Container Data Item: An Abstract Datatype for Efficient Container-based Edge Computing

Md Rezwanur Rahman*, Shirin Ebadi †, Adarsh Srinivasan‡, Varsha Natarajan§, Tarun Annapareddy ¶, Eric Keller‖ and Shivakant Mishra **,
Department of Computer Science, University of Colorado Boulder
‖ Department of Electrical, Computer and Energy Engineering, University of Colorado Boulder
Email:{*mdra7255, †shirin.ebadi, ‡adarsh.srinivasan, §varsha.natarajan, ¶ tarun.annapareddy,
‖ eric.keller, ** mishras }@colorado.edu

***Abstract*—We present Container Data Item (CDI), an abstract datatype that allows multiple containers to efficiently operate on a common data item while preserving their security and isolation semantics. Application developers can use CDIs to enable multiple containers to operate on the same data, synchronize execution among themselves, and control the ownership of the shared data item during runtime. CDI is designed to support applications comprised of a set of interconnected microservices, each implemented by a separate dedicated container. CDI preserves the important isolation semantics of containers by ensuring that exactly one container owns a CDI object at any instant and that the ownership of a CDI object may be transferred from one container to another only by the current CDI object owner. We present three different implementations of CDI that allow different containers residing on the same server as well containers residing on different servers to use CDI for efficiently operating on a common data item. The paper provides an extensive performance evaluation of CDI, along with a representative Augmented Reality application.**

***Index Terms*—Containers, data sharing, abstract datatype, shared memory, Edge, RDMA**

I. Introduction

A microservice-based architecture structures an application as a collection of loosely coupled microservices. Each microservice implements simple functionality with well-defined APIs. Containers provide a very convenient way to implement microservices [1] on an Edge server or the cloud, and have become the de facto standard for implementing microservices. However, since containers isolate their functionality in their own compute environment, any interactions between microservices entail inter-container communications across different IP spaces, typically using UDP, TCP, HTTP, messaging queues, or remote procedure calls (e.g. gRPC). This approach has two shortcomings. First, communication using transport layer protocols requires developers to worry about relatively low-level networking details. Changes in the use of an underlying protocol, e.g. from TCP to gRPC requires making significant changes in the application code. Second, communication over a (virtual) network involves data copying and user-kernel context switching, resulting in significant performance overhead.

In this paper, we propose the Container Data Item (CDI), an abstract datatype at the language runtime level that allows different containers to operate on the same data efficiently and transparently while preserving the container isolation semantics. A common operating style in a microservice-based IoT application is that an input data object passes through a sequence of microservices, being manipulated by each in turn. For example, in an Augmented Reality (AR) application discussed in detail in Section III-C, the frames in a video stream pass through the frame extractor, object detector, and frame combinator containers. Unfortunately, current implementations incur a significant data movement cost [2], [3].

CDI is specifically designed to support such applications. It provides transparency and efficiency by abstracting away the implementation details of data management (data sharing, data movement, etc.) and utilizing efficient suitable underlying communication mechanisms that minimize data movement and user-kernel context switches. For example, if the interacting containers are running on the same server, CDI ensures *zero-copy* (data sharing without CPU-initiated copying) and no user-kernel context switching. It provides transparency by ensuring that the application-level code remains the same irrespective of whether the interacting containers are on the same server or different servers. The low-level implementation details are managed transparently by the run-time system. Finally, each container in the sequence is still completely isolated from other application containers.

To demonstrate the utility of CDI, we provide three different implementations, CDI-SHM, CDI-RPC, and CDI-RDMA. CDI-SHM uses Unix System V shared memory and is applicable when interacting containers are on the same server. CDI-RPC and CDI-RDMA are applicable when interacting containers are on different servers. CDI-RPC uses gRPC as the underlying communication protocol, while CDI-RDMA uses RDMA (Remote Direct Memory Access) for memory copy between different servers. Since container-based microservice applications often run within a container orchestration framework such as Kubernetes, we have integrated CDI with Kubernetes.

The main contributions of this paper are as follows.

- CDI moves the support for container interactions from current I/O based methods to the language runtime level, providing transparency, encapsulation, and improved soft-

This work was supported in part by the National Science Foundation (NSF), through awards 2326835 and 2241818.

ware modularity, re-usability, maintainability, and interoperability.
- Our prototype implementation demonstrates a significant reduction in latency and resource usage, and an increase in throughput.
- We demonstrate the utility of CDI by building a popular Augmented Reality application.

II. Related Work

Inter-container communication in a microservice based architecture is primarily done using network protocols that work across IP spaces (TCP, UDP, HTTP, RPC, etc.) or by using message queues (RabbitMQ, Apache Kafka) implemented using these protocols. Due to the inherent overhead in these protocols (data copying, context switching, etc), exploring techniques for efficient inter-container communication is an active area of research in both academia and industry.

Shimmy [4], [5] presents a high-performance inter-container communication mechanism through shared memory channels targeting cloud and edge computing scenarios. It leverages shared memory for local communication and RDMA synchronization for remote interactions, providing significantly higher throughput and lower latency than TCP, UDP, and message queues. Ubaid et al. [6] have analyzed the inter-container network bandwidth and compute utilization on an RDMA-enabled Kubernetes cluster, comparing various networking frameworks and observed significant advantages over traditional networks for live migration. Xue et al. [7] have shown how deep neural networks can greatly benefit from using an RDMA based system rather than the traditional systems which rely on gRPC on TCP. Fent et al. [8] have extensively compared TCP with a RDMA or a shared memory based solution for a database management system, showing that performance improvement is due to fewer context switches in the kernel.

Kun et al. [9] has presented a comprehensive comparison of various container networking technologies, analyzing their performance degradation and overhead in a cloud environment. Dhmem [10] is a software library designed to manage shared memory across containerized workflow tasks with minimal code changes and performance overhead. It facilitates efficient cross-container communication by leveraging shared memory for local interactions and supports scalable performance and runtime communication configuration. A comparison with traditional message serialization methods and a thread-based approach highlights its superior performance in terms of throughput and latency across various workflow configurations, including simple, pipeline, and scatter-gather types.

FreeFlow [11], is a software-based virtual RDMA networking framework designed to bridge the gap between containerization and RDMA technologies in cloud-based applications. FreeFlow employs a virtual software switch on each server, offering transparent integration with applications running inside containers in a shared cloud environment and resulting in throughput and latency comparable to bare-metal RDMA.

III. Container Data Item

We define CDI as a language runtime level abstract data type designed to allow multiple containers to operate on a common data item. At a high level, a CDI object can be seen as consisting of three attributes, *key*, an id that uniquely identifies the object, *data*, which stores the data item, and *dataSize*, the size of the data item. In particular, the CDI runtime system provides three abstract data types, *CDI* that refers to a CDI object, *CDI_key* to uniquely identify CDI objects, and *CDI_container* to uniquely identify application containers. The unique identities provided by *CDI_key* and *CDI_container* are only within the context of the set of a application containers. For simplicity, *dataSize* is kept fixed when the object is created, and remains constant throughout the lifetime of the object. A *CDI_key* is usually a string, and a *CDI_container* is an integer. If two containers sharing an object are on the same server, *CDI_key* would refer to *key_t* if System V shared-memory is used and *filename* if POSIX shared-memory is used. Developers may assign a CDI object key either manually (reading from a file) or using a helper function. The *CDI_container* id is assigned to an application container during the application configuration. Operations are provided to create and destroy a CDI object, manipulate its value, and manage its access control.

A. CDI Semantics

- A CDI object is created by a container and that container is the owner of that object when created.
- A CDI object is owned by exactly one container at any instant.
- Only the owner of a CDI object can manipulate (read, write, destroy) the object.
- The owner of a CDI object can create a new CDI object that is an exact copy of the original CDI object. The new CDI object is owned by the creator container and is completely independent of any other CDI objects.
- The owner (C1) can transfer its ownership to another container (C2). In one atomic operation, C1 loses its ownership and C2 acquires ownership of the object.

B. CDI Operations

The following operations are supported on a CDI object:

- *CDI_create(CDI_key k, size_t s)*:
  - Returns 0 if a CDI object with key *k* already exists. Returns -1 if a new CDI object with key *k* cannot be created for any other reason such as insufficient memory. Otherwise, creates a new CDI object of data size *s* identifiable by key *k* and returns 1.
- *CDI_use(CDI_key k)*:
  - Associates the CDI variable with a CDI object with key *k* if that object exists and returns 1. Returns 0 if a CDI object with key *k* does not exist.
- *CDI_copy(CDI c, CDI_key k)*:
  - Returns 0 if a CDI object with key *k* already exists. Returns -1 if a new CDI object cannot be created for any other reason such as insufficient memory. Creates a new

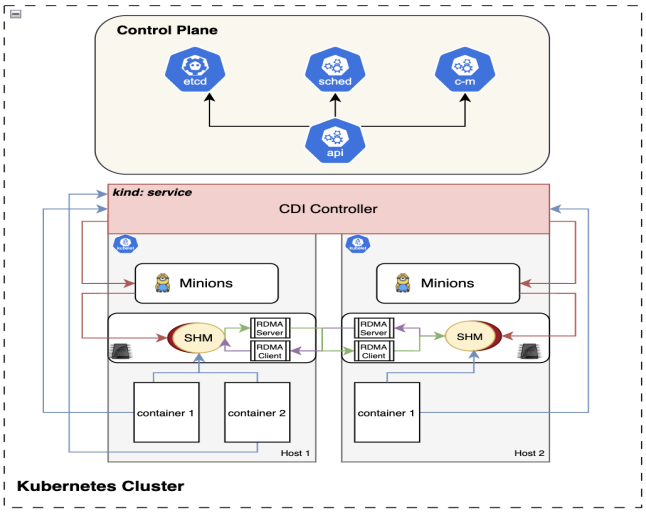


Fig. 1: CDI Architecture

CDI object identifiable by key *k* and returns 1; the new CDI object is an exact copy of *c*.

- *CDI_access( )*:
  - Blocks the calling thread until the container has obtained ownership of the CDI object.
- *CDI_transfer(CDI_container c_id))*:
  - Transfers the ownership of the CDI object to a different container identifiable by id *c_id*.
- *CDI_destroy( )*:
  - Destroys a CDI object.

C. Application of CDI: Augmented Reality

To demonstrate use of CDIs in a microservice-based application, we implemented an augmented reality application (Figure 2). We have primarily used Python the main implementation along with C for RDMA functionalities. The application hosts three containers: *frame extractor*, *object detector* and *frame combinator*. The *frame extractor* creates five CDI objects, c1, .., c5. Unique keys are provided for these objects (by console or configuration). In a relevant work on augmented reality [12], the authors found that the object detector container is the major performance bottleneck. The overall application performance can be improved by running multiple instances of this container in parallel, each operating on a different frame of a video. We have chosen to run five instances in our implementation. After storing frames in CDI objects c1, .., c5, the *frame extractor* container transfers the ownership of these objects to the corresponding object detector containers. Each *object detector* container waits for the ownership of its respective CDI object to perform its operations. After receiving the CDI object, it performs object detection and labeling on the frame stored in that object. For our case, we used YOLO-based [13] real-time object detection on videos. Next, it transfers the ownership of its CDI object to the *frame combinator* container to combines these frames to output a video sub-stream. Finally, it transfers the ownership of the CDI objects back to the *frame extractor* for reuse.

Frame extractor container code

```
CDI c1, c2, c3, c4, c5;
int max_image_size;
CDI_key k1, k2, k3, k4, k5;

 read six arguments from command line into
 max_image_size, k1, k2, k3, k4 and k5 respectively

c1.CDI_create(k1, max_image_size);
c2.CDI_create(k2, max_image_size);
c3.CDI_create(k3, max_image_size);
c4.CDI_create(k4, max_image_size);
c5.CDI_create(k5, max_image_size);

while (not end of video stream) {
   c1.CDI_access ( );
    extract next frame in c1
   c1.CDI_transfer(o1);
   c2.CDI_access ( );
    extract next frame in c2
   c2.CDI_transfer(o2);
   c3.CDI_access ( );
    extract next frame in c3
   c3.CDI_transfer(o3);
   c4.CDI_access ( );
    extract next frame in c4
   c4.CDI_transfer(o4);
   c5.CDI_access ( );
    extract next frame in c5
   c5.CDI_transfer(o5);
}
c1.CDI_destroy( );
c2.CDI_destroy( );
c3.CDI_destroy( );
c4.CDI_destroy( );
c5.CDI_destroy( );
```

Object detector container code

```
CDI c;
CDI_key k;

read one argument from command line into k

c.CDI_use(k);

while (1) {
   c.CDI_access( );
    do object detection in c and put labels
   c.CDI_transfer(fc);
}
```

Frame combinator container code

```
CDI c1, c2, c3, c4, c5;
CDI_key k1, k2, k3, k4, k5;

 read five arguments from command line into
 k1, k2, k3, k4 and k5 respectively

c1.CDI_use(k1);
c2.CDI_use(k2);
c3.CDI_use(k3);
c4.CDI_use(k4);
c5.CDI_use(k5);

while (1) {
   c1.CDI_access( );
   c2.CDI_access( )
   c3.CDI_access( );
   c4.CDI_access( );
   c5.CDI_access( );
    combine frames from c1, c2, c3, c4, c5
   c1.CDI_transfer (fc);
   c2.CDI_transfer (fc);
   c3.CDI_transfer (fc);
   c4.CDI_transfer (fc);
   c5.CDI_transfer (fc);
}
```

Fig. 2: Augmented Reality application using CDI

IV. Design and Implementation

A. Architecture

Figure 1 shows the architecture of our proposed system. Large-scale container based applications are managed by a container orchestration tool to orchestrate a set of worker nodes within a cluster. In our prototype, Kubernetes has been used as a proof of concept for implementing CDI. In Kubernetes, each worker node can run one or more Pods, each Pod hosting one or more containers. A Kubernetes cluster is comprised of a control plane (API Server, Scheduler, etc.) and the server nodes. The container orchestrator manages the entire infrastructure by deciding what containers to run and which hosts to run them on. This functionality is exposed to the administrators through an API. CDI extends a container orchestration framework by introducing two new components, *CDI Controller* and *CDI Minion*.

B. CDI Controller

The CDI controller exposes API functions for each of the CDI operations, namely *CDI_create*, *CDI_use*, *CDI_copy*, *CDI_access*, *CDI_transfer* and *CDI_destroy*. It also exposes *CDI_register(CDI_container c_id)* function. During configuration, application containers register with the CDI controller using this API. Unique container id values are provided by the developer, and the controller maintains a mapping of container id, container IP address and server IP address.

C. CDI Minions

An instance of a CDI Minion runs on every node and is responsible for managing storage and access control within

that node. CDI minions are implemented as a Kubernetes DaemonSet, which ensures that all current nodes run an instance of a CDI minion in a pod. When an application container invokes an API function of the CDI controller, the controller interacts with the minion(s) running on the relevant application container host(s) to execute the function. To manage access and isolation of CDI objects, a minion maintains a *Container Group* for each CDI object, which is a set of containers that can read and manipulate that CDI object, one at a time. In addition, it maintains the identity of the container that is the current owner of the object. A *Container Group* for a CDI object is created, and its owner identity is initialized when *CDI_create* is invoked. The membership of this container group is updated when *CDI_use* is invoked.

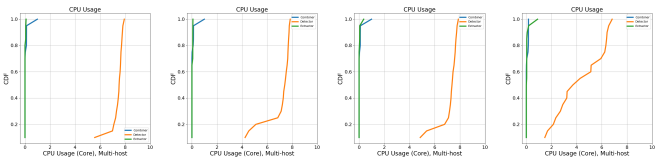
D. CDI Object Storage and Access Control

Minions create new CDI objects when *CDI_create* and *CDI_copy* functions are invoked by the host of that Minion. If a CDI object is shared between containers running on the same host, we use System V shared-memory for object storage. Inter-process communication is used, which makes the object shared between the host and the container. A container needs to know the *shm_id* of the shared memory, which is passed as a configuration file. If a CDI object is shared between containers running on different hosts, we leverage RDMA. RDMA enables direct copying from one region of memory from one host to another without involving the CPU. Finally, when the current owner of a CDI object invokes *CDI_destroy*, the controller contacts the appropriate minion(s), which deallocates the object memory as well as all metadata associated with that object. CDI Minions manage access to a CDI object, ensuring that only the current owner of the object has the read/write permissions for that object. When the *CDI_transfer* function is invoked, the controller instructs the appropriate minion(s) to update the owner id of the object. The minion(s) then updates the access control of the object and the owner id of the object. This is done only if the API function is invoked by the current owner of the object. If the two containers are on the same host, access control is changed by invoking *shmctl*. If the two containers are on different hosts, RDMA is used. The sequence of steps in this case are: (1) the minion on the current owner host revokes access permission from the current owner; (2) RDMA is used to copy the object; (3) the minion on the current host changes the owner id of the object to the new owner; (4) the minion on the next owner host changes owner id to the new owner and grants access permissions. This sequence ensures that a CDI object has at most one owner at any instant.

V. Evaluation

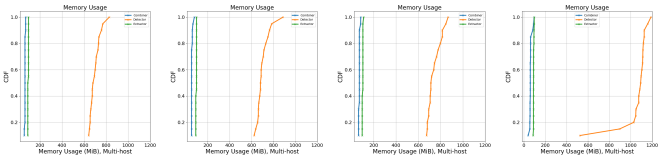
A. Setup

For raw measurements of the Augmented Reality application, we used m400 Cloudlab Nodes [14] with aarch64 architecture, dual-port Mellanox ConnectX-3 10 GB NIC (PCIe v3.0), 8 ARMv8 64-bit CPU cores and 65GB Memory



(a) (b) (c) (d)

Fig. 3: Multi-host setup of Augmented Reality Application. (a), (b), (c), and (d): CPU usage for TCP, REST, gRPC, and CDI-RDMA based implementations respectively.



(a) (b) (c) (d)

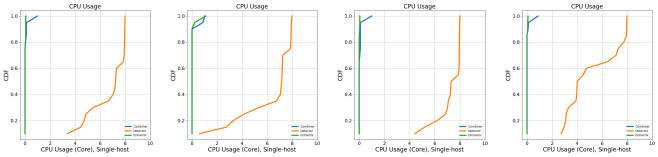
Fig. 4: Multi-host setup of Augmented Reality Application. (a), (b), (c), and (d): Memory usage for TCP, REST, gRPC, and CDI-RDMA based implementations respectively.

in our Kubernetes cluster. We dedicated 1 node to the master and 3 nodes to the workers.

B. Measurements

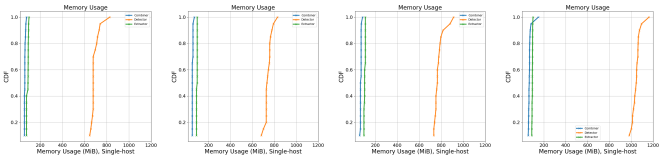
We first compared the performance of CDI-SHM and CDI-RDMA with TCP and gRPC. We measured latency for different message sizes. In the context of an application, the size of a message corresponds to the size of a data item that being operated on. Tables I and II show the mean latency for single-server and multi-server configurations, respectively. To reduce the chance of measurement errors, we repeated each experiment 100 times and reported the mean of each measurement. Both CDI-SHM and CDI-RDMA outperform TCP or gRPC and this performance gain increasingly gets better as the message size increases.

We attribute this performance improvement to the reduced data copying and context switching in CDI-SHM and CDI-RDMA. Further, no code changes are required in the application whether it is using CDI-SHM or CDI-RDMA, while code changes are required when moving TCP to gRPC or gRPC to TCP based implementations.



(a) (b) (c) (d)

Fig. 5: Single-host setup of Augmented Reality Application. (a), (b), (c), and (d): CPU usage for TCP, REST, gRPC, and CDI-SHM based implementations respectively.



(a) (b) (c) (d)

Fig. 6: Single-host setup of Augmented Reality Application. (a), (b), (c), and (d): Memory usage for TCP, REST, gRPC, and CDI-SHM based implementations respectively.

TABLE I: Latency (milliseconds) in a single host configuration.

	10 KB	100 KB	1 MB	10 MB
gRPC	1.25	1.49	6.2	39.12
TCP	0.13	0.19	0.69	16.9
Shared Memory	0.005	0.05	0.47	4.5

We have implemented the AR application described in Section III-C in four ways- using TCP for inter-container communication, using REST API, using gRPC for inter-container communication, and using CDI respectively. For CDI, we used CDI-SHM when all containers run on the same server and CDI-RDMA when different containers run on different servers. Figures 5 and 3 show CPU usage in the four implementations for single server and multiple servers respectively. We notice that there isn't any noticeable difference in CPU usage in frame extractor and frame combinator containers among the four implementations (very low CPU usage). As expected, the object detector container consumes most of the CPU time and here we see improvement in CPU usage in CDI-based implementations. Again, this can be attributed to reduced context switching and data copying.

As shown in Figures 6 and 4, memory usage on the other hand is higher in the CDI-based implementations when compared to TCP, REST, or gRPC-based implementations. This can partly be attributed to the fixed size of a CDI object. For example, in our experiments, we fixed the CDI object size to 10 MB, while these objects store video frames that were typically around 2 MB. TCP and gRPC-based implementations on the other hand use much less memory for storing these frames. Note that larger memory usage also negatively impacts latency performance in CDI-RDMA or CDI-gRPC implementations. For next steps, we are working with variable size CDI objects.

VI. Conclusion

We have presented Container Data Item, which allows multiple containers to operate on the same data item efficiently and transparently. A performance comparison with the current standard practices (TCP, RPC) shows that CDI provides a significant performance improvement in terms of latency and CPU usage. Finally, we demonstrate the generality of CDI by constructing an augmented reality application using CDI which showed a significant performance improvement. The key contribution of this paper is showing that CDI shifts data sharing among containers from I/O-based methods to the language level, allowing developers to focus on high-level design instead of implementation details. The underlying implementation of CDI can be changed transparently without the need for modification in application code. Using a variable-sized CDI object and other microservice patterns such as Scatter-gather, Streaming etc. are some interesting areas for further investigation. Additionally, allowing simultaneous read access to CDIs by multiple containers will be useful particularly for Publish-subscribe type of IoT applications.

TABLE II: Latency (milliseconds) in a multi-host configuration.

	10 KB	100 KB	1 MB	10 MB
gRPC	1.34	1.63	6.05	73.93
TCP	0.12	0.14	0.81	17.47
RDMA	0.014	0.094	0.87	8.64

References

[1] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li, "Microservices: architecture, container, and challenges," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. Macau, China: IEEE, Dec. 2020.

[2] K. Farkas, "Impact of TCP Variants on HTTP Performance."

[3] W. Bziuk, C. V. Phung, J. Dizdarevic, and A. Jukan, "On HTTP performance in IoT applications: An analysis of latency and throughput," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija: IEEE, May 2018.

[4] M. Abranches, S. Goodarzy, M. Nazari, S. Mishra, and E. Keller, "Shimmy: Shared Memory Channels for High Performance Inter-Container Communication."

[5] M. Khasgiwale, V. Sharma, S. Mishra, B. Thadichi, J. John, and R. Khanna, "Shimmy: Accelerating Inter-Container Communication for the IoT Edge," in *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*. Kuala Lumpur, Malaysia: IEEE, Dec. 2023.

[6] U. Abbasi, E. H. Bourhim, M. Dieye, and H. Elbiaze, "A Performance Comparison of Container Networking Alternatives," *IEEE Network*, vol. 33, no. 4, Jul. 2019.

[7] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast Distributed Deep Learning over RDMA," in *Proceedings of the Fourteenth EuroSys Conference 2019*. Dresden Germany: ACM, Mar. 2019.

[8] P. Fent, A. V. Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper, "Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. Dallas, TX, USA: IEEE, Apr. 2020.

[9] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An Analysis and Empirical Study of Container Networks," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. Honolulu, HI: IEEE, Apr. 2018.

[10] T. Hobson, O. Yildiz, B. Nicolae, J. Huang, and T. Peterka, "Shared-Memory Communication for Containerized Workflows," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Melbourne, Australia: IEEE, May 2021.

[11] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds."

[12] F. Hu, K. Mehta, S. Mishra, and M. AlMutawa, "Distributed Edge AI Systems," in *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*. Taormina (Messina) Italy: ACM, Dec. 2023.

[13] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, Jun. 2016.

[14] "CloudLab." [Online]. Available: https://www.cloudlab.us/