



Filesystem Sharding Tactics and Processes

White Paper

What is filesystem sharding?

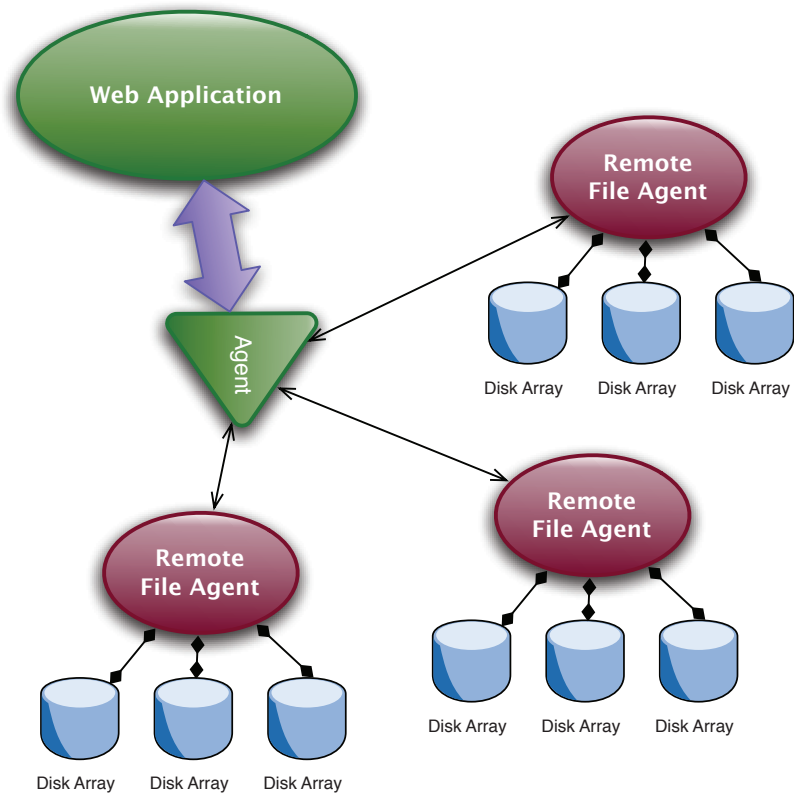
As disk IO based applications grow, they tend to run into limits of IO speed, filesystem size and stability. This trend is especially relevant in clustered web applications using a cluster aware filesystem such as GFS, NFS or Lustre FS. However it effects all applications which use disk to store assets at some point.

There are several low level tactics which can mitigate this issue in the short term, such as using high performance RAID arrays, and various tuning parameters in the Operating System hosting the application. Ultimately, these measures will fail to meet demand. At that point, filesystem sharding needs to be investigated.

This discussion is predicated upon the use of Linux as the host operating system. There are similar tools and concerns in the Windows environment, however they are different enough to warrant own treatment.

Tactics

There are two basic levels of filesystem sharding, basic file system sharding and application filesystem independence. Basic sharding will take an application to a certain point, and after that application level sharding needs to be implemented. Often, the authors have seen that basic sharding will allow an application to continue to grow while filesystem independence is implemented.



At a high level, basic sharding is accomplished by inserting an algorithm into the save and load modules, keying off of an unchangeable feature (such as username, file name, etc). You then share to different physical devices mounted under your main partition. Application level filesystem independence includes rewriting the save and load functions to abstract out the filesystem, allowing the application to use what ever file store it needs, where ever that filesystem resides. This is usually implemented as an agent. The application talks to an agent somewhere requesting assets. This allows you to abstract the file system to another machine or even another data center if needed.

Basic Sharding

Basic Sharding is accomplished by taking the shared file system and adding several mount points inside it. The only application change is a hashing algorithm to determine which file system is used. It can be as easy as inserting a snippet of code into the save and load methods.

As an aside, one of the limitations of many cluster file systems is performance degradation when there are too many files in one directory. In preparation for sharding the filesystem, it's often beneficial to use this hashing system to break up your filesystem before implementing complete filesystem sharding. One of the advantages is that this positions you for painless movement to the goal. Since Linux mounts filesystems as directories, it is a trivial task to move the existing directories, mount the new filesystems, and then move the data back. If you already have the hashing system in place and tested, this makes the move that much less stressful.

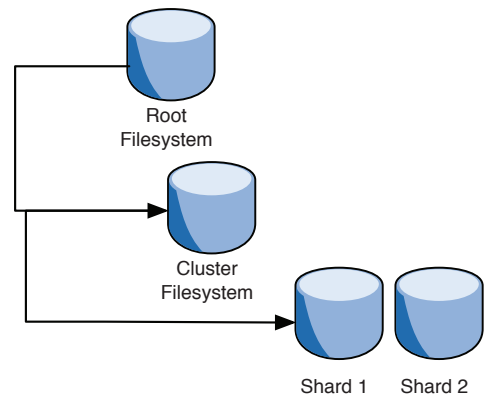
```
if user.name is odd
  ... use /data/shard_1
else
  ... use /data/shard_2
end
```

Logically the disk layout will be very simple,

```
/data
 /shard_1
 /shard_2
```

and the physical layout will be similar.

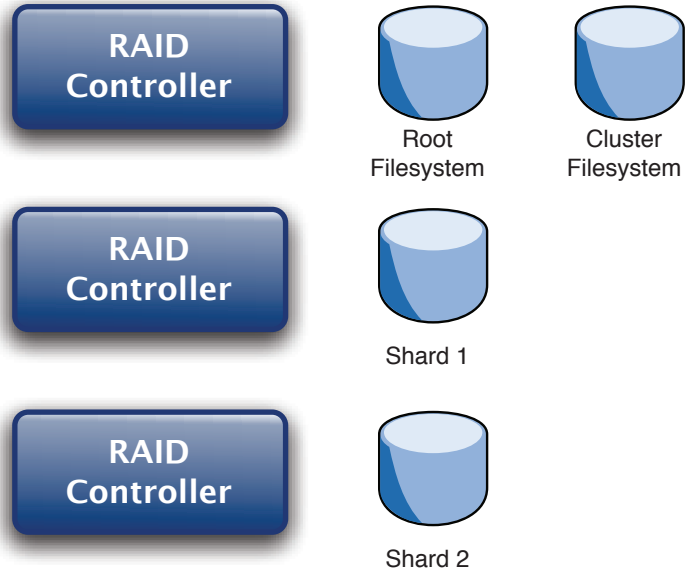
```
/dev/sda1 mounts to /
/dev/sdb1 mounts to /data
/dev/sdc1 mounts to /data/shard_1
/dev/sdd1 mounts to /data/shard_2
```



The advantages of this over a standard one filesystem approach are two fold. First, mount times and filesystem repair/check times are decreased, as the filesystems are each smaller. Secondly, filesystem maintenance can be performed on the live application with less undesirable side effects than simply taking the entire application off line. One can imagine a error rescue that displays a maintenance page when a users file share is not accessible, thereby allowing some users access to the application while other filesystems are being maintained.

From a physical layer, this also allows us to do some interesting things like splitting the filesystem to different RAID arrays. We can even split out the filesystems across different RAID controllers. We can also start to explore using expensive, fast RAID arrays only where you actually need them, and use slower, less expensive disk where warranted.

A more robust version of the sharding algorithm could look like



```
case user.name first character  
when = "abcd"  
  ... use shard_abcd  
when = "efgh"
```

etc...

It's logical disk layout would be

```
/data  
  /shard_abcd  
  /shard_efgh  
  /shard_ijkl  
  /shard_mnop  
  /shard_qrst  
  /shard_uvwx  
  /shard_yz
```

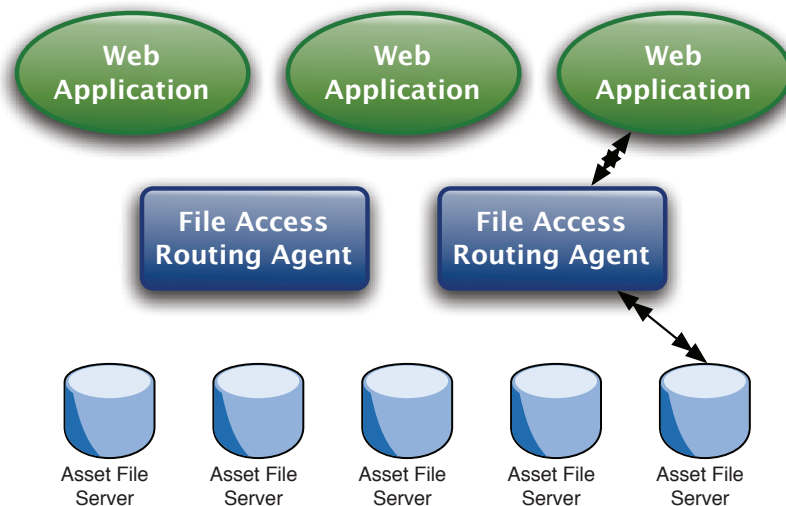
It's physical layout would be similar to the first example - each shard mount would be on it's own physical device.

Application level filesystem independence

Application level filesystem independence is an architectural choice, rather than a quick fix. When you reach the point where you are considering this as an option, you have probably already implemented basic sharding. A low level discussion of how to accomplish this is beyond this paper, however we can present some ideas that we have seen work in production sites.

The most basic implementation of this concept includes injecting logic into the save and load functions which call a module. This module then determines which file server it is going to talk to based on some algorithm. It then opens a socket to an application running on that server, and requests the filesystem operation via the remote application.

Conceptually the logical layout could look like this:

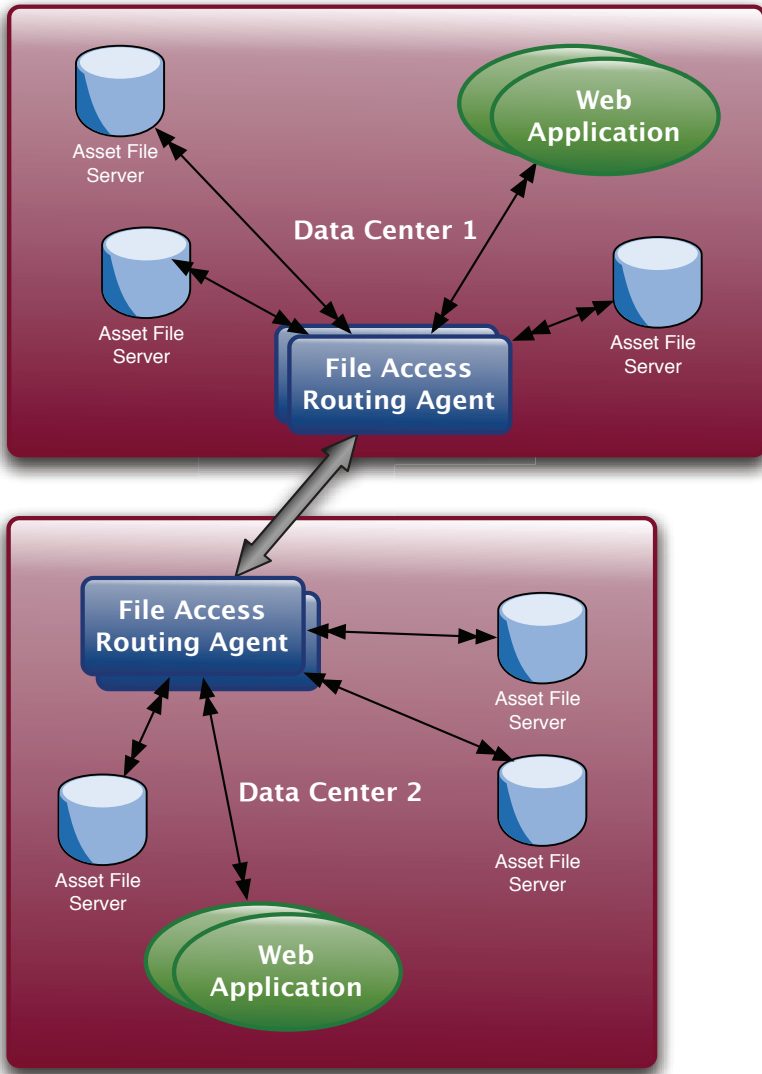


This level of abstraction allows us to do some interesting things. Advanced implementations could include ideas like geo-ip tagging, smart synchronization and advanced request forwarding.

For example, our file system agents can reside on different servers, in different data centers if we wish. We can build a peer to peer synchronization routine into the remote agents, allowing our application to spread load out amongst physical file servers and giving us easy failover and redundancy. In this case, each single asset could be written to several file servers

We would use geo-ip to redirect the request to a file server closest to the requester. Often we do not want to use this on the client machines, as their location might change. We see this keyed off of application server location.

Advanced request forwarding allows files not in the local store to be served up. The logic built into this idea should allow an agent to determine if it has a copy of the file in question. If it does not, it should check out a copy from the closest file server which does have a copy. This allows us to start trying and pre-cache items that the requester will need.



All of these are ideas which others have implemented with varying degrees of success.

Summary

Basic web frameworks can carry a website quite a ways as far as scaling goes. Using clustered filesystems can shoulder the load further. After exhausting the capabilities of vanilla deploys of each of these, you need to look into sharding to carry your application to the next level. Filesystem sharding is a good fit for high disk IO applications.