algorithms

4.1 Node Discovery Service 27
4.2 Function Discovery Service 30
4.3 Replication Algorithm 33
4.4 CRDT State Restoring 34

List of Listings

1 Using the SDK in Serverless Functions 37
2 gRPC Protobuf Example from MISO 51
3 Test Script Output 55
4 Modified faas-netes Function Deployment Handler 68
5 Example of Extended Node18 Template 69
6 Code for AllReduce with MISO 71

Glossary

MISO Middleware for Stateful Objects, a name for the software developed in this thesis. It comprises both the middleware itself and an SDK for serverless functions.. xi, xiii, 3, 4, 7, 16, 23–25, 27, 30, 39, 40, 54, 55, 57, 59–63, 65–74, 76, 85

MISO Stateful Object A MISO Stateful Object is a stateful object that can be accessed from within a serverless function handler. It combines one or more CRDT-based data types into a single object.. xi, xiii, 3–7, 18, 19, 23–27, 30, 31, 33–37, 40, 44–46, 53, 62, 63, 67–69, 72–74, 76

Acronyms

API Application Programming Interface. 1, 9, 19, 26, 29, 30, 34–36, 40, 41, 44, 46, 67, 72, 73

CNCF Cloud Native Computing Foundation. 10, 14

CRDT Conflict-free Replicated Data Type. xi, xiii, 3, 4, 6, 7, 12–14, 18–20, 24, 26, 27, 29–31, 33–37, 39, 40, 43–50, 55, 57, 59, 60, 62, 73–76

DF Durable Function. 18

DSR Design Science Research. 5

DWFlag Disable-Wins Flag. 14

EWFlag Enable-Wins Flag. 14, 36, 48

FaaS Function as a Service. 1, 3, 9, 11, 17–19

FDD Feature Driven Development. 6

GCounter Grow-only Counter. 13, 37, 47, 48

GSet Grow-only Set. 36, 49

IoT Internet of Things. 1, 2, 9

IPC Inter-Process Communication. 14

KinD Kubernetes in Docker. 39, 55, 61, 72, 76

MVRegister Multi-Value Register. 13, 36, 48

ORSet Observed-Remove Set. 13, 14, 36, 49

PNCounter Positive-Negative Counter. 13, 48

REST Representational State Transfer. 15, 19

RPC Remote Procedure Call. 14

RPS Requests per Second. xi, xiii, 60, 62–66

SDK Software Development Kit. xi, xiii, 4, 7, 16, 18, 24–28, 33–37, 39, 41, 42, 44–46, 49, 50, 53, 66–76

Bibliography

[1] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, *et al.*, "Serverless edge computing: Vision and challenges", in *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, ser. ACSW '21, New York, NY, USA: Association for Computing Machinery, Feb. 1, 2021, pp. 1–10, ISBN: 978-1-4503-8956-3. DOI: 10.1145/3437378.3444367.

[2] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful serverless computing with crucial", *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 3, 39:1–39:38, Mar. 7, 2022, ISSN: 1049-331X. DOI: 10.1145/3490386.

[3] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS track: Building stateful distributed applications with serverless architectures", in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19, event-place: Davis, CA, USA, New York, NY, USA: Association for Computing Machinery, 2019, pp. 41–54, ISBN: 978-1-4503-7009-7. DOI: 10.1145/3361525.3361535.

[4] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, *et al.*, *Serverless computing: One step forward, two steps back*, Dec. 10, 2018. DOI: 10.48550/arXiv.1812.03651. arXiv: 1812.03651[cs].

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, *et al.*, *Cloud programming simplified: A berkeley view on serverless computing*, Feb. 9, 2019. DOI: 10.48550/arXiv.1902.03383. arXiv: 1902.03383[cs].

[6] N. Shahidi, J. R. Gunasekaran, and M. T. Kandemir, "Cross-platform performance evaluation of stateful serverless workflows", in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 63–73. DOI: 10.1109/IISWC53511.2021.00017.

[7] P. Raith, S. Nastic, and S. Dustdar, "Serverless edge computing—where we are and what lies ahead", *IEEE Internet Computing*, vol. 27, no. 3, pp. 50–64, 2023. DOI: 10.1109/MIC.2023.3260939.

[8] C. Puliafito, C. Cicconetti, M. Conti, E. Mingozzi, and A. Passarella, "Stateful function as a service at the edge", *Computer*, vol. 55, no. 9, pp. 54–64, 2022. DOI: 10.1109/MC.2021.3138690.

[9] L. Baresi and D. Filgueira Mendonça, "Towards a serverless platform for edge computing", in *2019 IEEE International Conference on Fog Computing (ICFC)*, 2019, pp. 1–10. DOI: 10.1109/ICFC.2019.00008.

[10] Y. Harchol, A. Mushtaq, V. Fang, J. McCauley, A. Panda, and S. Shenker, "Making edge-computing resilient", in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, event-place: Virtual Event, USA, New York, NY, USA: Association for Computing Machinery, 2020, pp. 253–266, ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421278.

[11] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: Semantics for stateful serverless", *Proceedings of the ACM on Programming Languages*, vol. 5, pp. 1–27, OOPSLA Oct. 20, 2021, ISSN: 2475-1421. DOI: 10.1145/3485510.

[12] W. Schultz, T. Avitabile, and A. Cabral, "Tunable consistency in MongoDB", *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2071–2081, Aug. 2019, Publisher: VLDB Endowment, ISSN: 2150-8097. DOI: 10.14778/3352063.3352125.

[13] A. Vágner and M. Al-Zaidi, "Sharding and master-slave replication of NoSQL databases: Comparison of MongoDB and redis", in *Proceedings of the 12th International Conference on Data Science, Technology and Applications - Volume 1: DATA*, Backup Publisher: INSTICC, SciTePress, 2023, pp. 576–582, ISBN: 978-989-758-664-4. DOI: 10.5220/0012142700003541.

[14] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo, "Towards scalable and reliable in-memory storage system: A case study with redis", in *2016 IEEE Trustcom/BigDataSE/ISPA*, 2016, pp. 1660–1667. DOI: 10.1109/TrustCom.2016.0255.

[15] R. Shrestha, "High availability and performance of database in the cloud - traditional master-slave replication versus modern cluster-based solutions:" in *Proceedings of the 7th International Conference on Cloud Computing and Services Science*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2017, pp. 413–420, ISBN: 978-989-758-243-1. DOI: 10.5220/0006294604130420. (visited on 05/04/2024).

[16] B. Li, Q. He, F. Chen, *et al.*, "Cooperative assurance of cache data integrity for mobile edge computing", *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4648–4662, 2021, ISSN: 1556-6021. DOI: 10.1109/TIFS.2021.3111747.

[17] P. Lertpongrujikorn and M. A. Salehi, *Object as a service (OaaS): Enabling object abstraction in serverless clouds*, Aug. 5, 2022. DOI: 10.48550/arXiv.2206.05361.

[18] Apache Flink Stateful Functions. "Distributed architecture", [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/docs/concepts/distributed_architecture/ (visited on 04/21/2024).

[19] V. Sreekanti, C. Wu, X. C. Lin, *et al.*, "Cloudburst: Stateful functions-as-a-service", *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2438–2452, Aug. 2020, ISSN: 2150-8097. DOI: 10.14778/3407790.3407836.

[20] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types", in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400, ISBN: 978-3-642-24550-3.

[21] A. R. Hevner, "A three cycle view of design science research", *Scandinavian journal of information systems*, vol. 19, no. 2, p. 4, 2007.

[22] S. Aftab, Z. Nawaz, F. Anwer, *et al.*, "Using FDD for small project: An empirical case study", *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 3, 2019, Publisher: The Science and Information Organization. DOI: 10.14569/IJACSA.2019.0100319.

[23] S. Alsaqqa, S. Sawalha, and H. Abdel-Nabi, "Agile software development: Methodologies and trends", *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 14, no. 11, p. 246, Jul. 10, 2020, ISSN: 1865-7923. DOI: 10.3991/ijim.v14i11.13269. (visited on 04/16/2024).

[24] M. F. Bertoa, J. M. Troya, and A. Vallecillo, "Measuring the usability of software components", *Journal of Systems and Software*, vol. 79, no. 3, pp. 427–439, 2006, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2005.06.026.

[25] A. Zerouali, R. Opdebeeck, and C. De Roover, "Helm charts for kubernetes applications: Evolution, outdatedness and security risks", in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 523–533. DOI: 10.1109/MSR59073.2023.00078.

[26] C. Cicconetti, M. Conti, and A. Passarella, "On realizing stateful FaaS in serverless edge networks: State propagation", in *2021 IEEE International Conference on Smart Computing (SMARTCOMP)*, ISSN: 2693-8340, Aug. 2021, pp. 89–96. DOI: 10.1109/SMARTCOMP52413.2021.00033.

[27] E. Garbarino, "DaemonSets", in *Beginning Kubernetes on the Google Cloud Platform*, United States: Apress L. P, 2019, pp. 239–257, ISBN: 978-1-4842-5490-5.

[28] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance", in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19, event-place: Davis, CA, USA, New York, NY, USA: Association for Computing Machinery, 2019, pp. 37–42, ISBN: 978-1-4503-7038-7. DOI: 10.1145/3366623.3368139.

[29] N. Kratzke, "A brief history of cloud application architectures", *Applied Sciences*, vol. 8, no. 8, 2018, ISSN: 2076-3417. DOI: 10.3390/app8081368.

[30] GitHub. "Openfaas/faas-netes", [Online]. Available: https://github.com/openfaas/faas-netes (visited on 04/08/2024).

[31] N. Sukhija, E. Bautista, O. James, *et al.*, "Event management and monitoring framework for HPC environments using ServiceNow and prometheus", in *Proceedings of the 12th International Conference on Management of Digital EcoSystems*, ser. MEDES '20, event-place: Virtual Event, United Arab Emirates, New York, NY, USA: Association for Computing Machinery, 2020, pp. 149–156, ISBN: 978-1-4503-8115-4. DOI: 10.1145/3415958.3433046.

[32] A. Broniewski, M. I. Tirmizi, E. Zimányi, and M. Sakr, "Using MobilityDB and grafana for aviation trajectory analysis", in *OpenSky 2022*, MDPI, Jan. 10, 2023, p. 17. DOI: 10.3390/engproc2022028017. (visited on 04/13/2024).

[33] GitHub. "Openfaas/faas", [Online]. Available: https://github.com/openfaas/faas (visited on 04/08/2024).

[34] OpenFaaS. "Invocations", [Online]. Available: https://docs.openfaas.com/architecture/invocations/ (visited on 03/27/2024).

[35] GitHub. "Faas-netes/chart/openfaas", GitHub, [Online]. Available: https://github.com/openfaas/faas-netes/tree/master/chart/openfaas (visited on 04/08/2024).

[36] Y. Mao, Z. Liu, and H.-A. Jacobsen, "Reversible conflict-free replicated data types", in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, ser. Middleware '22, New York, NY, USA: Association for Computing Machinery, Nov. 8, 2022, pp. 295–307, ISBN: 978-1-4503-9340-9. DOI: 10.1145/3528535.3565252.

[37] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types", Inria–Centre Paris-Rocquencourt; INRIA, Research Report RR-7506, Jan. 13, 2011, p. 50. [Online]. Available: https://hal.inria.fr/inria-00555588.

[38] A. Bieniusa, M. Zawirski, N. Preguiça, *et al.*, "An optimized conflict-free replicated set", 2012. DOI: 10.48550/ARXIV.1210.3368. (visited on 03/26/2024).

[39] C. Baquero, P. S. Almeida, A. Cunha, and C. Ferreira, "Composition in state-based replicated data types", *Bull. EATCS*, vol. 123, 2017. [Online]. Available: http://eatcs.org/beatcs/index.php/beatcs/article/view/507.

[40] K. Indrasiri and D. Kuruppu, *GRPC: up and running: building cloud native applications with Go and Java for docker and kubernetes*, First edition. Sebastopol, CA: O'Reilly Media, Inc., 2020, OCLC: 1137594283, ISBN: 978-1-4920-5830-4.

[41] A. Fang, R. Zhou, X. Tang, and P. He, "RPCover: Recovering gRPC dependency in multilingual projects", in *ASE*, Issue: ISBN: 9798350329964 Journal Abbreviation: ASE, IEEE, 2023, p. 1939, ISBN: 9798350329964. DOI: 10.1109/ASE56229.2023.00108.

[42] Apache Flink Stateful Functions. "Stateful functions: A platform-independent stateful serverless stack", [Online]. Available: https://nightlies.apache.org/flink/flink-statefun-docs-release-3.2/ (visited on 04/21/2024).

[43] T. Tzu-Li. “Stateful functions internals: Behind the scenes of stateful serverless”. Section: posts. (Oct. 13, 2020), [Online]. Available: https://flink.apache.org/2020/10/13/stateful-functions-internals-behind-the-scenes-of-stateful-serverless/ (visited on 04/21/2024).

[44] C. Jaspan, M. Jorde, A. Knight, *et al.*, “Advantages and disadvantages of a monolithic repository: A case study at google”, in *ICSE-SEIP*, Issue: EISBN: 9781450356596 Journal Abbreviation: ICSE-SEIP, ACM, 2018, p. 234. DOI: 10.1145/3183519.3183550.

[45] npm. “Workspaces”. (Feb. 1, 2024), [Online]. Available: https://docs.npmjs.com/cli/v10/using-npm/workspaces (visited on 04/10/2024).

[46] GitHub. “Nestjs/nest”, [Online]. Available: https://github.com/nestjs/nest (visited on 04/08/2024).

[47] GitHub. “ReactiveX/rxjs”, [Online]. Available: https://github.com/ReactiveX/rxjs.

[48] M. Alabor and M. Stolze, “Debugging of RxJS-based applications”, in *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, ser. REBLS 2020, New York, NY, USA: Association for Computing Machinery, 2020, pp. 15–24, ISBN: 978-1-4503-8188-8. DOI: 10.1145/3427763.3428313.

[49] ReactiveX. “Introduction”, [Online]. Available: https://reactivex.io/intro.html (visited on 04/11/2024).

[50] RxJS. “Introduction”, [Online]. Available: https://rxjs.dev/guide/overview (visited on 04/11/2024).

[51] RxJS. “auditTime”, [Online]. Available: https://rxjs.dev/api/operators/auditTime (visited on 04/11/2024).

[52] Kubernetes. “Downward API”, Kubernetes Documentation. (Jul. 25, 2023), [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/downward-api/ (visited on 04/08/2024).

[53] Redis. “Deploy redis enterprise software for kubernetes”. Section: operate, [Online]. Available: https://redis.io/docs/latest/operate/kubernetes/deployment/quick-start/ (visited on 02/18/2024).

[54] MinIO. “Deploy the MinIO operator”, [Online]. Available: https://min.io/docs/minio/kubernetes/upstream/operations/installation.html (visited on 02/18/2024).

[55] MinIO. “Deploy a MinIO tenant”, [Online]. Available: https://min.io/docs/minio/kubernetes/upstream/operations/install-deploy-manage/deploy-minio-tenant.html (visited on 04/21/2024).

[56] GitHub. “Bojand/ghz”, [Online]. Available: https://github.com/bojand/ghz (visited on 04/08/2024).

[57] GitHub. "Helm-charts/charts/kube-prometheus-stack", [Online]. Available: https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack (visited on 04/21/2024).

[58] L. Lavazza, S. Morasca, and M. Gatto, "An empirical study on software understandability and its dependence on code characteristics", *Empirical Software Engineering*, vol. 28, no. 6, p. 155, Nov. 15, 2023, ISSN: 1573-7616. DOI: 10.1007/s10664-023-10396-7.

[59] G. A. Campbell, "Cognitive complexity: An overview and evaluation", in *Proceedings of the 2018 International Conference on Technical Debt*, ser. TechDebt '18, New York, NY, USA: Association for Computing Machinery, May 27, 2018, pp. 57–58, ISBN: 978-1-4503-5713-5. DOI: 10.1145/3194164.3194186.

[60] GitHub. "Deskbot/cognitive-complexity-TS", [Online]. Available: https://github.com/Deskbot/Cognitive-Complexity-TS (visited on 04/06/2024).