

# Getting started with HDFS on Kubernetes

My experience running a proof-of-concept of HDFS on Kubernetes



Tirumarai Selvan

Follow

Feb 12 · 6 min read

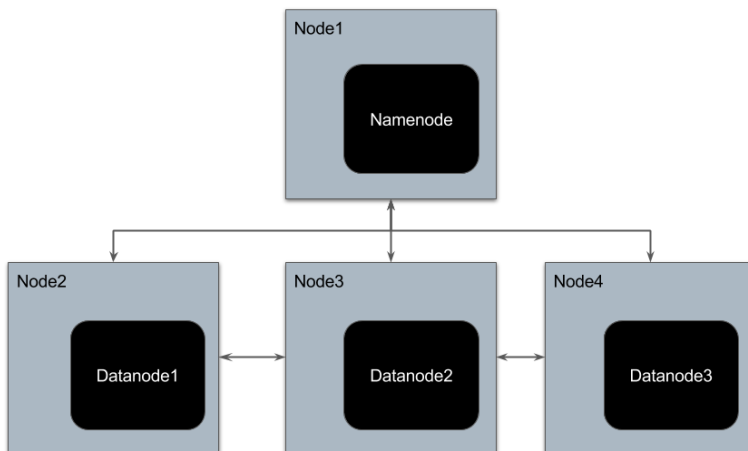
## Contents

1. [Basic architecture of HDFS](#)
2. [Architecture of HDFS on Kubernetes](#)
3. [Wrap namenode in a Service](#)
4. [Identify datanodes through Stateful Sets](#)
5. [Run fully distributed HDFS on single node](#)
6. [Next: Apache Spark on HDFS](#)

If you are a Kubernetes expert, then you can jump straight to the source code [here](#).

## Basic architecture of HDFS

Let's begin by recapping the traditional architecture of a completely distributed HDFS. The block diagram below illustrates this architecture:



Traditional HDFS architecture

Typically, there are dedicated VMs for running each HDFS daemon viz. namenodes and datanodes (we will use daemons and the nodes running the daemons interchangeably). Namenodes store the metadata of each file and datanodes store the actual data. There is one namenode (sometimes two for high availability but we will ignore it for our PoC) and multiple datanodes.

Each daemon communicates with the other using the host VM's network. This is an important point that will affect the way we run this on Kubernetes.

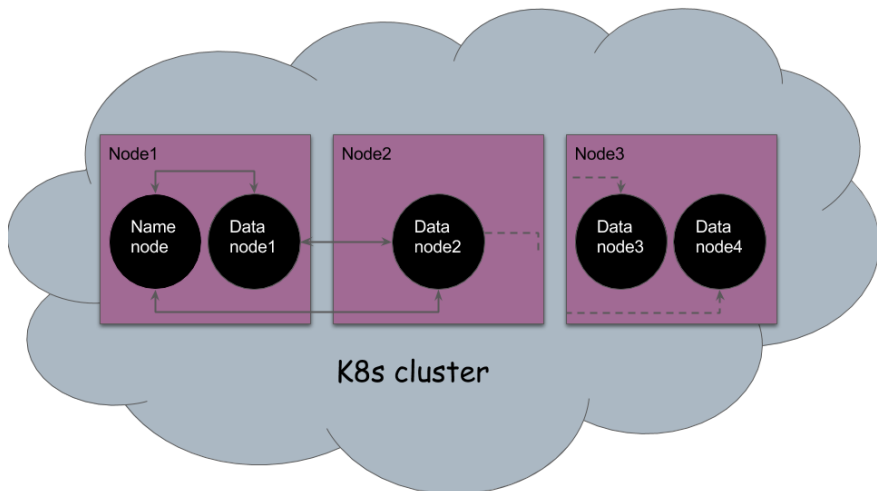
## Architecture of HDFS on Kubernetes

Now, let's understand how a typical application looks like on Kubernetes (also known as k8s).

Kubernetes is a container manager for a cluster of nodes. This means it deploys containers and manages their lifecycle on a cluster. A specific resource kind in Kubernetes specifies how a container should behave: should it be a long-running or batch process, should there be a single instance or multiple replicas, etc. A 'pod' is the simplest resource kind which is basically a single instance of few tightly coupled long-running containers.

So how does our application, a complex consortium of different processes interacting with each other, look like on Kubernetes?

At first glance, it seems simple. The different daemon processes will run in different pods and use their pod IPs to interact with each other. This looks something like below:



HDFS on Kubernetes architecture

Not so fast! Do you hard-code the namenode pod IP into your datanodes? What if the namenode pod goes down and comes back up on a different machine (changing its IP)? Same issue with datanode. How can they interact with each other if they keep changing IPs, machines and whatever else happens under Kubernetes management?

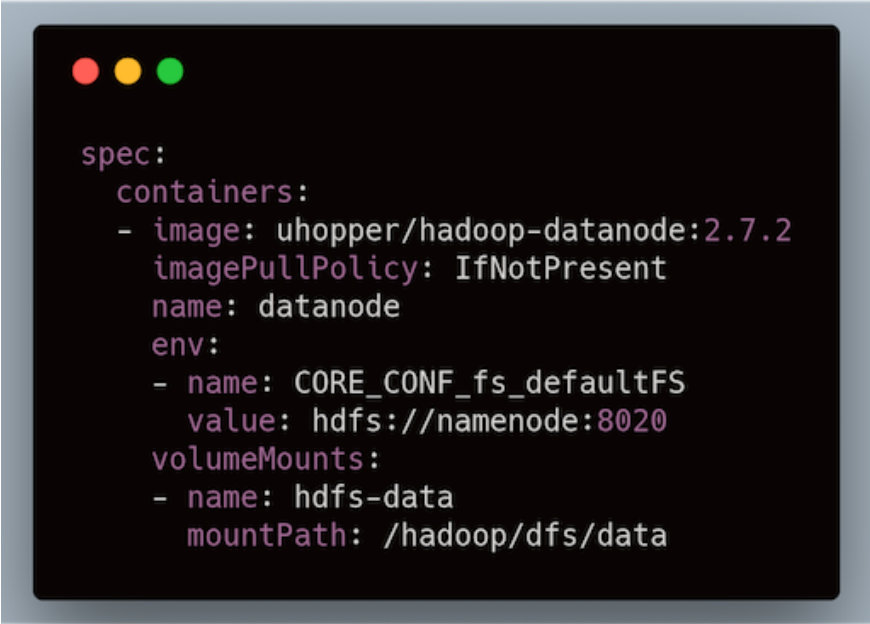
## So we have the following issues:

1. Namenode and datanodes can go down and come back up with a different pod IP. How do they keep interacting henceforth?
2. Namenode and datanodes can go down and come back up in a different machine. What happens to the data they were storing?

## Wrap namenode in a Service

The way k8s solves the problem of ephemeral nature of pods is to use a Service resource. A Kubernetes Service basically gives you a static IP/hostname in the cluster which load-balances incoming requests across selected pods. The pods are selected based on labels that are injected in the pod definition. So we can give our namenode pod a label say 'app: namenode' and create a service which selects pods with that label.

This gives us a static way to reach our namenode. The Datanodes are now happy. For e.g. datanodes can refer to the namenodes with Service hostname in their spec:



```
spec:
  containers:
  - image: uhopper/hadoop-datanode:2.7.2
    imagePullPolicy: IfNotPresent
    name: datanode
    env:
    - name: CORE_CONF_fs_defaultFS
      value: hdfs://namenode:8020
    volumeMounts:
    - name: hdfs-data
      mountPath: /hadoop/dfs/data
```

Datanodes can use K8s Service hostname

A HDFS client in the cluster can also use this hostname:



```
$ #kubectl exec -it namenode /bin/bash
$ hdfs dfs -ls hdfs://namenode:8020
```

Use HDFS client inside the cluster

## Identify datanodes through Stateful Sets

Now that we have seen a way to communicate with namenode regardless of pod behaviour, can we do the same for datanodes?

Unfortunately, no. A fundamental aspect of a Kubernetes Service is that it load-balances randomly across its selected pods. This means all the pods selected by the Service should be indistinguishable for the application to behave properly. This is not the case with our datanodes wherein each datanode has a different state (different data). Datanodes seem more like a case of “pets” not “cattle”.

Stateful applications such as this are quite common and hence Kubernetes provides another resource kind called Stateful Sets to help such applications. In a Stateful Set, each pod gets a sticky identity to its name, its storage and its hostname. Voila! Just what we needed. If each of our datanode has its sticky identity always present (even across pod rescheduling) then we can still be assured of stable communication and consistent data across the application.

In fact, we can make our namenode also stateful if we wished. Note that for all this to work, we need to configure our HDFS to use hostnames instead of IPs for communication. This is the default setting with the Docker images we are using.

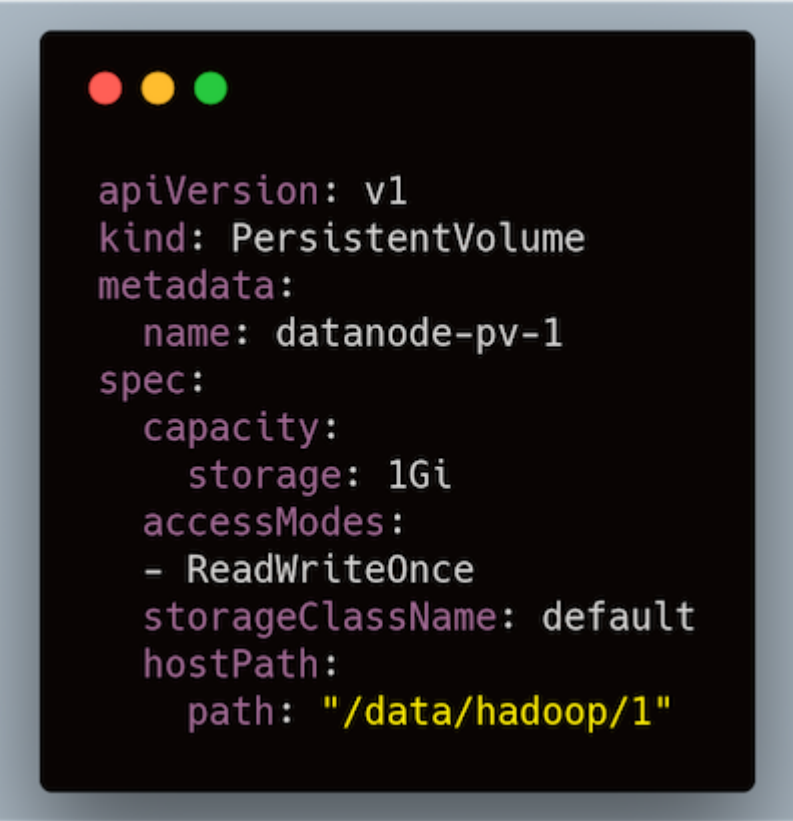
## Run fully distributed HDFS on single node

Are we done? Almost. Why not make all this work in a single node? But isn't that contrary to the concept of “distributed file” system? Well, in the Kubernetes world, the notion of distributed is at the container level.

So, if we have multiple pods, managing their own dedicated disk, running on a single node; it's distributed!

But for this PoC, we will do another trick. We will emulate multiple storage volumes on a single disk and attach them to our different pods (in more serious settings you will have separate disks backing each volume, but it's a matter of simple configuration). Kubernetes Persistent Volume (PV) resource kinds are perfect for this. Kubernetes PVs are a set of storage volumes available for consumption in your cluster. A pod can ask for a PV and it will be mounted in the pod.

We use a PV of type `hostPath` to use the underlying VMs local directory as its storage. And, we can create arbitrary number of them with a single disk with different `hostPath` values.

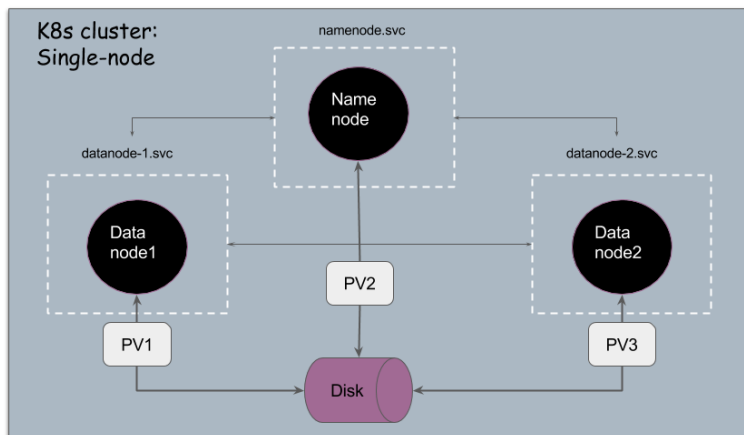
A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. It displays a YAML manifest for a PersistentVolume.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: datanode-pv-1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  storageClassName: default
  hostPath:
    path: "/data/hadoop/1"
```

Persistent Volume on `hostPath`

And that's it! We have a fully distributed HDFS running on a single node. This is what our final architecture looks like:





Single node fully distributed architecture

I've deployed this as a [Hasura Hub project](#) so that it's easy for you to try out. All you need to do is clone the project (which will clone the k8s spec files) and then deploy it on the Hasura free tier.

## Next: Apache Spark on HDFS

A HDFS by itself is not of much use. In our follow-up blogpost, we will show how to deploy Apache Spark on Kubernetes to process data stored in our new k8s HDFS.

Stay tuned!

If you want to deep dive into any aspect of this post or want to give your suggestions, feel free to hit the comments section below.

The Hasura platform is a PaaS + BaaS for the container era. Check it out here: <https://hasura.io/platform/>

. . .

**Hasura** gives you instant GraphQL APIs over any Postgres database without having to write any backend code.

## Subscribe to the Hasura Newsletter

We send about one mail a month

Sign up



I agree to leave Medium and submit this information, which will be collected and used according to [Upscribe's privacy policy](#).