Exploring the Effects of Virtualization in Spark

Andrew Sima
asima@college.harvard.edu
Harvard University

Jaylen Wang
jaylenwang@college.harvard.edu
Harvard University

William Cooper
wcooper@college.harvard.edu
Harvard University

Abstract

With the increase in popularity of cloud computing, and in particular serverless computing, the notion of multi-tenancy has become increasingly prominent. To achieve multi-tenancy along with scalability for their customers, cloud providers turn to virtualization software to contain user applications. A popular application-type that is run on such cloud infrastructures are distributed computation frameworks such as Apache Spark [1], as serverless options provide an easy way to run computations which require multiple machines and data spread across them. As a result, however, application developers and clients running such applications on cloud services must understand the overheads and effects of virtualization on the underlying computational architecture. This project aims to explore such effects through a set of experiments and propose some optimizations that could overcome any observed bottlenecks. The project designs and implements a set of experiments to explore such effects, as well as prototypes to model and test the larger system to gain further insights. Overall, we find that, surprisingly, virtualization through Docker can reduce networking overheads and, through our prototyping, that these networking overheads are significant.

1 Introduction

There has been an increase in popularity of cloud computing, and specifically serverless computing, as an avenue for performing distributed computation (with services like Amazon EC2[2], Amazon Lambda [3] and Microsoft Azure[4]). In order to maximally utilize the machines and resources of the servers, cloud providers must support multi-tenancy, or the ability to host multiple client programs on the same machine. This feature, if implemented correctly, allows clients to access 'virtual' CPUs, which may in reality be a combination of computing power from various different physical locations. To do this, cloud providers often use virtualization and containerization software to provide the desired layers of isolation between different client programs. Another advantage of using such containerization software is that it facilitates scaling; users' containers can easily be booted up or down based on particular needs. This way, a machine with many cores, lots of RAM/disk can be provisioned across client applications, where clients think that they are given an entire machine, even though only a portion of the resources have been allocated to them.

Some of the most popular applications that run on such services are distributed systems. Common such systems are Hadoop [5], Dask [6], and Spark [1]. These systems try to distribute large-scale computation and storage over multiple machines in a way that each machine computes effectively and in a way that is understandable to the user. These systems are natural choices to deploy on cloud services, such as those mentioned, where it's easy to scale up the number of machines needed for the application. Due to the virtualization described, however, these distributed computing frameworks must deal with the effects that virtualization has on their workload. Specifically, many of these workloads are bottle necked not on CPU utilization, but more on I/O tasks like networking, serialization/deserialization, and disk reading/writing [7].

This project aims to analyze the impacts of virtualization and multi-tenancy, particularly how virtualized networking affects existing distributed frameworks and their computation. For this project, Spark was chosen as the distributed framework to analyze, as discussed in §3.1. The project's first question concerns potential performance differences (i.e., gains or losses) from running multiple distributed compute nodes; in this project, Spark workers are tested first on the same machine and then within separate virtualized containers. The project's secondary focus is to determine what, if any, optimizations can be made for Spark (and other similar frameworks) that can reduce overhead weight when Spark workers are made aware that they are in containers located on the same machine. Detailed hypotheses for the below-outlined experiments which answer these questions are discussed in §5.

To answer these questions, we perform experiments where we first run multiple Spark worker computational nodes on the same machine as separate processes. We then introduce containerization, placing each Spark worker in separate containers to observe the effects. Using our experimental setup, we are able to compare the characteristics of running Spark in each environment and make certain conclusions about how Spark handles networking as well as how it interacts with containerization software like Docker. Additionally, due to a lack of time or expertise in Spark source code, we instead create a prototype of the Spark system architecture to test the effects of data serialization and networking, in order to determine the benefits of finding an optimized memory-share between separate processes.

The rest of the paper is organized as follows: Section 2 discusses related work and how the project utilizes the information or findings from each of them. Section 3 provides

the design of the experiments, as well as certain considerations taken into account for each of the design components. Section 4 describes how the experiments are setup and implemented. Section 5 then evaluates and analyzes collected data, and provides a number of other observations. Section 6 includes our speculations on reasons for these findings, and suggests a number of ways to test these speculations. Finally, Section 7 concludes the paper with our important findings as well as future work or experiments that would be useful.

2 Related Work

While this project used Docker to containerize and virtualize Spark nodes, there are other virtualization software used by cloud and serverless providers. Amazon EC2 itself uses Xen [8], a type-1 hypervisor that interfaces directly with bare-metal. Other cloud providers use higher-level hypervisor like KVM [9], which utilize interfaces directly within the Linux kernel to handle virtualization and isolation. One example of a KVM-based recent containerization software is Firecracker [10], used by Amazon Lambda. As a result, Firecracker is specially tailored for serverless computing and function-as-a-service applications, placing emphasis on isolation, portability, and being as lightweight as possible (providing a better option than QEMU). Firecracker, due to its focus on a minimalist design and isolation, does not provide any built-in support for shared-memory. As discussed later, Docker was ultimately chosen due to its use in similar Spark applications through Kubernetes and due to our familiarity with Docker.

Previous work has looked at optimizing data movement between workers in Spark. Davidson and Or first explain differences between Hadoop and Spark and observe how the shuffle stages are significant bottlenecks in many applications, beyond simple MapReduce examples [7]. They then profile applications on Spark and track the bottlenecks during the shuffle stage. They observe that the bottlenecks aren't in the CPU utilization or even networking, but the bottleneck is mainly in I/O wait time reading large amounts of data from disk. They propose using a smart columnar compression scheme that is tailored well to many of the workloads that Spark and Hadoop are used for, as well as have each core, rather than each map task, write to the same output files for each reducer. The work's focus is not on multi-tenancy and virtualization, as it only focuses on ways to optimize shuffle performance between machines. The work is also limited in its analysis, as it only looks at simple MapReduce examples, where other applications might have different characteristics that make their optimizations less effective.

There has also been previous work that has looked into implementing and optimizing for shared memory support between virtual machines. One of the most well-known of these is Virtio [11], which provides shared memory support for Linux containers. Further research has been conducted to look into further optimizing Virtio to reduce networking overheads [7].

One previous work that has looked at applying shared-memory into Spark, work done by Rang et al. They propose a caching system where Spark executors that are running in the same worker node, and hence on the same machine, have a shared cache layer. They utilize that Spark provides support for "off-heap" memory which is not handled by JVM, and thus does not undergo garbage collection. This off-heap space is given on a per-executor basis, so these authors make changes to Spark to allow for a shared off-heap memory region, where they utilize smart caching strategies to allow executors to directly share memory. However, this work does not address a fundamental situation in which multiple Spark workers are placed on the same machine, as the work's shared memory solution only works between executors within a single worker. The work is also not geared towards a study of the effects of virtualization on Spark, as multiple executors within the same Worker cannot operate within separate containers. The work, however, does make changes to Spark's source code (although not open-sourced) and also implements a caching policy that further reduces data movement on top of the gains of using shared-memory. They then perform an evaluation of their work to show significant gains, which is further evidence for our later hypotheses.

3 Design

There are three overarching choices we make when designing these experiments: the machine, the distributed framework, and the containerization method.

3.1 The Machine

Most contemporary operating systems feature aspects of virtualization in some way or another for a variety of reasons, including protecting the main kernel from attacks, separation of processes, and increased performance in some computations. We believe the Linux operating system to be the most straightforward to test virtualization with.

To avoid confounding variables in any extra unknown layers of virtualization, a bare metal Linux machine would suit our project most sufficiently; however, due to a lack of time and resources, we settle with a t2.xlarge Amazon EC2 instance running Ubuntu. We are provided 4 virtual CPUs (vCPUs) and a total of 16 GB of RAM. Using an EC2 instance, though, still allows for relatively fined-grained control of resources, as we can allocate cores and memory to workers using built-in Spark scripts.

3.2 The Distributed Framework

Apache Spark is one of the most widely-used distributed frameworks, known for its computational speed and especially for its utility and relative flexibility. It is able to achieve

its efficiency by heavily utilizing machine memory, storing its computation's intermediate data in RDDs (resilient distributed datasets). Typically, one would operate on a large dataset with Spark by using an API to communicate with a Spark cluster; however, the framework does provide a "Spark Standalone" mode, in which clients may download Spark source code locally and execute applications on non-commercially owned machines. This feature, along with its support for Python, makes Spark a good design choice for our experiment.

Dask was also another framework that was considered; however, because our application involved exploring networking overheads, Dask was found not to be a good choice primarily for the follow reason: generally, Dask is utilized not for big data applications, where networking overheads are the most prominent, but more for splitting up operating on smaller matrices over different machines and allowing for more parallelization with common Python numerical computations.

3.3 The Containerization Method

Initially, we considered three main containerization frameworks, including QEMU, Amazon Firecracker, and Docker.

QEMU is an older framework, with support for shared memory between containers. Unfortunately, it has been known to be buggy when integrated with larger cloud computing platforms. QEMU is also quite bulky and places a significant burden in terms of memory and disk space. The reason for this is that QEMU is extremely built-out in terms of support for various functionality like device drivers as well as shared memory support; however, most cloud providers don't need or want all these capabilities, which explains why recently it has not been the software of choice for cloud providers.

Amazon Firecracker is the main containerization software used by Amazon Web Services, and is now the backbone of services such as AWS Lambda. Though its performance has been proven on cloud services, we are deterred by its Rust source code and limited community-wide support. Firecracker does not provide shared-memory support, as they adopt a micro-VM methodology where only the necessary components are kept. This means that in order to use it for our initial purpose of implementing shared memory for Spark within containers, this would require altering Firecracker code as well, which was not feasible for the timescale of the project.

Finally, we arrive at Docker, a lightweight containerization framework which has been adopted by many systems communities and cloud providers as a primary method for virtualization. A Docker container runs on a Docker engine, which sits directly on the host OS, as can be seen in Fig. 3.3. Docker images provide a read-only snapshot of dependencies and configuration files to dictate container behavior. Although we did not end up implementing shared memory for lack of time, the Docker platform provides support for memory sharing across containers.

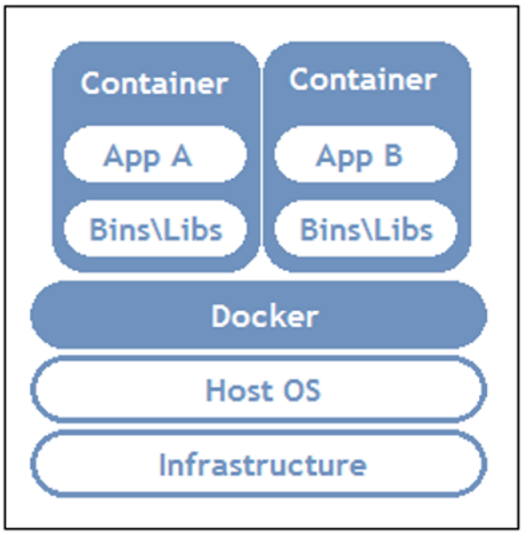

Figure 1. A Docker daemon runs in the background on the host OS. Containers manage their dependencies with the "Bins/Libs" module above, or images. On a bare metal instance, our containerization framework would sit closer to the hardware infrastructure.

3.4 Further Design Considerations and Prototyping

We made a number of other finer-grained design decisions throughout the research process. We use Spark's most recent stable version, 3.2.1. Spark 3.2.1's standalone module comes with its own resource manager and a configuration file. It allows us to configure and spin up a custom architecture on our EC2 instance. When spinning up the master, we take note of its IP address, which we assign to workers. We are able to manually assign a number of cores and an amount of memory for each worker. To further isolate workers for the containerization part of our experiment, we draw from open source code on Github (https://github.com/mvillarrealb/docker-spark-cluster) and use docker-compose to build containers to host workers. This repository is specifically built to accommodate Spark standalone, and is used to test smaller scale Spark applications on one or a few machines.

To perform any type of experiment, benchmarking and tracking tools are necessary. In testing Spark's performance with and without Docker, we initially looked to open-sourced tools that are specifically made to monitor Spark applications such as HiBench, SparkLint, and Dr. Elephant – all relatively popular benchmarking tools with sufficient documentation. However, these tools are primarily used to monitor Spark applications running on a remote cluster and don't provide the fine-grained details we wanted for a Standalone setup.

Due to time constraints, we felt were unable to delve deeper into each of these open source options. We considered using using the default Amazon Cloudwatch application, which provides statistics through a web UI for EC2 instance users. While it provides measurements of networking and CPU utilization, it is too coarse grained, and sometimes somewhat inaccurate.

Upon careful consideration, we decided instead to use our own application and data, as well as a default network and CPU tracker, sysstat, to monitor our deployment runs. sysstat includes a host of sub-tools, each responsible for different aspects of a Linux system. For tracking, we primarily make use of iostat, which reports CPU statistics and block device input and output statistics. We use sar, a sysstat cron scheduling tool which collects the output of iostat and stores it in a binary which can then be converted through the Linux sadf command to a user-friendly file form. From this point, we use the Linux SCP (secure copy) command to move data back and forth between local machines and our remote instance. Finally, we use Microsoft Excel to visualize networking and CPU information outputs.

4 Implementation

As described, this project entails two primary experiments: one where we experiment with Spark directly by comparing Spark without virtualization to Spark where workers are configured in separate Docker containers. The second experiment prototypes a Spark-inspired architecture where data is passed between workers, allowing us to more directly compare different alternatives to networking. This section describes how both are implemented.

4.1 Spark Experiments

In the first experiment, we configure Spark's worker nodes to work (1) as separate processes all running on the same EC2 instance, and then (2) within separate containers within the same instance. To set this up, we installed Spark on the instance and used scripts that allow users to set up a Standalone cluster, where a master and workers can be deployed on the same machine as separate processes. Spark by default gives users scripts to set this up and with different configuration settings that allow for a configured number of worker nodes, cores allocated, etc. For all our experiments we run with exactly on master, two worker nodes, and each worker node contains one executor that is allocated as many cores as Spark can receive.

We provide a bare comparison between containerized workers and non-containerized workers running two relatively representative algorithms, namely Pagerank and kmeans clustering, as both of these applications are common distributed algorithms that require communication between workers. Random data files are generated for these applications, with 100,000 and 5,000 data entries, respectively. We

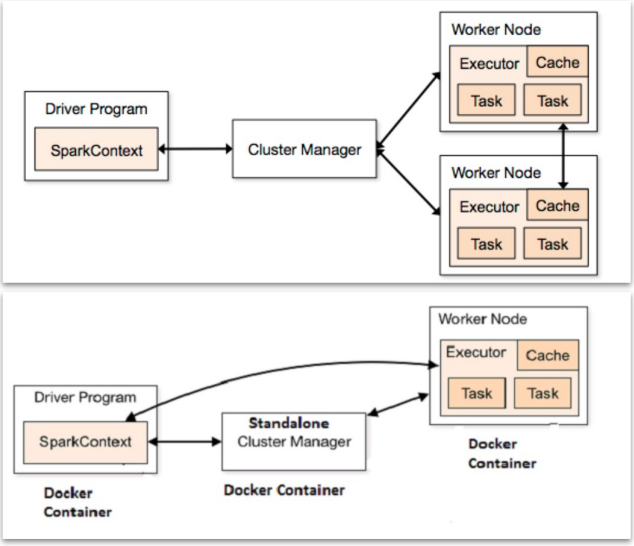

Figure 2. A basic Spark application control flow diagram. In Spark Standalone mode, all elements are run from the same physical machine. When running our containerized experiment, the worker nodes are located in separate docker containers.

spin up a preconfigured master node, along with two worker nodes (each allocated one core and one GB of memory). We create a Python script to run a particular computation, first on plain Spark Standalone, and then within Docker, while running sysstat in the background. The script provides a fair comparison as it takes out any of the time it takes for Docker to build the imagers and to start up the containers, both the non-Docker and Docker runs are timed from when the command is executed on the master node.

4.2 Spark Prototypes

We have formulated our second experiment to isolate the actual performance impacts of various factors including the use of a Docker container and repeated serialization and deserialization of data. We create a Spark prototype in Python, which simulates a computation via two methods: (1) by creating different processes which write and read intermediate data from shared data files located on the machine, and (2), by creating different processes which share data by networking, despite being on the same machine. The prototypes were tested using datasets of varying size, ranging from a few KBs to several hundred MBs. In all experiments, datasets used a `csv` format; data was read from the test file and sent to receiving processes without additional processing. All test datasets are courtesy of Kaggle. Process management within our prototype is handled using the `multiprocessing` library. For our prototypes without Docker usage, we used the `tempfile` library to model shared memory between processes, and the `socket` library to set up two TCP sockets

```
import multiprocessing as mpc

def send(**kwargs):
    ...perform send operation

def recv(**kwargs):
    ...perform recv operation

def main():

    ...perform setup

    # setup processes
    sendproc = mpc.Process(
        target=send, args=(...))
    recvproc = mpc.Process(
        target=recv, args=(...))

    # run processes
    sendproc.start()
    recvproc.start()
    sendproc.join()
    recvproc.join()

    ... perform cleanup
```

Figure 3. An illustration of our process management design. The writing task (sending packets over the network or writing to shared memory) is implemented by `send`, while the reading task (receiving packets over the network or reading from shared memory) is implemented by `recv`. Each of these tasks are then delegated to a process using the `mpc.Process` object so we can take advantage of simultaneous execution on multiple CPUs.

as a simple model of data exchange via networking. Following this, prototyping with Docker heavily depended on the Python SDK `docker` library. In this design, we continued with our approach using two processes, each now responsible for executing commands within a per-process container built from the Docker `python` base image.

In the Docker shared memory prototype, we used a volume mounted on both per-process containers to model shared memory, with each process reading and writing to a shared file located in this volume. In the Docker networking prototype, we first created a default Docker bridge network, and proceeded to attach each of our containers to network so that inter-containers discovery can occur. Then within each of these containers, TCP sockets were bound to respective ports on the host, and then used to send data to the other container.

```
import docker

# init client
client = docker.from_env()
api = client.api

def setup_docker_shm(**kwargs):
    ...pull Docker images

    # setup volume
    local = api.create_volume(volume_name)

    # setup containers with volume bind
    bind = f"{local['Mountpoint']}:{remote_path}"
    host_config = api.create_host_config(
        binds=[bind])
    container_id = api.create_container(
        image_name,
        host_config=host_config
        **options)

def setup_docker_net(**kwargs):
    ...pull Docker images

    # configure Docker network
    api.create_network(network_name)
    edpt_conf = api.create_endpoint_config()
    nw_conf = api.create_networking_config({
        network_name: edpt_conf
    })

    # create containers
    api.create_container(
        image_name,
        networking_config=nw_conf,
        **options)
```

Figure 4. Implementations for shared memory and networking in Docker containers. In the shared memory design, a volume is created and bound to each subsequent container created. In the networking design, a similar approach is taken: the network is created first, then bound to each container at the time of creation.

To investigate the effects of serialization and deserialization using this design, the `pickle` library was used to serialize data prior to packets being sent over the network. Note that serialization was not applied to the shared memory designs because many of Spark's performance improvements came from storing deserialized Java objects in-memory as opposed to incurring serialization overhead [1].

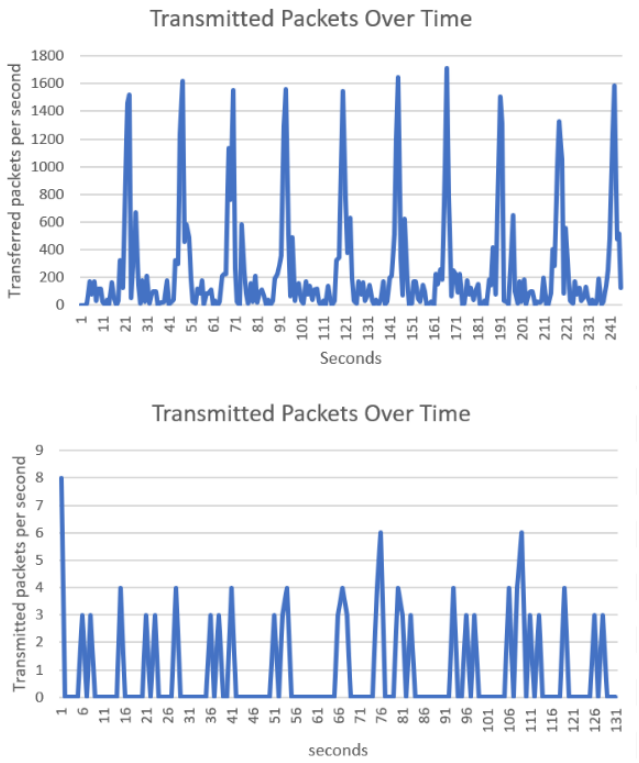

Figure 5. Packets transmitted over the network over time, for 20 iterations of a Spark kmeans clustering application. The top graph displays the kmeans computation without Docker, the bottom graph displays the computation without Docker. The spikes generally represent where a new computation begins.

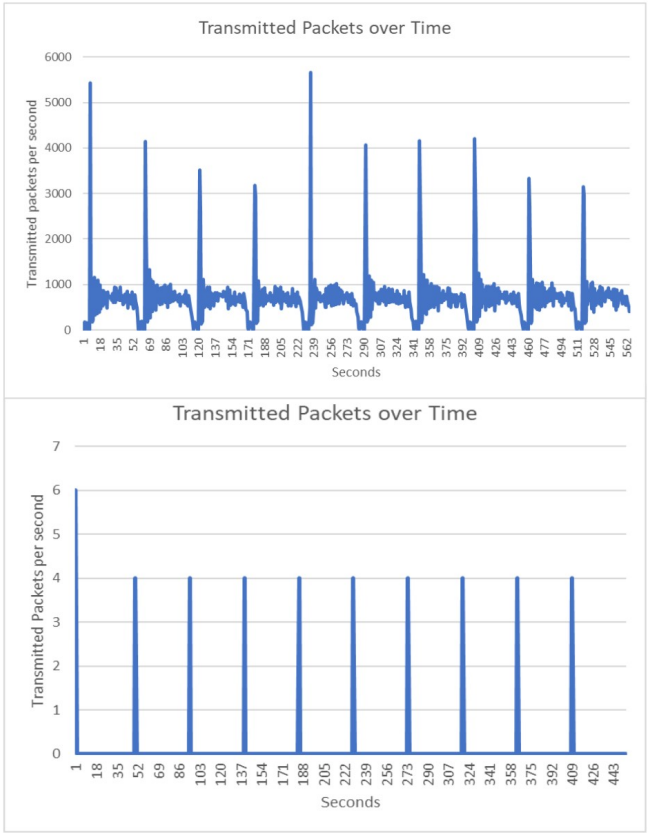

Figure 6. Packets transmitted over the network over time, for 20 iterations of a Spark pagerank application. The top graph displays the pagerank computation without Docker, the bottom graph displays the computation without Docker. The spikes generally represent where a new computation begins.

5 Evaluation

5.1 Hypothesis

Spark standalone workers do not make use of shared memory, and in our initial experiment, neither do the workers in isolated containers. Given that the inherent design of virtualization requires an additional layer of system call handling to handle virtual networking and dealing with the fact that Spark thinks the workers are running on separate machines, we expected all procedures using Docker to have noticeable performance lags compared to versions without Docker.

Moreover, we expect that the use of shared memory can provide a way to get around the additional networking overheads (like serializing and deserializing the JVM objects that Spark stores, as well as the overheads incurred through sending data over the network rather than accessing the same memory).

5.2 Experiment 1: Spark Virtualization

We have run two applications with our software setup, including Pagerank and kmeans clustering. We experienced surprising results for both: Spark applications that had workers isolated in Docker containers consistently ran faster than Spark applications with non-isolated workers. Details are displayed in Table 1 above. For kmeans clustering, isolated workers running in Docker containers are able to complete the task almost twice as fast as non-isolated workers. For the Pagerank algorithm, isolated workers take about 80% as much time as non-isolated workers.

Figure 7 displays CPU utilization of a single Pagerank application run on Spark, once with isolated workers and once without isolated workers. We see that the four vCPUs on our instance are almost all fully utilized at many points during the computations, though generally the Docker computation uses less power. The system's background processes use up slightly more CPU power for the Docker computation, presumably because Docker's daemon must run during deployment. Although we don't explicitly display kmeans computations' equivalent graphs, they are fairly similar in magnitude and oscillation periods.

	kmeans (no Docker)	kmeans (with Docker)	pagerank (no Docker)	pagerank (with Docker)
average runtime (s)	24.1752	13.0500	59.5073	47.3747
runtime variance (s)	0.5782	0.4476	0.0235	0.1068

Table 1. The average run time and variance of run times of 10 deployments each of pagerank with Docker, pagerank with no Docker, kmeans clustering with Docker, and kmeans clustering without Docker.

From our understanding of Spark's standalone module and the Docker containerization method, we expected that the isolated workers would communicate with one another by sending packets over the network. We expected a relatively stable number of packets to be sent and received by the workers for both Docker and non-Docker implementations, but we get a very surprising result: when workers are isolated in Docker containers, they send 0 packets over the network! A small number of packets (between 4 and 6) are transmitted at the beginning of each Pagerank iteration, but beyond this the workers are able to communicate without utilizing the network. Similarly, for the kmeans applications, fewer than a total of 20 packets are sent on the network per iteration. On the other hand, when workers are not in separate Docker containers, they send far more packets over the network. For Pagerank, a few thousand packets are sent at the beginning of each iteration, presumably between the master node and worker nodes. For the rest of the duration of each run, close to 1000 packets are continuously being sent every second over the network, and this value only drops off only towards the end of a computation. The kmeans graphs without Docker containers have the same structure, but with a consistent value of around 200 packets per second for most of the execution before dropping off, reflecting the smaller amount of data fed into this particular algorithm. We suspect that this disparity in packets transmitted per second between the Docker and non-Docker variations, coupled with the fact that data sent over a network between workers may need to be serialized and deserialized before use, could explain why Spark workers in Docker containers are actually more efficient than Spark workers sitting directly on the machine.

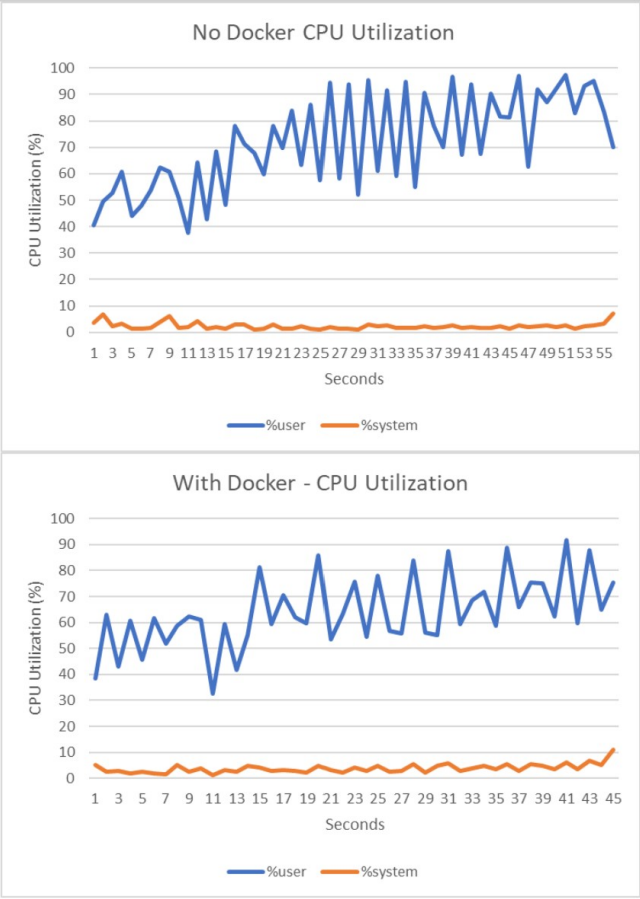

Figure 7. A combination of all four vCPUs utilization over the course of one Spark pagerank application, with and without Docker. The system typically does not use much power, and over time slightly more power is used by the non-Docker computation.

5.3 Experiment 2: Spark Prototyping

Prior to experimentation with serialization, we already see astounding performance differences between designs using Docker and those without. Examining Table 2 reporting on a data transfer of a few MBs, we see that the shared memory prototype has an approximate 8.5x slowdown and the networking prototype has an approximate 4.2x slowdown with Docker usage. These performance results were predictable, as we expected our experiments to demonstrate the cost of system call handling overhead for containers.

Furthermore, the results confirm our hypothesis that the use of shared memory provides significant performance enhancements. Simply examining the use of shared memory with Docker (`dockershm`) shows that the shared memory design outperforms the networking prototype even without the use Docker (`net`), by approximately 30%. Performing a comparison between the use of shared memory versus networking shows even more drastic performance differences; without Docker, a performance difference of ~12x is observed, whereas with Docker, a difference of ~5.9x is observed. Similarly these results were expected due to the predictable overhead from delivering packets over the network.

	shm	dockershm	net	dockernet	net (w/serialization)	dockernet (w/serialization)
average runtime (s)	0.0429	0.3662	0.5163	2.1603	0.5362	2.1961
runtime variance (s)	1.2220e-05	0.0037	0.0002	0.0159	0.0009	0.0164

Table 2. The average run time and variance of run times of 10 trials using a 6.5MB-sized dataset on each of the six prototypes. Note that `shm` stands for shared memory, whereas `net` stands for networking. Prototype names prepended with docker indicate Docker usage.

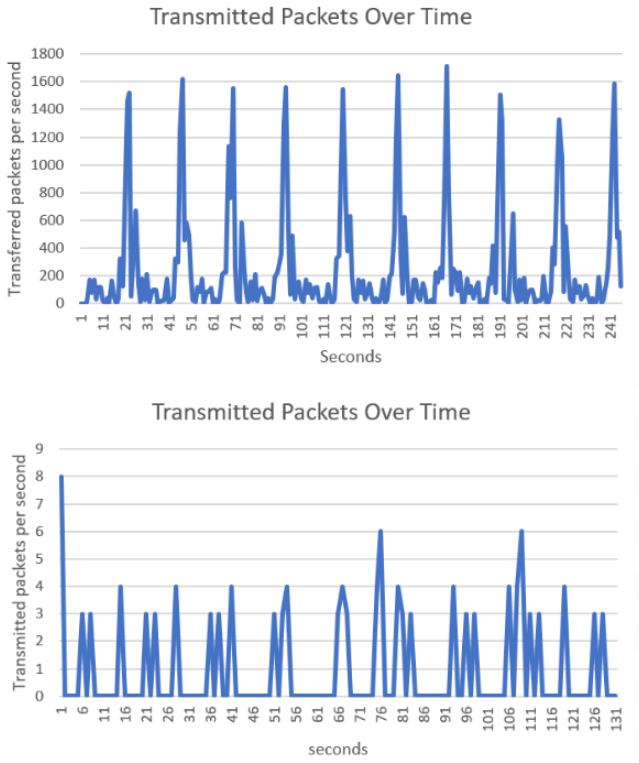

Figure 8. Packets transmitted over the network over time, for 20 iterations of a Spark kmeans clustering application. The top graph displays the kmeans computation without Docker, the bottom graph displays the computation without Docker. The spikes generally represent where a new computation begins.

Now turning to the investigation of serialization, we see that the prototypes employing serialization are only slightly slower than those without. Comparing the networking prototype without serialization (`net`) to that with serialization (`netserialize`), we only see a 0.02s average slowdown; similarly the docker networking prototype without serialization (`dockernet`) to that with serialization (`dockernetserialize`) only shows a 0.0358s average slowdown. Originally, we expected that the serialization performance overhead would be larger than currently measured, in accordance with the 10x overhead reported by Spark [1]. However, considering that this is overhead identified from a single point of transfer, and that the amount of data being transferred is tiny in comparison to that used in a MapReduce computation, we are more confident that the performance impact of serialization in practice is indeed significant. We also realized that the current serialization scheme using `pickle` is likely too simple to accurately model the Spark serialization strategy for various Java objects; in practice, the overhead from serializing Java objects is likely more than what we account for.

6 Discussion and Future Work

Through two distinct computations on sizeable data files, Pagerank and Kmeans clustering, we have seen that Spark workers isolated in Docker containers are able to complete computations faster than Spark workers sitting directly on the machine. We speculate that a primary reason for this is that Docker containers are actually sharing memory, because workers in Docker containers do not utilize the network to communicate with the master node and other workers. Due to time constraints, we were unable to further test this hypothesis, although research into Docker's memory-sharing functions would be a good direction to take this project.

Furthermore, in Table 1, we notice a large difference in Docker's speed-up between Pagerank and kmeans clustering applications – Pagerank is sped up by 20.4%, whereas kmeans is sped up by 46.0%. We come up with two hypotheses to explain the disparity in speed-up. Firstly, it is possible that an intermediate shuffling step, in which worker machines must transmit data to one another, is required more by the kmeans algorithm than the Pagerank algorithm. Proportionally, this would require more network traffic for a Spark kmeans algorithm running without Docker, bringing its total runtime far higher than the Docker alternative. Secondly, we note that our kmeans clustering application uses a datafile that is 17.5% the size of the Pagerank dataset (5,000 data entries but each is 7 integers, compared to Pagerank's 100,000 data entries of 2 integers each). We speculate that larger amounts of data in the Docker containers could lead to extra computation time, and that the added layer of virtualization makes more of a difference in this case.

To test the first hypothesis, we propose an experiment in which we find two applications with a large contrast in the number of required Spark shuffles. By conducting the experiment in the same manner as we did with Pagerank and kmeans, we could more closely evaluate the effects of

shuffling steps for workers in Docker containers. To test the second hypothesis, we propose an extension of the above experiments, in which the only modification would be that datasets for the Pagerank and kmeans applications would be far closer in size. This would allow us to evaluate whether or not data set sizes become a major drawback to placing Spark workers in Docker containers.

We notice a couple of other trends in our data. Firstly, it is evident for the Pagerank application deployment runs that a number of packets are transmitted at the beginning of each iteration, whereas there are more unexpected intermittent spikes in packets transmitted during each kmeans iteration. This indicates that changes in algorithm structure can fundamentally alter when and how often Docker containers need to use a network to communicate data. The kmeans algorithm features some sort of function which increases network transmission for Docker; if we were to identify this function we may be able to find an application that runs better without Docker than with.

Secondly, we notice that CPU utilization (Figure 5) is generally higher for the system when not operating in Docker containers, although for applications that do run in Docker containers, the system takes up a slightly higher percentage of CPU capacity. This suggests some sort of tradeoff between increasing background process presence and overall CPU usage, and we wonder how much this can be taken advantage of in the future. For example, if Docker was a heavier-weight containerization framework, and its background processes interfered even more with the behavior of container contents, could overall CPU usage be further reduced? More experiments can be formulated to test these questions.

Another continuation of our current experiment would entail running a Docker container network monitoring tool inside each worker container. This would give us further insight into how Docker could be intercepting network calls.

Other areas of work include expanding our prototype to more accurately simulate the network traffic and shared memory operations seen in Spark. The current design simply passes data between two processes, and makes assumptions concerning shared memory (simplifying to a shared file), serialization (simply using the `pickle` library), and networking (using a fairly straightforward networking bridge in docker). In contrast, the production Spark pipeline most certainly has much more complicated data exchange between its workers. The most severe of these factors would likely be serialization costs, as the Java objects used by Spark follow specific serialization patterns not captured by the default serialization provided by the `pickle` library. As a simple example, the Spark paper specifically states serializing closures when sharing data between machines, whereas other Java objects are stored deserialized in memory [1]. A more careful analysis of precisely which objects are serialized by Spark and how each of these serializations can be modeled in our design would lead to precision improvements in our prototype.

To improve upon various shared memory assumptions, we can instead write programs to use memory mapped IO instead of modeling shared memory using a file. This would remove overhead such as file opening and reading which invoke the kernel. Continuing on the idea of more advanced network monitoring, we can use this data to more accurately model which machines are actually receiving network packets, which would allow our prototype to better model the transmission and receive handling of Spark network packets.

7 Conclusion

Our project provides an exploration and study of the effects of virtualization on the communication and networking between workers within Spark. We find, surprisingly, that running Spark workers on the same machine in separate Docker containers provides a speedup as compared to running them as normal, separate Spark worker processes. We also find, through some networking monitoring, that this is likely caused by Docker preventing container communication from happening over the network, providing speedups. We then validate our hypothesis that shared memory could be used to speed up communication between workers on the same machine through a Spark-inspired prototype. We did this by using a Spark-inspire prototype where we ran controlled experiments that showed the overheads of using network vs. shared memory and how this interaction holds when processes are in separate containers. Overall, we verified that networking, and not just the distribution of work, is a significant design point for deploying distributed systems; and this networking can be made more complex through virtualization that is used in many cloud services.

Acknowledgments

To Eddie, for helping to come up with a cool project plan and providing the support for a great class in which we learned a lot. Also, to the rest of the class for the great feedback during the presentation and all the great discussion that followed.

References

[1] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A {Fault-Tolerant} Abstraction for {In-Memory} Cluster Computing," 2012, pp. 15–28. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[2] "Secure and resizable cloud compute – Amazon EC2 – Amazon Web Services." [Online]. Available: https://aws.amazon.com/ec2/

[3] "Serverless Computing - AWS Lambda - Amazon Web Services." [Online]. Available: https://aws.amazon.com/lambda/

[4] "Cloud Computing Services | Microsoft Azure." [Online]. Available: https://azure.microsoft.com/en-us/

[5] "Apache Hadoop." [Online]. Available: https://hadoop.apache.org/

[6] M. Rocklin, "Dask: Parallel Computation with Blocked algorithms and Task Scheduling."

[7] A. Davidson, "Optimizing Shuffle Performance in Spark," 2013. [Online]. Available: https://www.semanticscholar.org/

paper/Optimizing-Shuffle-Performance-in-Spark-Davidson/d746505bad055c357fa50d394d15eb380a3f1ad3

[8] 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," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.org/10.1145/1165389.945462

[9] A. Qumranet, Y. Qumranet, D. Qumranet, U. Qumranet, and A. Liguori, "KVM: The Linux virtual machine monitor," *Proceedings Linux Symposium*, vol. 15, Jan. 2007.

[10] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[11] M. J. P. January 29 and 2010, "Virtio: An I/O virtualization framework for Linux." [Online]. Available: https://developer.ibm.com/articles/l-virtio/