Docker and Storage

- Understanding Docker I/O

by George Crump, Lead Analyst

Designing a Docker Storage Infrastructure

A recent Storage Switzerland report covered the basics of Docker and Storage; what Docker is and how it impacts storage. Container technology and Docker specifically places unique demands on the storage infrastructure that most legacy storage architectures are ill-prepared to handle. The initial concern is developing a storage infrastructure that will support a container based dev/ops environment. The second concern is that these infrastructures will have to support containers in production as the value of something more granular than a virtual machine (VM) is understood.

The State of Docker Storage Today

Docker storage today direct attaches to specific servers and supports an agile dev/ops environment. Gradually, early Docker adopters are moving to shared storage as the need to create stateful containers and the need to share data between containers increases. The need to share data between containers or even have stateful access to data becomes more obvious as the traditional enterprise looks to expand the Docker use case beyond test/dev to production and true dev/ops. The infrastructures that support these shared Docker environments are often initially modeled after virtualized server environments. While container technology has some similarity with virtualized servers, the I/O pattern is not one of them.

What makes Docker I/O Unique?

The first difference between virtual server I/O and container I/O is that VMs are static, but containers are dynamic. A VM is typically created, many times from a template, but it is usually an instance in and of itself. It does not share an operating system or application with other VMs, the I/O coming in and going out of the VM is owned uniquely by that VM. Its I/O has no ties to other VMs. Also in most cases once created, a virtual server will remain in existence permanently.

A container, on the other hand, is dynamic. Containers are built, almost exclusively, from other containers commonly sharing both operating systems as well as application code. An I/O from one container can have a ripple effect on every container that stems from the master. The life span of a container can vary from a few seconds to years. The instant creation of thousands of containers for one specific task is not uncommon. Then as soon as that task is done, maybe minutes later, the containers are eliminated.

These differences from an I/O perspective are startling. All writes to the container are dynamic and have to be handled in real-time. There is no concept of pre-allocating capacity. Secondly, any modification that a container makes from the master has to be tracked separately, again in real-time. This real-time write I/O requires a storage system that can provide consistent, but not necessarily extreme, performance.

In addition to write heavy performance, containers also require decent read performance because many sub-containers made need code from the same master image. Again, high performance is not necessarily critical, but the ability to provide consistent performance to repeated access of the same data is.

Finally, the I/O of a container architecture can be highly variable. A Docker storage infrastructure can poke along for quite some time until there is a need to perform a test of a section of code, or a need to perform an analytics job across terabytes of unstructured data. Then the processes servicing these requests may create dozens, hundreds or even thousands of containers to make sure the code is either tested to its maximum or the analytics result is found quickly. Unlike virtualization, the creation of thousands of containers is not the big I/O challenge, dealing with the aftermath of those containers is. In a Docker environment there could be thousands of independent accesses to data, creating a random I/O stream that virtualization never sees. The storage infrastructure has to be able to handle a highly parallel I/O stream from hundreds, if not thousands, of containers.

The result is that container I/O stresses the write, read and random aspects of the storage infrastructure. Most vendors will respond to this demand similarly to how they responded to virtual desktop infrastructure, with all-flash storage. While flash may be a part of the Docker storage infrastructure design, it is unlikely that a pure flash infrastructure is justifiable for the use case. Instead, IT professionals need to look for a Docker storage infrastructure that provides consistent, scalable capacity and performance.

Designing A Storage Infrastructure To Match Docker

The first step in designing a storage infrastructure for Docker is to make sure it meets the requirements that were laid out in our prior report. At the top of these is to ensure that the environment is software defined. A software defined storage (SDS) solution brings many benefits, but the key benefit for Docker is its flexibility. A software first storage solution will have the adaptability to fit into Docker both today and in the future. Docker is still in its early stages, how it is used and the capabilities of the environment itself will evolve over time.

The second step is to assemble the storage infrastructure from the right hardware. Indeed, an SDS solution enables the use of commodity servers that act as storage nodes. These storage nodes though need to be moderately powered instead of high-end performance beasts to match the highly parallel I/O that containers can generate. The more cores the better, with a combination of flash and hard disk drives. In most cases the infrastructure is better off having 10 x $10,000 storage nodes than it is to have 4 x $25,000 storage nodes.

Also, while SDS does enable a build-it-yourself mentality, for most environments the money saved sourcing all the hardware and software is not worth the time and effort. Instead, look for a vendor that can provide a turnkey, but still price competitive, solution. A turnkey solution allows the organization to focus on what it should be focused on; developing applications instead of becoming storage integrators.

Finally, the design should include or integrate with a container management solution like Flocker. As Docker moves more into production, it needs to become more “stateful” but the container still needs to be portable. That means that the storage system should expose portable resilient volumes.

Conclusion

Docker is in an interesting state of its evolution. It is being widely adopted by development teams to be the foundation of a dev/ops initiative. At the same time Docker, and container technologies in general, are starting to catch the attention of production IT. A storage system selected today needs to first be designed as a distributed system that can keep pace with the unique architecture of containers, and then, secondly, it needs to have the flexibility to evolve with the environment as it picks up an increasing number of production responsibilities.