CRA: Enabling Data-Intensive Applications in Containerized Environments

Ibrahim Sabek*
University of Minnesota
Minneapolis, USA
sabek@cs.umn.edu

Badrish Chandramouli
Microsoft Research
Redmond, USA
badrishc@microsoft.com

Umar Farooq Minhas
Microsoft Research
Redmond, USA
ufminhas@microsoft.com

***Abstract*—Today, a modern data center hosts a wide variety of applications comprising batch, interactive, machine learning, and streaming applications. In this paper, we factor out the commonalities in a large majority of these applications, into a generic dataflow layer called *Common Runtime for Applications* (*CRA*). In parallel, another trend, with *containerization* technologies (e.g., Docker), has taken a serious hold on cloud-scale data centers, with direct implications on building next generation of data center applications. Container orchestrators (e.g., Kubernetes) have made deployment a lot easy, and they solve many infrastructure level problems, e.g., service discovery, auto-restart, and replication. For best in class performance, there is a need to marry the next generation applications with containerization technologies. To that end, CRA leverages and builds upon the containerization and resource orchestration capabilities of Kubernetes/Docker, and makes it easy to build a wide range of cloud-edge applications on top. To the best of our knowledge, we are the first to present a cloud native runtime for building data center applications. We show the efficiency of CRA through various micro-benchmarking experiments.**

I. Introduction

With the data explosion in businesses today, there is a need to deploy rich *dataflows* over the data. A dataflow consists of a graph of computation vertices that each holds state and reads/writes data to other vertices. In this paper, we start with the premise that such distributed stateful dataflows are a very general primitive, and encompass a wide range of applications deployed today. For example, a scan-based analytics system reads data from storage or in-memory caches (input vertices) and runs an analytics query using a dataflow of relational operator vertices. A streaming pipeline directly maps to an acyclic dataflow, whereas machine learning computations map to dataflows with iterative cycles in the dataflow graph. These systems consist of a dataflow of compute instances (actors) that communicate with one another using remote procedure calls that directly map to a dataflow abstraction.

Given the importance of (distributed) dataflows across diverse application domains, there is a strong need to make it easy for application developers to author dataflows that are distributed, efficient, scalable, resilient, and potentially long-running. Several frameworks and software layers have been proposed by the data processing and systems communities, in order to help create and deploy such applications. At the lowest level of the stack, we have raw virtual machines (e.g., in Infrastructure-as-a-Service cloud offerings) which are hard to deploy, manage, and program in a distributed setting. Most applications instead use a layer above the machine: data intensive applications such as map-reduce have historically used YARN [2] as a resource manager, whereas micro-services are deployed on Kubernetes [7] and Docker [6]. There is increasing interest in running data-intensive applications on Kubernetes (often abbreviated as *k8s*) and Docker as well.

Unfortunately, both YARN and Kubernetes offer bare-bones compute abstractions with no support for customized dataflows that are: (a) easy to deploy, (b) usable for offline and real-time dataflows or a mix of both, and (c) resilient to failures. Another option is to use specific dataflow systems such as Storm [5] to implement other applications. However, these systems make too many assumptions and choices that limit their general applicability across a broad range of applications. For example, they *prescribe* a particular data format, query model, and resiliency strategy (such as none, checkpoint-replay, or active-active). Thus, these solutions are not usable across the broad range of applications identified above. In addition, these solutions are not optimized for containerized environments that are becoming ubiquitous today.

As a result, today, there is severe fragmentation in the application ecosystem, where each system has created its own abstractions and implementations of its own building blocks to achieve its dataflow requirements at high performance. Examples include Storm [5], Spark [4], and Flink [1], which share no code commonalities today (apart from the lowest layer of YARN or Kubernetes). This fragmentation has resulted in repeated "re-invention of the wheel" with redundant re-implementation of significant parts of the stack that could have been shared. Further, this made it very hard to run such diverse applications in a shared environment.

We propose a new runtime layer called *Common Runtime for Applications* (*CRA*)[1]. CRA provides a generic distributed dataflow abstraction and deployment functionality *without* making choices that applications would like to control. At the same time, CRA offers significant functionality that makes it easier to build such applications. For example, CRA does not interfere with the *data plane* of the distributed dataflow,

*Work started during internship at Microsoft Research.

[1]CRA is available as open-source software, and can be downloaded at `https://github.com/Microsoft/CRA`.

exposing raw network streams between (virtual) endpoints, instead of a specific data format or protocol. This allows applications to choose their own data format and application protocol between dataflow vertices. CRA exposes the capability of running multiple copies of a vertex and switching over between them on failure, allowing applications to build dataflow graphs with active-active or active-standby resiliency models (in addition to the usual checkpoint-replay capability). CRA supports sharding primitives and rich communication patterns that enable complex and potentially long-running dataflows to be constructed, deployed, and maintained.

CRA exploits the recent advances in containerization to efficiently execute the physical deployments of dataflow graphs. The dataflow vertices are packaged into Docker containers, and we use a container orchestration framework such as Kubernetes [7] to deploy them. CRA leverages the features of the orchestrator (e.g., code deployment and worker instantiation, liveness and heartbeats for resiliency) and provides the additional functionality to make it possible to build resilient long-running dataflows with customized resiliency strategies, data delivery semantics, and scale out schemes.

With CRA, we were able to simplify the implementation of several dataflow-based systems such as (1) Quill [9], a distributed data-intensive temporal analytics engine; and (2) Ambrosia [11], a distributed programming framework based on reliable message delivery. In this paper, we focus only on the basic design and features of CRA. We also evaluate CRA using various micro-benchmarks. More details on CRA, its data processing layer, its applications, and a more complete evaluation can be found in our technical report [10].

II. CRA System Overview

A. Basic CRA Concepts and Design

1) Defining Vertices: At the simplest conceptual layer of CRA, we allow users to express their computation as a logical graph of computation units called *vertices*. Each *vertex* may be associated with named input and output *endpoints*, which are used to connect pairs of vertex instances. Input and output endpoints have to implement `FromStream` and `ToStream` functions, in order to receive and send data, respectively. Note that CRA provides a `Stream` abstraction to transfer bytes from one vertex to another. This enables applications to define their own data formats (e.g., columnar serialization) and send them another vertex without incurring the cost of extra memory copies. Further, endpoints can optionally enable the ability to communicate with other endpoints using shared memory, by implementing `FromOutput` and `ToInput` functions. These functions take objects as parameters, and allow a vertex to access the memory of objects belonging to the other vertex without incurring a copy. CRA is responsible for choosing this mode of transfer if source and destination vertices support it and if they are located on the same CRA worker.

2) Logical Graph Creation and Deployment: Given a set of vertex definitions as described above, one can programmatically register these definitions with CRA, instantiate vertices on a set of named CRA *worker instances*, and connect the vertices using *connections* (or edges), to/from the logical graph. The user programmatically interacts with the CRA *client library* to perform these operations.

3) Physical Deployment: CRA workers can be spawned off as processes from the OS shell. Alternatively, such workers can be packaged into Docker containers and deployed on a set of physical (or virtual) machines using a resource orchestrator such as Kubernetes. The orchestrator ensures that CRA worker instances are started (and re-started) as necessary on the cluster. These CRA workers use the metadata and take responsibility for maintaining the physical instantiation of the user-defined dataflow graphs in the presence of ongoing machine and connection failures.

B. Sharded Vertices & Endpoints

Building upon the concept of a graph of vertices, we provide a layer that exposes sharded equivalents of vertices and endpoints to users. A *sharded vertex* represents a certain number of copies (called *shards*) of a vertex instantiated in the data center, but referenced as a single entity. For clarity, we will refer to the normal vertices from Section II-A as *simple vertices*. Analogous to our endpoints from earlier (we will call them *simple endpoints*), we define the notion of *sharded endpoints*. A sharded input (or output) endpoint implements the sharded equivalent of `FromStream` (or `ToStream`), that takes an array of `Stream` objects as argument. Interestingly, sharded input and output endpoints may exist in both simple and sharded vertices. Sharded endpoints simply add the capability of receiving from (or sending to) a sharded source (or destination) vertex, respectively.

C. System Architecture

Figure 1 depicts the overall architecture of CRA. We have designed CRA as an embedded client library that is linked to code for three distinct entities:

- *Vertices and Endpoints*: The user code for vertices and endpoints are given a reference to the client library, which they can use to create or delete connections, instantiate new vertices, etc.
- *Deployers*: Code that is used to create vertices and connections, and instantiate them on specific worker instances (identified by names) may be written by deployment code, that lives outside vertices and endpoints.
- *Worker Bootstrap*: The worker instances themselves are written as a simple bootstrap program that uses the CRA client library to create a worker instance and start it in the current process. We offer the CRA worker as a Docker container that can be deployed on a cluster using resource orchestrators such as Kubernetes, that handle deployment (and re-deployment) and failure detection.

D. Metadata Management

CRA stores metadata in a key-value store. This is a pluggable module. Our implementation for Microsoft Azure uses Azure Table Storage to store such data. CRA stores a variety of metadata:

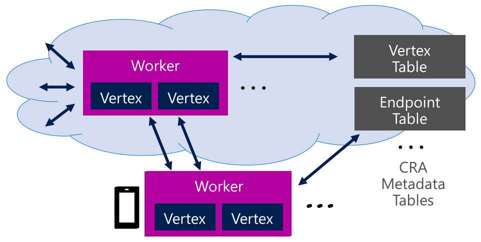


Fig. 1. CRA Overall Architecture

- List of logical vertex definitions, along with a pointer to files on storage that contain the binary related to the vertex.
- List of worker instances that have been defined. For workers that are active, additional information such as the IP endpoint and port are stored as part of the metadata.
- List of vertex instances associated with specific workers. A vertex may be instantiated on more than one worker – in this case, at most one instance may be designated as "active". Each instance is also associated with parameters that are associated with instantiating that specific instance.
- The list of logical connections between vertex endpoints, which represents the edges in the CRA dataflow graph.
- For sharded vertices, the sharded vertex names and list of instances associated with the shard.

E. *Worker Lifecycle*

Recall that the user registers a set of CRA worker instances, with a unique name for each worker. When a worker (with a unique name) is started by the resource orchestrator on some physical machine, it exposes a listening server on a registered port (whose information is present in the metadata entry for that worker instance). This allows remote workers and clients to establish connections and issue various requests such as connecting to an endpoint, notifying of a newly added vertex to the worker, and vertex removal.

When a worker is created, it first queries the metadata tables to identify what vertices need to be instantiated on that worker instance. For each vertex instance, CRA downloads the application code from the appropriate location in storage and dynamically instantiates the vertex object. It then downloads the vertex parameter and calls a special method `Initialize` on the vertex object, passing it the parameters registered during logical vertex instantiation and stored as part of the instance metadata, as described in the section on metadata management. Then, for each connection originating or ending at this vertex, CRA establishes the TCP connection from this endpoint to its matching remote endpoint. When a connection between two endpoints is established, CRA calls the `FromStream` and `ToStream` calls on the corresponding endpoints in order to invoke user code after a successful connection. This process is repeated for all connections to/from the instance.

F. *Handling Failures*

CRA handles both connection and node failures for the application. When a CRA worker fails, it depends on the resource orchestrator (such as Kubernetes) to detect this and reinstate the container elsewhere on the cluster. On worker re-creation, CRA goes through the process described above to reinstate the vertices and connections on that worker.

Other workers may be hosting vertices that are reading and writing to the failed vertex. When their network streams break, the endpoint code receives an exception that it can handle appropriately, and transfer up to CRA (which instantiated the `ToStream` or `FromStream`) code that was interacting with the network stream. The CRA worker then re-establishes the connection and returns control back to user code. It is possible that both sides of a connection try to establish the connection at the same point. CRA ensures that only one connection succeeds using backoff and retry logic on both sides. The protocol also supports the notion of requesting a connection with a parameter `killRemote`. When this is set to true, the CRA worker on the other side uses a "task cancellation" token to force the endpoint to kill its execution, and replaces the connection with the new incoming one. This feature is necessary because one side of the connection may be unaware that the connection is in a failed state, e.g., because the user logic on the other side is not even trying to use the stream at that point in time (e.g., waiting on a different stream or file).

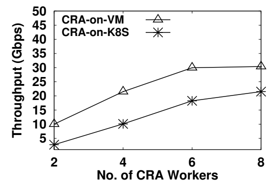
G. *Handling Vertex Replicas*

As mentioned earlier, CRA supports multiple replicas of the same vertex running on different worker instances. Only one vertex is designated as "active" at a given point. A vertex becomes active when it leaves the `Initialize` method. Thus, replicas typically stay in Initialize until they are ready to take over computation, at which point they exit `Initialize` and take over control. When vertex becomes active, existing connections that are broken (because the older active is no longer available) look up the current active destination in the metadata, find the newly activated replica, and establish connection to it. In parallel, the newly activated replica also proactively tries to create its outgoing and incoming connections, so that the distributed vertex graph is restored to the logically correct global state.

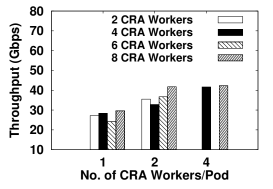
III. Experimental Evaluation

In this section, we evaluate CRA's performance using micro-benchmarks. In all experiments, we use a sharded broadcast scenario as an example. For a given number of vertices, we construct a fully connected mesh, i.e., each vertex is connected to every other vertex. Each vertex sends (receives) a fixed amount – 100MB – of data to (from) every other vertex. We use sharded broadcast as an example of the basic "data exchange" operator used in many analytical queries. Our target metric is measured throughput in Gigabits per second (Gbps). Unless otherwise noted, we run all experiments on two Azure VMs. Similarly, for K8s, we use an Azure Kubernetes Service (AKS) cluster with two agent nodes.

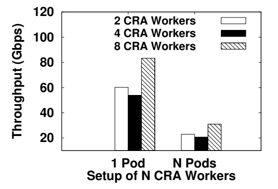
1) Overhead of Containerization: In the first experiment, we measure the overhead of containerization vs. VMs. We vary the number of CRA workers (or instances) in each VM/Pod



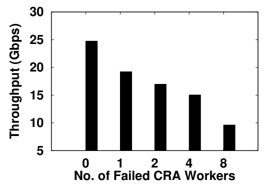
(a) Containerization Overhead



(b) Effect of Packing Workers



(c) Effect of Shared Memory



(d) Overhead of Failures

Fig. 2. Micro-benchmark Evaluation of CRA

and measure the throughput. Figure 2(a) shows the results of this experiment. Overall, throughput with VMs is between 1.4x to 3.6x higher as compared to K8s.

2) Packing CRA Workers on K8s Pods: In this experiment, we vary the number of workers packed in a K8s Pod and measure the impact on throughput. We show the results in Figure 2(b), where x-axis is the number of CRA workers per Pod, while the different colored bars represent the total number of CRA workers. We can see that as we pack more CRA workers in a Pod, we get higher throughput. This means that for network intensive applications, it is desirable to more tightly pack CRA workers into Pods.

3) Effect of CRA's Shared Memory Optimization: In this next experiment, we want to directly measure the impact of CRA's shared memory communication on throughput. We show the results in Figure 2(c). We explore two packing strategies (1) fully-packed: pack all CRA workers into a single pod, labeled as "1 Pod" (2) fully unpacked: one Pod per CRA worker, labeled as "N Pods". We run with 2, 4, and 8 CRA workers in total (represented as different bars). These results show that CRA's shared memory communication can improve throughput by up to 5x as compared to regular TCP sockets, showing the effectiveness of this optimization.

4) Effect of Failures on CRA: In this experiment, we study the impact of failures, which are quite common in practice, on CRA's performance. We fix the number of CRA workers to 8, running in two Pods (4 workers per Pod). We then randomly fail some of them. We show the results in Figure 2(d), with 0, 1, 2, 4, and 8 failed CRA workers. We see that CRA's performance gracefully degrades with the number of failures. This experiment shows that CRA effectively builds on top of the automatic detect and restart capability of K8s to provide fault tolerance for applications built on top of CRA.

IV. Related Work

Simiar to CRA, Hyracks [8], Dryad [12], and Nephele [14], adopt the notion of representing data processing as vertices and edges in a DAG. However, unlike CRA, these systems provide data processing engines of their own, and hence the other aspects of building and deploying applications (e.g., sharding, recoverability, elastic scaling) are built into the engine. This tight coupling makes these systems less flexible.

Apache Tez [13] and Apache REEF [3] facilitate building custom dataflow applications. To that end, these systems have similar goals as CRA. However, they are tied to the YARN ecosystem and have been exclusively developed to avoid "re-inventing the wheel" in the YARN community. In contrast, CRA is much more general purpose and can be widely used inside and outside of the YARN ecosystem. Further, CRA exposes raw network streams to applications and gives users full control over the data plane. CRA also exposes a capability to run highly resilient long-running workflows with support for different application-defined resiliency strategies such as active-active, and checkpoint-replay, as well as first-class support for making it easy to write applications with sharding and dynamic topologies.

V. Conclusions

In this paper, we presented the design and implementation of Common Runtime for Applications (CRA), a cloud-native runtime with a common interface to build a wide variety of data center applications. We showed how system and application designers can exploit next-generation virtualization technologies such as containers, which have become the de-facto building blocks for packaging and deploying cloud-scale applications. The CRA architecture enables data-centric applications to be first-class citizens on these emerging platforms. Finally, we welcome the community to contribute to and build on CRA, which is now available as open-source software.

Acknowledgments. We thank Mike Barnett, Gautam Bhakar, Darren Gehring, Jonathan Goldstein, Niel Lebeck, and Christopher Meiklejohn for their help and contributions.

References

[1] Apache Flink. https://flink.apache.org/, 2018.
[2] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2018.
[3] Apache REEF. http://reef.apache.org/, 2018.
[4] Apache Spark. https://spark.apache.org/, 2018.
[5] Apache Storm. http://storm.apache.org/, 2018.
[6] Docker. https://www.docker.com/, 2018.
[7] Kubernetes. https://kubernetes.io/, 2018.
[8] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.
[9] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar. Quill: Efficient, transferable, and rich analytics at scale. *VLDB Endow.*, 2016.
[10] B. Chandramouli, U. F. Minhas, and I. Sabek. CRA: A Common Runtime for Applications. Technical report, MSR-TR-2019-2, 2019. https://aka.ms/cra-tr.
[11] J. Goldstein et al. Ambrosia: Providing Performant Virtual Resiliency for Distributed Applications. Technical report, 2018. https://aka.ms/amb-tr.
[12] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.
[13] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *SIGMOD*, 2015.
[14] D. Warneke and O. Kao. Nephele: Efficient parallel data processing in the cloud. In *MTAGS*, 2009.