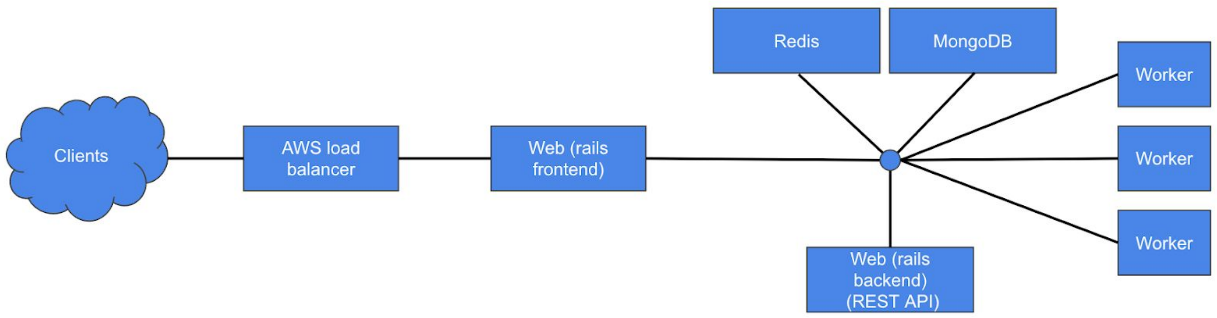
Introduction

This project is focused on evaluating OpenStudio-server and how ideas/concepts learned from **CSCI7000 - Datacenter Networking/Cloud Computing** could stand to benefit by incorporating some of these ideas/concepts.

OpenStudio is a cross-platform (Windows, Mac, and Linux) collection of software tools to support whole building energy modeling using EnergyPlus and advanced daylight analysis using Radiance. OpenStudio is an open-source project to facilitate community development, extension, and private sector adoption. OpenStudio includes graphical interfaces along with a Software Development Kit (SDK).

OpenStudio-server is an energy modeling framework that uses OpenStudio and EnergyPlus to run whole building energy modeling simulations in the cloud. The current architecture allows a user to deploy a containerized cluster by specifying a limited portion of the compute resource environment, such as the size of VM nodes, CPUs and memory. Currently, there are several known issues/challenges associated with the current OpenStudio-server architecture, which will be discussed in this paper. This project is particularly focused on addresses these issues/challenges by optimizing and incorporating concepts learned from this course.

The current architecture (illustrated below) is containerized and deployed using Docker Swarm. The various components including a load balancer, ruby-on-rails front-end/back-end web servers, Redis key-value store, MongoDB, and worker nodes that run the energy simulations, all run in there own isolated containers, or also commonly referred to as “pods.” Currently, there are some known issues and limitations with the current architecture which are discussed below.



Challenges/Issues

1. OpenStudio-server uses a Ruby-based CLI to provision the cluster resources in the cloud and Docker Swarm as the container orchestration manager. This deployment constrains users to use

AWS as the sole cloud provider. In addition, since code configures VM instances, VPC subnets, disk sizes, firewall rules, load balancers, proxies, etc., it's difficult to maintain and can lead to errors during cluster setup and tear down.

2. OpenStudio-server can not dynamically scale based on load. Although a user can choose n-number of nodes to run simulations, once created, these remain fixed and workers are automatically configured to run across all available nodes in the Docker Swarm cluster. While this makes it possible to run a large number of simulations, it makes it difficult for users to balance costs and performance as these are fixed in size. When simulations are not running (between different model setups for instance), the cluster will sit idle and waste resources and money.

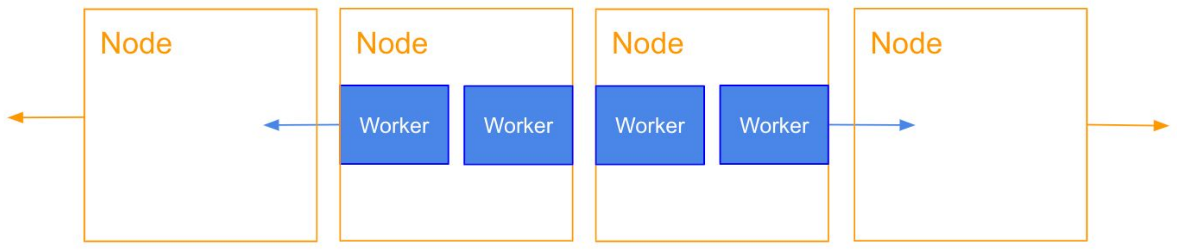
3. OpenStudio-server has a limitation in the number of simulations and/or types of simulations that can be run simultaneously. For instance, simulations that have high temporal resolution time steps can produce very large amounts of data. These data sets are outputted to an ephemeral disk for each worker node and the final result sets are sent over the network to a rails back-end REST server to be processed. If too many workers are sending data, the API server can become bottlenecked due to network and resource constraints on the API server pod. This behavior requires users to either scale back the temporal resolutions and the number of simulations to run.

Potential Solutions

1. Upgrade the current dockerized deployment from Docker Swarm to Kubernetes orchestration framework. Since the application is currently using the Ruby aws-sdk-gem, I used a Ruby Kubernetes gem client to handle the provisioning, deployment, and tear-down of the cluster. This was also converted to Helm chart, which is a Kubernetes package manager and is now becoming a popular choice for micro-service type applications to distribute deployments. Using Kubernetes along with Helm provides a more agnostic approach and now users can deploy Opentudio-server to cloud providers that offer Kubernetes services. Some of the notable ones include AWS, Microsoft Azure and Google Compute.

2. Kubernetes offers built-in auto-scaling features that can be triggered by metrics such as CPU utilization. Applications that are already configured to scale horizontally can leverage this feature by configuring a "horizontal pod auto-scaler (HPA)" which will increase the number of pods when CPU meets or exceeds a user-defined threshold. Enabling this feature in OpenStudio-server worker pods now can scale to demand. This reduces costs especially during

idle periods of time (e.g. between different energy model simulation results) as the cluster can scale down to minimal size. (See the illustration below).



3. Several enhancements could be made to relieve compute and network pressure on the rails REST API server under intensive application loads with large data sets. Since this application is committed to using a containerized microservice design, several options learned from class were explored to try and optimize the container overlay network. In particular, two experimental open-source tools, Microsoft “Free Flow” and “Slim” were evaluated to see if adopting either of these tools could make improvements to the network performance and ultimately reduce/eliminate the bottleneck.

   Slim and Freeflow accomplish similar goals by by-passing the container overlay networking stack. They both accomplish this by providing a custom libsocket.so lib that is loaded at run time and intercepts application TCP traffic and routes them to an associated router which then sends the traffic out via the host network. This approach differs from the traditional approach where the packet flows through the container network virtual NIC, virtual switch, and then through the host network. And since this is done on receiving side as well, it benefits both the sender and receiver and eliminates two network stack traversals for every packet. Freeflow was chosen as the test candidate for this study since it also supports RDMA.

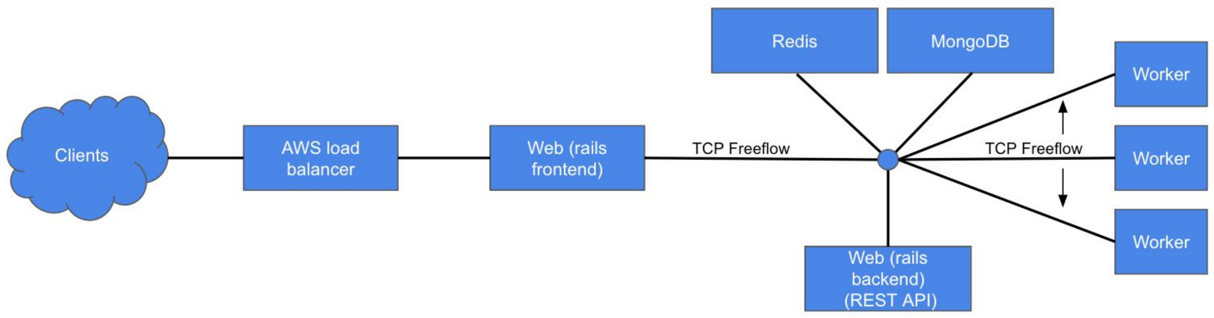
   Slim and Freeflow were both research papers discussed in class and were part of the discussion around container overlay networks.
   https://www.usenix.org/conference/nsdi19/presentation/zhuo
   https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/freeflow-2.pdf

   Both Slim and Freeflow source code is open-sourced on GitHub.
   https://github.com/danyangz/Slim
   https://github.com/microsoft/Freeflow

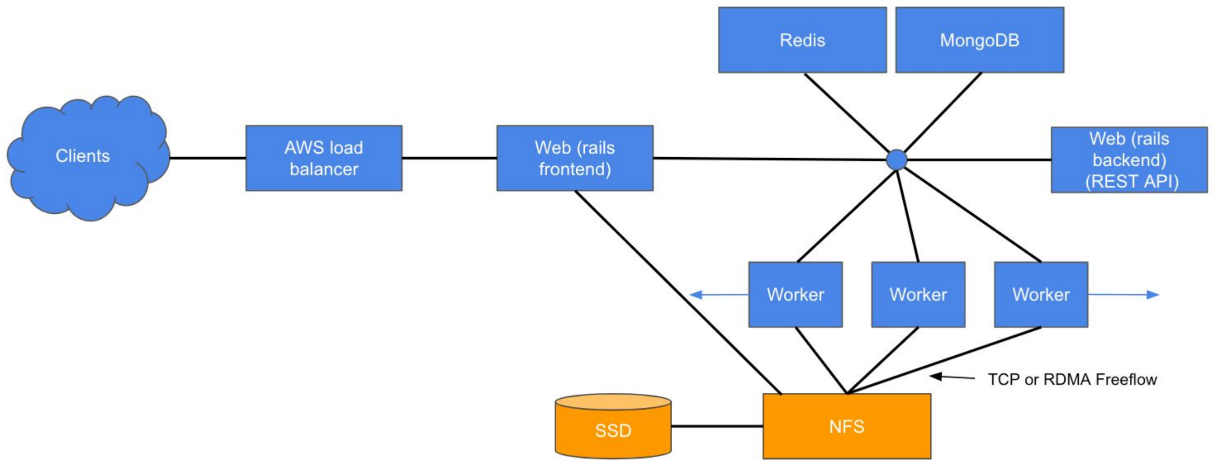
System Design (addressing issue #3)

The two potential solutions discussed below address issue #3 specifically.

1) Use Freeflow with TCP mode with worker pods and rails REST API pod to optimize the flow of traffic. This could relieve some of the burden by eliminating the need to process packets in the container network overlay stack. See the illustration below.



2) The second option would use Freeflow with TCP mode, or even better, using RDMA mode, combined with an NFS server as shared network storage for the worker and rails web front-end pod to share data. The workers would mount the NFS shared storage using TCP or RDMA depending on the mode, and all traffic would either flow using the optimized Freeflow TCP path or use RDMA depending on the configuration chosen.



The second option (NFS coupled with fast SSD) might be the best option of the two as you can remove the entire REST API data response traffic altogether. The front-end rails server could index and access the data results directly from NFS and serve these to the client. Clearly, this is dependent on a fast enough network shared storage coupled with performance SSD disks, but this design shows promise.

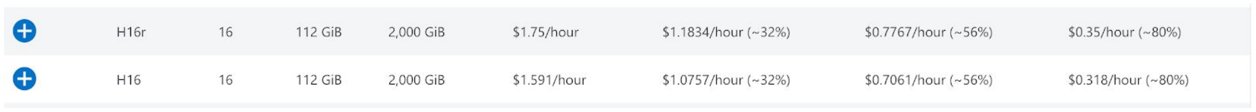
Evaluation/Findings

The first part of the evaluation is to determine if Freeflow can be implemented using Kubernetes as this is the chosen path for OpenStudio-server moving forward. Secondly, can Freeflow achieve the network speeds needed to support the application using shared NFS storage?

The first test was to deploy a small cluster setup with basic Kubernetes pods to understand and evaluate the network performance with Freeflow enabled and disabled and compare the results. Freeflow can work by optimizing TCP to achieve bare host speeds, and it can also use RDMA if you have RDMA enabled NICs (e.g. Mellanox 3x series). This test was to test latency and throughput using TCP mode only.

To conduct this study, Microsoft Azure was chosen as the testbed as it offers an RDMA enabled NIC VM instance types, which will be the next step after benchmarking TCP. The H-series High-Performance Compute series was the chosen VM instance type (below):

RDMA Enabled NICs: ResourceType: ComputeCores SKU: HC Series (H16r,) x2
Non-RDMA Enabled NICs: ResourceType: ComputeCores SKU: HC Series (H16,) x2



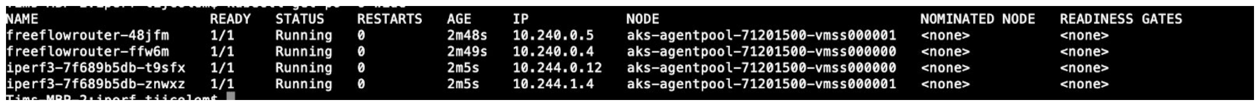
⊕	H16r	16	112 GiB	2,000 GiB	$1.75/hour	$1.1834/hour (~32%)	$0.7767/hour (~56%)	$0.35/hour (~80%)
⊕	H16	16	112 GiB	2,000 GiB	$1.591/hour	$1.0757/hour (~32%)	$0.7061/hour (~56%)	$0.318/hour (~80%)

https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/#h-series

The first test will be to confirm that Freeflow can be ran using TCP mode and achieve speeds equivalent to that of the VM host network speeds. First, several steps were required to provision the Kueberntes cluster:

- Create a Microsoft Azure Kubernetes cluster with 2x H-series H16 VMs.
- Deploy a compatible Container Network Interface(CNI) that works with Freeflow and Kubernetes. Weave was chosen in this case.
- Configure the Host Subnets and VNET subnets for Container Network Interface(CNI) and Freeflow.
- Create and deploy a Freeflow Kubernetes Router as a Daemon set (runs on each VN node by default).
- Finally, create and launch a container that links to shared lib freeflow/libsocket.so which intercepts TCP traffic and routes it to the freeflow router.

Example of a running Kubernetes cluster with router and iperf3 container:



```
NAME                    READY   STATUS    RESTARTS   AGE     IP            NODE                                  NOMINATED NODE   READINESS GATES
freeflowrouter-48jfm    1/1     Running   0          2m48s   10.240.0.5    aks-agentpool-71201500-vmss000001    <none>           <none>
freeflowrouter-ffw6m    1/1     Running   0          2m49s   10.240.0.4    aks-agentpool-71201500-vmss000000    <none>           <none>
iperf3-7f689b5db-t9sfx  1/1     Running   0          2m5s    10.244.0.12   aks-agentpool-71201500-vmss000000    <none>           <none>
iperf3-7f689b5db-znwxz  1/1     Running   0          2m5s    10.244.1.4    aks-agentpool-71201500-vmss000001    <none>           <none>
```

Using ping and iperf3 tools, I measured latency and throughput, respectively, for both TCP optimized and non-optimized by loading / unloading the freeflow/libsocket.so by unsetting ENV LD_PRELOAD variable.

Example outputs below from running iperf3 and ping (note that iperf3 pods are running on different vms nodes).

```
TCP FreeFlow Optimized
root@iperf3-7f689b5db-znwxz:/# iperf3 -b 0 -P 50 -c 10.240.0.5
...
[ 92]  0.00-10.00  sec  359 MBytes  301 Mbits/sec  2119        sender
[ 92]  0.00-10.00  sec  357 MBytes  300 Mbits/sec             receiver
[ 94]  0.00-10.00  sec  255 MBytes  214 Mbits/sec  1965        sender
[ 94]  0.00-10.00  sec  254 MBytes  213 Mbits/sec             receiver
[ 96]  0.00-10.00  sec  229 MBytes  192 Mbits/sec  1560        sender
[ 96]  0.00-10.00  sec  228 MBytes  191 Mbits/sec             receiver
[ 98]  0.00-10.00  sec  286 MBytes  240 Mbits/sec  2183        sender
[ 98]  0.00-10.00  sec  285 MBytes  239 Mbits/sec             receiver
[100]  0.00-10.00  sec  265 MBytes  222 Mbits/sec  1732        sender
[100]  0.00-10.00  sec  264 MBytes  222 Mbits/sec             receiver
[102]  0.00-10.00  sec  277 MBytes  232 Mbits/sec  1469        sender
[102]  0.00-10.00  sec  275 MBytes  231 Mbits/sec             receiver
[SUM]  0.00-10.00  sec  13.9 GBytes  11.9 Gbits/sec  99421      sender
[SUM]  0.00-10.00  sec  13.8 GBytes  11.9 Gbits/sec             receiver
```

```
TCP FreeFlow Non-Optimized
root@iperf3-7f689b5db-znwxz:/# iperf3 -b 0 -P 50 -c 10.240.0.5
...
[ 92]  0.00-10.00  sec  289 MBytes  242 Mbits/sec  1815        sender
[ 92]  0.00-10.00  sec  288 MBytes  242 Mbits/sec             receiver
[ 94]  0.00-10.00  sec  293 MBytes  246 Mbits/sec  1789        sender
[ 94]  0.00-10.00  sec  292 MBytes  245 Mbits/sec             receiver
[ 96]  0.00-10.00  sec  343 MBytes  288 Mbits/sec  2091        sender
[ 96]  0.00-10.00  sec  341 MBytes  286 Mbits/sec             receiver
[ 98]  0.00-10.00  sec  248 MBytes  208 Mbits/sec  1155        sender
[ 98]  0.00-10.00  sec  248 MBytes  208 Mbits/sec             receiver
[100]  0.00-10.00  sec  296 MBytes  248 Mbits/sec  1428        sender
[100]  0.00-10.00  sec  294 MBytes  247 Mbits/sec             receiver
[102]  0.00-10.00  sec  239 MBytes  201 Mbits/sec  1151        sender
[102]  0.00-10.00  sec  238 MBytes  200 Mbits/sec             receiver
[SUM]  0.00-10.00  sec  13.2 GBytes  11.3 Gbits/sec  73596      sender
[SUM]  0.00-10.00  sec  13.1 GBytes  11.3 Gbits/sec             receiver
```

TCP FreeFlow Optimized

```
root@iperf3-7f689b5db-znwxz:/# iperf3 -b 0 -P 50 -c 10.240.0.5
PING 10.244.0.12 (10.244.0.12): 56 data bytes
64 bytes from 10.244.0.12: icmp_seq=0 ttl=62 time=0.445 ms
64 bytes from 10.244.0.12: icmp_seq=1 ttl=62 time=0.372 ms
64 bytes from 10.244.0.12: icmp_seq=2 ttl=62 time=0.391 ms
64 bytes from 10.244.0.12: icmp_seq=3 ttl=62 time=0.381 ms
64 bytes from 10.244.0.12: icmp_seq=4 ttl=62 time=0.336 ms
64 bytes from 10.244.0.12: icmp_seq=5 ttl=62 time=0.384 ms
64 bytes from 10.244.0.12: icmp_seq=6 ttl=62 time=0.372 ms
64 bytes from 10.244.0.12: icmp_seq=7 ttl=62 time=0.349 ms
64 bytes from 10.244.0.12: icmp_seq=8 ttl=62 time=0.423 ms
64 bytes from 10.244.0.12: icmp_seq=9 ttl=62 time=0.360 ms
64 bytes from 10.244.0.12: icmp_seq=10 ttl=62 time=0.344 ms
64 bytes from 10.244.0.12: icmp_seq=11 ttl=62 time=0.367 ms
64 bytes from 10.244.0.12: icmp_seq=12 ttl=62 time=0.387 ms
64 bytes from 10.244.0.12: icmp_seq=13 ttl=62 time=0.407 ms
64 bytes from 10.244.0.12: icmp_seq=14 ttl=62 time=0.348 ms
64 bytes from 10.244.0.12: icmp_seq=15 ttl=62 time=0.361 ms
```

TCP FreeFlow Non-Optimized

```
root@iperf3-7f689b5db-znwxz:/# ping 10.244.0.12
PING 10.244.0.12 (10.244.0.12): 56 data bytes
64 bytes from 10.244.0.12: icmp_seq=0 ttl=62 time=0.410 ms
64 bytes from 10.244.0.12: icmp_seq=1 ttl=62 time=0.456 ms
64 bytes from 10.244.0.12: icmp_seq=2 ttl=62 time=0.451 ms
64 bytes from 10.244.0.12: icmp_seq=3 ttl=62 time=0.419 ms
64 bytes from 10.244.0.12: icmp_seq=4 ttl=62 time=0.456 ms
64 bytes from 10.244.0.12: icmp_seq=5 ttl=62 time=0.469 ms
64 bytes from 10.244.0.12: icmp_seq=6 ttl=62 time=0.416 ms
64 bytes from 10.244.0.12: icmp_seq=7 ttl=62 time=0.484 ms
64 bytes from 10.244.0.12: icmp_seq=8 ttl=62 time=0.441 ms
64 bytes from 10.244.0.12: icmp_seq=9 ttl=62 time=0.465 ms
64 bytes from 10.244.0.12: icmp_seq=10 ttl=62 time=0.470 ms
64 bytes from 10.244.0.12: icmp_seq=11 ttl=62 time=2.647 ms
64 bytes from 10.244.0.12: icmp_seq=12 ttl=62 time=0.469 ms
64 bytes from 10.244.0.12: icmp_seq=13 ttl=62 time=0.450 ms
64 bytes from 10.244.0.12: icmp_seq=14 ttl=62 time=0.512 ms
64 bytes from 10.244.0.12: icmp_seq=15 ttl=62 time=0.489 ms
```

With this setup, using iperf3 under repeated attempts, I did not measure any noticeable difference in total throughput vs optimized versus non-optimized, however, it is worth noting that total throughput on the host was the same speed of 11.5-12 Gbits /sec, so in this case, the virtual network was able to keep pace with the host network. The Freeflow authors claim that benefit can be seen when running a link that can support a full 40 Gb / sec, so more work needs to be done to evaluate the performance using NICs and links.

However, while there was no improvement in throughput, there was consistent improvement with latency. When Freeflow was enabled “optimized” there was indeed a latency reduction of 0.10 ms on average.

Further Evaluation/Findings

Further work needs to be done to investigate the performance of Freeflow. Using a cloud provider such a Microsoft Azure, it was difficult to adequately measure performance due to variable link speeds and possible rate-limiting done after the packets leave the VM NICs, which as a user, you have no control over. An improved testbed would be to provision bare-metal machines, configured with appropriate NICs, install Kubernetes from scratch, and use dedicated links to fully isolate the environment to truly understand the performance gains.

The next step will be to test freeflow using RDMA support with Mellanox drivers, but further work needs to be done to investigate using Freeflow RDMA with compatible Mellanox NICs as the drivers I attempted to use did not compile with Freeflow.

One of my wishlist items, from a diagnostic perspective, would be to have a tool such as traceroute that can provide the entire packet trace including any packer encapsulation (e.g. VXLAN) and time the differences between network hops. Searching for a tool such as this doesn’t seem to exist, although some vendors provide it (e.g. Juniper) and that is assumed to only work with their hardware.

Related Work / Conclusion

The main goal/purpose of this project was to investigate various approaches to optimize the network as it relates to containerized applications. The traditional approach of having packets traverse both the container overlay network and then the host network is redundant. While Freeflow and Slim both offer a portable approach to by-pass container overlays to reduce time spent processing packets, these are both experimental and do not show promise for continued support. I think there will be real use cases for this as network speeds become increasingly larger and containerized applications become more mainstream. This might indeed become a service tier offered by cloud providers.