Perforce Administration

Optimization, Scalability, Availability and Reliability

Michael Mirman
Perforce Administrator
MathWorks, Inc.
Abstract

Perforce Software Configuration Management system is designed to be scalable. Necessary infrastructure includes such maintenance tasks as regular checkpointing, rolling journals, testing disaster recovery plans, and load balancing. As our Perforce user base grows, we are striving to minimize downtime and provide reasonable response time, while running more and more maintenance tasks. We hope our experience will help other Perforce administrators plan their growing systems.

Perforce at MathWorks

Company’s source code is company’s jewels. Perforce is the system that protects it. As of this writing, MathWorks has one main production server, about 500 users, and several million archive files. We deploy almost all possible triggers, several daemons, and partially mirror our own bug database into Perforce. In addition to P4, P4V, P4Perl, P4Java, Emacs, and P4Eclipse interfaces, we support a modified version of P4DB\(^1\).

\(^1\) There are several versions of P4DB publicly available. Our version is close to the version at [http://www.releng.com/downloads3/index.html](http://www.releng.com/downloads3/index.html)
**Architecture Overview**

We use multiple proxies, p4broker, and multiple replicas for different purposes. We’ll describe the details later.

![Architecture Diagram]

**Using proxies**

We install proxies at different locations, and we use the anycast routing methodology\(^2\) to route senders to the topologically nearest node using the same destination.

---

This means that all our Perforce users use the same default P4PORT=perforce:1666. However, depending on which computer user issues perforce commands on, the requests are routed to a specific proxy.

For example, we currently have three proxies at different geographic locations. When user issues a request to perforce:1666, the proxy is chosen based on the location of the user.

In addition to providing cache for syncing files, using anycast with proxies gives us the ability to fail over. Anycast routing is supported by a few system processes, started at the boot time. If those processes go down, another host would automatically take over routing within seconds.

We also have a watchdog cron that detects if a proxy process goes down but anycast processes keep running. In that case, the cron script can be configured either to bring down anycast processes to let another proxy host to take over routing, or (our current choice) send email to the administrators for an immediate action.

Every proxy host is running a command like this:

```bash
/local/perforce/bin/p4p -p 1666 -t host:port \ -r /local/perforce/1666 -d -L /local/perforce/logs/p4p.1666.log
```

where `host:port` identifies the location of a p4broker instance (see below).

In our experience, using anycast with multiple proxies increased Perforce Availability through the automatic fail-over.

**Replication**

**How it's done (2009.2 solution)**

When the replicate command became available, we implemented the following procedure.

Say, the master server is perforce-00, and perforce-01 is to be the host for our replica. Then, on perforce-01 we would run

```bash
p4 -p perforce-00:1666 replicate \ -s SOMEDIR/replica.state -J SOMEDIR/journal \ SOMEDIR/admin.support/p4admin_replicate -port 1666 -srchost perforce-01 -srctop DATADIR
```

`p4admin_replicate` is our script to replicate data synchronously\(^3\). This was implemented before the “p4 pull” command became available, and this method is still used for some replicas.

---

\(^3\) The script is available in Perforce public depot at

The `p4admin_replicate` script reads journal records from STDIN and passes them to the following command:

```
p4d -r root -f -jrc -
```

References:

http://www.perforce.com/perforce/doc.current/manuals/p4sag/10_replication.html and
http://www.perforce.com/perforce/doc.current/manuals/p4sag/aa_p4d.html#1043673

Some features of this script:

Only certain journal records trigger the need to copy data. For example, we ignore db.rev[cdhp]x records because there are no associated file changes.

Directories are grouped for efficiency, and we use the Perl module Parallel::ForkManager\(^4\) to parallelize `rsync` of selected directories.

We use this `rsync` command:

```
rsync -av --delete "$srchost:$srctop/$dir_nospace/" "$dir_nospace/"
```

Note the following:

- We recurse the directory. It may be an overkill, but “smarter” selection is error-prone;
- We preserve modification times of the files;
- The verbose mode allows introspection in case “p4 verify” finds missing revisions;
- We have to quote arguments in case if there are spaces;
- We use `--delete` because subdirectories and files related to deleted shelved changes as well as in a rare case of obliterate get removed in the source and we need to mirror it.

Result: the average replica age is typically 4-5 minutes, including data. However, certain conditions may greatly affect the replica age. For example, maintenance tasks may change using the filer cache, or copying a top depot directory may take a long time. In these cases the replica age may jump significantly.

**Work-in-progress – 2010.2 solution**

Perforce 2010.2 release has a new command “p4 pull” and several configurables to help the replication.

For the full replica we replaced the replicate command described above with configurations like this:

```
-> p4 configure show Replica
Replica: monitor = 1
Replica: server = 1
Replica: startup.1 = pull -i 2 -J /perforce/1666/journal
Replica: startup.2 = pull -i 1 -u
```

\(^4\) See [http://search.cpan.org/~dlux/Parallel-ForkManager-0.7.5/ForkManager.pm](http://search.cpan.org/~dlux/Parallel-ForkManager-0.7.5/ForkManager.pm)
We run p4d with the –J option. Therefore, we need to use the –J option in the first pull command, which replicates the journal records. Note that the -J option in “p4 pull” is available in p4d Build 284433 and later.

The number of “pull –u” commands depends on the amount of load on the server.

As of p4d Build 295040, there is a problem with running multiple “pull-u” processes. This will be fixed in 2011.1. For now, we limited the number of pull processes to one.

We use “p4 pull -l” to monitor pending file transfers to see whether we need to add more pull commands in the configuration.

Occasionally, “pull -l” returns messages about errors during transfers. Some of these messages go away automatically when the files are accessed. Others indicate actual errors. In our experience, they are related to either oblitered revisions or deleted shelved changes. This is expected to be fixed in one of the 2010.2 patch releases.

We also created a new meta-data only replica with the only purpose to create checkpoints every night. The configuration of that replica is:

-> p4 configure show Replica2
Replica2: monitor = 1
Replica2: server = 1
Replica2: startup.1 = pull -i 4 -J /perforce/1666/journal

**Comparing two ways to replicate**

“p4 pull” is definitely more efficient than our script p4admin_replicate used in the earlier solution.

A certain reason to use a script with the “p4 replicate” command rather than “p4 pull” is if you want to filter journal records and take different actions. We have an instance of this script for the backup archive maintenance: the script reads the journal records, but does not replicate them anywhere. It only replicates archive files on a backup server.

**Monitoring the age of your replica**

We find it important to know the age of some replicas. For example, if a replica goes stale beyond a certain threshold, we may want to alert an administrator or change routing (see the p4broker section below).

The way we did it is the following.

Every ten seconds a cron job sets a special counter to the current time:

```
# This counter is used to monitor how close our replicas are to the master
* * * * * for n in 1 2 3 4 5 6; do export DT=`date`; echo "$n. $DT";
/export/db/perforce/1666/bin/p4 -p perforce:1666 counter keepinsync "$DT";
sleep 10; done
```

Now we can compare the value of this counter on the master and on the replica. The difference between the values gives us the lag between the servers.
Example:

```
-> replica_age -p perforce:1666 -r replica1:1666 -v -sec
Server perforce:1666: Mon Apr  4 14:26:01 EDT 2011
Server replica1:1666: Mon Apr  4 14:13:42 EDT 2011    739
-> replica_age -p perforce:1666 -r replica2:1667 -v -sec
Server perforce:1666: Mon Apr  4 14:26:22 EDT 2011
Server replica2:1667: Mon Apr  4 14:26:22 EDT 2011    0
```

Note that this method does not give any indication whether the archive is in sync. However, if no transfers are pending (“p4 pull -l” output is empty), then we assume the archive is in sync.

**Load Balance**

We monitor the time users hold different locks as Google suggested\(^5\), although we do not terminate processes automatically. In fact, we mostly only collect the information about the locks and command timing for later introspection.

We don’t wait until we notice a slowdown. If we need to investigate a performance problem we need all the data ready for analysis. That’s why **we start collecting the data about what might cause a slowdown as early as possible.**

This monitoring helps us identify users and workflows putting significant load on the system. It also helps us determine whether a remote office may benefit from deploying a new proxy.

Many continuous integration build systems submit at least a few commands every so often. Depending on the number of build systems and the time interval they use, the load may become noticeable.

For example, we have found out that one particular group, which deployed a build system based on TeamCity submits approximately 120,000 “changes –m 1 //...” commands every day. That’s almost three commands every two seconds, and from only one small group.

We deployed replica servers (we’ll talk about the details later), installed p4broker, and started rerouting build systems to replicas.

In our experience, **using replicas is a good way of reducing the load on the main server.**

**Using p4broker**

We use p4broker for several reasons.

First, p4broker helps distribute the load, routing certain read-only requests to replicas.

Second, p4broker can be used to provide read-only access to Perforce while the master server is under maintenance.

To accomplish these goals, we have a watchdog cron job that runs every five minutes and checks the availability of all the servers as well as the replica age. This job updates the broker configuration file as needed. P4broker does not have to be restarted. It automatically picks up the change in the configuration file.

If all the servers are up and running, and the replica age is within the set boundaries, the configuration file redirects all p4db (web) and build systems queries to a replica.

If the replica age reaches the set threshold, the routing to the replicas is removed from the broker configuration file and email is sent to the administrators.

If the main server is down, all read-only requests are redirected to a replica, and other user commands will result in the message that the server is in maintenance mode.

**Offline Checkpointing**

As of this writing, creating a new checkpoint at MathWorks takes an hour and a half. Even when it took half of this time, regardless of the time of the day when we ran it, some users would complain that their commands would hang. This is typical for companies where people work long hours or have offices around the world.

Our first solution used NetApp filer snapshots\(^6\). When we deployed replicas we implemented a simpler solution: creating a checkpoint on one of the replicas.

On the replica, we create a new checkpoint by the command

\[
p4d -r root -z -jd path
\]

Our replicas run with the “-J off” option, which turns off journals. This way we don’t have to worry about truncating journals on the replicas.

---

On the master server we roll journal every 30 minutes (journals are not compressed because replication does not support compressed journals). All checkpoints from the replica and journals from the master are copied to a backup filer, from which everything gets copied on tape once a day.

There is no synchronization between rolling journals on the master, replication process on the replica, and creating checkpoints on the replica. This gives us flexibility in when we do different maintenance tasks, but poses a challenge (more below) when we need to determine what journals we have to replay to rebuild the database.

**Disaster Recovery Plan (Preparation)**

At the proxy level we have redundant machines, where one takes over if another goes down.

At the p4broker level, the machine is an H/A virtual machine, and it gets rebooted if needed.

A copy of the archive data on the backup file server is maintained during the day by rsync. A checkpoint is created every night and copied to the backup file server. Every time when the master rolls journal, the latest journal also gets copied to the backup file server. Checkpoints, journals, and the archive get copied on tape once in 24 hours.

What is kept on the backup file server (and tape) is sufficient to restore the database to the last roll of the master journal, which is no more than 30 minutes.

Currently we do not have redundant servers at different locations and we don’t fail-over automatically.

**Disaster Recovery Plan (Execution)**

To restore the database, we run a script, which first finds the latest checkpoint and uses it to create db files from scratch. Then, the script needs to play sequentially all the journals from a certain point in time. Due to the mentioned independence of replication, rolling journals and creating checkpoints, using journal counters causes a race condition, sometimes leading to not replaying some journal records, which in turn causes inconsistency errors in the “p4d –xx” report.

Instead, to determine which journals to replay we use the @ex@ records\(^7\) in the checkpoint.

The first @ex@ record in the checkpoint contains the time when we started creating this checkpoint. Starting with the journal determined by the journal counter, walking back in time, we need to find the first journal that has an @ex@ record with the time preceding or equal the time from the first checkpoint record.

---

\(^7\) The described implementation had been done before the new @nx@ type was introduced in 2010.2. See the Release Notes ([http://www.perforce.com/perforce/r10.2/user/relnotes.txt](http://www.perforce.com/perforce/r10.2/user/relnotes.txt)) for the description of the @nx@ records.
Example of restoring server.

Created on a replica server:

```
-r--r--r-- 1 perforce users 3656847585 Nov 28 18:59 perforce-01-blr.1666.ckp.42909.gz
```

Created on the master:

```
-r--r--r-- 1 perforce users 14967 Nov 28 17:52 journal.jnl.42906
-r--r--r-- 1 perforce users 34766 Nov 28 18:22 journal.jnl.42907
-r--r--r-- 1 perforce users 12319 Nov 28 18:52 journal.jnl.42908
-r--r--r-- 1 perforce users 118816 Nov 28 19:22 journal.jnl.42910
-r--r--r-- 1 perforce users 51032 Nov 28 20:22 journal.jnl.42911
```

First, all old database files are removed and recreated by

```
p4d  -r root -z -jr path/perforce-01-blr.1666.ckp.42909.gz
```

Now we’ll find out what journals we should replay.

Find the time from the first @ex@ record in the checkpoint file:

```
my $ex_time_ckp;
if ( open my $PIPE, ' -|', '/bin/zcat $ckp' ) {
    while ( <$PIPE> ) {
        if ( /^@ex@ @d+ @d+$/ ) {
            $ex_time_ckp = $1;
            last;
        }
    }
} else {
    warn "WARNING: Cannot open pipe to zcat: $!";
}
```

From the checkpoint file name $counter=42909. Collect all the available journals and start walking back
starting with 42909 looking for the @ex@ record with time <= time in the previous step:

```
my @all_journals = glob("$backups/journal/journal.jnl.*");
my @jnl_nums = map { /\.(\d+$/ ? $1 : () } @all_journals;
READ_JOURNAL:
    # journals are sorted from the most recent to the oldest
    for my $n ( sort { $b <=> $a } grep { $_ <= $counter } @jnl_nums ) {
        my $jnl_file = "$backups/journal/journal.jnl.$n";
        open my $JNL, '<', $jnl_file 
            or do { 
                print "Unexpected error opening file $jnl_file: $!\n";
                next;
            };
        while ( <$JNL> ) {
            my ($time) = /^@ex@@d+$/;
            or next;
            print "Found in $jnl_file @ex@ with time=",
                scalar(localtime $time), "\n";
```

Perforce at MathWorks
if ( $time <= $ex_time_ckp ) {
    $new_counter = $n;
    last READ_JOURNAL;
} else {
    next READ_JOURNAL;
}

Now $new_counter contains the journal number to start replaying.

Here is the snippet from the log:

>> We got this time from /nasbackups/perforce/checkpoint/perforce-01-b1r.1666.ckp.42909.gz: Sun Nov 28 18:30:22 2010
>> Found in /nasbackups/perforce/journal/journal.jnl.42909 @ex@ with time=Sun Nov 28 18:52:01 2010
>> Found in /nasbackups/perforce/journal/journal.jnl.42908 @ex@ with time=Sun Nov 28 18:22:01 2010

Depending how frequently the master rolls the journal and how much time checkpointing takes, you may have to walk back for more than one journal.

Now we can play all the journals sequentially:

my $last_replayed;
for ( sort { $a <=> $b } grep { $_ >= $counter } @jnl_nums ) {
    $last_replayed = $_;
    my $jnl = "$backups/journal/journal.jnl.$last_replayed";
    # We'll use -f to overcome benign errors
    # Journal record replay failed!
    # This was recommended by Perforce Support.
    print "Replaying journal\n";
    system "$p4d -r $dbroot -f -jr $jnl";
}

We calculate $last_replayed in case if we restore the database on a replica and have to restart replication after we’re done restoring the database. In that case, we need to update correctly the replica state file we use in the –s option of the “p4 replica” command.

# modify the replica.state file according to the last journal we replayed
print "Updating $replica_state:\n",
"the new counter from which we'll start replicating is $last_replayed\n";
open my $STATE, '> ', $replica_state
or die "Unexpectedly, cannot rewrite $replica_state: $!";
print $STATE "$last_replayed\n";
close $STATE;
Testing

We have two stacks: production and test, which we keep very similar to each other. Every server host runs monitoring jobs. We also use Nagios\(^8\) to monitor certain processes, as well as the CPU load and disk space usage.

Test servers are used for daily testing of the restore procedure using the latest checkpoint and journals. This is followed by full verification (“p4 verify”, “p4d –xx”, “p4d –xx”), although during the week “p4 verify” is done only against head revisions.

Restore procedures on two different test servers are staggered. This means that they may use the same checkpoint, but they use a different number of journals. Any problem with the checkpoint usually becomes known within 24 hours.

Maintenance Procedures on the Production Stack

<table>
<thead>
<tr>
<th>Master</th>
<th>Replica-1 (full)</th>
<th>Replica-2 (meta-data only)</th>
<th>Replica-3 (full)</th>
</tr>
</thead>
</table>
| • Review Daemon  
• Mirroring some Perforce information to an SQL db  
• Partial mirroring of our own bug database into Perforce  
• Roll journal every 30 minutes and copy journals to backup storage  
• only on Sundays: Database verification (p4d –xx and p4d –xx) | • Replication with synchronous data update  
• Watchdog to create broker configuration file  
• Database verification (p4d –xx and –xx)  
• “p4 verify” – during the week: only head revisions; on a weekend – full verification | • Create daily checkpoints and copy them to backup storage  
• Database verification (p4d –xx and –xx) | • Mirroring some Perforce information to an SQL db  
• Replication with synchronous data update for the replica itself and backup storage  
• Database verification (p4d –xx and –xx)  
• “p4 verify” – during the week: only head revisions; on a weekend – full verification |

This replica is a warm standby and used for periodic controlled fail-over.

Fail-over Planning

We do not have an automatic fail-over procedure because we want to assess the situation before acting.

\(^8\) [http://www.nagios.org/](http://www.nagios.org/)
Fail-over is achieved by changing the broker configuration file and making the standby server the new master.

The standby server cannot be contacted directly (protect table makes sure every contact to the server is made through one of the proxies, and there is no proxy for the standby server). We have not started using the read-only mode available in 2010.2, but we plan to start using it at least for some replicas.

**Minimize downtime during upgrade**

Consider the following sequence of steps to minimize the downtime during a major upgrade:

- Prepare all binaries (p4d, p4p, p4broker, p4) in the right locations on the servers, so a restart of any of them would start the new version of that executable.
- Make sure your license files allow your upgrade.
- Configure p4broker configuration file to redirect all read-only access to a replica. All other requests would respond to the user with the message that the server is in the maintenance mode.
- Wait until the replication process is finished.
- Stop the master server (p4 admin stop). This typically brings down the replication processes. Alternatively, kill them.
- Upgrade the master (p4d –xu) and restart it.
- Restart p4broker (this is one of the service interrupts).
- Reconfigure the broker configuration file without using any replicas.
- Upgrade every replica server (p4d –xu) and restart replication.
- Reconfigure the broker configuration file using replicas.
- Restart proxies (this is another service interrupt).

**Conclusion**

- Testing your Disaster Recovery plan is essential. Test servers with identical to production architecture provide good environment for this testing.
- The following options may increase Perforce Availability:
  - Anycast with multiple proxies to provide the automatic fail-over;
  - High-Availability VM for p4broker;
  - Offload checkpointing to a warm standby metadata-only replica.
- Using p4broker to redirect some requests to read-only replicas helps balance the load.
• Replicate data synchronously on replicas that need them.

• “Set it and forget it” - Administration is easy when you have automated most functions.