IFIP Networking 2024 1570997126
2024 IFIP Networking Conference (IFIP Networking)

Deduplicator: When Computation Reuse Meets Load Balancing at the Network Edge

Md Washik Al Azad
University of Notre Dame
malazad@nd.edu

Spyridon Mastorakis
University of Notre Dame
mastorakis@nd.edu

***Abstract*—Load balancing has been a fundamental building block of cloud and, more recently, edge computing environments. At the same time, in edge computing environments, prior research has highlighted that applications operate on similar (correlated) data. Based on this observation, prior research has advocated for the direction of "computation reuse", where the results of previously executed computational tasks are stored at the edge and are reused (if possible) to satisfy incoming tasks with similar input data, instead of executing incoming tasks from scratch. Both load balancing and computation reuse are critical to the deployment of scalable edge computing environments, yet they are contradictory in nature. In this paper, we propose the *Deduplicator*, a middlebox that aims to facilitate both load balancing and computation reuse at the edge. The *Deduplicator* features mechanisms to identify and deduplicate similar tasks offloaded by user devices, collect information about the usage of edge servers' resources, manage the addition of new edge servers and the failures of existing edge servers, and ultimately balance the load imposed on edge servers. Our evaluation results demonstrate that the *Deduplicator* achieves up to 20% higher percentages of computation reuse compared to several other load balancing approaches, while also effectively balancing the distribution of tasks among edge servers at line rate.**

***Index Terms*—Edge Computing, Computation Reuse, Load Balancing, Middleboxes**

I. Introduction

Middleboxes have been proposed and widely deployed over the years across both private and public networks [1], [2]. Among the most widely deployed middleboxes are load balancers, which are fundamental components of creating scalable web services and networked environments, such as cloud data centers [3]. With the deployment of applications that require ultra low user-perceived latency and produce massive volumes of data (*e.g.,* augmented and virtual reality applications, smart city applications), edge computing has emerged as a paradigm that brings computing resources (small-scale data centers) physically close to user devices and applications.

In edge computing environments, prior studies have highlighted that applications operate on temporally, spatially, and contextually similar data [4], [5]. In other words, applications at the edge may request the processing of similar data by offloading computational tasks with similar input data for the same services (*e.g.,* image or video annotation). The execution of such tasks often yields the same output/outcome (execution results), resulting in the execution of duplicate (redundant) computation. To eliminate duplicate computation and reduce task completion times, the direction of "computation reuse" has been proposed, where edge servers store the results of executed tasks to reuse them and satisfy incoming similar tasks (tasks with similar input data for the same services) [6], [7].

In line with these observations, in this paper, we argue that both load balancing and computation reuse are vital functions for the realization of scalable edge computing environments. Nevertheless, these functions are contradictory in nature. On the one hand, load balancing aims to distribute the load equally among the available edge servers. On the other hand, computation reuse aims to distribute tasks with similar input data to the same edge server(s), which causes load imbalances, since: (i) tasks with similar input data are not uniformly generated at the edge (*e.g.,* certain applications/services may be more popular than others); and (ii) due to (i), certain edge servers may receive and execute more tasks than others.

In this paper, we shed light into the following **research questions**: "is it possible to achieve both the, yet contradictory in nature, functions of load balancing and computation reuse in edge computing environments? If so, to what extent and what are the tradeoffs between these functions?". To answer these questions, we propose the *Deduplicator*, a middlebox that aims to facilitate the reuse of computation, while at the same time balancing the load imposed on edge servers. Our work makes the following contributions:

- We present the *Deduplicator* middlebox design, which takes advantage of Locality Sensitive Hashing (LSH) [8] to identify and deduplicate similar computational tasks offloaded by user devices. It also features mechanisms to collect information about the usage of edge servers' resources, manage the addition of new edge servers and the failures of existing ones, and ultimately balance the load imposed on edge servers.
- We implement a *Deduplicator* prototype on top of Nginx, a popular open-source web server, HTTP proxy, and load balancing framework [9]. Our experimental evaluation demonstrates that the *Deduplicator* prototype identifies and deduplicates similar offloaded tasks with minor performance overhead, being able to distribute them to available edge servers at line rate. In addition, it is able to effectively balance the tradeoffs between load balancing and computation reuse at the edge, facilitating the realization of both functions.

II. Motivation and Prior Work

Let us consider a multi-player Augmented Reality (AR) gaming use case to showcase the necessity of *Deduplicator* as a middlebox to facilitate both computation reuse and load balancing in edge computing environments. As we illustrate in Figure 1, user 1 offloads two similar subsequent

ISBN 978-3-903176-63-8 ©2024 IFIP

978-3-903176-63-8/24/$31.00 ©2024 IEEE

frames/snapshots (1a and 1b) captured by an AR headset to edge servers, and these frames are distributed to two different servers. In this case, it is not possible to reuse the results of the first frame's processing to satisfy the processing of the second frame, thus both frames need to be processed from scratch. To achieve computation reuse, we need mechanisms to introduce reuse awareness in the task distribution process, so that tasks with similar inputs are distributed to the same edge server(s).

Let us further consider the case of users who play the AR game in an collaborative environment (*e.g.,* users 1, 2, and 3 in Figure 1). Users can meet at a physical location in the real world, where they interact with virtual objects in a collaborative manner (*e.g.,* Pokemon Go application). The frames offloaded by these players are similar to each other, since they depict the same physical location from different angles/distances. As such, the corresponding tasks should be distributed to the same edge server to facilitate computation reuse (*e.g.,* tasks 1a, 2, and 3 in Figure 1). Distributing similar tasks to the same edge server(s) for a wide spectrum of applications may overload certain servers, while leaving others underutilized, thus causing load imbalances among servers. This signifies the need in edge computing environments to facilitate computation reuse, while at the same time balancing the load imposed on edge servers. To this end, in this paper, we propose the *Deduplicator*, a middlebox that aims to simultaneously achieve these functions.

How is the *Deduplicator* different than traditional load balancing: Several load balancing techniques have been proposed over the years. These techniques do not consider the similarity between the input data of tasks, thus they do not facilitate computation reuse during the distribution of tasks towards servers. For example, load balancing based on consistent hashing [10] provides a session-persistent connection between a user and a server by hashing the header fields of user requests. A naive solution could be to replace the hash function used (consistent hashing) with LSH and hash the task input data to forward tasks with similar inputs to the same server(s). Although this approach increases the probability of reuse, as our evaluation results show in Section IV-C, it will also cause severe load imbalances among servers. On the other hand, the *Deduplicator* provides mechanisms to simultaneously facilitate computation reuse and balance the load among edge servers.

III. *Deduplicator* Design

A. Design Overview

We assume the existence of small-scale data centers at the edge, called "cloudlets" [11]. A cloudlet consists of heterogeneous servers with computing and storage resources. Each server may offer a set of services (*e.g.,* object recognition, face detection) to users or execute the code of processing functions offloaded directly by user devices.

The *Deduplicator* operates as a middlebox in edge computing environments, which is placed in front of available edge servers (Figure 1). It aims to achieve the following **functions**: **(i) balance the load among available edge servers**; and **(ii) facilitate the reuse of computation**. In the context of (i), the *Deduplicator* distributes computation tasks among the available edge servers in ways that equalize the load among them. In the context of (ii), the *Deduplicator* distributes similar tasks to the same edge servers, enabling servers to reuse the results of tasks that they have previously executed.

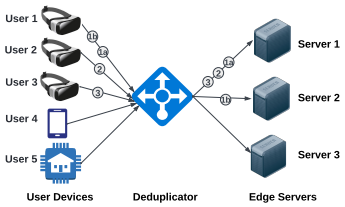


Fig. 1: An edge computing environment where the *Deduplicator* receives tasks offloaded by user devices and distributes them to edge servers.

User devices offload computation tasks as HTTP/HTTPS POST requests, which identify the requested services (in the URL field) and carry the input data (in the request body). In the case of HTTPs, a user device first creates a TLS session with the middlebox so that the requested service and input data can be extracted. The *Deduplicator* intercepts HTTP/HTTPS POST requests and makes use of Locality Sensitive Hashing (LSH) to identify requests for similar computation, which can be reused by edge servers. To achieve that, the *Deduplicator* slices the space of LSH values and distributes (assigns) these slices among edge servers. Each edge server becomes responsible for executing tasks with input data hashed to values that fall in the slice assigned to it. The *Deduplicator* collects information about the usage of server resources over time and redistributes the hash value space among edge servers, so that the load is balanced while facilitating computation reuse. Finally, it features mechanisms to identify newly added servers and server failures, and redistributes the hash value space to achieve its primary functions.

B. Task Offloading and Execution Process

Tasks are created by user devices and are offloaded to edge servers. Each task is represented as an HTTP/HTTPS POST request. In the URL field, the URL of the requested service is included. The input data to be processed is included in the request's body. Each task also carries a similarity threshold, so that applications can indicate the minimum threshold of similarity with tasks previously executed by edge servers, which would be acceptable for reuse. We discuss how an appropriate similarity threshold can be determined for an application below. An offloaded task will be received by the *Deduplicator*, which is placed in front of available edge servers and distributes the task towards one of these servers based on the locality sensitive hash of the task's input data. This hash can be produced based on one of the following mechanisms:

- **Hashing by the *Deduplicator*:** The *Deduplicator* receives an offloaded task and hashes its input data through LSH. This mechanism is fully transparent to users, since LSH is performed by the *Deduplicator*. However, this could incur

overhead on the *Deduplicator*, which aims to distribute offloaded tasks at line rates.

• **User assisted hashing:** Users apply LSH to the task input data and attach the hash value to the task. This mechanism requires user involvement, however, it does not require the *Deduplicator* to apply LSH to the task input data.

As we mentioned in Section III-A, the *Deduplicator* slices the space of potential LSH values and assigns these slices among edge servers. In other words, each edge server becomes responsible for executing tasks with input data hashed to values that fall in the slice assigned to it. For example, as we explain in Section III-C and illustrate in Figure 2a, the first edge server S_1 is responsible for executing tasks with input data hashed to values 0 - 21844. Once the *Deduplicator* receives a task, it will produce the locality sensitive hash of the task's input data (or extract the hash attached to the task) and distribute the task based on the resulting hash value. For example, if the hash value of a task's input data falls under the slice of S_1, the task will be distributed to S_1.

Once an edge server receives an offloaded task, it will search for previously executed similar tasks that have been stored by the server. If such a task is found, its execution results are returned to the user device that offloaded the task. For the selection of a previously executed similar task, edge servers will make use of the similarity threshold attached to tasks. Specifically, in response to a new offloaded task t_{new}, edge servers will select and return a stored task t_{stored} for the same service as t_{new}, so that: (i) the input data of t_{stored} has the highest similarity (nearest neighbor) among stored tasks to the input data of t_{new}; and (ii) the similarity between the input data of t_{stored} and t_{new} exceeds the similarity threshold that is acceptable by the application that offloaded t_{new}. If there is no previously executed task that satisfies these conditions, t_{new} will be executed from scratch and its execution results will be stored by the server for reuse in the future.

Deciding on appropriate similarity thresholds: For a specific application, a developer or a service provider can select an approximate initial similarity threshold value based on the application's nature. Edge servers will initially reuse computation based on this approximate threshold, and the *Deduplicator* will adjust the threshold by sampling the reuse accuracy of groups of tasks over time. The sampling process can take place by having servers execute from scratch small groups of tasks $g_1, g_2, ..., g_n$ for which they have results that they could potentially reuse and compare the results of the execution of $g_1, g_2, ..., g_n$ with the results of the tasks that could be reused. Based on this process, the *Deduplicator* can estimate the reuse accuracy and adjust the similarity threshold accordingly.

C. Hash Value Space Slicing

For every hash function, there is a space (range) of resulting hash values. The same applies to the *Deduplicator*, which slices the LSH value space and distributes slices to available edge servers. Each edge server receives and executes offloaded tasks with input data resulting in hash values that fall in the slice assigned to this server when hashed through LSH.

In Figure 2a, we present an example of a hash space with a hash length equal to 2 bytes (potential values 0 - 65535). In this example, the *Deduplicator* has distributed the space equally among the three available edge servers: the first slice includes hash values 0 - 21844 and is assigned to the first server (S_1), the second slice includes hash values 21845 - 43689 and is assigned to the second server (S_2), and the third slice includes hash values 43690 - 65535 and is assigned to the third server (S_3). The *Deduplicator* forwards tasks to servers according to input data hash values: 0 - 21844 to Server $S1$, 21845 - 43689 to Server $S2$, and 43690 - 65535 to Server $S3$.

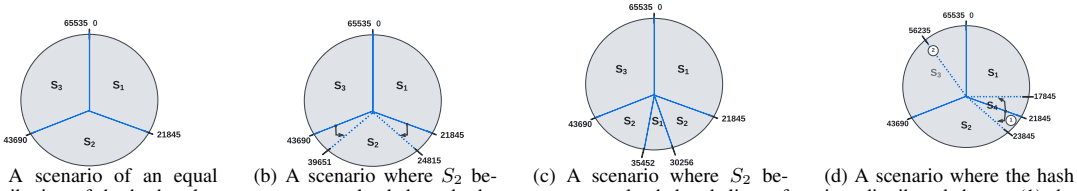
The *Deduplicator* begins its operation with an equal distribution of the hash space among edge servers. Such an equal distribution may cause load imbalances among edge servers, since the distribution of tasks in the hash space is unlikely to be uniform over time. Over time, the *Deduplicator* collects usage information about the resources of edge servers (*e.g.,* CPU and memory usage), identifies overloaded and underutilized edge servers, and redistributes the hash value space among servers to balance the load without impairing the ability of the edge computing environment to achieve computation reuse.

Hash value space redistribution strategies: A challenge that arises during the redistribution process of the hash value space has to do with which values in a slice (and the associated tasks) impose substantial load on each edge server. To address this challenge, the *Deduplicator* employs two strategies to redistribute the hash value space among edge servers in order to alleviate load imbalances:

• **Adjusting the size of existing slices:** This strategy applies to cases where significant load is imposed due to hash values (and associated tasks) close to one or both "edges" of a slice. For example, in Figure 2b, we present a scenario where S_2 becomes overloaded and a substantial part of the load is due to values close to the edges of the slice assigned to S_2. In this case, one or both edges of the slice can be redistributed to edge server(s) possessing adjacent slices (S_1 and/or S_3 in our example). In other words, the slice assigned to the overloaded server becomes smaller, while the sizes of adjacent slices grow.

• **Creating slices of finer granularity:** This strategy applies to cases where significant load is imposed due to hash values (and associated tasks) that are not close to an edge of a slice. For example, in Figure 2c, we present a scenario where S_2 becomes overloaded and a substantial part of the load is due to values closer to the middle of the slice assigned to S_2. In this case, the *Deduplicator* creates slices of finer granularity. Specifically, it creates a slice with values in the middle of the initial slice assigned to S_2. As a result, the newly created slice (which includes hash values 30256 - 35452 in our example) will be assigned to another server (S_1 in our example). In this way, the offloaded tasks that fall under this new slice (and subsequently part of the load imposed to S_2) will now be distributed by the *Deduplicator* to S_1. The *Deduplicator* can also perform the reverse process, thus being able to aggregate multiple fine-grained slices to create a coarse-grained slice, if needed during its operation.

Both strategies aim to redistribute part of the load from



(a) A scenario of an equal distribution of the hash value space among edge servers.

(b) A scenario where S_2 becomes overloaded and the slice assigned to S_2 shrinks.

(c) A scenario where S_2 becomes overloaded and slices of finer granularity are created.

(d) A scenario where the hash space is redistributed due to (1) the addition and (2) the failure of a server.

Fig. 2: Distribution (slicing) of a hash value space (hash length equal to 2 bytes) among three edge servers.

overloaded edge servers to servers with available resources. This is achieved by adjusting the created slices, so that part(s) of the hash value space are re-assigned (from overloaded servers to servers with available resources). As a result, tasks are distributed by the *Deduplicator* to edge servers based on the updated slices.

When a redistribution of the hash value space takes place, offloaded tasks with input data resulting in the redistributed hash values will be executed by a different edge server. For example, offloaded tasks will be executed by S_1 instead of S_2 for the redistributed hash values in the scenarios of Figures 2b and 2c. In such cases, the execution results of tasks, which correspond to the redistributed hash values and are stored by S_2 in order to be reused, can be transferred from S_2 to S_1. This enables S_1 to reuse the results of previously executed tasks that correspond to redistributed hash values right away (warm start) instead of having to first execute offloaded tasks from scratch for the redistributed hash values in order to store their results and reuse them in the future (cold start).

D. Collecting Resource Usage Information From Edge Servers

To redistribute the hash value space, the *Deduplicator* needs to collect information about the resource usage of available edge servers over time. Such information may include CPU and memory usage, among others. The collection of this information is achieved through the following mechanisms:

- **Explicit notifications:** Edge servers send explicit notifications to the *Deduplicator* over time to indicate their current resource usage. This can happen either periodically or once the resource usage of an edge server exceeds a certain threshold.
- **Piggybacking information on HTTP/HTTPS POST responses:** Edge servers piggyback resource usage information on HTTP/HTTPS POST responses, which contain the results of the execution of offloaded tasks. Once a response that carries such information is received by the *Deduplicator*, the *Deduplicator* extracts this information from the response and the response is returned to the user device that offloaded the corresponding task (HTTP/HTTPS POST request).

Ideally, we would like the *Deduplicator* to collect CPU and memory usage information per hash value. However, as the size of the hash value space increases, it becomes infeasible to maintain such information on a per hash value basis. To this end, edge servers maintain such information for a range (group) of hash values. The size of this range is typically smaller than an individual slice (finer granularity) and it is determined, so that it provides sufficient granularity without imposing significant overhead.

E. Edge Server Additions and Failures

The *Deduplicator* is able to identify additions of new edge servers and failures of existing servers. In both cases, the hash value space needs to be redistributed. When adding a new server, a slice of the hash value space needs to be created and assigned to the newly added server. An approach to achieve that is for the *Deduplicator* to create a new slice out of the slice of the most loaded edge server. For example, in Figure 2d, let us assume that the most loaded edge server is S_2 and a new edge server S_4 is added. The *Deduplicator* will re-assign a part of S_2's slice to S_4, thus creating a slice for S_4. Optionally, the *Deduplicator* can also re-assign a part of a slice adjacent to S_2's slice (the slice of S_1 in our example) to S_4 in order to create a slice of a more substantial size, so that additional load is handled by S_4 (more offloaded tasks are distributed to S_4). The exact part of the hash value space to be assigned to S_4 depends on the load imposed by the different hash values in the slice of S_2, and potentially the different hash values in the slice adjacent to S_2's slice.

In the case of an edge server failure (S_3 in the example of Figure 2d), the *Deduplicator* will re-assign parts of the slice of the failed server to adjacent slices. In our example, the *Deduplicator* will re-assign a part of S_3's slice to S_1 and the remaining part of S_3's slice to S_2. The *Deduplicator* aims to re-assign the slice of S_3 to adjacent slices, so that the corresponding servers (S_1 and S_2) do not become overloaded (based on their current load and the load to be re-assigned). Load imbalances that occur after the initial redistribution of the hash value space described above (due to the addition of a new server or the failure of an existing server) will be alleviated through subsequent redistribution events of the hash value space among servers as described in Section III-C.

During the addition of a new edge server, the new server sends an explicit notification to the *Deduplicator*. Once this notification is received by the *Deduplicator*, the process of creating a slice for the new server begins. On the other hand, server failures are identified due to the absence of: (i) responses to offloaded tasks distributed by the *Deduplicator*; and (ii) explicit notifications sent by the edge servers to the *Deduplicator* to indicate the usage of their resources.

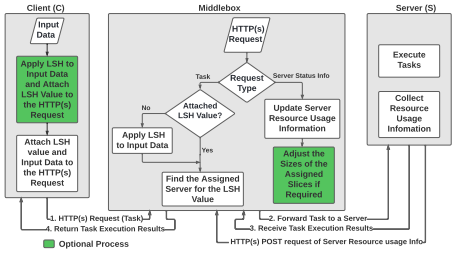


Fig. 3: Overview of the *Deduplicator* implementation.

IV. Evaluation

A. Deduplicator *Prototype Implementation*

We implemented a *Deduplicator* prototype [1] as a module of Nginx [9]. The *Deduplicator* is deployed as a middlebox. Figure 3 shows an overview of our *Deduplicator* implementation and how the *Deduplicator* interacts with clients and servers. The *Deduplicator* mainly performs two operations: (i) receives and forwards HTTP/HTTPS requests (tasks) from users to edge servers based on the assigned hash value ranges (slices) and resource availability of the servers; and (ii) collects resource usage information from servers and updates the sizes of assigned slices as described in Section III.

The task distribution process begins when the middlebox receives HTTP/HTTPS requests from users. Depending on whether users hash the task input data, the *Deduplicator* either hashes the input data attached to the received HTTP/HTTPS requests or extracts the hash values of input data from the received requests to decide to which server a task should be distributed in order to facilitate reuse.

To redistribute slices of the hash value space, servers send resource utilization information to the *Deduplicator*. Subsequently, the *Deduplicator* updates the resource information of servers and decides whether redistribution is needed. If redistribution is needed, the slice sizes are adjusted and the stored execution results of tasks that correspond to the adjusted hash space may be transferred to a new server from previously assigned server(s) as described in Section III-C.

We have implemented an adaptive approach, so that our *Deduplicator* prototype can adjust the granularity of slices. Let us assume that there are n edge servers available at a certain time and the LSH hash size is b bits. Initially, the assigned hash space per edge server will be $\frac{2^b}{n}$. If an edge server S_i, where $1 \leq i \leq n$, receives t_i tasks during the interval between two consecutive hash space redistribution events, then the size of the hash space H_i assigned to S_i will be:

$$H_i = \frac{2^b}{n} + 2^b\left(\frac{1}{n} - \frac{t_i}{\sum_{i=1}^{n} t_i}\right) \quad (1)$$

The hash value range R_i assigned to S_i will be:

$$R_i = \begin{cases} [max(R_{i-1}) + 1, max(R_{i-1}) + H_i], & \text{if } 1 < i \leq n \\ [0, H_i - 1], & \text{if } i = 1 \end{cases} \quad (2)$$

[1]We make our *Deduplicator* implementation code available to the research community at https://github.com/malazad/Deduplicator.

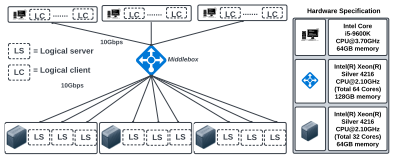


Fig. 4: Experimental topology.

TABLE I: Edge services and real-world datasets used for the evaluation of the *Deduplicator*.

Dataset	Dataset size	Correlation between data	Edge services	Processing granularity
MNIST	70K	Low	Digit recognition	Medium
ASL [30]	87K	Low	Recognition of ASL letters	Medium
Hololens [31]	189K	High	Action recognition	Fine
Pandaset [32]	49K	Low	Obstacle detection	Fine
CCTV	10K	High	Vehicle detection	Coarse

The second part of Equation 1 determines whether the range R_i assigned S_i needs to be adjusted, so that the slices assigned to edge servers become more fine- or coarse-grain.

We implemented a user traffic generator, which attaches input data to HTTP/HTTPS POST requests (tasks) and offloads certain numbers of tasks per second. We implemented the LSH semantics through the FALCONN library [8]. We implemented edge server programs, which perform different data processing operations (*e.g.,* object detection)and collect information about the resource usage of edge servers.

B. *Evaluation Setup*

Experimental environment: We deployed the *Deduplicator* prototype on the testbed of Figure 4 to evaluate its performance and tradeoffs. We used three physical machines as traffic generators to emulate users who offload tasks. Each generator has hundreds of logical clients, and each logical client sends a specific number of requests (tasks) per second. We also used three physical servers, which offer different services, such as object detection and recognition. Each physical server hosts three logical servers to scale up our experimental setup. All traffic generators and servers are connected to the middlebox through 10Gbps Ethernet links. We run each experiment ten times and we present the average results in Section IV-C.

Datasets: To evaluate the *Deduplicator* prototype, we use five real-world image datasets as task input data with different degrees of correlation between data (Table I). Each edge service may also require a different processing granularity.

Evaluation metrics: We consider the following metrics:

1) *Time overhead per request:* The time required to receive, process, and forward a task by the middlebox.

2) *Percent of reuse:* The percent of offloaded tasks that reused the results of previously executed tasks.

3) *Percent of distributed tasks per edge server:* The number of tasks distributed to each edge server by a middlebox as a percentage of all the tasks offloaded by user devices (clients).

4) *Accuracy of reuse:* The percentage of reused tasks, where reuse was accurate. The reuse of a previous task to satisfy an incoming task is accurate if the results of the reused task and the results of the incoming task would have been the same, if the incoming task had been executed from scratch.

Approaches for comparison: First, we compare different approaches that can be used to facilitate computation reuse as part of the *Deduplicator* design. These approaches include:

TABLE II: Percent of reuse for different datasets.

Number of servers	Approach	MNIST	ASL	Hololens
6	Reuse ideal	18.61	51.84	61.53
	Reuse vanilla	13.96	43.40	56.11
	Reuse mini-buckets	11.76	41.40	53.40
	Reuse adaptive	10.39	39.91	51.51
9	Reuse ideal	18.61	51.84	61.53
	Reuse vanilla	12.65	42.14	53.12
	Reuse mini-buckets	10.45	41.82	50.22
	Reuse adaptive	9.71	39.87	49.89

1) *Reuse ideal:* This is the ideal case for reuse, represented as if a single edge server exists that has adequate resources to execute all offloaded tasks. This approach results in the maximum percentage of reuse possible for each dataset.
2) *Reuse vanilla:* This is a static approach, which distributes the hash value space equally among edge servers and this distribution does not change afterwards. This approach is equivalent to replacing consistent hashing with LSH and applying the hash function to task input data instead of headers.
3) *Reuse mini-buckets:* For n available edge servers, the whole hash value space is first divided into n slices. Each slice is further divided into n smaller slices, and each smaller slice is assigned to a server. This approach is also static in the sense that the size of each smaller slice and the assignment of smaller slices to edge servers do not change over time.
4) *Reuse adaptive:* This is a hybrid approach where the hash value space is initially distributed equally among edge servers. After this initial equal distribution, this approach adapts to the conditions (load) at the edge by adjusting the granularity of slices based on Equations 1 and 2 presented in Section IV-A.

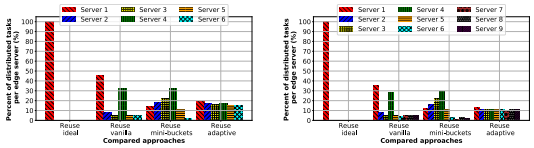
Second, we compare the *Deduplicator* to off-the-shelf load balancing approaches that Nginx offers (*e.g.,* round robin, random, least connection, consistent hashing).

C. Evaluation Results

In this section, we present results on the trade-offs of the *Deduplicator* and we compare it to different load balancing approaches. Due to space limitations, we present results for three of the used datasets (MNIST, ASL, and Hololens datasets of Table I). Nevertheless, we have verified that the results and the trends presented below hold for all datasets of Table I.

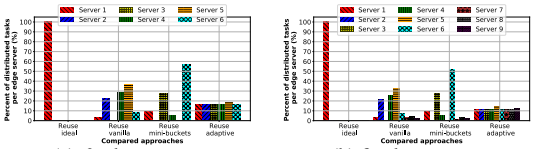
1) Deduplicator *Design Tradeoffs:* **Time overhead:** For all approaches (reuse ideal, reuse vanilla, reuse mini-buckets, and reuse adaptive), we evaluate the time needed by the *Deduplicator* to distribute a task for varying sizes of input data (up to 5MB). Our results show that the *Deduplicator* needs up to 1.76ms to distribute a task, which is roughly the same amount of time as other load balancing approaches (*e.g.,* round robin, random, consistent hashing) implemented by Ngnix. Specifically, the time needed for the *Deduplicator* to distribute a task is 1% lower to 3% higher than the time needed for other load balancing approaches. All approaches are able to distribute traffic at line rate (a total of 30Gbps) for various mixes of tasks and input data sizes.

Percent of reuse: As shown in Table II, reuse decreases up to 12% in the case of the reuse adaptive approach compared to the ideal reuse case. Compared to static approaches (reuse vanilla and reuse mini-buckets), reuse adaptive results in a reuse reduction of less than 5%. This reduction is due to the redistribution of the hash value space performed by reuse adaptive to balance task distribution among servers.



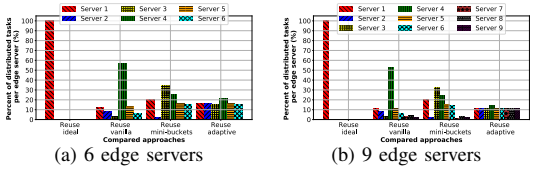
(a) 6 edge servers (b) 9 edge servers

Fig. 5: Load distribution among edge servers (MNIST).



(a) 6 edge servers (b) 9 edge servers

Fig. 6: Load distribution among edge servers (ASL dataset).



(a) 6 edge servers (b) 9 edge servers

Fig. 7: Load distribution among edge servers (Hololens).

Percent of distributed tasks per edge server: Our results (Figures 5, 6, and 7) show that the reuse adaptive approach distributes tasks substantially more uniformly among servers compared to other approaches. Reuse adaptive also redistributes the hash value space among servers, so that load imbalances are identified and mitigated. Static approaches (reuse vanilla and reuse mini-buckets) are less effective, since they do not redistribute the hash value space in cases of load imbalances. Finally, the reuse ideal approach distributes all tasks to a single server, thus imposing all the load to this server (*e.g.,* server 1 in the results of Figures 5, 6, and 7).

Accuracy of reuse: For all approaches, the *Deduplicator* facilitates the reuse of computation with accuracy that exceeds 90% across all datasets (in some cases even reaching 100%).

Conclusion: The reuse adaptive approach achieves the best trade-off between reuse and load balancing. Reuse adaptive can effectively balance the distribution of tasks among edge servers with a minor reduction of the reuse percentage (less than 5%) compared to static approaches. To this end, for the results presented in Section IV-C2, we implement the *Deduplicator* with the reuse adaptive approach.

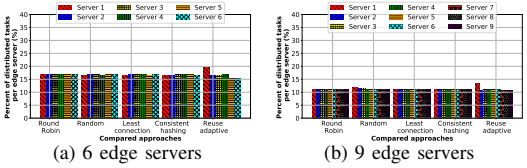
2) Comparison to Different Load Balancing Approaches:

Time overhead: We evaluate the time needed by the *Deduplicator* and other load balancing approaches to distribute a task for varying sizes of input data (up to 5MB). Our results show that the *Deduplicator* requires roughly the same amount of time to distribute a task as other load balancing approaches (up to 1.76ms per task, which is 1% lower to 3% higher than the time needed by other load balancing approaches). We experimented with different traffic mixes that consist of tasks with various input sizes and we verified that the *Deduplicator* is able to distribute traffic at line rate (a total of 30Gbps).

Percent of reuse: Our results in Table III show that the *Deduplicator* achieves up to 20% higher percent of reuse than

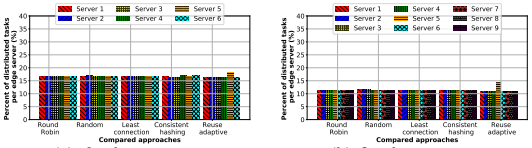
TABLE III: Percent of reuse for different datasets.

Number of servers	Approach	MNIST	ASL	Hololens
6	Round Robin	6.03	23.76	32.59
	Random	6.01	23.09	32.62
	Least connection	6.51	23.34	32.87
	Consistent hashing	6.80	25.54	32.76
	Reuse adaptive	10.03	38.77	51.56
9	Round Robin	4.04	20.67	26.78
	Random	4.01	20.11	26.82
	Least connection	4.50	20.14	26.91
	Consistent hashing	4.76	20.23	26.87
	Reuse adaptive	9.71	35.87	46.94



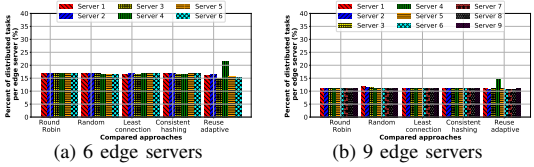
(a) 6 edge servers (b) 9 edge servers

Fig. 8: Load distribution among edge servers (MNIST).



(a) 6 edge servers (b) 9 edge servers

Fig. 9: Load distribution among edge servers (ASL dataset).



(a) 6 edge servers (b) 9 edge servers

Fig. 10: Load distribution among edge servers (Hololens).

other load balancing approaches, since the *Deduplicator* is aware of the semantics of reuse, thus being able to identify and distribute similar tasks to the same edge server(s). Our results also show that the percent of reuse severely degrades for load balancing approaches that are not aware of the reuse semantics as we increase the number of servers. This degradation for the *Deduplicator* is much smaller as we increase the number of servers, since it can identify similar tasks independent of the number of available servers.

Percent of distributed tasks per edge server: Figures 8, 9, and 10 show that the *Deduplicator* balances task distribution among servers with minor deviations (less than 5%) compared to other load balancing approaches, thus mitigating load imbalances that occur due to reuse. These deviations are primarily due to the fact that in datasets with high correlations, such as the Hololens dataset, higher amounts of data typically correspond to only a few hash values in the overall hash value space. In such cases, the *Deduplicator* creates over time slices of finer granularity (as described in Section III-C) to balance task distribution among servers. Nevertheless, the fact that different amounts of data are hashed to individual hash values over time results in these minor deviations of task distribution.

Accuracy of reuse: Our results show that the *Deduplicator* achieves 3-14% higher reuse accuracy than the compared approaches. The *Deduplicator* distributes tasks to edge servers that are likely to reuse previously executed tasks, while other approaches are not aware of the reuse semantics. In addition, the *Deduplicator* consistently achieves reuse accuracy that exceeds 90%, reaching for certain datasets almost 100%.

Conclusion: The *Deduplicator* increases the percentage of reuse by up to 20% compared to load balancing approaches that are not aware of the computation reuse semantics. The *Deduplicator* also balances the distribution of tasks among edge servers with minor deviations (less than 5%) compared to non computation reuse aware load balancing approaches.

V. Conclusion

In this paper, we presented the *Deduplicator*, a middlebox that balances the load and facilitates computation reuse in edge computing environments. The *Deduplicator* design features mechanisms to identify and deduplicate similar tasks offloaded by user devices, collect information about the usage of edge servers' resources, manage the addition of new servers and the failures of existing servers, and ultimately balance the load imposed on servers. Our evaluation results showed that the *Deduplicator* balances the distribution of tasks among edge servers with minor deviations compared to non computation reuse aware load balancing approaches and achieves higher percentages of reuse than these approaches.

Acknowledgements

This work is partially supported by the National Science Foundation through awards CNS-2104700 and CNS-2306685, as well as by the U.S. Army Engineer Research & Development Center (ERDC) through HPC PET special project 5025.

References

[1] V. Sekar, S. Ratnasamy, M. K. Reiter *et al.*, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011, pp. 1–6.

[2] J. Sherry *et al.*, "Making middleboxes someone else's problem: Network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[3] J. Zhang *et al.*, "Load balancing in data center networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2324–2352, 2018.

[4] M. W. Al Azad and S. Mastorakis, "The promise and challenges of computation deduplication and reuse at the network edge," *IEEE Wireless Communications*, 2022.

[5] P. Guo, B. Hu, R. Li *et al.*, "Foggycache: Cross-device approximate computation reuse," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, 2018, pp. 19–34.

[6] P. Guo and W. Hu, "Potluck: Cross-application approximate deduplication for computation-intensive mobile applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 271–284.

[7] M. W. Al Azad and S. Mastorakis, "Reservoir: Named data for pervasive computation reuse at the network edge," in *2022 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 2022, pp. 141–151.

[8] A. Andoni *et al.*, "Practical and optimal lsh for angular distance," *Advances in neural information processing systems*, vol. 28, 2015.

[9] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[10] X. Wang and D. Loguinov, "Load-balancing performance of consistent hashing: asymptotic analysis of random node join," *IEEE/ACM Transactions on Networking*, vol. 15, no. 4, pp. 892–905, 2007.

[11] A.-C. Nicolaescu, S. Mastorakis, and I. Psaras, "Store edge networked data (send): A data and performance driven edge storage framework," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[12] "Unvoiced," https://github.com/grassknoted/Unvoiced, 2022.

[13] "Ar egocentric dataset for action recognition," https://www.kaggle.com/datasets/cyrilruosch/ar-egocentric-dataset-for-action-recognition, 2022.

[14] "Pandaset by hesai and scale ai," https://pandaset.org, 2022.