Life on the Edge: Monitoring and Running A Very Large Perforce Installation

Dan Bloch, Google

March 24, 2007

Abstract

Although Perforce does a remarkable job of scaling, Google's main server is at the limits of what Perforce can do, and eternal vigilance is the price of being able to provide acceptable service to our users. This talk discusses the monitoring tools and techniques used at Google and the actions that we (or the tools) take in response to the information provided. Some new Perforce features, notably the performance tracking information introduced in Perforce 2005.2, are discussed in detail.

A key result which applies to smaller sites is that server load is often caused by a few users or a few commands, and the ability to identify these allows improvement to performance without the purchase of additional hardware or software.

Introduction

Perforce scales astonishingly well, but many if not most sites run into performance issues sooner or later. The two standard solutions to performance problems are buying faster hardware or splitting the depot.

Hardware upgrades are indeed often the right solution, but they involve expense, downtime, and uncertainty as to whether the new hardware will really solve the problem. Google is already at the limits of what can be done through hardware improvements, so most of our improvements have to come from the way we use our server.

Splitting a depot is a great deal of work, involving changes to supporting tools and procedures (e.g., backups, authentication, builds), user education, and some amount of ongoing increased workload for both users and administrators forever. We have embarked on this course as well (we're now running a half dozen smaller servers), but most sites will choose to avoid splitting their depot if at all possible.

Google has a very large, very active, fast growing Perforce server. As far back as two years ago, we started encountering performance problems, and investigation into server utilization found that significant load was due to unnecessary commands which could be eliminated. Refinement of that investigation has continued to this day. Over the last year and a half we have improved our server's responsiveness while more than doubling the number of users, the size of the database, and the volume of transactions per day. This is the result of work on many fronts including incremental hardware and software upgrades, aggressive database cleanup, and moving some users to satellite servers, but the most significant factor has been monitoring server use at a

detailed level and taking ongoing action to prevent unacceptable commands from running. At this point most routine enforcement is automated.

This paper discusses the monitoring provided by Perforce and additional tools built at Google on top of Perforce's native monitoring (available in the Perforce public depot). The discussion applies to Perforce installations of all sizes, although since there is an investment in the administrator's time, the processes may not be of interest to sites unless they're experiencing performance problems. Even at Google we only run most of these tools on our main server, but they're available on smaller servers in case of need. Server size is of course relative to machine size--a small installation running on a small machine can benefit from monitoring as much as large installation running on a fast machine.

This paper describes Perforce 2006.1 and some Perforce 2006.2 features, running on Linux. Some results are not applicable to servers running on Windows. In particular, on Windows it's not possible to kill individual Perforce commands on the server using OS commands.

Note that parts of this paper discuss Perforce internals which, though they have been stable for many releases, are undocumented and subject to change.

Perforce at Google

Google's main Perforce server is very large, very active and requires 24x7 uptime. It has more than 4000 active users, 200 GB of metadata, and performs more than 2 million commands a day. In addition to this main server we run a number of smaller servers, read-only replicas, and proxies worldwide. We use an extensive layer of tools built on top of Perforce, and make heavy use of triggers. Perforce at Google is used by IT and other departments in addition to Engineering. Our operating platform consists of an HP DL585 with 4 dual-core Opterons and 128 GB of RAM running Red Hat 2.6 Linux and Perforce 2006.1. The depot is on a NetApp filer, and the database and journal are on RAID-10 local disk using the ext3 file system.

We do a number of things right: we set MaxScanrows and MaxResults to limit large queries; we aggressively clean up inactive clients, labels, and users; we make use of proxies and read-only replicas; and we upgrade to new versions of Perforce within three or four months of their release to take advantage of performance enhancements.

Unfortunately, we also do a number of things which are less than ideal from a performance standpoint: we have a very complex protect table, our users run many automated scripts, we have many large (30,000+ file) clients and branches, and we make heavy use of temporary clients. We are working to change some of this, but much of it arises from Google's culture: within limits, any engineer is free to use Perforce in any way he or she sees fit.

Perforce Architecture

In order to understand performance, it's necessary to understand something about the Perforce architecture.

The Perforce Process Model

There is a single Perforce server, **p4d**, which runs on the server machine. Each command run by a user results in a child p4d process on this server machine. Processes can be identified by their **pid** (process ID), which is shown or used by various OS-level tools (ps, top, kill) and shown by Perforce in **`p4 monitor show`** output and in the error log. An important point is that any command, from any client (e.g., from P4V, P4SQL, or the P4API), appears to the server as if it came from the **`p4`** command line. That is, the set of possible commands is defined entirely by the Perforce Command Reference.

The Perforce Database and Locking

The Perforce server stores all its information except for the content of users' files in a database. The database is a modified version of BerkeleyDB. Physically, this consists of about thirty-five database files in the **P4ROOT** directory on the server machine, with names like “db.have”, “db.labels”, and “db.counters”. This is the data which is saved in checkpoints. Details about the database schema can be found at http://www.perforce.com/perforce/doc.062/schema/index.html.

Discussion of the database files themselves is outside the scope of this paper, but their existence is tremendously significant from a performance standpoint, because the p4d processes take locks on them while executing commands.

Locks are an OS-level feature which prevents multiple processes from modifying a file at the same time, or from getting inconsistent views of a file. Two types of locks are used: exclusive ("write") locks and shared ("read") locks. When a process has a write lock on a file, no other process can either read or write to the file. When a process has a read lock on a file, other processes can also read it (they will take locks on the same file), but no process can write to it.

From an administrator's or a user's point of view, the consequences of this are immensely significant. When a long-running command is holding a lock, all other access to that database file (or all write access, if it's a read lock) is blocked. Since some files are used by almost all commands, this has the effect of stopping almost all activity on the server.

This is the source of a widespread misconception that the Perforce server is single-threaded. The server is very much multi-threaded, but database locks can block all the threads, making it effectively single-threaded for the duration that locks are held. The typical examples of this are big submits or integrates, which on an installation of our size can block the server for many minutes.

Note that the operative phrase is "for the duration that locks are held". If locks were held for the entire time a command was run, Perforce wouldn't work at all. Perforce allows as much concurrency as possible by holding locks for the shortest possible time. In particular, locks are never held while files from the depot are being accessed (p4 sync, p4 print, p4 submit) or while triggers are being run. This means that slow depot access or poorly written triggers will never block database access to other commands. This also means that syncs continue to run while the server appears to be hanging for other commands.

A final significant point about lock time is that from a hardware point of view, this is a disk I/O bottleneck on the volume holding the Perforce database. The locks are being held for the amount of time it takes the server to do seeks, reads, or writes on that disk. Anything you can do to speed up disk access will improve this situation.

Perforce has recognized the importance of database locking in their most recent releases. Perforce 2005.2 added the time spent holding locks to the diagnostic information in the error log, and Perforce 2006.2 added a settable **MaxLockTime** resource limit. Both of these are discussed in more detail below.

Additional detail on database locking can be found in my presentation from the 2006 Perforce European User Conference, **Performance and Database Locking at Large Perforce Sites**.

Monitoring Tools Provided By Perforce

`p4 monitor show`

The **`p4 monitor show`** command lists commands currently running. For some reason this functionality is disabled by default. If when you try to run it you see, "Monitor not currently enabled", you'll need to enable it by running "`p4 counter -f monitor 1`", and restarting the server. Output looks like this:

```
% p4 monitor show
```

```
 3114 R build      00:00:09 changes
 4394 R alexh      00:00:01 have
 4419 R dbloch     00:00:00 monitor
 ...
```

The first number is the process ID. Users with p4 admin access can use the `-l` flag, which provides the command arguments as well:

```
% p4 monitor show -l
 3114 R build      00:00:09 changes -m 1 -s submitted ...
 4394 R alexh      00:00:01 have /home/alexh/google/search/trace.h
 4419 R dbloch     00:00:00 monitor show -l
 ...
```

or the `-e` flag, which includes the IP address, client, and client program (p4, p4v, etc.) as well:

```
% p4 monitor show -e
 3114 p4/v99999 172.27.75.60  R build      unittest_c 00:00:09 changes
 4394 p4/v99999 172.24.8.91   R alexh      alexh-src  00:00:01 have
 4419 p4/v99999 172.25.203.43 R dbloch     dbloch     00:00:00 monitor
 ...
```

As of Perforce 2006.1 you may, if you choose, set the monitor counter to 2 instead of 1 to include idle processes in the output. Idle processes exist if an application uses the p4 API, and holds a connection open to the server so it can run multiple commands. P4V and P4WIN both do this. Idle processes use no significant resources on the server, and typically aren't of interest.

`p4 monitor show` is the simplest tool at your disposal. It may be all that's needed at small sites, but when you have more than a hundred commands running at any one time, visual inspection of the output isn't all that helpful.

One drawback of p4 monitor is that the p4 monitor show command itself can hang when the server is hanging. A workaround for this is described in my previous presentation.

The Error Log

The error log is the primary tool available for server analysis of all kinds. It's specified with the `-L log` option to the p4d command. This option should always be specified, since otherwise error messages go to stderr and will be lost. Additionally I highly recommend setting the debug mode to 2 or 3 (`-v server=3`), which causes the server to log start and stop records for all Perforce commands, turning the error log into an event log.

This is invaluable for data mining and after-the-fact performance analysis, and is also very useful for answering user questions like "who deleted my client?" It does make the error log much larger than it would be otherwise (Google's sometimes reaches a gigabyte in a day) so it's important to rotate it and delete or archive the old logs.

Log output looks like this:

```
Perforce server info:
        2006/01/20 00:10:04 pid 23520 nora@nora-maps 172.30.0.103 [p4]
          'user -add -c 2002444 //depot/google/search/maps.cc'
        2006/01/20 00:10:16 pid 23520 completed 12.512s 20+100us 0+0io
          0+0net 0k 23pf
```

(line breaks added), where data includes start and stop time, pid, user, client, IP address, client program, command, and (on the "completed" line, elapsed time, CPU time, io and network operations, memory used, and page faults.

Some commands, such as sync, have additional “compute end” log entries. Note that only the CPU time in the first “compute end” line is valid for these. `p4 submit` commands have additional `dm-CommitSubmit` and `dm-SubmitChange` entries for different phases of the submit.

If a process is killed, there will be a “Process 10927 exited on a signal 15!” line instead of a “completed” line.

The simplest use of the error log is to grep for commands of interest, e.g., by a specific user, but it's possible to do arbitrarily complex processing on it.

Perforce 2005.2 added "performance tracking" output: when a command exceeds a certain threshold of resource use, additional entries are printed to the error log. The thresholds are dependent on the number of licensed users, e.g., for a server of our size information will be printed if a command takes more than a minute of elapsed time, 10 seconds of CPU time, etc. These entries are always printed, irrespective of the `-v server` value describe above. There is a `-v track` option which can turn off performance tracking or change the thresholds, but in our experience it has never been necessary to modify this.

A fairly simple example of this output is

```
Perforce server info:
        2007/03/22 23:13:10 pid 26154 config@config-mp 172.26.64.72
         [p4/v99999] 'user-resolve -as'
--- lapse 3180s
--- usage 90656+13715us 0+0io 0+0net 0k 148pf
--- db.resolve
---   pages in+out+cached 11240+11878+64
---   locks wait+held read/write 0ms+811ms/34441ms+31ms
```

Interpretation of this format is left as an exercise to the reader (and to a script, described below). These entries, and arguably the whole error log, are intended for use by Perforce product support, but are certainly amenable to use by local administrators. The format of the entries is undocumented and subject to change, but has been stable for the two releases since it was introduced. For our purposes, by far the most interesting information is the time spent holding and waiting for locks.

The performance tracking information also includes messages "killed by MaxResults", "killed by MaxScanRows", etc., as appropriate.

Other Logs

The following are not part of a monitoring strategy, but are mentioned for completeness.

Journal - Perforce logs all database modifications to a journal file for the purpose of recovering the database in case of emergency. Although diagnostic information isn't the goal of the journal, it is possible to use it for this purpose. This is quite difficult, though, and needed only in extraordinary circumstances. Use of the journal file for diagnostic purposes is outside the scope of this paper.

Audit Log - Perforce 2006.1 added an audit log, specified with an optional `-A auditlog` option to p4d, which logs every file access by every user. This log (not used at Google) is very large, and is typically not useful for diagnostic purposes. Audit logging is disabled by default.

Experience at Google

Initial Work

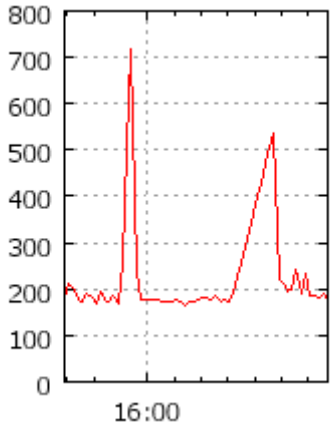
A year and a half to two years ago, performance became unacceptable to our users, with the server regularly blocked for half a minute or more out of every five minutes. A manual investigation followed, using tools such as "p4 monitor show" and the OS utility "top", and the problem was tracked down to a small number of frequently run automated jobs and fixed.

Server Spikes and `locks.pl`

This solved the urgent issue, but we still had problems. Several times a day, the server would stop responding to commands. To the user, it appears that the server was hanging. To the

administrator, this was visible as a spike in the number of p4d processes. Spikes could last for one minute, five minutes, ten minutes, or longer. At that time, the cause of these was a mystery. They turned out to be caused by long-running p4 commands holding locks.

Two Server Spikes



The diagram here shows two server spikes. Time is on the horizontal axis, and Perforce process count is on the vertical axis. The first spike is caused by a `p4 submit` command. Since it's blocking all other commands, the number of processes waiting on it goes up very steeply. The second is caused by a `p4 integrate` so only write commands are waiting on it. It lasts longer but never blocks as many processes.

The tool which came out of this was the **locks.pl** script, which uses "p4 monitor show" and the **`lsof`** command, available in most versions of Unix, to report on which specific Perforce commands are holding locks while a spike is in progress. At Google this script is set to run every half minute when the process count goes above 250 processes, and to send mail to Perforce administrators for possible action.

Based on this information, we are able to kill the offending command if appropriate, find the root causes of commands, and if possible change scripts or work procedures so they don't happen again. The investigation into this was the subject of my presentation last year.

Automation

With increased understanding of the offending commands, we expanded locks.pl's monitoring role to also kill known offending commands. A representative sample of the commands the script now kills is:

- `p4 integrated` (on a branch)
- `p4 changes -i` (on a branch)
- `p4 filelog ...` (I don't know why people do this, but they do)
- `p4 integrate //depot/path/... @nnnnnn` (note accidental space before @)
- `p4 opened -a` (on a path with a wildcard)
- `p4 fstat -P -Ol //depot/...pattern` (generated by p4v)
- `p4 files //depot/path/.../path2/...` (multiple wildcards)
- and finally, all integrates which take longer than 10 minutes

A time threshold is associated with each of the commands, usually a minute but sometimes two or three if the commands have legitimate uses (perhaps in small branches). The script occasionally kills an innocent bystander, that is, a process that is not the cause of a server hang, but this is infrequent. This capability of the script is the end result of about a year of tuning.

When the script kills a command it notifies p4 administrators by e-mail, and we usually send mail to the user explaining that his command was killed and why, and offering to help him with alternatives.

Note that this script targets read commands only, uses the **`kill`** system call with a SIGTERM signal, which allows Perforce to do cleanup on the server side, and as an added safeguard checks that the process is not holding any write locks before killing it. In the general case, killing Perforce processes by means other than "p4 monitor terminate" is done at your own risk. It is possible to corrupt the Perforce database if you kill processes while they are writing to the database.

Finally, note that while this capability of the script has been invaluable over the last year, it has been made less essential by the introduction of the MaxLockTime limit in Perforce 2006.2 (described below).

Further Automation - Terminating Commands Proactively

Killing commands this way, or by MaxLockTime, is reactive in that by the time the command is killed it has already blocked the server for half a minute or more. It turns out that we are able to identify a set of commands which should never be run.

These are typically run by P4V or P4WIN, so we have little direct control over them. The "p4 fstat -P -Ol //depot/...pattern" command in the list above is an example of one such. This is generated by default when a user does a "find file" search. Our depot is so large that this command will never succeed, but it will block the server until it fails with "MaxScanRows exceeded" or we kill it.

Our solution is to run a script which tails the log, searches for known bad commands, and kills them as soon as they're run. For that particular command, we send the user a suggestion for an alternate command. For most, no action need be taken.

Other commands killed by this script are "p4 opened -a" called from P4V or P4WIN, and "p4 fstat" or "p4 dirs" of //specs/client/*. This script kills about twenty commands a day, so it's giving us back ten or fifteen minutes a day of time during which the server would otherwise be unavailable.

Another use we've found for this script is that every once in while a rogue automated job starts doing some kind of expensive query on a regular basis. By adding a regular expression for that command to the list of commands to be killed, we can shut it down immediately without waiting to contact the user, or disabling his account, which would block any other work that he's doing.

The Perforce 2005.2 Error Log and `lockers.pl`

As mentioned above, the performance tracking information added to the error log in Perforce 2005.2 adds locking information which can be invaluable for tracking down performance bottlenecks. Unfortunately, it isn't very easy for human beings to read. Consider the example below, of a sync which locks database tables for a little over a minute:

```
Perforce server info:
        2007/03/25 03:12:41 pid 29576 build@build 172.24.6.10
          [updatebuild.py/v57] 'user-sync //...'
--- lapse 72.6s
--- usage 67412+5241us 0+0io 0+0net 0k 0pf
--- db.have
---   locks read/write 1/1 rows get+pos+scan put+del 0+1+1459792 11+0
---   locks wait+held read/write 0ms+70827ms/0ms+1ms
--- db.resolve
---   locks wait+held read/write 0ms+70826ms/0ms+0ms
--- db.rev
---   pages in+out+cached 259914+0+64
---   locks read/write 1/0 rows get+pos+scan put+del 0+10+7960042 0+0
---   locks wait+held read/write 0ms+57724ms/0ms+0ms
```

Consider further that it's embedded in an error log with millions of lines. I've written a script, `lockers.pl`, availailable in the Perforce Public depot in **`//guest/dan_bloch`**, which parses the error log and lists all commands with lock time over a specified threshold (default is fifteen seconds), which will extract the above and print it as:

```
2007/03/25 03:12:41 pid 29576 build@build 172.24.6.10
        [updatebuild.py/v57] 'user-sync //...'
    db.have: read: 70.827
    db.resolve: read: 70.826
    db.rev: read: 57.724
```

This script is related to `locks.pl` in that they both report on commands holding locks, but while `locks.pl` is used on the live server in real time, `lockers.pl` is for post-analysis. It can be used either to investigate a performance problem immediately after it has happened, or to look at logs on a daily or weekly basis for trends and significant users of lock resources.

Summary of Monitoring at Google

p4d Process Count

In our experience the p4d process count is the one best metric for system performance. This is a single number which is a measure of system health. Every p4d process represents a user waiting on a command (or an idle process--there's a baseline which can be ignored). We have three different tools monitoring this:

- The dashboard, shown in the diagram above, is posted to a web page and is visible to administrators and all users at all times.
- Also as described above, a script runs `locks.pl` and sends mail whenever the p4d count goes above a given threshold.
- Finally, a third script pages the on-call administrator when the count goes over a higher threshold and stays there for several minutes.

Related

The following two scripts, described in this paper, are arguably not monitoring scripts per se, but are closely related to monitoring work:

- The script that tails the error log and kills known bad commands.
- `lockers.pl`, which is run manually to look for commands causing problems on the server.

Additional Monitoring

The following are typically not referred to on a daily basis:

- We have an OS performance dashboard which shows variables such as system load, disk reads and writes, and network traffic in real time.
- Our first monitoring tool was a script which runs "p4 edit" every five seconds and logs the elapsed time. While this is an extremely accurate record of our users' experience, it hasn't been a real success as we haven't found a good way of visualizing the huge amount of data it generates.
- We have a very complicated script which looks for changelists with more than 10,000 files, and reports submit time, measured in seconds holding a lock on db.working, per thousand files. This turns out to be an interesting metric for overall disk performance on the database volume.
- We run a nightly script which parses the error log and reports actual errors. In practice, it turns out that these are rarely interesting.
- Another nightly script generates a report on the most active users each day.

MaxLockTime

Google is running Perforce 2006.1 as of this writing, but Perforce 2006.2 introduces an extremely interesting new resource limit, **MaxLockTime**. This works similarly to MaxScanRows and MaxResults. It specifies the a time threshold after which a command holding locks will be terminated. It's applied to users on a per-group basis. Note that submits are never terminated.

This could be used instead of `locks.pl` (with less granularity of control) as a way of killing long running commands holding locks. Since `locks.pl` involved a lot of effort in tuning, non-Google sites would be well advised to investigate MaxLockTime first. Before activating it, `lockers.pl` can be used to find out which commands would be effected by different values for MaxLockTime.

Conclusion

Performance issues are different at each site, and can be due to a number of factors, but at Google and probably at many large sites the most pervasive is database lock contention. Lock contention is often caused by a relatively small number of commands, and overall performance can be greatly improved if you're able to identify these commands and eliminate or reduce them. The performance tracking information introduced in Perforce 2005.2 provides enough information to identify these commands, but the form is hard for humans to use. A script, `lockers.pl`, is provided which makes it easy to extract this information.

References

Dan Bloch, Performance and Database Locking at Large Perforce Sites,
http://www.perforce.com/perforce/conferences/eu/2006/presentations/Google.pdf