Petal: Distributed Virtual Disks

Garrett Kolpin
CS 739: Distributed Systems
University of Wisconsin, Madison

Lecture Date here
Spring 2006

1 Overview

Petal aims to provide a distributed, fault tolerant, and scalable data storage system. Clients view the Petal system as a group of virtual disks. Clients interface with Petal as they would any other block device, by specifying a virtual disk and an offset. This allows nearly any file system to use Petal for data storage. Doing this only requires a change to the current disk driver.

2 Problem Statement and Assumptions

It is the goal of the Petal designers to provide a globally accessible storage system that can tolerate single component failures, is easy to administer, has load balancing capabilities, and is possibly geographically distributed.

Petal provides an abstract block device in the form of a virtual disk. There is an important distinction that must be made between Petal and other distributed file systems such as NFS or AFS. Petal is a distributed storage system, not a distributed file system. That is, the interface provided by Petal is a block-level interface. This has some important implications. First, since Petal has a block-level interface, existing software that works on the block level can be used rather than having to rewrite new software that will only work with the storage system. For instance, existing file systems that rely on block-level interfaces will work just fine on top of Petal. *Also, having a block interface makes the system simpler, and the designers can focus more on providing mechanisms rather than policies. The policy decisions can then be left up to the designers of the file systems which will interface with Petal.*

Because of the distributed nature of Petal, some different assumptions about how the disks are accessed have to be made. Current file systems assume that they have sole access to the disk volume. However, in the case of Petal, multiple clients have access to the same disk volumes at the same time.

3 Design

3.1 Virtual to Physical Translation

The problem in this case is that clients will ask for blocks in the form ¡disk-identifier, offset¿. This must be translated into the form ¡server-identifier, disk-identifier, disk-offset¿. This translation occurs by first going to the global map and retrieving the redirect server. This will then respond with the physical location of the request which is essentially an enumeration of the servers over which the virtual disk is spread. From this, it is possible to find the actual physical location of the block.

The V. directory and the global map are both small tables, while the physical map has the most information stored in it. *The tables are stored this way probably because it is more reliable since there is less to remain consistent across servers. As a general rule of thumb, there is probably no need to share information you don't need to share.*

Because of the distributed nature of Petal, there are other assumptions that can be made with normal disk in-

terfaces that no longer apply here. First, it cannot be assumed that blocks with a close virtual address are necessarily physically close to one another on disk. In fact, two virtually 'close' blocks may actually be on two physically different machines. Furthermore, the way in which space gets allocated differs substantially from most file systems. Using a regular file system on a local disk, the amount of space required for the file system is allocated at file system creation time. However, Petal utilizes a lazy allocation scheme where space doesn't actually get allocated until a write operation. It is also possible to initially allocate a huge amount of space because it is unlikely that the space will be needed immediately, and it can be allocated even without the required amount of physical storage available. If the storage demands ever reach the limit of the physical disks, then more disks can simply be added. Lazy allocation also helps to avoid fragmentation problems as data is written in large contiguous blocks. *If an application absolutely needed knowledge about whether the required space actually exists, then in order to preallocate space, one could simply write data to the number of blocks needed.*

3.2 Snapshots

There are various goals for the snapshot system. First, we don't want to recreate all the data. Doing this would require space to store unnecessarily large amounts of data. Instead, we can essentially just keep track of the changes. By storing the 'deltas', we can create a snapshot that is space efficient. Also, after the snapshot process, there may now exist different data at the same offset. In order to keep track of the most recent data, an epoch number is incremented which will allow Petal to differentiate between different snapshot data. *One implication of the snapshot is that the lowest level of Petal needs to keep track of the epoch numbers. This is because the server will only receive a virtual disk and offset from the client. It's clear that the client will have no knowledge of the epoch numbers itself, which necessitates the server keeping track of this information.*

3.3 Reconfiguration

In Petal, one goal is to be able to scale the system easily and without much administrative overhead. Thus, it is relatively easy to add disks to machines and add or remove machines from the network. We'll first look at the process of adding a new disk to a machine. The essential goal here is to have some method to place data on the new disk. There are a couple of approaches that can be taken. Data could be taken off the existing disks forcefully and placed on the new disk in order to create balanced load across the disks. Another approach is to do nothing and rely on the system to place newly written data to the new disk. *There is not necessarily an assumption that all disks on a machine will have the same amount disk capacity. If disks have varying capacity on a single machine, then the machine local maps will need to be maintained in order to know which disks have what capacity.*

Now we'll deal with the case of adding a new machine to the network. This is handled through a three step process. First, a new global map needs to be created which affects all servers. Second, all the virtual disk directories need to be changed so that the new global map is referenced. Again, this affects all servers. Finally, data has to be redistributed across the newly added machine. This is especially important when striping over all disks. New writes go to the new global map, but reads go to the old one. However, the reader should check with the new global map to make sure it will not be reading stale data from the old location. The downside to always having readers check the new data is that there is now twice as much network traffic during reads, and the result is that whole reconfigurations could take hours to complete. Therefore, a policy decision was made to minimize the performance pains. The problem is solved by moving data incrementally. The virtual disks address range is divided into three types: old, new, and fenced. Reads bound for the old or new regions go to the old and new global maps. Reads bound for the fenced region are redirected to the new or old maps as appropriate. The only downside is that this could tie up a disk that has a hot spot on it. Thus, the portions of the disk that are to be moved are selected randomly so that hot spot data is not moved all at once.

3.4 Data access and recovery

The basic method to store data is to spread the data to more than one node. This is done for load balancing and for better operation under failure. Thus, when using chained declustering both a primary and secondary copy

are kept for blocks. Reads can be serviced from either the primary or secondary copies. However, write requests must always be serviced by the primary server first. Since blocks are locked upon writing, we need this ordering to avoid deadlocks. Without this order, two concurrent writers could obtain exclusive locks on one of the primary or secondary copies of the blocks, and they will reach deadlock at this point because neither will release the lock they currently hold. Once the server has determined that it is the primary server for a block of data, it will send out a write request to the secondary copy. Busy bits are also set when writing to help during recovery from a failure. If a failure were to occur, the busy bits would be used to determine which blocks were being written at the time the failure occurred. *At this point, some decision will have to be made about how to bring both copies of the data to a consistent state. Upon recovery, if the primary and secondary copies of a block are inconsistent and only half of the block was written, then the only way to bring them both back to a reasonable state is to use a log. In the absence of a log, one or the other copies has to be chosen to be the 'most consistent' copy, and the other will bring itself to consistency with it.*

4 Evaluation

Table 1 shows a comparison of Petal and a local disk. From the table it is apparent that a local disk is faster for blocks of all sizes. Petal performs reasonably for small reads and writes, but performance degrades for larger block sizes. Writes are slower than reads because writes must go to neighboring servers as well. Clients don't issue writes in parallel; instead the primary sends the data to the secondary which results in higher latency. Waiting for the log to be properly updated is also a source of added latency during write operations.

In Figure 7 we see a glimpse of the scalability of Petal. *However, they only show Petal scaling with four servers. It would be nice to see how Petal scales in an environment with many more servers.*

The authors give no indication about the performance of reconfiguration, recovery, etc. It would have been nice to see performance numbers for reconfiguration because of the time spent describing the reconfiguration mechanisms in the design section.

5 Conclusion

To conclude, Petal provides an easy to manage distributed storage system that has the ability to be scaled easily. It provides a very simple block interface and can give clients the view of multiple virtual disks. The replication and data distribution is completely invisible to the user so that there is no need to run specialized client software besides the Petal device driver.