Scaling Perforce Servers and Storage For Film Assets

written by Mike Sundy and David Baraff
additional research by Shaun Brown
Pixar, March 2011

Overview

In the past four years, Pixar went from one Perforce server to over ninety. We had to address several infrastructure scaling issues to meet this increased demand, from growth in the size of the environment to storage and server architecture. This document will detail some of the ways Pixar uses Perforce and how we manage storage and servers.

Studio Growth

1 code server grew to 90+ servers used for all types of assets: code, art, textures, models, flat images, videos, etc.

Pixar Perforce environment as of March 2011:
~1000 users
70 GB db.have on largest server
12 million operations per day
30+ VMWare servers
40 million changelists (10 million if you exclude automated testing depot)
currently on 2009.1 but in process of migrating to 2010.1

Types of Data We Store in Perforce

art - reference and concept art - inspirational art for film
dept - department-specific files. E.g. Creative Resources has special blessed marketing images.
studio - company-wide reference libraries - e.g. animation reference, config files, flickr-type of company photo site
tech - show-specific data, e.g. models, textures. pipeline for the shows.
tools - code for our central tools team, software projects
exotics - patent data, casting audio, data for live action shorts, story gags, theme park concepts, intern art show, post production, etc.

Storage

115 million files in Perforce
19 TB of storage

We use several techniques to manage our storage growth:

- Use the +S filetype for the majority of generated data. This automatically purges old data. We never have to go back more than a couple of revs on generated data. If we did, we'd regenerate it. This saved 40% of the space for Toy Story 3 (1.2 TB).
- Work with the teams to migrate data out of Perforce which doesn't need to be versioned. On one show, they were storing nearly 2 TB of .tid scene file data which was fine on a fileshare.
- De-dupe files - see explanation below. So far, de-duping has saved us from creating 1 million files and saved nearly 1 TB of disk space.

At Pixar, all of our asset perforce servers use the "full file format" to store files (i.e. not the RCS-delta scheme). We have noticed three common patterns:

1. Checking in a file with the exact same contents as the last file.
2. Checking in a file, modifying it, and then immediately reverting back to the previous version and checking it in.
3. Deleting a file, then re-adding the deleted file back in at the same location.

Both of these result in duplicate copies of the same file in the repository backend. While (1) can in theory be avoided by certain p4 settings (the "submit-unchanged" options settings), there are reasons why relying on this to eliminate duplicates is less than optimal at Pixar.

Our shared trigger setup uses the checksum data to detect the three cases above; note that detection involves simply comparing the newly added repository file with the repository file either one or two versions prior. (One could, in theory, be more aggressive, and check even more past versions, if necessary.) If the checksums match, the newly added file, which is now known to be a duplicate, is (atomically) replaced with a hardlink to the original copy of the file. This has proven simple, reliable, and effective. Note that the probability of two files differing yet having identical checksums is vanishingly small; as a result, the cost of detecting duplicates in this fashion is near zero.

Servers

All of our Perforce servers are run on a VMWare ESX cluster and we use RHEL 5 for the OS. Virtualization made it easy to scale from 1 Perforce instance to 90.

For details on our System Architecture, see the 2009 whitepaper: http://www.perforce.com/perforce/conferences/us/2009/Presentations/Harrison_Sundy-PerforceatPixar_Templar-paper.pdf

Benefits of Virtualization:

-Quick to spin up new servers.
-Stable and fault tolerant.
-Easy to administer remotely.
-Doesn't take up valuable datacenter space, cooling, power, etc.

Gotchas with Virtualization:
-We did have a severe performance problem with one of our largest servers when the datastore space grew to over 90%. Once we moved it to a less used datastore, the performance went back to normal.
-It takes some jockeying to ensure load stays balanced across multiple nodes. We weren't comfortable with letting VMWare automatically migrate VM's for us, but it didn't save our settings of which VM lived where when we would restart them. So that requires periodic manual administration to make sure that our highest loaded servers are distributed across different nodes.

Speed of Virtual Servers:

We ran several different benchmarks comparing the speed of virtual servers to physical servers. We used Perforce's Benchmark Results Database tool: http://kb.perforce.com/brdb. The results were good: the virtualized servers were 95% as fast as the physical servers for the branchsubmit benchmark and 85% as fast for the browse benchmark. However, we tend to be command-line or API-driven Perforce clients, so the browse functions aren't as important to us as a studio which uses P4V as the primary client.

Quick Server Setup in 1 Hour

This assumes that the storage is already in place. Here's the process to build a new Perforce server from scratch:

1. Clone a VM (Virtual Machine) ~30 minutes for a 70 GB VM.
2. Prep the VM. E.g. set up mounts, update LDAP records, etc. - 15 minutes
3. Run David Baraff's squire script to build a new p4 instance - 8 seconds.
4. Validate and test new Perforce instance - 15 minutes.

Administering Multiple Servers via "superp4"

Suppose one day you realized that your clever idea of automatically preventing a "host" specification from being in anyone's client had a small bug - you wrote the following in the triggers table:

```
    noHostInClient form-out client /usr/local/bin/removeHostFromClient %formfile
%
```

where "removeHostFromClient" simply takes the "host" line out of a client before

delivering it to the user. However, you've just realized that the above slows down every single "p4 client -o" command, which is done a lot in your world. In retrospect, you see that it would be cheaper to strip off the "host" information whenever someone attempts to modify the client, which is far less frequent.

In short, you wish you had written "form-out" in the above line, and you have now to fix it in seventy-two different depots. Right now.

At Pixar, we would simply run the command:

```
$ superp4 -table triggers -script fix-nohost-trigger.py
```

where fix-nohost-trigger.py contains this code:

```
def modify(data, depot):
    return [l.replace("noHostInClient form-out", "noHostInClient form-in") for l in
data]
```

This would modify all the p4 depots at Pixar in the appropriate fashion. Superp4 is a simple script that allows a user to modify any perforce "table" (i.e. triggers, groups, users, etc.) across a specific set of depots, all depots, or depots matching certain criteria. The ability to operate on many depots utilizes an always up-to-date database of current depots, their ports and triggers, and keywords that describe the purpose of the depot.

The command also contains a "restore" feature that automatically snapshots the table in question across all modified depots, in a safe directory, allowing the administrator to quickly roll-back changes across large numbers of depots (a necessity when you plan to modify dozens of critical p4 areas in a hurry).

The superp4 script has a very simple implementation. Since our database has a record of the names of all p4 depots, along with a category tag for each of them, users of the superp4 script can specify depots to be modified by category, or as an explicit list of depots with wildcard matching. The database also has the ports for each depot, so it is simple to simply loop over all the affected depots, pull the relevant table from each one, and run the data through the user-supplied python code to modify it.

Once this is done, the user has a change to see a graphical diff of the contents, inspect the (potentially) changed files on disk, etc. Finally, the user can allow the modified data to be inserted into each p4 depot. To aid reliability, the unmodified contents for each depot are stored in a restore directory, and the superp4 command can be run directly on the restore directory to quickly fix mistakes. Since the script itself is python, it is trivial to execute a user’s supplied python fragment and we use python itself as the specification for making changes to the depots.

Finally, we control the behaviors of the triggers themselves for each depot using configuration files that are python dictionaries. The config files reside at well known locations, for example:

//<DEPOT>/admin/p4/unit-config

for each depot. The superp4 script can also be used to operate on a named configuration file, and the python fragment in this case is used to modify the contents of the dictionary in the configuration file.

Scaling Challenges

//spec/client filled up

One surprise we had is that our //spec/client/... area kept filling up. It turns out our tools group runs a bunch of automated tests to create one-time clients and then deletes them. This filled up the Linux-based filesystem and users would start getting a "RCS token too big!" error when making any client updates. We had to write a script to auto-purge this spec client data, sometimes multiple times per day. There's hope with the new spec.hashbuckets tunable, but that seems to just make it take more files to fill up the filesystem. We think we're stuck with auto-purging for now.

user-written triggers are often sub-optimal

We had one user write a trigger which ran out of his /home directory. Everyone was curious why submits started failing - we tracked it down to the server losing its mount to the user's home directory. This was only the tip of the iceberg. It was also difficult to deduce which trigger was causing a hang as they all show up as 'dmCommitSubmit'. We had users use the 'Program' variable to distinguish their trigger from others and that made troubleshooting easier with p4 monitor. To reduce our dependencies, we banned user-written triggers and implemented a queuing system for asynchronous processing, which my colleague Mark Harrison will talk about at the 2011 conference.

reverting pending changelists still consumed server space

There was a bug in the version of Perforce we were using which meant even files reverted would still upload files to the server's repository but not clean them up when they were reverted. This meant we had orphaned files consuming large amounts of disk space.

monitoring became much more difficult

Going from 1 to 90 server presented a lot more monitoring to stay on top of. My colleague Mark Harrison wrote a web-based monitoring dashboard called templaRX which tells us in an instant if any servers are having an issue serving p4 operations.

cap operations from your renderfarm

Our initial design of the templar system optimized for renderfarm access to the NFS link trees, and the renderfarm was not supposed to need to talk to Perforce.

Of course, the renderfarm quickly needed to talk to Perforce and would often overwhelm our Perforce server. Thousands of CPUs vs. 2 isn't a fair fight. So we implemented a cap on the number of concurrent p4 operations which could be run on the renderfarm.

beware of automated tests

One depot with 200k files had 4.2 million opened files from automated tests. Any operation that took a lock on db.working or db.locks was hanging for several minutes. We spoke with the appropriate team and they rewrote their tests to clean up opened files.

Summary

In summary, Perforce scales well for large numbers of binary assets if managed properly. It can be used for a wide variety of data and clients. Using virtualization provides a cost-effective and fast turnaround to new depot requests. Using the +S filetype and de-duping binary data helps keep space consumption to a reasonable level.