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

Valentin Goronjic
Distributed Systems Group
TU Wien, Austria
publications@valentingc.dev

Stefan Nastic
Distributed Systems Group
TU Wien, Austria
snastic@dsg.tuwien.ac.at

***Abstract*—Serverless functions typically depend on external services to manage the application state, which can be difficult at the Edge due to high latency and network costs. Current solutions for stateful serverless functions at the Edge either have limited support for data locality or require mutual consensus for write operations which is hard to achieve at the Edge. This paper introduces MISO, a novel middleware for serverless computing that enables stateful serverless functions across the Edge-Cloud continuum. The middleware provides MISO Objects offering data locality. It is interoperable with existing serverless platforms and allows concurrent state modifications in a decentralized manner. The main contributions of our work include: i) A novel conceptual model to maintain application state in serverless functions called MISO Objects, ii) MISO middleware and an SDK for serverless functions, and iii) The asynchronous state replication mechanism of MISO Objects using an overlay network to optimize data transfer and resource consumption. Our evaluation demonstrated that MISO outperforms the state-of-the-art by up to 243% in terms of total execution time for AllReduce-type operations. Furthermore, the state replication exhibits O(n) scalability regarding time, throughput, memory usage, and data volume. We further demonstrate that our work can seamlessly be integrated into an open-source serverless platform and that our SDK requires up to 150% fewer lines of code and exhibits up to 75% less cognitive complexity than the state-of-the-art.**

***Index Terms*—serverless, faas, middleware, CRDT, stateful objects, edge-cloud continuum**

I. Introduction

The Function as a Service (FaaS) paradigm has emerged as a popular way to develop applications. They abstract away the complexity of provisioning computational resources from developers by running custom functions in response to events or API calls [1]–[4]. Serverless functions are increasingly used in the Edge-Cloud continuum [1], [2], [5], [6]. However, these functions are stateless and typically depend on remote services to maintain application state [3]. This dependency poses a significant challenge at the Edge, for example, due to latency issues. The characteristics of the Edge, including heterogeneous devices and sites [1], [5] that might fail unexpectedly [7] adds to the complexity managing state in the Edge-Cloud continuum.

To address these issues, research has extended existing serverless platforms with a solution to manage state, such as the Crucial [3] framework and the stateful FaaS platform proposed by Baresi *et al.* [8]. Some solutions are specifically designed for specific FaaS providers, such as Azure Durable Functions [9]. In contrast, other researchers have proposed novel serverless paradigms, like Object as a Service (OaaS) [10], or entirely new FaaS platforms like Cloudburst [11] which incorporate state management.

Many of these solutions operate adjacent to an existing (often cloud-based) FaaS platform and lack direct integration with the serverless platform. As serverless computing is currently not standardized across FaaS platforms [12], [13], this lack of interoperability increases development efforts for application developers. They typically have to manage separate development environments for various FaaS platforms, requiring multiple different tools [12]. Adopting entirely new FaaS platforms, hence, introduces additional complexity for developers due to potentially new abstractions, toolchains, programming models, and development workflows. Moreover, direct integrations facilitate data locality for lower latency, which is often missing in current state-of-the-art solutions.

Traditional storage systems, such as database management systems or key-value stores like MongoDB [14], [15] and Redis [15], [16] can also be used to maintain state in serverless functions. However, they often require mutual consensus for state modifications, which becomes increasingly challenging in a large and distributed Edge-Cloud continuum. To circumvent this issue, other solutions for stateful serverless functions use a centralized queue for state modifications regarding the same entity [9], or output immutable objects instead of modifying existing ones [10]. There is a lack of a solution that provides data locality, interoperability with existing serverless platforms, and facilitates concurrent state modifications in a decentralized manner.

In this paper, we introduce MISO[1], an open-source middleware for serverless computing that is based on Conflict-Free Replicated Data Types (CRDTs). The middleware allows serverless functions in the Edge-Cloud continuum to maintain application state across function invocations. Our middleware introduces MISO Objects, which are locally available on the continuum's devices and nodes that execute the serverless functions. The MISO Objects are specifically designed to support concurrent modification while eliminating the need for a central authority to perform data synchronization and update merging. Instead, the updates are done independently

[1]https://github.com/polaris-slo-cloud/miso

2694-0825/24/$31.00 ©2024 IEEE
DOI 10.1109/IC2E61754.2024.00013

by the MISO Objects in a decentralized manner. Further, our middleware ensures low-latency access to the MISO Objects and offers seamless integration with existing open-source serverless platforms by exposing MISO Objects as well-known data structures. MISO middleware is part of Polaris SLO Cloud[2], a SIG of the Linux Foundation Centaurus project[3], a novel open-source platform for building unified and highly scalable public or private distributed Edge, Cloud, and 3D continuum systems.

The main contributions of this paper include:

1) *MISO Objects*, a novel state management model to access and modify application state within serverless functions. MISO Objects run locally on the nodes that execute serverless functions and can be accessed transparently in serverless functions. Serverless functions concurrently access or modify MISO Objects via Inter-Process Communication (IPC).
2) *MISO middleware*, including the definition of the architecture, core runtime mechanisms, and a Software Development Kit (SDK) for serverless functions to use MISO Objects. It provides data locality and does not depend on a central authority for data synchronization to ensure applicability to the Edge-Cloud continuum. Furthermore, it is designed to be integrated into existing serverless platforms which we demonstrate practically with OpenFaaS.
3) *A state replication and merging mechanism that enables asynchronous replication of MISO Objects* using an overlay network to propagate changes across the system. The overlay network is used to make the replication process more efficient by only replicating to nodes that run the affected serverless function. This ensures that the limited resources in the Edge-Cloud continuum are not being utilized unnecessarily and thus optimizes the algorithm toward data transfer and resource consumption.

We evaluate MISO on a serverless implementation of an AllReduce use case. Our middleware outperforms both Redis Enterprise and MinIO in terms of total execution time. Specifically, it is 26.7% faster than Redis Enterprise and over twice as fast as MinIO. Our experiments also show that MISO's state replication algorithm exhibits $O(n)$ scalability with respect to the required replication time, throughput, process memory usage, and data volume. We further demonstrated the middleware's seamless integration capabilities with an existing open-source serverless platform, OpenFaaS. This integration required only little modifications to the function deployment handler of OpenFaaS to set environment variables. Moreover, our SDK required up to 150% fewer lines of code and exhibited up to 75% less cognitive complexity compared to the state-of-the-art. This suggests that our middleware not only enhances performance but also contributes to reducing the code complexity.

[2]https://polaris-slo-cloud.github.io/

[3]https://www.centaurusinfra.io/

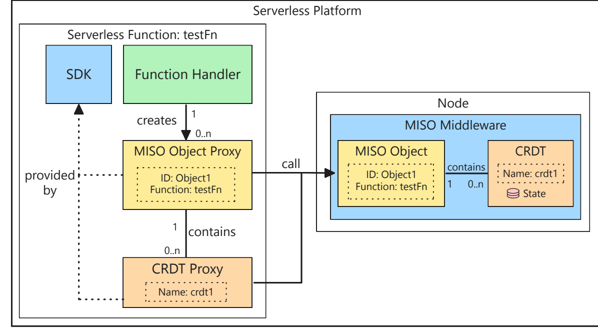

Figure 1. Conceptual Model of MISO Objects

II. MISO's Conceptual Model and Architecture

A. MISO Objects

Figure 1 depicts the conceptual model of MISO Objects. They are stateful objects that serverless functions can use to maintain the application state. One serverless function can create multiple such objects. MISO Objects bundle multiple CRDT-based data types into a single object. MISO Objects are *distributed*, as their state is replicated across all nodes that run the same serverless function. This is because their underlying data types, CRDTs, are designed to be replicated. This provides data locality to serverless functions. MISO Objects can be *modified concurrently and independently* by serverless functions, and MISO does not limit the function concurrency on a single node. Coordinating state modifications with other MISO Objects or serverless functions is not required. Furthermore, due to the utilized data types, MISO Objects are *eventually consistent* across multiple nodes. On a per-node level, MISO provides a read-your-writes consistency level. The asynchronous state replication of MISO disseminates updates to all nodes running the same serverless functions, which then merge the state updates with their local states. MISO Objects, therefore, eventually converge to the same value across nodes. The merging of the state happens at the data-type level, and the algorithms are built into the data types themselves.

Every MISO Object is identified by an ID, which can be set manually or automatically generated. As MISO Objects are linked to a serverless function, the ID only needs to be unique for each serverless function. Multiple replicas of the same serverless function can share the same MISO Object, even if they are executed on different nodes. Every CRDT-based data type within a MISO 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 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.

As serverless functions are stateless, they alone cannot maintain the state of MISO Objects across function invocations. For this reason, the lifecycle of MISO Objects is managed by the MISO middleware. This includes creating, re-

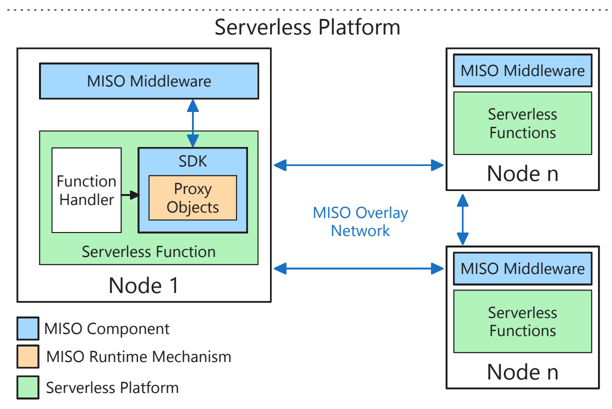

Figure 2. Overview of MISO Midldleware Architecture

trieving, modifying, and replicating the state of MISO Objects. To access and modify MISO Objects from serverless functions, the MISO SDK provides proxy versions of the MISO Objects and the CRDT-based data types they encompass. The proxies can be retrieved from the MISO Object proxy object, and the configuration process of the proxies so that the correct middleware instance can be invoked is transparent to developers. The proxy versions of the CRDT-based data types behave like regular data types but internally call the middleware whenever an operation is executed against them. The middleware ensures that the data of the MISO Objects is available on every node of the serverless platform that runs this serverless function by replicating the state asynchronously.

To demonstrate the concept further, we now discuss a hypothetical example of a serverless function dealing with IoT sensor data readings of a smart car with the id *12345*. The function accesses a single MISO Object with the ID *TripData-12345*, where *12345* stands for the car's ID. This MISO Object contains two CRDT-based data types: i) a Counter with the name *distanceDriven* which stores the total distance driven during the trip, and ii) a Set with the name *alerts* which contains all extraordinary events that arise during the current trip (e.g., low fuel). The serverless function could access other MISO Objects with different IDs to maintain other sensor readings, such as engine parameters. This logical separation and grouping of data types into MISO Objects offers flexibility to developers and allows for a clear separation of concerns.

B. Middleware Architecture

Figure 2 provides an overview of MISO's architecture. The system is designed to be modular and flexible enough to be integrated into 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 Objects are located in close proximity to the serverless functions. The middleware is responsible for managing MISO 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 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.

The middleware is combined with an SDK for serverless functions. It enables serverless functions to use the middleware and provides proxy versions of MISO Objects and the data types they contain. Developers can use the data types as if they are 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 II-C. 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 2. This network is mainly used for the replication of updates whenever a MISO Object is modified. To make the process of sending updates more efficient, the overlay network needs to provide information, such as on which nodes a particular function is currently being executed.

Figure 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 for readability. 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 MISO-specific 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 IV.

C. MISO 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

```javascript
// function handler of the serverless function
module.exports = async (event, context) => {
    const so = context.statefulObject;
    const counter = so.getGCounter('totalSum');
    await counter.add(1);
    return { data: await counter.getValue() };
};
```

Listing 1. Example Usage of MISO SDK to develop Serverless Function (OpenFaaS)

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 FaaS platforms that support a runtime for serverless functions in which the SDK is written.

1) API and Programming Model: There are two key abstractions that are exposed to developers of serverless functions: `MisoObjectProxy` and `CRDT Proxies`.

The `MisoObjectProxy` is the most important programming abstraction of the SDK and is responsible for the following tasks:

1) Providing serverless functions the functionality to create and modify proxy versions of MISO Objects, which is transparent to developers
2) Abstracting away the complexity of communicating with the MISO middleware, where the lifecycle of the MISO Objects is managed
3) Providing mechanisms to register and unregister serverless functions with the overlay network of the MISO middleware. This is necessary so the overlay network can properly provide the information required in the replication process.
4) Providing instances of proxies of CRDT-based data types that can be accessed via MISO Objects. They behave like regular data types but transparently proxy the operations to the middleware.

`CRDT Proxies` provide a proxy instance of a given CRDT. They can be accessed via the MisoObjectProxy and behave like regular data types, but internally the operations are delegated to the middleware. This is because the middleware manages the lifecycle and state. The proxies can purposely not be instantiated without the `MisoObjectProxy`, as otherwise, developers would have to manually configure the details on how these data types communicate with the middleware. The list of supported CRDT-based data types currently includes:

1) Enable-Wins Flag (EWFlag),
2) Grow-only Counter (GCounter),
3) Positive-Negative Counter (PNCounter),
4) Grow-only Set (GSet),
5) Multi-Value Register (MVRegister), and
6) Observed-Remove Set (ORSet).

2) Usage in Serverless Functions: In this Section, we describe an example of how the SDK can be used in the code of serverless functions.

Listing 1 gives an example of using the SDK in a JavaScript-based serverless function for OpenFaaS. In this case, a MISO Object is injected into the *context* argument of the serverless function, as we integrated MISO with the serverless platform as described in Section IV-C1. The serverless function then creates a GCounter, increments it by 1, and returns the current value.

III. Runtime Mechanisms

A. Constructing MISO Overlay Networks

All instances that run MISO create an internal overlay network, consisting of a node- and function discovery service. The overlay network is required for the replication module and has two main tasks. The first 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 middleware can efficiently replicate MISO Objects, making sure only nodes that actually run replicas of the same serverless function receive the updates. This increases the efficiency of the replication process, saving bandwidth and processing power compared to replicating updates to all discovered nodes.

The *node discovery* service is responsible for discovering other nodes that run the MISO middleware. To accomplish this, 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 an extended period. We provide a practical implementation that relies on mDNS, but there could be other strategies in the future.

The function discovery service provides the replication part of the middleware a list of nodes that need to receive the update whenever MISO Objects are modified. To achieve this, serverless functions need to register and unregister with the overlay network. This can be done via the MISO SDK and is transparent to developers of serverless functions. The function discovery service stores a list of serverless function replicas for every previously discovered node.

B. Replication and Merging of MISO Objects

MISO disseminates modifications to MISO Objects 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 1. Whenever the state of a MISO Object is modified, this data eventually needs to be replicated (line 2). To achieve this, an update is generated for every such modification, which is then wrapped in a *ReplicationTask* (lines 3-4). This task is then queued in the *ReplicationService* and replicated asynchronously after a

Algorithm 1: State Replication Algorithm

```
   input : fds // FunctionDiscoveryService
   input : object // MISO Object
   input : crdt // CRDT that is modified
   input : fnName // 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     task = crdtTasks.getLatestTask()
 8     targetNodes = fds.getReplicationTargets(fnName)
       // 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
```

Algorithm 2: Restoring State of MISO Objects

```
   input : fds // FunctionDiscoveryService
   input : object // MISO Object
   input : crdtType // Type of modified CRDT
   input : crdtName // Name of modified CRDT
   input : fnName // 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
```

configurable delay (lines 5-6). The replication module must ensure that exactly one replication happens in the configured replication interval, as long as the data type within a MISO Object is being modified. Our algorithm always uses the latest queued task *for each CRDT within a MISO Object* (line 7). This is because MISO uses state-based CRDTs, where every state update carries the entire state. Afterward, the overlay network is utilized to determine which nodes need the modified data. The function discovery service knows which discovered nodes run the same serverless function (line 8). 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. In case the network transmission fails, the request is retried on the network level (line 13). 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 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 (line 11). 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 (line 17). This function must be associative, commutative, and idempotent, as this is a requirement of state-based CRDTs [17]. The replication algorithm described in Algorithm 1, 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.

C. Restoring State from MISO Objects

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. Algorithm 2 describes the process of restoring MISO Object states. When a serverless function modifies a CRDT of a MISO Object via the SDK, the middleware checks if the node running this serverless function locally has such a CRDT (line 3). If true, the requested operation is executed without initializing the state. If the data is not present locally, the overlay network provides a list of nodes running replicas of the same serverless function (line 4). Every node in the list is then asked for the current state of the CRDT (lines 5-11), and the state of the first node that answers is then initialized (lines 12-15). 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 initialization process is transparent to developers of serverless functions when they modify MISO Objects (line 17).

IV. Evaluation

In this section, we present a multifaceted evaluation of MISO that encompasses both quantitative and qualitative aspects. We start with studying the performance overhead of MISO (Subsection IV-A), followed by examining the replication algorithm in more detail (Subsection IV-B). We conclude with a qualitative evaluation of MISO's usability and interoperability with an existing serverless platform (Subsection IV-C).

A. Performance Overhead

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

1) Experiment Definition: The objective of this experiment is to reduce a numeric array to a single number as fast as possible. At the beginning, an array of numbers is generated and split into multiple chunks. The serverless function then sums up the chunk to an intermediate sum and stores this information. When all intermediate results have been calculated, they are reduced to a single number. The serverless function is, hence, called *n+1* times (*n* times for writing intermediate results, *1* time to retrieve the overall result). 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). The usage of those well-known solutions for state management enabled us to use the same serverless platform and function for all software systems, with the only difference being the utilization of a different state management solution in the serverless function code.

2) Experiment Results: Figure 3 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. We ran all solutions in a cluster of 5 nodes except MISO which we

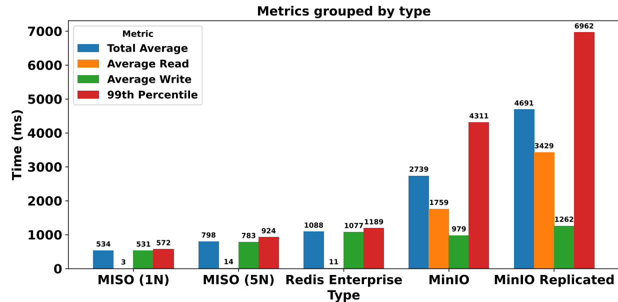

Figure 3. AllReduce Results - Grouped Bar Chart

additionally ran in a single-node cluster for a baseline measure. A detailed analysis of the results is presented in the subsequent sections.

a) MISO: As visible in Figure 3, the measured average read time (14ms) and write time (783ms) for MISO are lowest for all three utilized solutions. 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. However, this also means that the serverless function does not have to manually compute the intermediate sums, as this is essentially built into the data type itself. The difference in read times between our solution and Redis Enterprise in this setting is marginal. The average read and write times of MISO are significantly lower than what we have measured with MinIO. This negligible difference underscores the efficiency of our middleware in this use case.

b) Redis Enterprise: As visible in Figure 3, the read times for Redis Enterprise in this use case were 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 3, was 1077ms, which is almost 300ms higher than what we have measured for MISO.

c) MinIO: As depicted in Figure 3, the observed average read and write times in this experiment for MinIO were substantially higher than those of MISO and Redis Enterprise. The read-time average was 1759ms in a single cluster and 3429ms in a replicated cluster. The write-time average was 979ms in a single cluster and 1262ms in a replicated cluster. The times of creating the bucket in the beginning and removing the files and bucket after each test run were not taken into account in our measure.

d) 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-

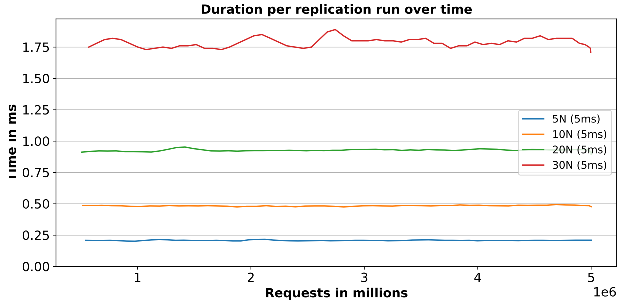

Figure 4. Replication Time over Time

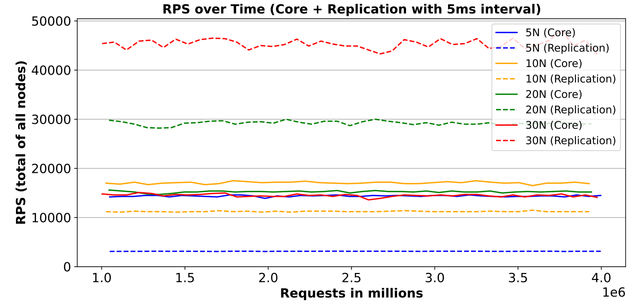

Figure 5. Requests per Second over Time

based data structures offer great performance for use cases that can utilize their potential, as our presented AllReduce use case.

B. 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 RPS (i.e., the sum of all MISO-related RPS on all participating nodes),
3) Replication Data Volume,
4) Process Memory Usage.

1) Experiment Definition: We perform a stress test on the middleware to evaluate the replication algorithm. As the prototype implementation of the middleware exposes its operations through gRPC, we use an open-source gRPC benchmarking tool for this [18]. The benchmark repeatedly calls a gRPC endpoint with multiple threads. We increase a PNCounter concurrently and then study the metrics of the replication algorithm with varying nodes. Our test simulates concurrent data modification on all participating nodes. We have used 10 concurrent threads and performed 5 million increases to a PNCounter with a replication interval of 5ms.

2) Experiment Results:

a) Average Replication Time: Figure 4 depicts how the replication time changed over time during the experiment. It 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. 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. The total average replication times were 0.22 ms, 0.51 ms, 0.97 ms, and 1.83 ms for 5,10,20, and 30 nodes, respectively.

b) Requests per Second: Figure 5 demonstrates how the total RPS rate changed during the experiment execution time with 5 million total requests. The total rate consists of the core requests that modify a CRDT and the replication requests to propagate state modifications across the system. The average core RPS rate is depicted in a solid line in

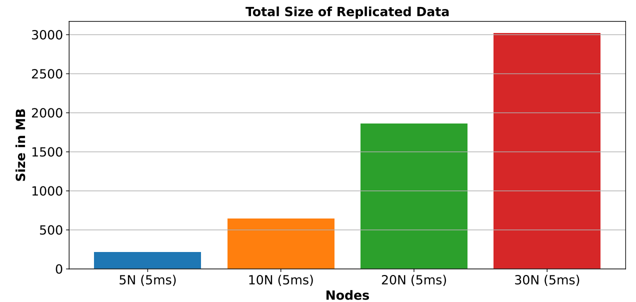

Figure 6. Replication Data Volume

Figure 5 was between 13 800 and 16 300 core requests per second, depending on the cluster size. The replication requests, depicted in a dashed line in Figure 5, rose from approximately 3 100 to 43 300 RPS with 5 and 30 nodes, respectively. This is an increase in replication requests by approximately 1297% while the cluster size increased by 600%. Our results indicate that MISO can handle increasing workloads. We leave out the initial and last 1 million requests to show results with equal load and without initial connection setup.

c) Replication Data Volume: Figure 6 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. 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.

d) Process Memory Usage: Figure 7 shows the maximum observed process memory usage of the middleware during the experiment, including all components. The maximum memory usage for 5 nodes was lowest, with 216 MB. This has increased to 251 MB with 30 nodes. 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.

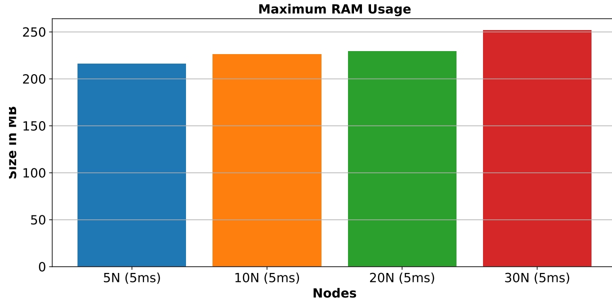

Figure 7. Maximum Process Memory Usage

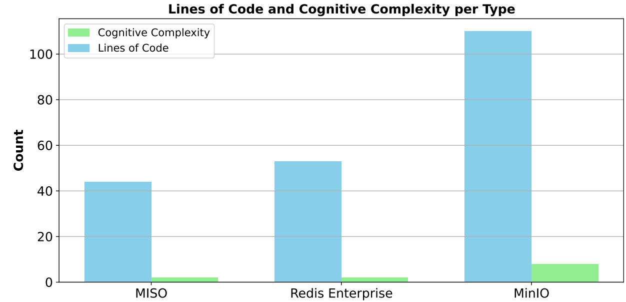

Figure 8. Comparison of Lines of Code and Cognitive Complexity

Table I. 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

e) Impact of Replication Interval: Table I shows 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. We have directed all requests to a single MISO node in a cluster of 20 nodes, which replicated the modifications to the other 19 nodes using a varying replication interval. Our results show that the configured replication interval heavily influences the performance of the middleware and the replication process. The higher the replication interval is set (i.e., less frequent updates), the higher the RPS scores are, and the request latency for core middleware operations decreases. Starting from a replication interval of around 200ms, further increases in the replication delay had a negligible impact on the overall performance. 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.

C. 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.

1) Integrability: This section demonstrates the process of integrating both the middleware and SDK with OpenFaaS. The principle of how we integrated the middleware and SDK also applies to other serverless platforms and programming languages.

a) Integrating the Middleware: The necessary steps to integrate the middleware with OpenFaaS can be summarized as follows:

1) Adding the MISO middleware to the Helm chart template of the OpenFaaS provider.
2) Setting environment variables in the function deployment handler of the faas-netes provider by utilizing the Kubernetes Downward API (node name and IP address, serverless function name).
3) Building the provider image locally and changing the image pull policy so that the updated image is used.

b) Integrating the SDK: To integrate our SDK with OpenFaaS, we have added it as a dependency to the serverless function template for NodeJS. In the template, we also register the serverless function with the overlay network of the MISO middleware before the function is executed. This means that developers of serverless functions do not have to deal with registering/unregistering the serverless function with our middleware, which contributes to usability.

2) Usability: A major part of software development cost is poor code understandability [19]. This is because inspecting and maintaining poorly understood code is hard, and a lot of time is spent there. Refactoring hard-to-understand code sections improves maintainability.

In this section, we use two metrics to measure the *understandability* of MISO's SDK. The first is *Lines of Code*, a widely used traditional code measure. The second is *Cognitive Complexity*, a newer metric introduced in 2018 [19]. Campbell [20] describe how the score is calculated.

We recall the experiment previously mentioned in Section IV-A. In this experiment, we have utilized three different state-of-the-art 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.

Figure 8 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 measured for MISO and Redis Enterprise.

D. Threats to Validity

The analysis of the performance overhead and replication algorithm was performed on a KinD[4] cluster running on a Ubuntu 22.04.3 LTS VM, powered by an AMD EPYC 7742 processor with 64 cores and 128 GB of RAM. This simulation of nodes enabled us to evaluate MISO in different settings. Still, this setting does not fully replicate the same conditions as if we used distinct (virtual) machines. We did not simulate an artificial network latency in our experiments.

V. Related Work

The literature mentions multiple proposals for stateful serverless functions. They can be categorized according to:

a) Data Locality: In terms of where data is maintained, some research approaches either provide data locally to serverless functions or, alternatively, rely on remote storage services. Cloudburst [11] is a novel stateful serverless platform that relies on caching on the nodes executing serverless functions to provide local and low-latency access to frequently used data stored in a remote key-value store. On the other hand, the *Crucial* [3] framework, the *Object as a Service* [10] paradigm introduced, and Apache Flink Stateful Functions [21] manage the state separately from the node running the serverless function. Shahidi *et al.* [22] 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. Our work maintains the data of MISO Objects locally on the nodes that execute the serverless functions. This means that the data of MISO Objects can therefore be retrieved from the same local node that also runs the serverless function via IPC, which is beneficial for latency.

b) Interoperability: Other solutions for stateful serverless functions either extend existing serverless platforms or propose entirely new stateful serverless platforms. Crucial [3] is a framework that works with FaaS platforms that offer a Java runtime and an API to upload/call serverless functions. Baresi *et al.* [8] propose a prototype stateful serverless platform for the Edge that extends OpenWhisk, an existing serverless platform. Certain solutions, such as Durable Functions [9], are tailored to a specific serverless platform and not generalizable to multiple serverless platforms. Cloudburst [11] is a novel serverless platform that provides state management. Similarly, Lertpongrujikorn *et al.* [10] have introduced a new paradigm called Object as a Service (Oaas) to manage state for serverless functions, and their prototype is based on top of Knative. [6] propose a system model of a new stateful FaaS platform tailored to the Edge. New stateful serverless platforms and paradigms provide state management for serverless functions, but they come at a cost as they potentially introduce new programming models, data structures, or toolchains. This contributes to an increased complexity in the learning curve experienced by developers. Similarly, a solution for stateful serverless functions should ideally work for multiple platforms. MISO is deliberately designed in a way that it interoperates with existing open-source serverless platforms. Our work does not rely on any particular component of the FaaS platform besides setting environment variables in the serverless function containers. This adds to the independence of MISO and is a fundamental design decision to enable interoperability with various serverless platforms.

c) Concurrent State Modification: Many solutions for managing application states (e.g., key-value stores or database management systems) require mutual consensus when modifying the state. This can, for example, be achieved by electing primary nodes for writes with consensus protocols. Examples of this are MongoDB [14], [15] and Redis [15], [16], where writes are only possible against primary nodes [16], [23]. Burckhardt *et al.* [9] propose a solution for stateful serverless functions that uses use a centralized queue to sequentially perform writes regarding the same entity [9]. Similarly, in [8] the authors propose that all requests belonging to a certain session are routed to the same container instance which mitigates the need to replicate data across nodes, but therefore limits horizontal scaling of the serverless function across different nodes. All these proposed solutions work best when there are stable nodes that do not fail, which might not always be the case in the Edge-Cloud continuum [1], [7]. MISO Objects relax the requirement for mutual consensus for state modifications and work in a decentralized way, offering automatical reconciliation in case of conflicting updates due to their semantics. 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 [17]. MISO Objects do not require the coordination of state modifications between nodes and can be modified independently by serverless functions.

Acknowledgment

This work is sponsored by the Austrian Research Promotion Agency (FFG), under the project No. 903884.

VI. Conclusion

In this work we have presented MISO, a CRDT-based serverless middleware for the Edge-Cloud continuum. It provides MISO Objects for serverless functions, which combine multiple CRDT-based data types into a single object. The objects are accessed from serverless function handlers using proxies, and the state and lifecycle are managed by the middleware instance that runs on the node executing the serverless function. The middleware is designed to be integrated into

[4]https://kind.sigs.k8s.io/

multiple open-source serverless platforms, provides data locality, and does not depend on a central authority for data synchronization.

We evaluated MISO's performance in an AllReduce operation, where the total experiment time was 26.7% lower than with Redis Enterprise and almost 2.5 times lower than with MinIO. We have also performed an in-depth assessment of the replication algorithm, where we demonstrated that the replication process exhibits $O(n)$ scalability with respect to the required replication time, throughput, process memory usage, and data volume. The integration with OpenFaaS, a popular serverless platform, was seamless and required only minimal code changes to provide MISO-specific environment variables to serverless functions. Finally, our SDK required up to 150% fewer lines of code and exhibited up to 75% less cognitive complexity than the other state-of-the-art.

In the future, we intend to extend our work in several directions. MISO Objects are currently only stored in memory. While this makes it fast to access their data, this also means that there is the potential for data loss. In the future, we plan to address this by exploring the possibility of persisting MISO Objects locally on the nodes that run the MISO middleware while retaining the benefits of CRDTs. Additionally, the data types exposed by the SDK do not exactly match the typical data types offered by programming languages. This is because there is no one-to-one mapping of conventional data types and CRDTs. In the future, we aim to explore how we can offer native interfaces for common data types while still using the data types provided by MISO Objects. We envision that overcoming this challenge includes combining multiple CRDTs into new data types. We also intend to transform MISO's SDK into a stand-alone programming model and combine it with our previous efforts in the field (e.g., [24]) to better support the development of large-scale, pervasive, serverless applications. Lastly, we aim to investigate the usage of δ-CRDTs [25] to improve network utilization.

References

[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] S. Nastic, S. Dustdar, R. Philipp, F. Alireza, and T. Pusztai, "A serverless computing fabric for edge & cloud", in *4th IEEE International Conference on Cognitive Machine Intelligence (CogMi)*, 2022. DOI: 10.1109/CogMI56440.2022.00011.

[3] 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.

[4] S. Nastic, "Self-provisioning infrastructures for the next generation serverless computing", *SN Computer Science*, vol. 5, no. 6, pp. 678 –693, 2024. DOI: 10.1007/s42979-024-03022-w.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] P. Castro, V. Isahagian, V. Muthusamy, and A. Slominski, "Hybrid serverless computing: Opportunities and challenges", in *Serverless Computing: Principles and Paradigms*, R. Krishnamurthi, A. Kumar, S. S. Gill, and R. Buyya, Eds. Cham: Springer International Publishing, 2023, pp. 43–77, ISBN: 978-3-031-26633-1. DOI: 10.1007/978-3-031-26633-1_3.

[13] M. Kumar, "Serverless architectures review, future trend and the solutions to open problems", *American Journal of Software Engineering*, vol. 6, no. 1, pp. 1–10, 2019, ISSN: 2379-528X. DOI: 10.12691/ajse-6-1-1. [Online]. Available: http://pubs.sciepub.com/ajse/6/1/1.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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).

[22] 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.

[23] 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.

[24] S. Sehic, F. Li, S. Nastic, and S. Dustdar, "A programming model for context-aware applications in large-scale pervasive systems", in *Proceedings of the IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2012)*, IEEE Computer Society, 2012, pp. 142–149, ISBN: 978-1-4673-1428-2.

[25] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based CRDTs by delta-mutation", in *Networked Systems*, A. Bouajjani and H. Fauconnier, Eds., Cham: Springer International Publishing, 2015, pp. 62–76, ISBN: 978-3-319-26850-7.