CRA: A Common Runtime for Applications

MSR Technical Report, MSR-TR-2019-2

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Umar Farooq Minhas
Microsoft Research
ufminhas@microsoft.com

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

ABSTRACT

Today, a modern data center hosts a wide variety of applications comprising batch, interactive, machine learning, and streaming applications. A large majority of these applications can be abstracted as a "distributed dataflow" graph. Even though this commonality exists and can be exploited in the applications' design and implementation, each application is typically written from scratch, resulting in significant inefficiencies in developer productivity. In this paper, we factor out the commonalities for many types of big data applications, into a generic dataflow layer called *Common Runtime for Applications* (*CRA*). In parallel, another trend, with *containerization* technologies, has taken a serious hold on cloud-scale data centers, with direct implications on the design, implementation, and deployment of next-generation of data-center application. Container engines (e.g., Docker and CoreOS) and cloud-scale container orchestrators (e.g., Kubernetes and Docker Swarm) are two important technologies that enable this trend. Container orchestrators 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 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. To show the practicality of our approach, we built a distributed analytics engine on top of CRA, namely Quill. We show through in-depth micro- and macro-benchmark results, that CRA provides significant performance improvement over an unoptimized implementation on modern cloud platforms. CRA is available as open source, and can be downloaded at `https://github.com/Microsoft/CRA`.

1 INTRODUCTION

With the growth in data volumes acquired by businesses today, there is a need to deploy rich *dataflows* over the data. A dataflow consists of a graph of computation vertices that each hold state and read/write data to other vertices in the graph. Dataflows can operate on both historical (bounded) and real-time (unbounded) datasets. 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 over the cloud and edge 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. Consider an application in a completely different domain of microservices. 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. Similarly, Web and cloud-edge [24] applications consist of client vertices that communicate with Web frameworks such as PHP and NodeJS running on vertices in the cloud, which communicates result requests back to the clients – another instance of a distributed dataflow.

1.1 Problem and Today's Solutions

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 [3] as a resource manager, whereas microservices are deployed on Kubernetes [14] and Docker [12]. 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 barebones compute abstraction 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 instead use specific dataflow

*Work started during internship at Microsoft Research.

systems such as Storm [8] to implement other applications. However, these layers do too much: they make too many assumptions and choices that limit the general applicability of the framework across a broad range of applications. For example, they *prescribe* a particular data format, query model, specific resiliency strategy (such as none, checkpoint-replay, or active-active), scale out scheme, data delivery semantics (e.g., exactly once or at most once), data processing semantics (e.g., offline or real-time queries), network topology, and distribution. As a result, these solutions are not usable across the broad range of applications identified above. For instance, we would not consider running an offline job on Storm if efficiency were a concern. Finally, 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 their own building blocks to achieve their dataflow requirements at high performance. Examples include Storm [8], Spark [7], and Flink [2], 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.

1.2 Our Solution: CRA

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, 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. Figure 1 shows where CRA fits in the full application stack.

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

CRA has two layers of abstraction. First, users define a *logical* dataflow topology as a directed graph of named vertices. Vertices are associates with endpoints that may be connected via edges (or connections). Vertices may either be single (one logical copy of the computation) or sharded (with N copies of the computation). Applications create the logical dataflow programmatically, and specify code for each vertex and endpoint in the system. Endpoints communicate with other endpoints via standard network stream connections that are provided by the runtime, or using shared memory if possible.

Second, users specify a *physical* deployment topology for the dataflow. The physical topology consists of a set of CRA *workers*, each with a unique name. A worker is an OS process that can host one or more vertex instances. Multiple different vertices may be instantiated on the same worker instance, and vertices on the same worker may communicate via shared memory for performance (e.g., when reading from a vertex hosting cached input datasets). A sharded vertex is mapped to a set of worker instances. Further, users may map a vertex to more than one worker. In this case, only one copy is designated as "active", while the other instances are used as active or passive standby copies.

Interestingly, CRA's physical deployment is one step removed from actual execution on a cluster as follows. CRA workers are packaged into Docker containers, and we use a container orchestration framework such as Kubernetes [14] to deploy the worker instances. The orchestration framework handles hard problems such as code deployment and worker instantiation, liveness and heartbeats for resiliency, and monitoring tools. CRA leverages these features of the orchestrator and provides the additional functionality to make it possible to build resilient long-running dataflows with customized resiliency strategies, data delivery semantics, data processing and query semantics, and scale out schemes.

We have used CRA to implement several concrete dataflow applications. Examples include (1) a data-intensive analytics application (Quill [25]) that enables offline and real-time analytics over temporal data using Trill [26] (a real-time streaming library); and (2) a resilient distributed actor [22] framework based on reliable message delivery between actors.

This paper focuses on the design and features of CRA, and describes how we built support for data-intensive applications in CRA. With CRA and its data layer, we were able to re-build a previous system (Quill) for Kubernetes/-Docker in less than 200 lines of code. We use CRA/Quill to perform a detailed analysis of performance of data-intensive applications in the Kuberbetes/Docker environment, in comparison to naive deployments of today's big data analytics solutions such as Spark. We believe the latter contribution is interesting in its own right. It provides new insights into

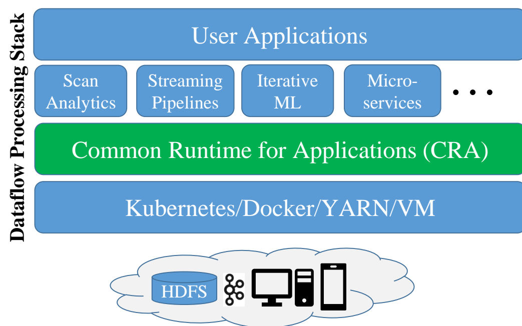


Figure 1: CRA Application Stack

how good a fit new infrastructures such as Kubernetes and Docker are for data-intensive applications, and explores how features in these infrastructures may be optimally leveraged for performance.

The rest of the paper is organized as follows. We start with background on containerization in Sec. 2. We provide an overview of CRA and the interface it exposes to application developers in Sec. 3. This is followed by a description of CRA's system architecture in Sec. 4. We describe our data processing layer and how we used it to implement Quill in Sec. 5. We provide an extensive evaluation of data-intensive computation in containerized environments in Sec. 6. We discuss the implication of these results as well as other extensions to CRA in Sec. 7. Finally, we discuss related work and conclude the paper in Sec. 8 and 9, respectively.

2 CONTAINERIZATION BACKGROUND

There has been a growing industry trend to move away from "monolithic" software architectures due to their rigid design, towards a "microservices" architecture. In such an architecture, a service comprises many loosely coupled, smaller applications – that talk to each other using well-defined, light-weight APIs (e.g., REST). Further, this architecture allows individual applications to be independently modified and deployed, making the larger service more flexible and agile.

Containerization technology, popularized by Docker [12] and CoreOS [11], is the key enabler of microservices. A container defines a standard packaging format, which when combined with the accompanying tools make it much more simpler to package, deploy, and run individual applications in a microservices architecture, across different environments. Containers are also key to deal with the *matrix of hell* [18], which is a result of a more complex software stack, that needs to run on a more complex and heterogeneous hardware stack.

Container technologies are sometimes referred to as "light-weight virtualization", in contrast to the heavy-weight hypervisor based virtualization technologies [19, 21]. The hypervisor (e.g., VMWare, or Xen) in a traditional virtualization setting is replaced by a containerization engine (e.g., Docker Engine). The virtual machine (VM) format is replaced by a light-weight container format. The main difference is that every VM carries a copy of a full operating system (OS) image, and the binaries and libraries needed to run the applications hosted inside the VM. On the other hand, containers share a single host OS image, and the needed binaries and libraries, whenever possible. As a result, containers are faster to deploy. However, we note that in a real cloud setting (e.g., Amazon AWS, Microsoft Azure), it is a common practice to run containers on top of virtual machines.

2.1 Docker

Docker [12] is arguably the most popular container format, and is supported by all the major cloud vendors including Microsoft Azure, Amazon AWS, Google Cloud, and IBM Cloud. Below we present an overview of the Docker terminology and the workflow. For more details we refer the reader to [29].

At the heart of the Docker ecosystem is *Docker Engine*, which is analogous to the hypervisor in the traditional VM settings. Given container specifications in a **Dockerfile**, a Docker Engine allows to build Docker **container images**, as well as to instantiate and run those images, as containers on the target host(s). To enable sharing, the images built on a local host can then be pushed to a central repository called **Docker Registry**, where other developers can search and download previously built Docker images. Finally, to deploy an application on a target host, a container image for that application is **pulled** from a Docker Registry and run on the host by the Docker Engine.

One of challenges of dealing with containers at scale is the need to efficiently manage their build, deployment, and monitoring across large clusters. This is typically achieved by **container orchestrators** such as Docker Swarm [13], Mesosphere [16], and Kubernetes [14]. In this work, we use containers orchestrated by Kubernetes, which we present next.

2.2 Kubernetes

Kubernetes is arguably the most popular container orchestrator, invented and deployed at scale, at Google [32] and later open-sourced [14]. Kubernetes is commonly used to deploy "elastic, distributed microservices", and automates the deployment, maintenance, scheduling, and scaling of containers across a cluster. Further, it provides mechanisms for resource provisioning, scheduling, auto-restart, and auto-scaling.

The main unit of deployment in a Kubernetes cluster is a **Pod**, which is simply an abstract resource that bundles one or more Docker containers together. An important point to note is that all the containers hosted inside a single Pod, share the resources such as IP address and port space, and storage. Further, all the containers in a Pod are always co-located on a single host, and are always co-scheduled. We refer the readers to [27] for more details on Kubernetes.

2.3 Data-intensive Computing on Containers

The open-source containerization community has mainly focused on “stateless” services. Although, there have been some recent advances in providing support for data-intensive services, which are “stateful”, the industry is lacking a good understanding of what it takes to run data-intensive services in these new environments. One of the goals of our work is to devise new architectures for container-friendly data-intensive computing and to shine a bright light on the strengths and weaknesses of running such services on container platforms.

3 CRA OVERVIEW AND INTERFACE

3.1 Requirements

Our goal is to create abstractions that one can use to build and deploy a distributed graph of objects that can send and receive data from one another. From an application developer’s perspective, the key requirements for such abstractions are:

- ***Format Independence***: Ability to send and receive data in an application-specific format and protocol, without interference from the runtime.
- ***Communication***: Ability to communicate between components via either message passing (TCP across nodes, fast loopback TCP within nodes) or shared memory, depending upon the application’s capability and physical placement.
- ***Location Independence***: Ability to write an application with multiple distributed components without having to specify the physical location of each component.
- ***Failure Independence***: The application should not worry about restarting components or establishing network connections in case of failure.
- ***Sharding***: The application must be able to shard its state and computation transparently, and possibly vary the number of shards dynamically during the deployment lifetime.

From the deployer’s perspective, the key requirements are the ability to programmatically control the physical placement of components on machines, containers within machines, the ability to run multiple copies of computations in order to implement failover, and the ability to pack multiple components within the same OS process or container.

In CRA, we leverage (1) Kubernetes and Docker for resource orchestration; and (2) Resilient Cloud Tables and Storage (e.g., Azure Tables and Blobs in Microsoft Azure) for decentralized metadata and persistent state (e.g., binaries, application state, logs) management. Kubernetes provides us with a resource orchestrator that packs Docker containers within a cluster of physical machines (or VMs). It also provides us with tools to monitor containers, check for failures using heartbeats, and recover failed containers on another machine.

3.2 Basic CRA Concepts and Design

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. Vertices are themselves packed into CRA *worker instances*, a shared-memory concept which roughly corresponds to an operating system process. CRA worker instances can be spawned off as processes from the OS shell. Alternatively, in Kubernetes, we can host one or more CRA workers in each Pod, and let Kubernetes deploy these Pods on a set of physical (or virtual) machines.

We describe these concepts with a simple running example where we have a pair of vertices connecting to and from each other, sending and receiving a fixed sequence of integers.

3.2.1 Defining Vertices. The following code fragment shows the *definition* of a simple CRA vertex that (a) writes a fixed sequence of integers to an output endpoint; and (b) reads a sequence of integers from an input endpoint and writes them to the console.

```
class IntExchangeVertex : IVertex {
  public IntExchangeVertex() { }

  public void Initialize(object arg) {
    int numInts = (int)arg;
    AddEndpoint("in1", new IntReader(numInts));
    AddEndpoint("out1", new IntWriter(numInts));
} }
```

As shown, vertices implement an `Initialize` function which CRA uses to create a vertex instance with user-defined parameters. The `AddEndpoint` function is used to define two CRA endpoints (an input and an output), passing each of them the number of integers to send/receive. We next define the two endpoints to send/receive the sequence of integers:

```
class IntReader : IInputEndpoint {
  int numInts;
  public IntReader(int numInts) {
    this.numInts = numInts;
```

```
  }

  public void FromStream(Stream s) {
    for (int i=0; i<numInts; i++)
      Console.WriteLine(s.ReadInt());
} }

class IntWriter : IOutputEndpoint {
  int numInts;
  public IntWriter(int numInts) {
    this.numInts = numInts;
  }

  public void ToStream(Stream s) {
    for (int i=0; i<numInts; i++)
      s.WriteInt(i);
} }
```

Input and output endpoints have to implement `FromStream` and `ToStream` respectively, in order to receive and send data. 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 vertices support it and if the source and destination vertex are located on the same CRA worker.

3.2.2 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 or from the logical graph. The user programmatically interacts with the CRA *client library* to perform these operations, as shown next.

```
var cl = new ClientLibrary();
cl.DefineVertex("IntExchangeVertex",
       () => new IntExchangeVertex());

cl.InstantiateVertex("inst01", "vert01",
             "IntExchangeVertex", 5);
cl.InstantiateVertex("inst02", "vert02",
             "IntExchangeVertex", 5);

cl.Connect("vert01", "in1", "vert02", "out1");
cl.Connect("vert02", "in1", "vert01", "out1");
```

Here, the `DefineVertex` function call associates the string "`IntExchangeVertex`" with the creation of an instance of type `IntExchangeVertex`. The second parameter to the function `DefineVertex` is a *lambda expression*, an anonymous function that can be serialized and sent over the wire for invocation at a remote location, either immediately or in the future (e.g., when an instance is restored after failure).

The function `InstantiateVertex` takes an instance name, a vertex name, a vertex definition, and an initialization parameter as arguments. It logically creates an instance of the vertex with the given definition, on the named CRA worker. The created vertex is associated with the given vertex name. In our example, two vertices with the definition of `IntExchangeVertex`, called `vert01` and `vert02`, are instantiated on CRA workers `inst01` and `inst02` respectively. Finally, we create the connection between the endpoints of the two vertices using the CRA `Connect` function, which takes two {vertex name, endpoint name} pairs, one for the source and another for the destination, to be connected.

3.2.3 Physical Deployment. Note that the instantiation of the graph described above is logical. For example, we do not require the CRA worker instances be deployed a priori, before issuing these commands. The commands result in metadata being created in CRA tables, that will allow the two instances, when created (perhaps at a later point in time), to host the corresponding vertices. CRA workers are packaged into Docker containers and deployed 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. For example, when a machine fails, the vertices sending data to (or receiving data from) that node see an exception in their network stream, which propagates to the CRA worker code, which is responsible for re-establishing the connection and returning control back to user code. We discuss CRA's contract with user code in more detail in Section 4.

3.3 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 3.2 as *simple vertices*. We assume that the number of shards is fixed at deployment time, and revisit this assumption in the context of dynamic topologies in Section 7.3.

Analogous to our endpoints from earlier (we will call them *simple endpoints*), we define the notion of *sharded endpoints*, identified by endpoint names as before. 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. In this section, we assume that the total number of shards N for a sharded vertex is fixed during its deployment. The ability to assign zero or more shards to each container allows us to re-balance load. For example, we can have one container with N shards at one extreme, and up to N containers with one shard each at the other extreme. In order to scale beyond this limit, CRA also allows the number of shards to vary over the lifetime of a vertex; we cover such *elastic* sharded vertices in Sec. 4.

The existence of simple and sharded vertices gives a rise to four types of connections in a CRA graph:

- **Simple-to-simple**: These are simply the point-to-point connections introduced in Section 3.2; both connection ends consist of simple vertices with simple endpoints.
- **Simple-to-sharded**: A simple vertex with a sharded output endpoint can connect to a sharded vertex with a simple input endpoint. This corresponds to a 1:many connection from the simple vertex to the sharded vertex members.
- **Sharded-to-simple**: A sharded vertex with a simple output endpoint may connect to a simple vertex with a sharded input endpoint. This corresponds to a many:1 connection from the sharded vertex members to the simple vertex.
- **Sharded-to-sharded**: Finally, a sharded vertex with a sharded output endpoint may connect to a sharded vertex with a sharded input endpoint. This corresponds to a many:many (or mesh) connection from the members of the sharded source vertex to the members of the sharded destination vertex.

For example, a simple vertex that broadcasts an integer sequence to each member of a sharded destination vertex might be implemented as follows:

```
class IntBroadcastVertex : IVertex {
  public IntBroadcastVertex() { }

  public void Initialize(object arg) {
    int numInts = (int)arg;
    AddEndpoint("out1", new ShardedIntWriter(
        numInts));
} }

class ShardedIntWriter : IShardedOutputEndpoint {
  int numInts;
  public ShardedIntWriter(int numInts) {
    this.numInts = numInts;
  }

  public void ToStream(Stream[] s) {
    for (int i=0; i<numInts; i++)
      for (int j=0; j<s.Length; j++)
        s[j].WriteInt(i);
} }
```

Similarly, a sharded source vertex, where each source shard broadcasts a non-overlapping range of integers to each member of a sharded destination vertex, is shown next. Note that the only difference between a simple and a sharded vertex is the additional parameter (shard ID) in the call to `Initialize`.

```
class ShardedIntBroadcastVertex : IShardedVertex {
  public ShardedIntBroadcastVertex() { }

  public void Initialize(int shardId, object arg)
      {
    int numInts = (int)arg;
    AddEndpoint("out1", new ShardedIntWriter(
        numInts, shardId));
} }

class ShardedIntWriter : IShardedOutputEndpoint {
  int numInts, sourceShardId;
  public ShardedIntWriter(int numInts, int
      sourceShardId) {
    this.numInts = numInts;
    this.sourceShardId = sourceShardId;
  }

  public void ToStream(Stream[] s) {
    for (int i=0; i<numInts; i++)
      for (int j=0; j<s.Length; j++)
        s[j].WriteInt(s.Length*sourceShardId + i);
} }
```

The destination vertex and its sharded input endpoint are defined symmetrically (not shown). The deployment API for sharded vertices is very similar to that for the basic case: `DefineShardedVertex` is used to create the definition, while `InstantiateShardedVertex`, which now takes a set of instances and the number of shards per instance as parameters, is used to instantiate all the shards of the vertex (details omitted for brevity).

Figure 2 shows two CRA worker instances. There are two sharded vertices, each spanning both worker instances. Thus, each vertex has two shards. There is a single sharded connection between the two sharded vertices, which translates to a 2x2 cross-bar connection between the two vertices, as shown, with two physical output endpoints on each shard of

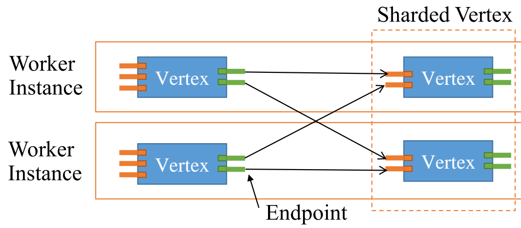


Figure 2: CRA Vertices and Endpoints

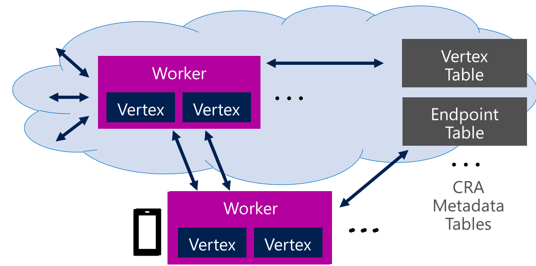


Figure 3: CRA Overall Architecture

the source vertex and two physical input endpoints on each shard of the destination vertex.

4 SYSTEM ARCHITECTURE

4.1 Overview

CRA is designed 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. For instance, an SQL query compiler may be responsible for translating the query into a graph that is deployed and executed.
- *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 (re-)deployment and failure detection.

We first describe how and what metadata is handled by CRA. We then describe how workers operate, and interact with vertex and endpoint code. We then describe the threading model and cover how system management is performed using CRA. Figure 3 depicts the overall architecture of CRA.

4.2 Metadata Management

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

- 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 instantiation parameters that are associated with instantiating that specific instance. For example, this may include the query subplan that the vertex instance is supposed to execute.
- 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.

4.3 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 or from the instance.

Note that even if the user defines a connection from A to B, CRA may physically establish the connection from B to A. This is important in cloud-edge scenarios, where the cloud machine may not be able to access the edge device, whereas the edge device can access the cloud machine. Similarly, connections from a public machine to a machine inside an organization's private cloud is made possible in the same way.

4.4 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 reestablishes 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, until its local in-memory table of connections contains the connection. 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).

4.5 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.

4.6 Flexible Threading Model

The interfaces shown in the previous section are *synchronous*. In this case, vertices can spawn a set of threads to manage their own concurrency within the process. This is useful if the vertex application is multi-threaded, such as Trill [26]. CRA also supports *asynchronous* versions of the interfaces, so that a pool of threads can be shared by all the endpoints on the worker process. The latter is a better fit for applications such as microservices that share a pool of threads to perform small units of activities.

5 DATA PROCESSING IN CRA

5.1 The Dataset Abstraction

To efficiently build dataflow applications, we provide a *Dataset* abstraction that allows users to create and process datasets using CRA. By implementing such an abstraction, CRA knows how to deploy datasets inside CRA vertices, and processes them in an efficient manner. The proposed abstraction is used to define keyed datasets (e.g., `Dataset<TK, TP>` where `TK` and `TP` are the key and payload types of each tuple in the dataset, respectively). The user is responsible for implementing the following three types of functions:

Serialization/Deserialization. To support data movement operations, CRA should understand how to send and receive the dataset tuples between different vertices. To that end, and similar to CRA endpoints, we provide `ToStream` and `FromStream` functions that users should implement to serialize and deserialize the dataset tuples to and from streams, respectively. These functions will be invoked internally within CRA endpoints. We also provide `ToObject` and `FromObject` variations to utilize shared memory endpoints.

Rekeying. Since many applications (e.g., key-value stores and map-reduce programs) and queries (e.g., group-by) require rekeying the data for efficient processing, we provide a `Rekey<TK2>(...)` function that associates each tuple in a dataset of type `Dataset<TK1, TP>`, with a new key type `TK2`. `Rekey` function takes a lambda expression as input, which selects the new key type `TK2` from the payload, and returns a new dataset of type `Dataset<TK2, TP>`. Rekeying is a *transformation* operation which does not actually change the key of each tuple in the dataset when it is called, however, it only returns a new dataset instance with the appropriate output type and waits for an action that triggers the actual computation later (similar to Spark [7]).

Actions. We provide a `Subscribe(...)` function that triggers the immediate computation of all *transformations* (e.g., rekeying) on the dataset till that point, and returns the computation results. This function is particularly useful for client applications to use as an entry point for execution. For example, a user can implement a subscribe function that accepts a lambda expression that will be invoked for every tuple in the results and log it. For example, `Subscribe(e => new LogToConsole(e))` logs every tuple to the console.

5.2 The Sharded Dataset Abstraction

We propose a *Sharded Dataset* abstraction that leverages both the extensibility of sharding concepts in CRA and the *Dataset* abstraction presented in this section. The sharded dataset provides a small, yet, rich set of data transformation and movement operations that enables building distributed dataflow plans in an efficient and scalable manner. Below, we highlight the main features of the sharded dataset abstraction:

5.2.1 Creation and Transformation. A sharded dataset is conceptually a set of data shards, where each shard is implemented with the *Dataset* abstraction. The CRA client library supports creating a sharded dataset through the function `CreateShardedDataset(...)` that takes a lambda expression defining a constructor for creating the dataset on each shard. For example, we can create a sharded dataset named `s` that consists of 2 shards of a dataset of type `IntDS` as follows:

```
var s = cl.CreateShardedDataset(n=>new IntDS(n),
    2);
```

In this example, the lambda expression takes a shard id `n` as argument and constructs an instance of `IntDS` (which implements the *Dataset* abstraction) per shard. We assume each instance creates a list of 100 keyed tuples, where each tuple has an integer key `K` and two random integers `R1` and `R2` as payload. During execution, CRA will create `s` on a sharded vertex that has 2 shards. Note that dataset creation is a transformation operation where it just associates the lambda expression to the sharded dataset instance upon its call. The actual creation will be described later in Section 5.2.3.

Transformation. We support a general transformation operation, namely `Transform(...)`, that transforms a dataset from one type `T1` to another type `T2`. It takes a lambda expression that executes on every shard independently when execution is triggered, and produces a new sharded dataset. There is no data movement across shards in this operation. In our running example, suppose we wish to create a new dataset `s1` that uses the same key in `s`, but, selects only the first random integer `R1` from the payload. This can be done through transformation as follows:

```
var s1 = s.Transform(a => a.SelectR1());
```

We also support multi-input variation of the transformation operation, which operates on two sharded datasets at the same time, and outputs a single sharded dataset. In addition to `Transform(...)`, we provide a sharded variation of the `Rekey(...)` operation. The sharded dataset forwards the rekeying lambda expression to each data shard to run independently, without any data movement. As an example on sharded rekeying, assume we wish to rekey the tuples in `s1` to have a new dataset `s2` with keys of values 0 and 1 only:

```
var s2 = s1.Rekey(a => a % 2);
```

5.2.2 Data Movement. We provide a generic `Move(...)` operation to efficiently support moving data across dataset shards. Basically, this operation accepts two lambda expressions that capture the plan for sending and receiving data between different shards. These expressions are as follows:

- *Splitter* expression which executes on each shard that will send data. This expression defines how to produce a set of M datasets that will be sent to M shards.
- *Merger* expression which executes on each shard that will receive data. This expression defines how to produce a single dataset out of the L datasets that will be received from L shards.

Using the splitter and merger expressions, users can implement most, if not all, operations needed to organize or duplicate the shard contents in specific ways. This includes *resharding* for load-balancing, *broadcasting* for duplicating every shard on every output shard, and *multi-casting* for duplicating every shard on zero or more output shard. For example, to implement a broadcast operation among the N shards of a sharded dataset, the splitter should produce N exact copies of the dataset on each sender shard, and the merger should produce one dataset from N datasets received on each receiver. In our running example, suppose we wish to broadcast the data of sharded dataset $s2$ among its shards and produce a new sharded dataset $s3$:

```
var s3 = s2.Move(n => new BroadcastSplitter(n),
            n => new BroadcastMerger(n));
```

The lambda expressions take the shard id n as a parameter to construct the proper splitter and merger for each shard.

5.2.3 Deployment and Execution. We provide a function named `Deploy()` to deploy a dataflow plan defined on a sharded dataset. Once the deployment operation is called, the CRA client library extracts the operations topology expressed by the sharded dataset API, and transforms it into a set of *tasks* that will be executed as CRA sharded vertices. A task is a set of logical operations (e.g., create, transform, rekey) that can be executed inside a sharded vertex. Transformations are pipelined and packed into a single task. However, a data movement operation breaks the pipeline into

Operation Type	Quill API	CRA Sharded Dataset API
Transformation	Query	Transform
	Rekey	Rekey
	Reshard Broadcast Multicast Redistribute	Move
Actions	ToMemory ToStorage ToBinaryStream Subscribe	Subscribe

Table 1: Quill Re-implementation Using CRA Sharded Dataset.

two tasks; one that ends with a split operation, and another that starts with a merge operation. In our running example, we can trigger the deployment as follows:

```
var s4 = s3.Deploy();
```

The deployed plan will consist of two tasks. The first task executes the *create*, *transform*, *rekey*, and *split* operations, while the second task executes the *merge* operation. Note that deploying the plan never triggers the execution. To start execution, we should call a subscribe call as follows:

```
s4.Subscribe(()=>new LogToConsole());
```

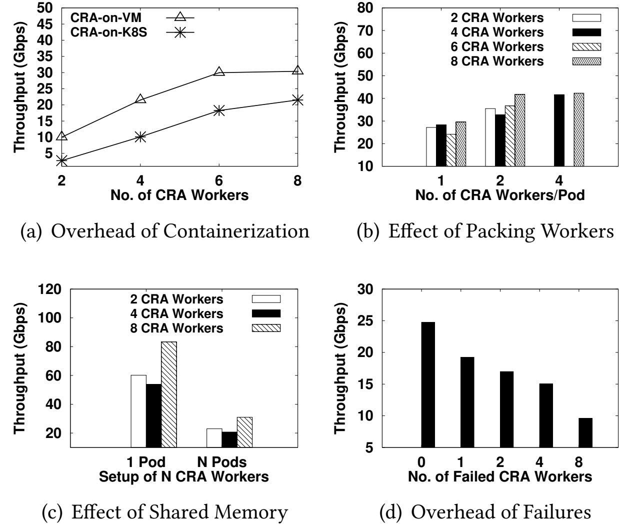
This forwards the lambda expression to the subscribe operation of each dataset shard and waits till the whole execution finishes and logs the results.

5.3 Case Study: Quill-on-CRA

We show a case study on the applicability of CRA in building efficient dataflow applications. We used CRA to re-implement Quill [25], a high-performance distributed platform for offline and real-time analytics over temporal data. We refer to our re-implementation of Quill as *Quill-on-CRA*.

Implementing StreamableDataset. Quill is based on a sharded temporal dataset, named *ShardedStreamable*, which is built on top of a real-time streaming library called Trill [26]. So, our first step in implementing Quill-on-CRA was providing a new dataset type, named *StreamableDataset*, that basically implements the *Dataset* abstraction presented earlier in this section, using the ShardedStreamable constructs.

Re-implementing Operations. By having the StreamableDataset implementation, we gain access to the whole set of operations that are provided by the CRA sharded dataset abstraction on top of StreamableDataset. Thus, our second step was re-implementing the original Quill API using these sharded dataset operations. Table 1 shows which sharded



(a) Overhead of Containerization (b) Effect of Packing Workers

(c) Effect of Shared Memory (d) Overhead of Failures

Figure 4: Micro-benchmark Evaluation of CRA

dataset operation is used to implement each Quill functionality. Note that different data movement operations (e.g., Reshard and Broadcast) are implemented by providing different split and merge expressions as described in Section 5.2.2. Analogously, different action operations (e.g., ToMemory and ToStorage) are implemented using different subscribe lambda expressions. In total, the implementation of Quill-on-CRA has less than 200 lines of code. This shows the practicality of CRA in building powerful dataflow applications. We extensively evaluated Quill-on-CRA in Section 6.2 and showed its efficiency and scalability.

6 EXPERIMENTAL EVALUATION

6.1 Micro-benchmarking Evaluation

Our goal in this section is to (1) measure the overhead of containers in K8s vs. virtual machines, (2) explore different packing strategies for deploying data intensive applicatins on K8s, and (3) measure the impact of specific CRA's optimizations on performance and fault-tolerance.

Setup: For all the experiments in this section, 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 of data to (from) every other vertex. We fix this parameter to 100 MB in our experiments. We use sharded broadcast as an example of the basic "data exchange" operator quite commonly used in many analytical queries. Hence, studying its performance is important. For these experiments, our target metric is the measured throughput in Gigabits per second (Gbps). Unless otherwise

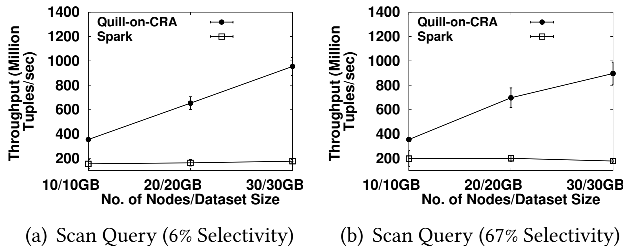
noted, for this section, we run all of experiments on two Azure VMs. Similarly, for K8s, we use an Azure Kubernetes Service (AKS) cluster with two agent nodes.

6.1.1 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 and measure the throughput. Figure 4(a) shows the results of this experiment. Overall, throughput with VMs is between 1.4x to 3.6x higher as compared to K8s. This experiment shows that the networking subsystem in K8s imposes a much higher overhead as compared to VMs. Our experience (results not presented here) shows that this overhead can be overcome by opening more parallel TCP connections when running in K8s.

6.1.2 Packing CRA Workers on K8s Pods. As noted earlier, CRA provides full flexibility over deployments – allowing the developers to pack different number of CRA workers inside a given container (VMs or Pods) to achieve the best performance. 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 4(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 (moving from left to right in the figure). This means that for network intensive applications, it is desirable to more tightly pack CRA workers into Pods. As we will show later, this not only results in higher throughput at K8s level, but CRA can exploit specific optimizations to get even better performance.

6.1.3 Effect of CRA's Shared Memory Optimization. As described earlier, CRA can detect if two workers are running on the same machine (sharing the same IP address), in which case, we short circuit the communication path to use shared memory instead of regular TCP sockets. Previous experiment showed that more tightly packing CRA instances into Pods improves throughput. 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 4(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.

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



(a) Scan Query (6% Selectivity) (b) Scan Query (67% Selectivity)

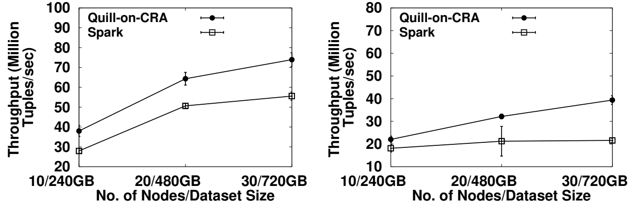
Figure 5: Big Berkeley Data Benchmark (Scan Query)

6.2 Berkeley Big Data Benchmark Evaluation

Our goal in this section is to evaluate the performance of Quill-on-CRA (Section 5.3), which is a case study on building efficient dataflow applications using CRA, in comparison to a state-of-the-art big data analytics platform. We use SparkSQL [20], with Spark v2.2, as our baseline.

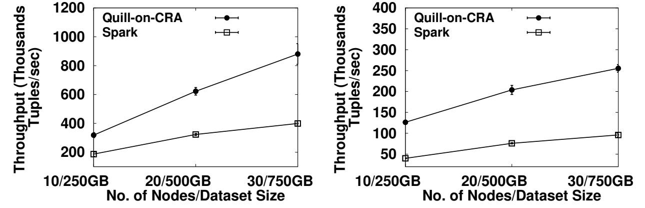
Deployment. We are interested in the Kubernetes deployment. Similar to micro-benchmark evaluation, we use Azure Kubernetes Service (AKS) to deploy both systems. As best configuration, we assign two workers per pod for each of CRA and Spark, where each AKS agent node has only one pod.

Benchmark. We compare the performance of both systems, while running the big data benchmark [30] that was used to evaluate both original Quill [25] and SparkSQL [20]. The benchmark contains three types of typical analytical SQL queries; *scan*, *aggregate* and *join*, which run on two types of datasets; *rankings* and *uservisits* (Schema is omitted due to space constraints). As instructed in the benchmark, we fix the size of each dataset per agent node to be 1GB (18M tuples) and 25GB (152M tuples) for rankings and uservisits datasets, respectively. However, to evaluate the scalability of both systems, we vary the number of agent nodes from 10 to 30 in all experiments, and hence, in total, we have up to 750GB data from the two datasets. Since we assign two workers per node, this also means that we have from 20 to 60 running workers for each of CRA and Spark. Our target



(a) Aggregate Query (52M Groups) (b) Aggregate Query (231M Groups)

Figure 6: Big Berkeley Data Benchmark (Aggregate Query)



(a) Join Query (65K Groups) (b) Join Query (231M Groups)

Figure 7: Big Berkeley Data Benchmark (Join Query)

metric is the measured throughput by each query in tuples per second.

6.2.1 Scan Query. We first start with the scan query which basically selects two columns, pageURL and pageRank from *rankings* dataset where the pageRank is higher than a certain value X. Figures 5(a) and 5(b) show the throughput results for the two systems when using a scan selectivity of 6%(low value) and 67%(high value), respectively. Overall, the throughput with Quill-on-CRA is between 2.2x to 5.4x higher than Spark in both selectivity cases. The lower throughput of Spark comes from the high overhead of scheduling the operation during the execution time. In contrast, Quill-on-CRA never incurs such overhead because of the lightweight runtime management in CRA.

6.2.2 Aggregate Query. The next query in the benchmark is an aggregate query over the *uservisits* dataset which computes the total revenue originated per sourceIP. Figures 6(a) and 6(b) report the throughput results for Quill-on-CRA and Spark when the number of groups is 52M and 231M, respectively. In general, Quill-on-CRA achieves higher throughput than Spark for two main reasons. First, the execution time of this query is bounded by the amount of data that will be shuffled among workers, and since CRA utilizes the shared memory communication between processes within the same pod, its performance will be better than Spark which employs network communication between workers even within the same pod. Second, the aggregate query plan that is generated by Quill is much more efficient than the one generated by the Spark optimizer as shown in [25]. Note that the average improvement ratio in case of 231M groups (1.5x) is higher than the average in case of 52M groups (1.2x). This confirms the first reason mentioned earlier as the amount of data shuffled in case of 231M groups is higher than 52M groups, and hence the CRA shared memory communication will be utilized much more.

6.2.3 Join Query. The last query is a join between *rankings* and *uservisits* datasets to obtain the sourceIP and associated pageRank that gave rise to the highest revenue for 20 years. Figures 7(a) and 7(b) show the results in case of having 65K and 231M groups according to sourceIP, respectively. Similar to the aggregate query, this experiment confirms our observation about increasing the average improvement ratio when the number of groups increases (i.e., increasing the amount of data shuffling). The average throughput of Quill-on-CRA is 1.9x and 2.7x higher than Spark with 65K and 231M groups, respectively.

7 DISCUSSION

7.1 Takeaways on Data Intensive Use of Kubernetes/Docker

As noted earlier, Kubernetes and the accompanying ecosystem has been solely focused on "stateless" (web) services. Only recently the community has turned its attention to supporting data intensive (stateful) applications as a first class citizen (e.g., by proposing Stateful sets [15]). Although CRA's design is independent of a particular resource orchestrator, CRA takes a very important step in demonstrating how to build data intensive, stateful applications on top of Kubernetes. We summarize our preliminary findings below:

- **Networking Overhead:** Containers impose additional networking overhead. Our experiments and experience show that, the per-connection throughput achieved from within a container could be, in many cases, much lower than the corresponding throughput with a VM. This is related to how networking layers are designed and implemented in Kubernetes, with more layers of abstraction, and hence a higher overhead. Architects of data intensive applications and platforms need to be aware of this limitation and work around it, for example, by opening more parallel connections when running inside containers.
- **Flexible Deployment:** When it comes to containerization platforms, such as Kubernetes, there is a lot of flexibility in choosing how to pack processes into Docker containers and how to pack those containers inside Pods.

CRA exploits this flexibility at its very core. Our experiments show that packing multiple processes in Pods can enable CRA to exploit shared memory communication, which can be up to 5x more performant, as opposed to communicating over TCP sockets. This is an important design choice.

- **Exploiting Orchestrator Capabilities:** Kubernetes solves many infrastructure level problems, including service discovery, failure detection, auto-restart, and replication. In general, it is important to follow a layering approach and fully exploit these capabilities at the higher layers. In our current design, CRA builds on top of the failure detection, and auto-restart capabilities of Kubernetes. Going forward, there is a possibility to deeply integrate CRA's elastic sharding with StatefulSets [15] in Kubernetes.

7.2 Other Applications on CRA

Beyond SQL, spatial, and temporal analytics, CRA supports the creation of vertices that can handle diverse applications such as machine learning, real-time stream processing, iterative, graph computation, distributed microservices, and key-value stores. For instance, machine learning and iterative computations can create sharded topologies with self-loops to implement the iteration. Real-time stream processing using long-running vertices that are made durable with checkpoint-replay, active-active, and active-standby forms of resiliency. Distributed microservices and three-tier architectures (clients, Web servers, databases) can also be mapped to a multi-level CRA topology across edge and cloud. A distributed key-value store uses the key-value requests as a stream from clients to servers that host the state, with responses sent asynchronously along reverse connections.

7.3 Dynamic Topologies

Since CRA applications may be long-running, there is often a need to change the topology, such as adding or removing vertices in the topology or allowing the number of shards in a sharded vertex to grow or shrink over time (e.g., to handle workload variations). CRA can support dynamic topologies by explosing the ability to add or remove vertices at any point. Further, it exposes information related to sharded to the CRA vertices and endpoints. Briefly, when a vertex is instantiated, it is provided with *sharding information* that identifies how many vertices are in the shard, and on which set of worker instances the shards are instantiated. Similar information is provided to vertices that connect to a sharded input or output endpoint. By providing and updating the information to all interested parties whenever the vertex topology changes, an application can support a dynamic topology according to its application-level semantics. While dynamic topologies are supported in CRA, full support for dynamic shards is currently being added to CRA; thus, the details and evaluation of this capability are outside the scope of this paper.

7.4 Detached Mode for CRA

By default, CRA vertices are written in the context of a base class that registers the vertex with the system. However, in some cases, we may want existing code to easily connect to vertices in the CRA vertex graph, without actively participating itself. For example, an edge device may wish to connect to an ingress vertex in the cloud (e.g., when devices are connecting to a sharded server exposing a REST API). For such scenarios, CRA supports a *detached vertex* mode, where it can programmatically connect to an existing vertex and get back a network stream over which it can send and receive data. This facility makes it easy to integrate external data sources to the CRA ecosystem of vertices.

8 RELATED WORK

Simiar to CRA, Hyracks [23], Dryad [28], and Nephele [33], adopt the notion of representing data processing as vertices and edges in a DAG. However, these systems provide a different level of abstraction and have different goals. As opposed to CRA, which provides a framework to build dataflow style processing engines on top, 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 [31] is probably the most closely related to our work. Tez is an open-source framework which, at its heart, provides a library for YARN that facilitates building dataflow style processing engines. Similar to CRA, using the Tez API, one can define an arbitrary DAG representing a custom dataflow. Tez promotes component re-use among different verticals built on top of YARN [3]. For example, Hive [4], and Spark [7] can be built using the Tez APIs, but applications still have to take ownership of resiliency strategies and use other frameworks to target non-YARN deployments. Apache REEF [6] and Apache Twill [10] facilitate building applications on YARN but provide a lower level interface than Tez, and hence are more general purpose. Apache REEF provides a library for building applications which can be ported to multiple resource orchestrators such as Apache YARN and Apache Mesos. Apache Twill provides a Java-like multi-threaded programming API for writing YARN applications. Twill's main goal is to reduce the complexity of developing YARN applications by offering a programming framework and common functionalities needed by many large scale distributed applications. To that end, all these systems have similar goals as CRA. However, unlike CRA,

Tez, REEF, and Twill are tied to the YARN ecosystem and have been exclusively developed to avoid "re-inventing the wheel" in the YARN community. CRA is much more general purpose and can be widely used inside and outside of the YARN ecosystem, as we show in this paper. 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.

Apache Spark [7] and Apache Flink [2] provide an API and a general purpose data processing engine. Similar to CRA, they also have a notion of a DAG used to represent data processing. However, CRA provides a lower-level API which gives much more flexibility to the application programmer to build and deploy custom dataflow graphs on top. It is possible to implement the Spark and Flink processing engines on top of the common runtime provided by CRA. As an example, in this paper, we showed how to build a general purpose data processing layer, namely Quill [25], on top of CRA, with only 200 lines of code. We showed the generality of the CRA API, the value of reusing existing components, and the significant impact it has on improving developer productivity.

DBMSs expose SQL for computations, but extending them with custom functionality, such as implementing new shuffle or join strategies, is difficult. A majority of the new generation of big data processing systems, such as Drill [1], Impala [5], Tajo [9], and Presto [17] also have similar limitations. They are custom data processing engines, built from scratch, and perform well for specific workloads. On the other hand, in CRA, a developer does not need to start from scratch, has full control over data processing, and is free to define and execute arbitrary applications, as long as they can be expressed as a distributed, possibly cyclic, sharded dataflow graph.

9 CONCLUSIONS

In this paper, we started with the goal of significantly changing the way big data applications are designed and implemented, with a particular focus on boosting developer productivity. 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. Our experimental results on Quill-on-CRA showed a significant performance advantage of CRA dataflows vs. unoptimized data flows. Overall, we believe our research and system present interesting insights into how data-intensive applications can exploit containerization technologies. Further, we believe these insights are quite valuable for the larger community. Last, we welcome the community to build on CRA, which is now available as open-source software.

REFERENCES

[1] Apache Drill. https://drill.apache.org/, 2018.
[2] Apache Flink. https://flink.apache.org/, 2018.
[3] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2018.
[4] Apache Hive. https://hive.apache.org/, 2018.
[5] Apache Impala. https://impala.apache.org/, 2018.
[6] Apache REEF. http://reef.apache.org/, 2018.
[7] Apache Spark. https://spark.apache.org/, 2018.
[8] Apache Storm. http://storm.apache.org/, 2018.
[9] Apache Tajo. http://tajo.apache.org/, 2018.
[10] Apache Twill. http://twill.apache.org/, 2018.
[11] CoreOS. https://coreos.com/, 2018.
[12] Docker. https://www.docker.com/, 2018.
[13] Docker Swarm. https://docs.docker.com/engine/swarm/, 2018.
[14] Kubernetes. https://kubernetes.io/, 2018.
[15] Kubernetes StatefulSets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, 2018.
[16] MESOSPHERE. https://mesosphere.com/, 2018.
[17] Presto. https://prestodb.io/, 2018.
[18] The Matrix of Hell. https://blog.docker.com/2013/08/paas-present-and-future/, 2018.
[19] VMWare Hypervisor. https://my.vmware.com/en/web/vmware/evalcenter?p=free-esxi6, 2018.
[20] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.
[21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.
[22] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical report, 2014.
[23] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.
[24] B. Chandramouli, J. Claessens, S. Nath, I. Santos, and W. Zhou. Race: Real-time applications over cloud-edge. In *SIGMOD*, 2012.
[25] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar. Quill: Efficient, transferable, and rich analytics at scale. *VLDB Endow.*, 2016.
[26] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *VLDB Endow.*, 2014.
[27] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running: Dive into the Future of Infrastructure.* O'Reilly Media, Inc., 2017.
[28] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[29] K. Matthias and S. P. Kane. *Docker: Up & Running: Shipping Reliable Containers in Production*. O'Reilly Media, Inc., 2015.

[30] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[31] 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.

[32] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *EuroSys*, 2015.

[33] D. Warneke and O. Kao. Nephele: Efficient parallel data processing in the cloud. In *MTAGS*, 2009.