Thursday, April 04, 2019

[**Kubernetes 1.14: Local Persistent Volumes GA**](https://kubernetes.io/blog/2019/04/04/kubernetes-1.14-local-persistent-volumes-ga/)

**Authors**: Michelle Au (Google), Matt Schallert (Uber), Celina Ward (Uber)

The [Local Persistent Volumes](https://kubernetes.io/docs/concepts/storage/volumes/#local) feature has been promoted to GA in Kubernetes 1.14. It was first introduced as alpha in Kubernetes 1.7, and then [beta](https://kubernetes.io/blog/2018/04/13/local-persistent-volumes-beta/) in Kubernetes 1.10. The GA milestone indicates that Kubernetes users may depend on the feature and its API for production use. GA features are protected by the Kubernetes [deprecation policy](https://kubernetes.io/docs/reference/using-api/deprecation-policy/).

What is a Local Persistent Volume?

A local persistent volume represents a local disk directly-attached to a single Kubernetes Node.

Kubernetes provides a powerful volume plugin system that enables Kubernetes workloads to use a [wide variety](https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes) of block and file storage to persist data. Most of these plugins enable remote storage – these remote storage systems persist data independent of the Kubernetes node where the data originated. Remote storage usually can not offer the consistent high performance guarantees of local directly-attached storage. With the Local Persistent Volume plugin, Kubernetes workloads can now consume high performance local storage using the same volume APIs that app developers have become accustomed to.

How is it different from a HostPath Volume?

To better understand the benefits of a Local Persistent Volume, it is useful to compare it to a [HostPath volume](https://kubernetes.io/docs/concepts/storage/volumes/#hostpath). HostPath volumes mount a file or directory from the host node’s filesystem into a Pod. Similarly a Local Persistent Volume mounts a local disk or partition into a Pod.

The biggest difference is that the Kubernetes scheduler understands which node a Local Persistent Volume belongs to. With HostPath volumes, a pod referencing a HostPath volume may be moved by the scheduler to a different node resulting in data loss. But with Local Persistent Volumes, the Kubernetes scheduler ensures that a pod using a Local Persistent Volume is always scheduled to the same node.

While HostPath volumes may be referenced via a Persistent Volume Claim (PVC) or directly inline in a pod definition, Local Persistent Volumes can only be referenced via a PVC. This provides additional security benefits since Persistent Volume objects are managed by the administrator, preventing Pods from being able to access any path on the host.

Additional benefits include support for formatting of block devices during mount, and volume ownership using fsGroup.

What’s New With GA?

Since 1.10, we have mainly focused on improving stability and scalability of the feature so that it is production ready.

The only major feature addition is the ability to specify a raw block device and have Kubernetes automatically format and mount the filesystem. This reduces the previous burden of having to format and mount devices before giving it to Kubernetes.

Limitations of GA

At GA, Local Persistent Volumes do not support [dynamic volume provisioning](https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/). However there is an [external controller](https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner) available to help manage the local PersistentVolume lifecycle for individual disks on your nodes. This includes creating the PersistentVolume objects, cleaning up and reusing disks once they have been released by the application.

How to Use a Local Persistent Volume?

Workloads can request a local persistent volume using the same PersistentVolumeClaim interface as remote storage backends. This makes it easy to swap out the storage backend across clusters, clouds, and on-prem environments.

First, a StorageClass should be created that sets volumeBindingMode: WaitForFirstConsumer to enable [volume topology-aware scheduling](https://kubernetes.io/docs/concepts/storage/storage-classes/#volume-binding-mode). This mode instructs Kubernetes to wait to bind a PVC until a Pod using it is scheduled.

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: local-storage

provisioner: kubernetes.io/no-provisioner

volumeBindingMode: WaitForFirstConsumer

Then, the external static provisioner can be [configured and run](https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner#user-guide) to create PVs for all the local disks on your nodes.

$ kubectl get pv

NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE

local-pv-27c0f084 368Gi RWO Delete Available local-storage 8s

local-pv-3796b049 368Gi RWO Delete Available local-storage 7s

local-pv-3ddecaea 368Gi RWO Delete Available local-storage 7s

Afterwards, workloads can start using the PVs by creating a PVC and Pod or a StatefulSet with volumeClaimTemplates.

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: local-test

spec:

serviceName: "local-service"

replicas: 3

selector:

matchLabels:

app: local-test

template:

metadata:

labels:

app: local-test

spec:

containers:

- name: test-container

image: k8s.gcr.io/busybox

command:

- "/bin/sh"

args:

- "-c"

- "sleep 100000"

volumeMounts:

- name: local-vol

mountPath: /usr/test-pod

volumeClaimTemplates:

- metadata:

name: local-vol

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "local-storage"

resources:

requests:

storage: 368Gi

Once the StatefulSet is up and running, the PVCs are all bound:

$ kubectl get pvc

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

local-vol-local-test-0 Bound local-pv-27c0f084 368Gi RWO local-storage 3m45s

local-vol-local-test-1 Bound local-pv-3ddecaea 368Gi RWO local-storage 3m40s

local-vol-local-test-2 Bound local-pv-3796b049 368Gi RWO local-storage 3m36s

When the disk is no longer needed, the PVC can be deleted. The external static provisioner will clean up the disk and make the PV available for use again.

$ kubectl patch sts local-test -p '{"spec":{"replicas":2}}'

statefulset.apps/local-test patched

$ kubectl delete pvc local-vol-local-test-2

persistentvolumeclaim "local-vol-local-test-2" deleted

$ kubectl get pv

NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE

local-pv-27c0f084 368Gi RWO Delete Bound default/local-vol-local-test-0 local-storage 11m

local-pv-3796b049 368Gi RWO Delete Available local-storage 7s

local-pv-3ddecaea 368Gi RWO Delete Bound default/local-vol-local-test-1 local-storage 19m

You can find full [documentation](https://kubernetes.io/docs/concepts/storage/volumes/#local) for the feature on the Kubernetes website.

What Are Suitable Use Cases?

The primary benefit of Local Persistent Volumes over remote persistent storage is performance: local disks usually offer higher IOPS and throughput and lower latency compared to remote storage systems.

However, there are important limitations and caveats to consider when using Local Persistent Volumes:

* Using local storage ties your application to a specific node, making your application harder to schedule. Applications which use local storage should specify a high priority so that lower priority pods, that don’t require local storage, can be preempted if necessary.
* If that node or local volume encounters a failure and becomes inaccessible, then that pod also becomes inaccessible. Manual intervention, external controllers, or operators may be needed to recover from these situations.
* While most remote storage systems implement synchronous replication, most local disk offerings do not provide data durability guarantees. Meaning loss of the disk or node may result in loss of all the data on that disk

For these reasons, local persistent storage should only be considered for workloads that handle data replication and backup at the application layer, thus making the applications resilient to node or data failures and unavailability despite the lack of such guarantees at the individual disk level.

Examples of good workloads include software defined storage systems and replicated databases. Other types of applications should continue to use highly available, remotely accessible, durable storage.

How Uber Uses Local Storage

[M3](https://eng.uber.com/m3/), Uber’s in-house metrics platform, piloted Local Persistent Volumes at scale in an effort to evaluate [M3DB](https://m3db.io/) — an open-source, distributed timeseries database created by Uber. One of M3DB’s notable features is its ability to shard its metrics into partitions, replicate them by a factor of three, and then evenly disperse the replicas across separate failure domains.

Prior to the pilot with local persistent volumes, M3DB ran exclusively in Uber-managed environments. Over time, internal use cases arose that required the ability to run M3DB in environments with fewer dependencies. So the team began to explore options. As an open-source project, we wanted to provide the community with a way to run M3DB as easily as possible, with an open-source stack, while meeting M3DB’s requirements for high throughput, low-latency storage, and the ability to scale itself out.

The Kubernetes Local Persistent Volume interface, with its high-performance, low-latency guarantees, quickly emerged as the perfect abstraction to build on top of. With Local Persistent Volumes, individual M3DB instances can comfortably handle up to 600k writes per-second. This leaves plenty of headroom for spikes on clusters that typically process a few million metrics per-second.

Because M3DB also gracefully handles losing a single node or volume, the limited data durability guarantees of Local Persistent Volumes are not an issue. If a node fails, M3DB finds a suitable replacement and the new node begins streaming data from its two peers.

Thanks to the Kubernetes scheduler’s intelligent handling of volume topology, M3DB is able to programmatically evenly disperse its replicas across multiple local persistent volumes in all available cloud zones, or, in the case of on-prem clusters, across all available server racks.

Uber’s Operational Experience

As mentioned above, while Local Persistent Volumes provide many benefits, they also require careful planning and careful consideration of constraints before committing to them in production. When thinking about our local volume strategy for M3DB, there were a few things Uber had to consider.

For one, we had to take into account the hardware profiles of the nodes in our Kubernetes cluster. For example, how many local disks would each node cluster have? How would they be partitioned?

The local static provisioner provides [guidance](https://github.com/kubernetes-sigs/sig-storage-local-static-provisioner/blob/master/docs/best-practices.md) to help answer these questions. It’s best to be able to dedicate a full disk to each local volume (for IO isolation) and a full partition per-volume (for capacity isolation). This was easier in our cloud environments where we could mix and match local disks. However, if using local volumes on-prem, hardware constraints may be a limiting factor depending on the number of disks available and their characteristics.

When first testing local volumes, we wanted to have a thorough understanding of the effect [disruptions](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/) (voluntary and involuntary) would have on pods using local storage, and so we began testing some failure scenarios. We found that when a local volume becomes unavailable while the node remains available (such as when performing maintenance on the disk), a pod using the local volume will be stuck in a ContainerCreating state until it can mount the volume. If a node becomes unavailable, for example if it is removed from the cluster or is [drained](https://kubernetes.io/docs/tasks/administer-cluster/safely-drain-node/), then pods using local volumes on that node are stuck in an Unknown or Pending state depending on whether or not the node was removed gracefully.

Recovering pods from these interim states means having to delete the PVC binding the pod to its local volume and then delete the pod in order for it to be rescheduled (or wait until the node and disk are available again). We took this into account when building our [operator](https://github.com/m3db/m3db-operator) for M3DB, which makes changes to the cluster topology when a pod is rescheduled such that the new one gracefully streams data from the remaining two peers. Eventually we plan to automate the deletion and rescheduling process entirely.

Alerts on pod states can help call attention to stuck local volumes, and workload-specific controllers or operators can remediate them automatically. Because of these constraints, it’s best to exclude nodes with local volumes from automatic upgrades or repairs, and in fact some cloud providers explicitly mention this as a best practice.

Portability Between On-Prem and Cloud

Local Volumes played a big role in Uber’s decision to build orchestration for M3DB using Kubernetes, in part because it is a storage abstraction that works the same across on-prem and cloud environments. Remote storage solutions have different characteristics across cloud providers, and some users may prefer not to use networked storage at all in their own data centers. On the other hand, local disks are relatively ubiquitous and provide more predictable performance characteristics.

By orchestrating M3DB using local disks in the cloud, where it was easier to get up and running with Kubernetes, we gained confidence that we could still use our operator to run M3DB in our on-prem environment without any modifications. As we continue to work on how we’d run Kubernetes on-prem, having solved such an important pending question is a big relief.

What’s Next for Local Persistent Volumes?

As we’ve seen with Uber’s M3DB, local persistent volumes have successfully been used in production environments. As adoption of local persistent volumes continues to increase, SIG Storage continues to seek feedback for ways to improve the feature.

One of the most frequent asks has been for a controller that can help with recovery from failed nodes or disks, which is currently a manual process (or something that has to be built into an operator). SIG Storage is investigating creating a common controller that can be used by workloads with simple and similar recovery processes.

Another popular ask has been to support dynamic provisioning using lvm. This can simplify disk management, and improve disk utilization. SIG Storage is evaluating the performance tradeoffs for the viability of this feature.

Getting Invovled

If you have feedback for this feature or are interested in getting involved with the design and development, join the [Kubernetes Storage Special-Interest-Group](https://github.com/kubernetes/community/blob/master/sig-storage/README.md) (SIG). We’re rapidly growing and always welcome new contributors.

Special thanks to all the contributors that helped bring this feature to GA, including Chuqiang Li (lichuqiang), Dhiraj Hedge (dhirajh), Ian Chakeres (ianchakeres), Jan Šafránek (jsafrane), Michelle Au (msau42), Saad Ali (saad-ali), Yecheng Fu (cofyc) and Yuquan Ren (nickrenren).