Workload Mobility Across Datacenters

Kaj Suhonen, Tuomas Päivärinta, Kari Päivärinta

ArctiqDC – Arctic datacenters 2019 - 2021

ArctiqDC – Arctic Datacenters project aims to strengthen the regional data centre industry's products, services, solutions and offerings to customers (parties) outside the region, nationally or internationally. This should be done by demonstrating and proving that; Investing and operating data centres in Arctic regions have low and among the lowest investment and operating costs in the world in terms of cooling and power distribution

ICE
RI.SE SICS NORTH

UNIVERSITY OF OULU

OULUN ENERGIA

Oulun DataCenter
Northern Data Cloud

H66

Hushållnings sällskapet

AURORA DATACENTERS

ÄLVSBYNS KOMMUN

sftec

Business Rovaniemi

LAPIN LIITTO

xarepo

OULU | BusinessOulu

Interreg
Pohjoinen
Euroopan aluekehitysrahasto

EUROOPAN UNIONI

CONTENT

Introduction

This research was done for the InterReg funded project ArctiqDC. It consists of 11 partners from Finland and Sweden, with a funding of 1 430 000€. The goal for ArctiqDC is to strengthen the regional datacenter products and services and provide them to other parties. The project has been divided into smaller activities, such as Datacenter Automation, of which this research is a part of.

The main goal of this paper is to map the needed components/elements to support workload migration across any datacenter. This has been divided into two sections: 1. Setting up the required infrastructure at the source and the target clusters and 2. Performing the migration of the workload between these clusters. Migration should support applications with and without stateful data. The secondary goal was to implement the needed functionality with existing open-source projects that have a noticeable community activity around them to remain relevant. Thirdly, we create a proof-of-concept setup to estimate what level of downtime an application might experience during the migration phase.

In the scope of ArctiqDC – project, the results of this research could be used as a basis for automating load migration between datacenters driven with different metrics. For instance, the cooling capacity of a naturally cooled datacenter located at an arctic climate might experience a relatively sudden cooling power shortage requiring to reduce the power load. Or a hydropower and/or wind turbine powered datacenter experiences a shortage in the power supply requiring to migrate loads to another location with an excess offering of resources.

This paper aims to extend Martin Ternerborgs thesis written for the ArctiqDC “Enabling container failover by extending current container migration techniques”, from a more practical view.

Application Container and Orchestration

Virtual machines have long been the standard to provide virtualization of the underlying physical infrastructure. The virtualization layer drastically improves workload mobility among other things as the live migration of a virtual machine has evolved into a trivial task. However, this is usually designed to work in intra-datacenter migrations with high bandwidth networking. To support migrations between any datacenter, we need to migrate the workload over WAN connections.

Container technology has gained a massive increase in popularity in the last few years, especially with the introduction of Docker. The major difference is that a container runs on the host machine's kernel rather than running their own kernel as traditional virtual machines do [1]. As the container shares the host operating system and libraries, they essentially hold only the application itself making them more lightweight compared to virtual machines. Often these container images are hosted in public repositories, such as Docker Hub, making them easily accessible. These characteristics makes containers a promising technology in workload migration across datacenters.

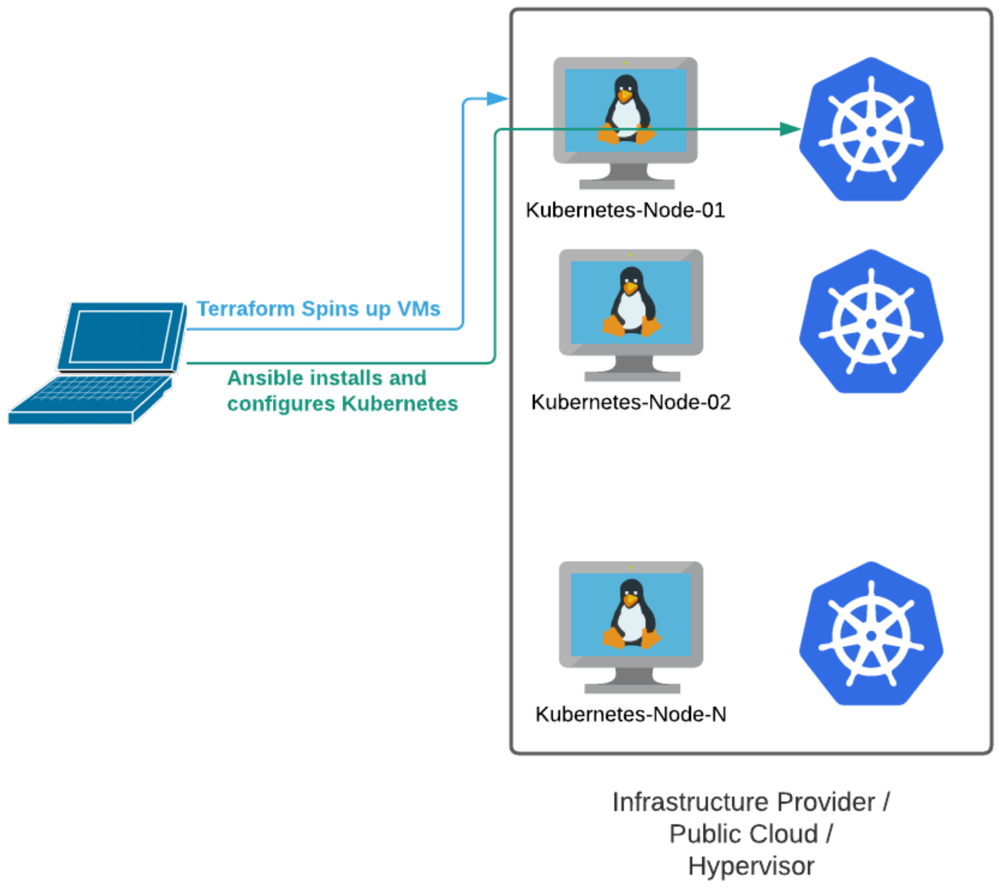
A modern container-based software is often divided into multiple smaller applications, often called as microservices. These microservices communicate with each other over the network, usually through a well-defined REST API. Segmenting the software into these smaller components increases the horizontal scalability, as the developer can simply increase the count of the running instances of a specific microservice. However, this segmentation increases the operational complexity of the software, for example the life cycle management of the containers and the managing of their networking requirements.

Open-source project called Kubernetes has developed into the de facto container orchestration system that handles many of the challenges that are introduced with microservice application architecture. Not only solving the above networking and life cycle management challenges, Kubernetes also makes it trivial to scale the application and/or the application platform (i.e. the nodes running containers). Most importantly, it increases the level of automation and portability of a container-based application. [2]

Kubernetes Infrastructure as a Code

To be able to provide true application mobility across datacentres, we need to have a declarative way to spin up and maintain a common application platform on any environment (i.e., bare metal / private cloud / public cloud). Infrastructure as a code is a process that has been evolved to reduce or remove the need for repetitive manual tasks to create and maintain infrastructure environments (i.e., virtual machines and their networking). The principal is to describe the desired infrastructure in versioned configuration files and the IaC process is responsible for creating it in a reproducible way.

For this approach we have chosen Terraform to spin up the needed infrastructure and Ansible to install the application platform (Kubernetes) on it. Both tools are opensource with a strong community maintaining them. We can consider Terraform as a “driver” that needs to be configured according to the target environment. For instance, Terraform needs a plugin, or a “provider”, to interact with the target hypervisor / cloud platform. Also, for example, there may be differences on how to provision and configure networking.


Terraform Spins up VMs
Ansible installs and configures Kubernetes
Kubernetes-Node-01
Kubernetes-Node-02
Kubernetes-Node-N
Infrastructure Provider / Public Cloud / Hypervisor

Pic. 1. Terraform + Ansible Installing Kubernetes

Implementation

The implemented Terraform approach is configured to install mainstream Linux virtual machine(s) (NODE_COUNT and MASTER_COUNT) with custom resources according to the below configuration file. It also supports multiple clusters (donated with CLUSTER_ID). Terraform is responsible on getting the virtual machines to a state where they respond to ssh requests.

Terraform:

```
locals {
  cluster_id      = 0
  worker_count    = 3
  master_count    = 3

  vm_template     = "Ubuntu-18.04-base"
  master_memory   = 12228
  master_cpus     = 2
  node_memory     = 12228
  node_cpus       = 2

  worker_add_disk = [
    "512000",
    "512000",
  ]
}
```

Once the environment responds to ssh – requests, Terraform will call Ansible which will install and configure the required Kubernetes components on the virtual machines:

Terraform:

```
resource "null_resource" "master-playbook" {
  provisioner "local-exec" {
    command = "ansible-playbook master-playbook.yml"
  }
}


resource "null_resource" "worker-playbook" {
  provisioner "local-exec" {
    command = "ansible-playbook worker-playbook.yml"
  }
}
```

Ansible:

```
 - name: Disable swap
   command: swapoff -a
   when: ansible_swaptotal_mb > 0

 - name: Add Kubernetes repo signing key
   apt_key:
     url: https://packages.cloud.google.com/apt/doc/apt-key.gpg
     state: present

 - name: Adding Kubernetes repo
```

```
  apt_repository:
    repo: deb https://apt.kubernetes.io/ kubernetes-xenial main
    state: present
    filename: kubernetes.list

- name: Install Kubernetes
  apt:
    name: "{{ packages }}"
    state: present
  register: task_result
  until: task_result is success
  retries: 20
  delay: 2
  vars:
    packages:
      - kubelet
      - kubeadm
      - kubectl
```

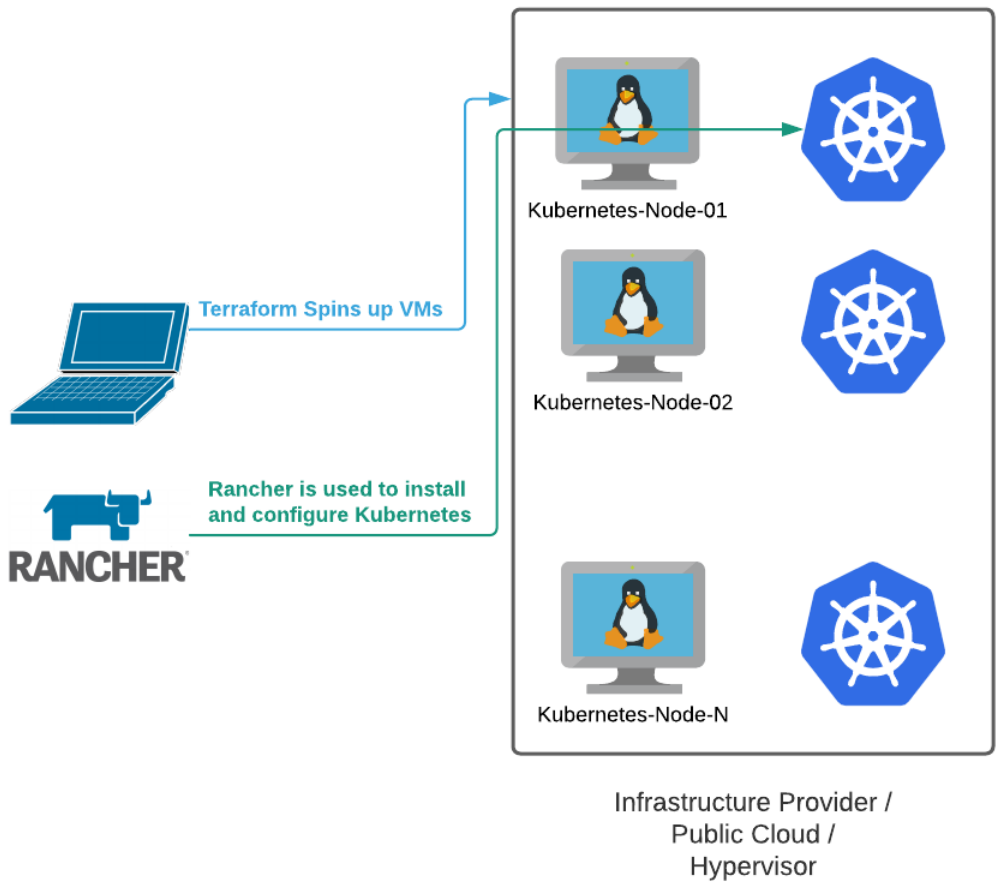
Outcome and findings

The outcome for this sub – project was a click to deploy Kubernetes infrastructure with the ability to scale Kubernetes resources. However, it was quickly discovered that the approach is lacking some basic functionality such as the ability to upgrade the Kubernetes version of a running cluster, which is expensive to implement. Also, as the mainstream Kubernetes receives frequent updates, it would require constant upkeep of the Kubernetes installation and configuration part of the IaC (the Ansible part). Therefore, Kubernetes installation and configuration had to be replaced with an opensource solution with a strong development team behind it.

Open-source Kubernetes Infrastructure Management

Multiple different open-source Kubernetes provisioners was considered such as Kubespray and Rancher from which the latter was chosen into this project. The advantage of Rancher approach appeared to be stronger community and better handling / visibility of multiple Kubernetes cluster.

From the previous IaC – implementation (Terraform + Ansible) we kept the Terraform implementation to create the infrastructure for the cluster, and then used Rancher to install Kubernetes on it. For the sake of time saving, we did not automate the Kubernetes installation the same way as it was automated in the previous IaC – implementation.


Terraform Spins up VMs
RANCHER
Rancher is used to install and configure Kubernetes
Kubernetes-Node-01
Kubernetes-Node-02
Kubernetes-Node-N
Infrastructure Provider / Public Cloud / Hypervisor

Pic. 2. Terraform + Rancher installing Kubernetes

Outcome and findings

Rancher solved many of the issues we had in the IaC – implementation such as Kubernetes cluster lifecycle management and visibility / manageability of multiple Kubernetes clusters. The manual phase of the new

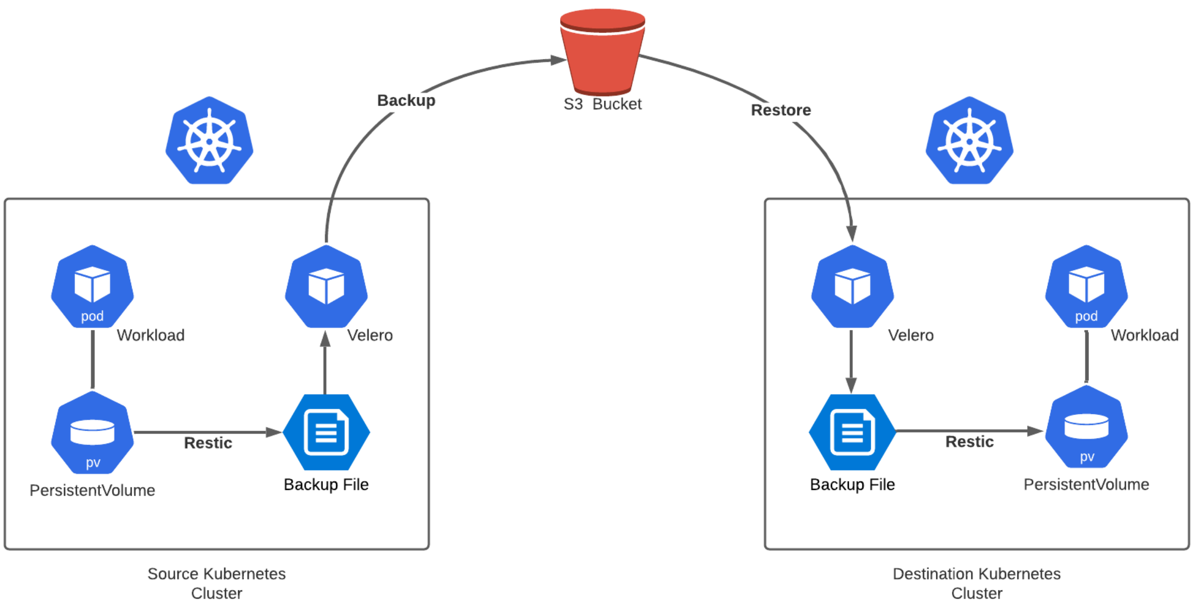
Kubernetes cluster installation could rather easily be automated into the Terraform – script but was chosen not to in order to save time. Also, Rancher could possibly replace the need for Terraform – script altogether with the use of a “Node Driver” – implementation. In that way a new Kubernetes cluster creation would be 100% automatic and handled by Rancher.

Application Mobility

A modern Kubernetes based application has multiple components that needs to be migrated including but not limited to: the container workloads (eg. deployments), networking (e.g. services / ingresses) configuration (e.g. configmap) and the stateful data. In many cases a stateless application could be migrated from cluster to another simply by exporting the required components in a yaml – formatted definition file (for instance the whole namespace where the application is running) and importing it to the new cluster. This imposes a requirement of that the container reposition can be reached from the target cluster, such as a public Docker Hub Container Library.

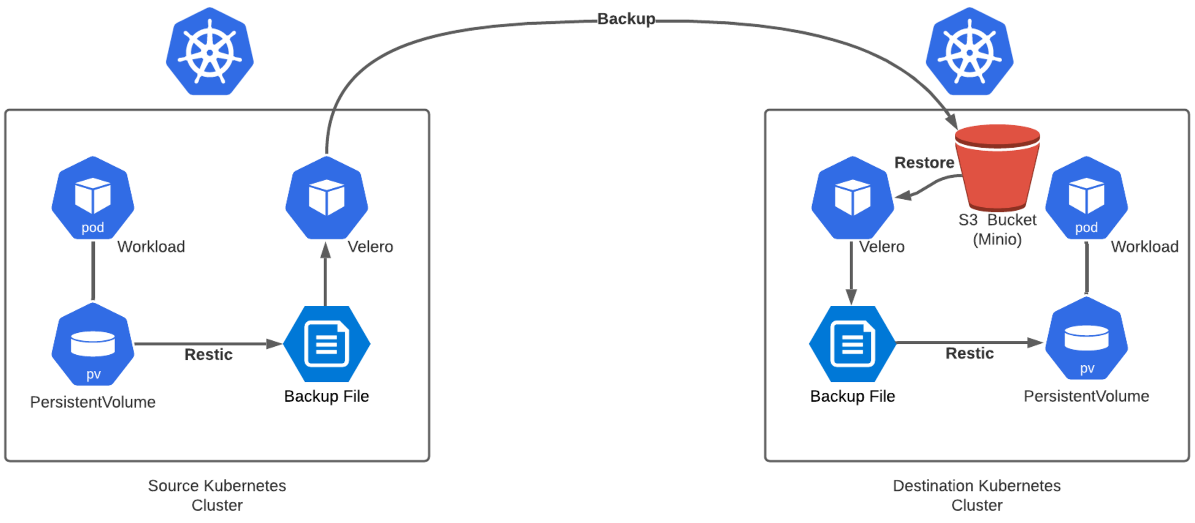
However, stateful applications, such as databases, needs to also migrate their data. The proposed method for migrating the workload should support both the stateless and stateful applications. An open source project named Velero was chosen due to its popularity and versatility. Velero's primary use case is a backup solution for Kubernetes based workloads, but it can be used to migrate from one cluster to another supporting both stateful and stateless applications.

The usual workflow for backing up a workload is to an external S3 storage as shown in Pic.3.



Pic. 3. Velero+Restic with External S3 Storage

However, as our goal is to migrate workloads from cluster A to cluster B, it might be favourable for the target S3 storage to be as close as possible to the destination cluster B. Therefore, we chose to host the target S3 storage on the destination cluster B as shown in Pic. 4. The S3 storage is provided with an open-source project named Minio.



Pic. 4. Velero+Restic with S3 hosted at the Destination Cluster

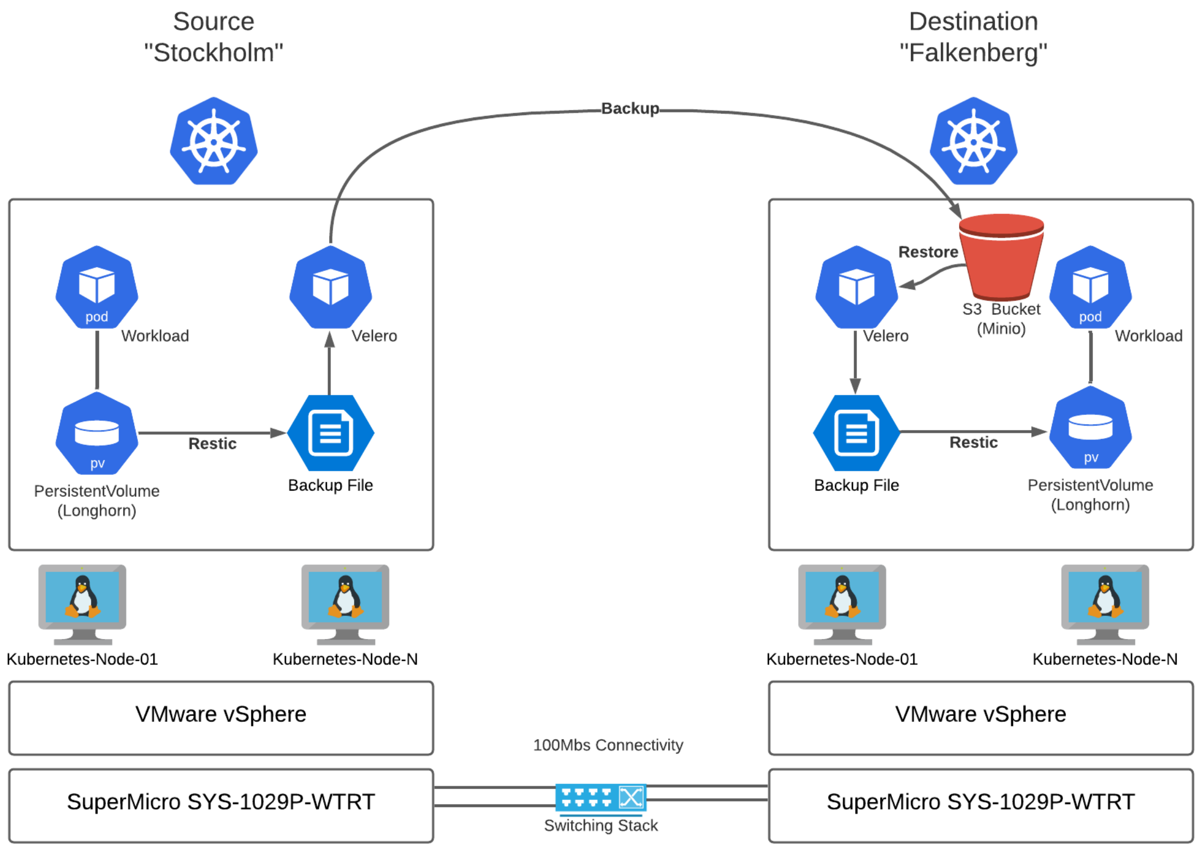
Velero uses a "storage provider – plugin" to support backing up the state of the container workload (i.e., the persistent volume of the container). For instance, there is its own plugin for AWS and Google based K8s implementations. However, as we are using Longhorn to provide persistence for the workloads, we need to use an open-source tool Restic in conjunction with Velero to backup those volumes.

Restic supports incremental backups i.e., transferring only the changed data of the stateful application. Therefore, Velero in conjunction with Restic should support the warm migration method, or pre-copy migration, as described in [1].

Proof of Concept Setup

Our setup consists of two Kubernetes clusters, a source and a destination cluster. We have configured a 100Mb link in between to simulate a typical network speed over internet. Due to time constraints, we use a proprietary hypervisor VMware vSphere but it could easily be replaced with a open-source solution such as KVM. Chosen hypervisor plays insignificant role in the test setup.

The original plan was to migrate workloads between the datacenters of RISE and Oulun DataCenter, both participating in the ArctiqDC – project, but this had to be given up due to time constraints.

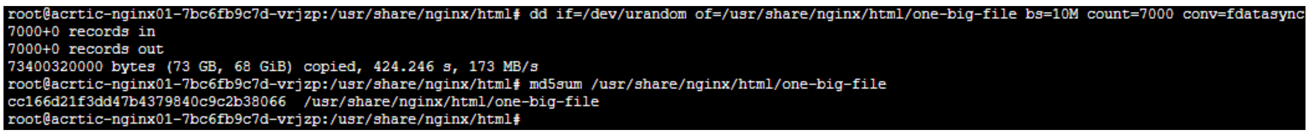


Pic. 5. Overview of the Test Setup

We chose three scenarios to test warm migration of an application with different characteristics: a file-based application with a single large file, another with many small files and a database. With these different characteristics we try to evaluate how efficiently Velero with Restic performs with transferring / restoring only the changed data, which is a key factor in warm migration latency / downtime [1].

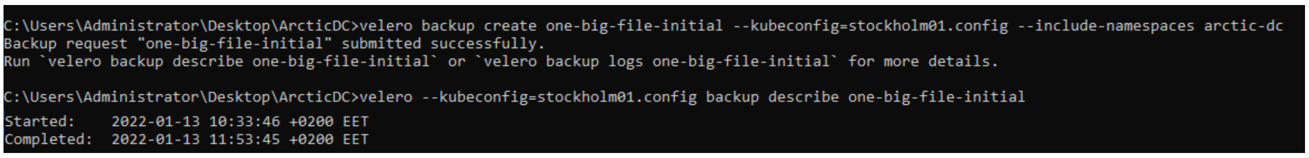
Migrating an Application with a Single Large File

Our first test evaluates how Velero with Restic handles an application with a single / few large files. We start by generating a file with random data in the source cluster (Stockholm) and calculating a MD5 hash value for the file (to check the integrity of the file when performing restore to the target cluster). The size of the file is around 70GB.



```
root@acrtic-nginx01-7bc6fb9c7d-vrjzp:/usr/share/nginx/html# dd if=/dev/urandom of=/usr/share/nginx/html/one-big-file bs=10M count=7000 conv=fdatasync
7000+0 records in
7000+0 records out
73400320000 bytes (73 GB, 68 GiB) copied, 424.246 s, 173 MB/s
root@acrtic-nginx01-7bc6fb9c7d-vrjzp:/usr/share/nginx/html# md5sum /usr/share/nginx/html/one-big-file
cc166d21f3dd47b4379840c9c2b38066  /usr/share/nginx/html/one-big-file
root@acrtic-nginx01-7bc6fb9c7d-vrjzp:/usr/share/nginx/html#
```

We then create a backup of the application and measure the time for the initial transfer of the data.

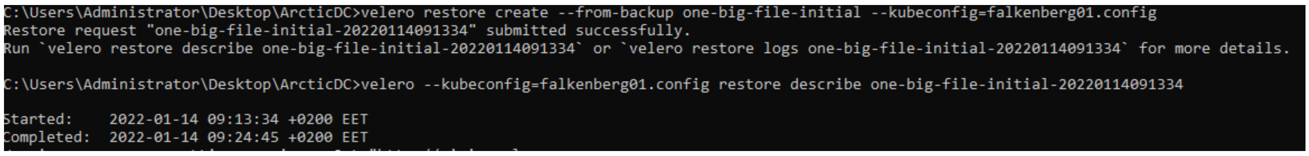


```
C:\Users\Administrator\Desktop\ArcticDC>velero backup create one-big-file-initial --kubeconfig=stockholm01.config --include-namespaces arctic-dc
Backup request "one-big-file-initial" submitted successfully.
Run `velero backup describe one-big-file-initial` or `velero backup logs one-big-file-initial` for more details.

C:\Users\Administrator\Desktop\ArcticDC>velero --kubeconfig=stockholm01.config backup describe one-big-file-initial
Started:    2022-01-13 10:33:46 +0200 EET
Completed:  2022-01-13 11:53:45 +0200 EET
```

The initial transfer took around 80 minutes to complete (~0.88GB per minute).

We then proceed to test the restore of the initial data into the target cluster (Falkenberg).



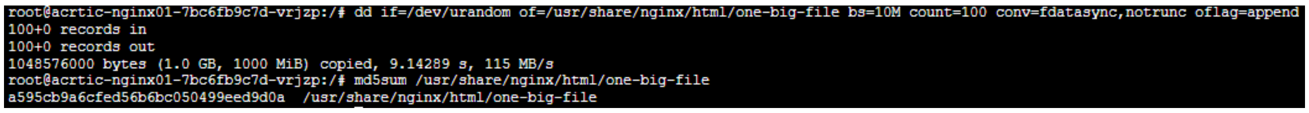
```
C:\Users\Administrator\Desktop\ArcticDC>velero restore create --from-backup one-big-file-initial --kubeconfig=falkenberg01.config
Restore request "one-big-file-initial-20220114091334" submitted successfully.
Run `velero restore describe one-big-file-initial-20220114091334` or `velero restore logs one-big-file-initial-20220114091334` for more details.

C:\Users\Administrator\Desktop\ArcticDC>velero --kubeconfig=falkenberg01.config restore describe one-big-file-initial-20220114091334

Started:    2022-01-14 09:13:34 +0200 EET
Completed:  2022-01-14 09:24:45 +0200 EET
```

The restore took around 11 minutes (~6.36GB per minute). As expected, the restore speed is much higher as the data is already present at the target cluster.

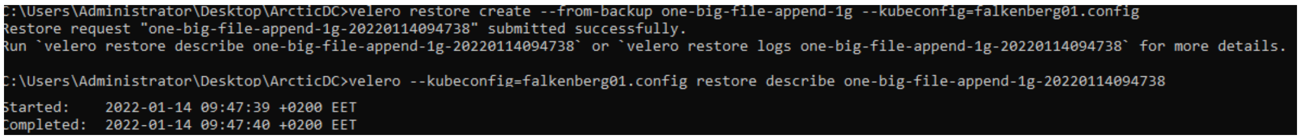
We continue the test by introducing some delta to the source application (Stockholm) by appending it with 1GB of random data and then transferring it.



```
root@acrtic-nginx01-7bc6fb9c7d-vrjzp:/# dd if=/dev/urandom of=/usr/share/nginx/html/one-big-file bs=10M count=100 conv=fdatasync,notrunc oflag=append
100+0 records in
100+0 records out
1048576000 bytes (1.0 GB, 1000 MiB) copied, 9.14289 s, 115 MB/s
root@acrtic-nginx01-7bc6fb9c7d-vrjzp:/# md5sum /usr/share/nginx/html/one-big-file
a595cb9a6cfed56b6bc050499eed9d0a  /usr/share/nginx/html/one-big-file
```

The delta transfer of 1GB took around 8 minutes (~0.13GB per minute).

We end the test by trying to restore the delta on top of the initial restore at the target cluster (Falkenberg).



```
C:\Users\Administrator\Desktop\ArcticDC>velero restore create --from-backup one-big-file-append-1g --kubeconfig=falkenberg01.config
Restore request "one-big-file-append-1g-20220114094738" submitted successfully.
Run `velero restore describe one-big-file-append-1g-20220114094738` or `velero restore logs one-big-file-append-1g-20220114094738` for more details.

C:\Users\Administrator\Desktop\ArcticDC>velero --kubeconfig=falkenberg01.config restore describe one-big-file-append-1g-20220114094738

Started:    2022-01-14 09:47:39 +0200 EET
Completed:  2022-01-14 09:47:40 +0200 EET
```

It was noted that the delta restore took only a second indicating that the process did not complete as expected. By examining Velero logs we notice the restore process skipping the delta of the stateful data:

Skipping persistentvolumes/pvc-e08ee6ec-c0a4-4b63-b00c-a00f48fe28d7 because it's already been restored.

Therefore, we can conclude that with our test setup and configuration, Velero with Restic supports warm migration at the source cluster (i.e. sends only the delta of the application data) but not at the target cluster (i.e. restore is always a full restore).

Migrating an Application with Many Small Files

We perform a similar test, now with 7000 files the size of 10MB and simulate a delta of 100 files. Therefore, the initial transfer and the delta sizes are directly comparable to the above test (70GB initial data, 1GB delta data).

Initial transfer was recorded to take around 80 minutes (~0.88GB per minute) and the restore 11 minutes (6.36GB per minute). The transfer of the delta took place in 1 minute 15 seconds (~1GB per minute). Delta restore failed similarly as in the above test, which was expected.

Migrating a Database Based Application

For a more real-world scenario we chose a MySQL database populated with generated data. This scenario is meant to represent a typical web application where a relatively small and static frontend workload (such as WordPress) stores its state in a potentially large and dynamic database (such as MySQL).

We create a simple table:

create table arctic.person (

```
  id int auto_increment primary key,
  long_name text,
  age int,
  birth_day date
);
```

And initially populate it with 1M of dummy rows:

```
for n in {1..1000000}; do
    mysql -uroot -ppassword -e "insert into arctic.person (long_name, age, birth_day) select
repeat(md5(rand()), 1000), floor(rand()*100), date_add(date('1900/01/01'), interval floor(rand()*100)
year);"
done
```

This resulted in a 37GB of data on the disk. Velero transferred this initial state to the target cluster in 42 minutes (~0.88GB per minute).

The introduced delta data was an additional 27k rows occupying around 1GB of physical disc. The transfer time was around 4 minutes (~0.25GB per minute).

The restore operation at the target cluster took 6 minutes for the initial data

Summary of the PoC

Application Characteristic	Initial Data GB	Delta Data GB	Initial Data Migration Speed GB/min	Delta Data Migration Speed GB/min	Restore Speed at the Target Cluster* GB/min
Single Large File	70	1	0.88	0.13	6.36
Many Small Files	70	1	0.88	0.80	6.36
MySQL Database	37	1	0.88	0.25	6.16

*In our configuration, restore was always a full restore i.e. delta restore was not supported

We can see that the initial data migration speed is the same in all scenarios, which might indicate that the bottleneck was the 100Mb network speed that was configured to simulate migration over public internet. Also, the restore speed at the target cluster is the same in all scenarios. The most interesting finding here is the delta data migration speed difference between the single large file and many small files – scenarios. The finding would indicate that Restric spends majority of the time to calculate what has changed in the source data file(s), making it less efficient in migration scenarios where there are a few large files experiencing small changes. The delta migration speed of the MySQL– scenario would indicate that the database is stored as a single large file / few large files.

Results

The main goal for this research was to map what elements/components needs to be implemented to support application migration across any datacenter. This was divided into two sections: 1. setting up the infrastructure supporting application mobility and 2. the actual migration of the application. The secondary goal was to identify which of these elements could be implemented with existing open-source projects. Thirdly, a PoC setup was created to estimate what level of downtime an application might experience during the migration phase, which is the key factor for a practical migration. The findings were expected to be the basis for automatic load migration between datacenters driven with different metrics such as the availability of natural cooling capacity.

It was seen that setting up and maintaining the Kubernetes application platform was effectively achieved with open-source projects Terraform + Rancher. Also, application migration was successfully performed with open-source projects Velero + Restic + Minio.

As stated earlier, our configuration did not support delta restore at the target cluster. Instead, the application migration workflow would go roughly:

1. Perform initial data migration (initial backup)
2. Perform periodic delta migrations (delta backup)
3. Schedule a switchover to the target cluster. Downtime starts, perform the final delta migration.
4. Perform full data restore at the target cluster (initial + all deltas). Downtime ends.

Depending on the application, the downtime appears to be dominated by the speed of the full data restore at the target cluster, which in our case was around 6GB per minute. Therefore, even with this preliminary configuration, a relatively large application of 100GB could be migrated with a downtime around 20 minutes, showing the potential of this approach.

However, to be a production suitable solution for automatic load migration scenarios described in the introduction, the downtime needs to be significantly lower. The above downtime might be suitable for some cases, but the availability for an enterprise application should be near 100% even during the migration phase. Therefore, enhancing the current setup to support live migrations should be the target before load balancing between datacenters can be achieved.

Further Development

The current Terraform + Rancher implementation for setting up the application platform should be made 100% Rancher implantation by replacing the Terraform with the Rancher Node Driver to handle setting up the infrastructure. This would result in a cleaner solution but more importantly allow a higher level of automation (or at least make it cheaper to implement).

The current implantation of the application migration by Velero causes a downtime defined by the sum of the last delta migration from the source cluster + full restore at the target cluster. It should be investigated how the configuration should be altered to support delta restore to drastically reduce the downtime needed (defined by the delta rate of the application). Also, solutions for near zero downtime (live migration) should be investigated to further increase the potential of automatic load migration.

Once the application is ready on the target cluster, end user traffic needs to be routed to it. The simplest approach would probably be a DNS entry with a low TTL, however more sophisticated solutions might need to be investigated when trying to minimize application downtime. This was not in the scope of this research since the current migration implementation causes a downtime that is well over DNS delay.

References

[1] Martin Terneborg. “Master’s Thesis: Enabling container failover by extending current container migration techniques”. In Luleå University of Technology 2021

[2] Brendan Burns, Joe Beda & Kelsey Hightower. “Kubernetes Up & Running”. By O’Reilly Media 2019