

Improving Performance at Remote Sites

Lloyd Cha & Vishnu Mohan P.

Montalvo Systems

Introduction:

Fast inexpensive means of communication have fostered the popularity of distributed development by worldwide engineering teams. It is now common for companies to employ developers who work closely together despite being geographically distant¹. Physical interactions are being replaced by exchanges over e-mail, instant messaging, and audio/video conferencing. This paper itself is an example of remote collaboration, as its two authors have never met in person.

To enable distributed development, team members require real-time access to a common set of data regardless of their geographic location. To simply support and to enforce common methodology, tools should be designed to be site-agnostic. For projects that can be divided into different areas of responsibility, it is often suitable for sites to maintain their own local source control depots. Periodic releases or other mechanisms can be used to propagate changes from one site to another. The Perforce Remote depot feature can be used to automate this process². Sharing code in this manner works well when the groups operate independently. However, for tightly integrated development teams, this model is unsuitable because it does not provide real-time access to the entire development tree.

Wide Area Network (WAN) connectivity solutions have been rapidly improving in cost and bandwidth. However, long latency times between sites continue to present problems and are subject to unyielding rules such as the speed of light. Caching solutions such as the Perforce Proxy server provide some relief from these delays, but still have limitations. To work around these issues, further optimization is required.

Perforce Deployment at Montalvo Systems:

Montalvo Systems is a fabless semiconductor startup company with engineers concentrated at two geographically distant sites. Development work is distributed among tightly integrated hardware and software teams which span across both sites. The structure of the design, verification, and implementation teams requires that the entire repository of data be available in real-time for both reading and writing.

The source code checked in to our Perforce repository consists primarily of hardware design source files along with supporting software, verification test benches, scripts, and documentation. Hardware description files are similar in appearance to software source code. However, semiconductor design flow is more involved than the software development process owing to the complexity of the functional verification, implementation and physical design steps that must happen before a working silicon can be produced.³

1 Modgil, Deepak, "Distributed Development at Symbian," Perforce European User Conference, 2006.

2 Vinayak, Tony, "Distributed Development With Perforce Software," Perforce User Conference, 2005.

3 Johnson, Anders and Gary Holt, "'S' is for 'Source': The Role of the Build System in Configuration Management," Perforce User Conference, 2005.

Most engineers perform edits on a limited set of files, but require a substantial portion of the entire development tree to run simulations and regression tests. Frequent collaboration among engineers from both sites requires immediate visibility of changes to everyone working on a particular task. This requirement precludes the use of a shared software model.

Because of the tight integration of the two teams, there are no simple usage patterns that allow read-only caches or other optimizations which require dividing the depot. Changes must be seen rapidly across sites because engineers from both locations could be actively working on the same set of files as they debug and fix problems together.

Engineers at Montalvo Systems interact with Perforce through a wrapper script dubbed `mtv` (purportedly an abbreviation for “Montalvo”). `Mtv` automatically creates and manages clientspecs to ensure that all users are using consistent views of the depot. In addition, `mtv` is used to fetch releases, defined as a set of files known to have basic functionality, and to submit release candidates to the release server.

Native Perforce commands are passed directly through `mtv`. In addition, `mtv` includes additional functionality built around Montalvo's release process. The most commonly used commands in `mtv` are the `get` and `update` commands. The `get` command is used to start a fresh workspace populated with a release while the `update` command is used to sync an existing client workspace to a particular release.

The `get` process uses Perforce commands to create a new clientspec, retrieve the list of files and versions that comprise a release, and then `sync` to the release using that filelist. The `update` process is more complex because it needs to understand the existing client workspace state before retrieving new data. To perform an `update`, `mtv` analyzes the target release filelist, the current release filelist, and the current state of the client workspace to determine which files need to be updated. The script then issues the native Perforce commands to obtain the necessary files.

Evolution of methodology:

The initial deployment of Perforce Montalvo consisted of a Perforce Server in our main office and a Perforce Proxy server in our remote office. The Perforce Proxy implements a model of a central repository with remote site caching⁴. This architecture is designed so that users at remote sites will always get the most up-to-date state of the files in the depot of the Perforce Server. The local caches speed up access to the data for remote users. To ensure that engineers in the remote office have rapid response times, periodic updates of the most recent releases were run to preload the proxy's cache directory⁵.

As the size of our releases increased, the time required for users to retrieve new releases at the remote site grew to be much longer. Internal processing done by `mtv` contributed part of the delay. However, because performance at the remote site was degenerating at a much faster rate than in our local office, we suspected the majority of the time was spent executing the underlying Perforce commands. We instrumented `mtv` to confirm our hypothesis and we created some tools to obtain benchmark data for the underlying Perforce operations. We learned that the time consumed by `mtv` activities outside of Perforce was fairly consistent for users at both sites, but that the `p4 sync` process took much longer to execute in the remote office.

Despite the fact that the cache was preloaded to ensure that the versioned files were available locally, the Perforce commands used by `mtv` to create or update client workspaces in the remote office took longer than we expected. None of the suggestions from Perforce System

4 Vinayak, *op. cit.*

5 Perforce 2007.3 System Administrator's Guide, p. 158, December 2007.

Administrator's Guide⁶ applied to our situation. Neither the proxy nor the server were CPU limited, so disabling compression did not help. Deployment of additional Perforce Proxy servers would distribute load on the subnet local to the remote client workspaces but would further saturate our WAN link and would not affect the slow update times we measured using a single active client.

Perforce Proxy performance has been reported to provide less of a speed improvement when syncing sets of predominantly small files⁷. For these situations, the `-e` flag is recommended to prevent the proxy from caching small files. This effectively disables the proxy for most of our data and limits update speed to what can be obtained by accessing the server directly. We concluded that the Perforce Proxy provides insufficient speed improvement when deployed by itself in situations such as Montalvo's, and that to achieve suitable performance we would need to further dissect the `p4 sync` and proxy communication processes.

Turbo mode:

The Perforce Proxy server stores only the versioned file data. It does not store comments or other metadata and does not keep a copy of the have list to track of the state of each client workspace. It's purely a one-way cache of user-generated content. As a consequence, Perforce sync operations still require significant exchanges of information between the server, the proxy, and the client. Any Perforce operations that involve metadata require consulting the server. When the average file size is large, this traffic is not a significant component of the overall delay. But for large sets of smaller files, such as the situation at Montalvo, the per-file cost of this overhead is too high to ignore.

The `p4 sync` command performs the following operations internally:

- Determine which files and versions to retrieve.
 - lookup information in `db.label` if filespec uses `@<label>`
 - lookup information in `db.have` if filespec uses `#have`
 - translate filepath to `depotspec`

[Ed. Note: this isn't the full picture of 'p4 sync' operation]

- Retrieve actual data from proxy or server
 - update proxy cache if required
- Write data to disk at Perforce client workspace
- Update the have list on the Perforce Server

We conducted some experiments to isolate the contributions of the various activities to the overall time consumed. Our test cases provided explicit list of filepaths and revisions requested, so no lookups in the server's `db.label` or `db.have` tables were required [Ed. The `db.have` lookup is still required]. The server only needed to translate the filepath to a `depotspec` format before retrieving the file. We eliminated the cache update from the benchmark times by running the command multiple times and ignoring the first invocation which automatically populated the cache with the requested data if needed.

This left us with the tasks of updating the client workspace and updating the have list on the server. The `p4 sync` command performs both tasks, but does not give visibility to how much time is required by each individual subtask. The `p4 flush` command can be used to update the have list without doing any data retrieval, but there is no way in the 2005.1 version of Perforce to

6 Ibid.

7 <http://maillist.perforce.com/pipermail/perforce-user/2007-November/022755.html>

retrieve data from the server or proxy without performing the have list update (the '-p' option to p4 sync was not available until Perforce version 2007.2).

Sample results from one particular test case are shown in the following table. This represents the underlying Perforce operations for an update to a test release number from an empty client workspace:

command	Execution time
p4 sync -f <filepathlist>	14 min 33 sec
command	Execution time
p4 flush <filepathlist>	1 min 1 sec
p4 sync -f <filepathlist>#have	5 min 56 sec
TOTAL	6 min 57 sec

Note that in both cases, the same amount of work is performed, but by executing the server update and the client workspace update as separate operations, less than half the time is consumed compared to the standard monolithic p4 sync.

We inferred from our results that some sort of information exchange is being performed as each individual file is being updated. This theory is consistent with the have list being updated once per file rather than being batched for optimum efficiency. This is a robust way to handle the update, since the state of the server and client workspace will always be in sync even if the command is aborted. We suspected we could improve on performance if we were willing to compromise on safety by first updating all the server information in one batch and updating the client workspace information in a separate batch.

We created a turbo mode for mtv which essentially replaces the p4 sync command used internally with the following pair of commands:

```
p4 -p<server>:1666 flush <file1#rev1> <file2#rev2> ...
p4 -p<proxy>:1999 sync -f <file1>#have <file2>#have ...
```

We believe that using p4 flush is the fastest practical way to update the have list on the server. Only one exchange of information between client and server is required, avoiding the need to handshake as the state of each file is updated. While processing the flush command, the Perforce Server assumes that nothing is changing on the client workspace or proxy and does not communicate with the remote site.

After the server have list is updated, the p4 sync -f <file>#have command synchronizes the client workspace to the server state for <file>. Since we are only requesting a refresh of what the server thinks that the client workspace already has, the state on the server database is not touched. Removal of files by this method is a special case. The script is written to ignore deleted files on the flush step, and to remove them using on the p4 sync -f <file>#none at the sync step.

The following table shows some benchmark results for various updates:

Operation	Native Proxy	Turbo mode
Update of 1000 releases	9 min 17 sec	6 min 38 sec
Update of 500 releases	5 min 21 sec	3 min 59 sec
Update of 100 releases	2 min 50 sec	2 min 4 sec
Update of 6 releases	35 sec	31 sec

Remote mode:

Additional performance improvement required further minimizing exchanges of information across the WAN. We broke down the update process into its fundamental components and carefully analyzed the flow of data. The following table shows the locations of data for the example of a user at the remote site performing an update:

Source/Destination	Physical location
Source of user request	client host (remote)
Source of filelist	Proxy (remote) or Server (local)
Source for user data	Proxy (remote)
Destination of have list update	Server (local)
Destination of user data	client workspace (remote)

The interesting fact we discovered is that many of the transactions can be done on one site or the other without corresponding across the WAN provided that the proxy cache has already been populated with the requested user data. The only part of the transaction that has to happen at the remote site is the actual transfer of the data files from the proxy to the client workspace.

Data	Source	Destination
Filelist (files/versions)	Server (local)	Script memory (local)
Have list	Script Memory (local)	Server (local)
User data	Proxy (remote)	client workspace (remote)

The initial mtv command request consists of a project name and release number. The mtv script uses that information to retrieve the appropriate filelist from a special location in the Perforce depot. These operations can all be done without interacting with the local client, and therefore can be done completely at the same site as the Perforce Server. Instead of executing the Perforce command from the remote site, better performance can be obtained by sending the entire command line to the local site for execution.

This new flow, remote mode, is similar to the flush/sync method described for turbo mode, with the key difference in that the flush step is initiated from a machine in the local site even though it refers to a client workspace that only exists in at the remote site. This flow takes advantage of the fact that the p4 flush operation does not involve the client workspace's filesystem.

The remote client starts by sending a command to the host at the server site via ssh to initiate the process. The analysis of the workspace state and the update of the server have list are handled next. These steps only involve LAN traffic at the main site. To complete the update a script is generated to standard output and returned to the client on the remote site. The client runs this script to perform the final p4 sync -f <files>#have.

Remote mode achieves an additional speed boost by dividing the sync to the #have versions into groups of fifty files. These sync commands are executed in parallel from the remote site.

Users typically execute remote mode using a one-line shell alias. The procedure begins by using secure shell (ssh) to execute a shell command on a machine at the local site that runs mtv

with the `--remotemode` flag. `mtv` performs the update until the `p4 sync` step, which must be executed at the remote site. To transfer this information back to the remote client host, `mtv` generates a Perl script to standard output that can be retrieved by originating wrapper script and executed on the initiating host. The resulting command looks like:

```
ssh <main-site-host> mtv update --remotemode | perl -
```

The following table shows benchmark results for several test cases:

Operation	Native Proxy	Remote mode (typical ⁸)
Update of 1000 releases	9 min 17 sec	41 sec
Update of 500 releases	5 min 21 sec	38 sec
Update of 100 releases	2 min 50 sec	29 sec
Update of 6 releases	35 sec	18 sec

The remote mode offers outstanding performance, but requires a cumbersome usage model. Only the most savvy users were willing to put up with it. Worse yet, in practical usage, engineers experienced random hangs and generally poor reliability. The dependence on several different machines and processes in the pipe made debugging and tracing the problems intractable. In the future, we intend on revisiting this method to iron out these bugs.

Bypass mode:

An alternative approach to improving performance is to improve the efficiency of data retrieval from the proxy cache. The final `p4 sync -f <file>#have` method of transferring data used by turbo mode and remote mode still requires looking up information in the `db.have` database on the server to determine which versions to fetch. Furthermore, there appears to be additional communication between the proxy and server to determine if the proxy has the required data. We hypothesized that the fastest way to retrieve data was to read the versioned files directly from the local storage cache used by the proxy rather than using the proxy daemon itself. Separating the operations in this way also means that updating the server can be performed independently from the task of delivering the data to the client workspace.

We considered two approaches to retrieving data from the cache directory. The simplest approach is to make the cache directly accessible from any machine used to update local Perforce client workspaces and to write a client-side script that could read the repository data from the cache directory. If security is a concern, making the cache data available in this manner would be impractical, but in our environment most engineers have read-access to the entire repository anyways. A more sophisticated approach would be to develop a custom daemon to run on the proxy server which would read data from the cache directory and deliver it to the client process connecting via a TCP port.

In the interest of minimizing the effort required to demonstrate proof-of-concept, we opted to implement the direct access approach. This method necessitated moving the cache from directly attached storage on the proxy machine to network-attached storage (NAS) on our central fileserver. Since we required only read-access to the cache, the existing operation of the Perforce Proxy was unaffected.

The Perforce Server stores much of its depot data using variations of the traditional RCS format. The Perforce Proxy uses this same format in its cache to store local copies of versions

⁸ The remote mode is prone to random hangs. These results omit the cases where the update hung.

that have been read at least once by one of its clients. Raw RCS commands can be used to access these files and reconstruct user data. The specific storage method used by Perforce for a file depends on the filetype selected by the user⁹. Before interpreting the data from the cache, mtv must determine what format is being used. Filetype information is stored only in the server's database and is not cached on the proxy. Therefore, the most efficient way to determine the filetype is to bypass the proxy and query the server directly for this information. Mtv uses the p4 files command to do this. A sample query and result follows this format:

```
% p4 -p server files //depot/file/location#1234
//depot/file/location#<rev> - <action> change <change#> (<filetype>)
```

//depot/file/location is the name of the file in the depot in Perforce syntax, <rev> is the requested revision number of that file, <action> is the action taken at that revision: add, edit, delete, branch, or integrate, <change#> is the number of the changelist that this revision was submitted in, and <filetype> is the Perforce file type of this file at that revision.

Using the depot format to specify the file location facilitates retrieval of the versioned data in the proxy cache without needing to translate the information through the clientspec [Ed. This technique is highly unreliable and won't work for branched files]. Revisions of text filetypes are stored in RCS format¹⁰, using deltas unless the +C flag is used, in which case the entire source file is stored in the <filename>.d directory. Binary filetypes are stored in their entirety and are optionally compressed in gzip format unless the "u" flag is specified. Symbolic inks are stored as RCS text files with the content of the file being the destination of the link.

The bypass mode delivers consistently fast updates without the reliability problems encountered by remote mode.

Operation	Native Proxy	Bypass mode
Update of 1000 releases	9 min 17 sec	5 min 35 sec
Update of 500 releases	5 min 21 sec	3 min 2 sec
Update of 100 releases	2 min 50 sec	1 min 34 sec
Update of 6 releases	35 sec	35 sec

With these enhancements, the client workspace can be updated with minimal involvement from the Perforce Server over the WAN. The list of files and versions in the release is computed locally by the mtv script. The server is only consulted to get the file type information. File retrieval is done directly from the proxy's cache using methods that do not require the proxy to interact with the server.

Performance summary:

The following table summarizes our performance measurements. The normalized performance rate relative to the baseline setup using the Perforce Proxy directly is the ratio of the time required for the baseline to the time required by the new mode.

Update mode	Normalized performance rate			
	1000	500	100	6
Native Proxy	1.00	1.00	1.00	1.00

⁹ Perforce, op. cit., p. 30.

¹⁰ <http://www.gnu.org/software/rcs>

Update mode	Normalized performance rate			
turbo	1.40	1.34	1.37	1.13
remote	13.59	8.45	5.86	1.94
bypass	1.66	1.76	1.81	1.00

We also collected some performance profile data to highlight the different areas targeted for optimization by the various methods. The following table shows update profile for the various modes when updating text files that have an average file size of 13KB.

Update mode	portion of total time elapsed	
	Flush	Sync
Native Proxy	Difficult to calculate	
turbo	26%	74%
remote ¹¹	1%	95%
bypass	87%	13%

This table shows the same information for binary files averaging 329KB in size:

Update mode	portion of total time elapsed	
	Flush	Sync
Regular	Difficult to calculate	
turbo	18%	82%
remote ¹²	2%	85%
bypass	94%	6%

From this data, we can see the effects of the various modes

Caveats and Disclaimers:

The Perforce system is designed to be robust and is careful to not leave client workspaces in inconsistent states. However, our performance needs were so critical that our focus was to achieve maximum possible performance even if exceptions and abnormal terminations put the integrity of the client workspace at risk. Provided that failures are still detected and reported, we were willing to compromise on the error recovery in favor of achieving high performance. Readers looking to implement these concepts should understand that the techniques we described carry substantial risk and are not supported by Perforce Software or the authors of this paper.

11 For remote mode, the remaining 4% of time (approximately 7-8 secs) is used to filelist information across sites. This value is constant regardless of the number of files synced.

12 Remaining 13% of the time (approximately 7-8 seconds) is used to transfer filelist information.

Our benchmark data is based on the 2005.1 version of Perforce Server which is several generations old. Substantial improvements have been made to the internal protocols since then, particularly with the database schema on the server side¹³. Newer versions have a significantly different performance profile compared to the statistics we collected during development of our tools.

Some of the concepts discussed in this paper reflect educated guesses on the internal workings of Perforce operations based on empirical data and extrapolation from documented information.

Conclusions and future work:

Our implementation is still a work in progress and we believe there is still room for further improvement. We plan to concentrate on improving the reliability of remote mode and further optimizing bypass mode.

Solving the inconsistent behavior problems of remote mode and debugging the various hangs that plague its usage in practical operation are necessary to gain user acceptance. A daemon on a host at the server site could be written to handle the p4 flush required in the first stage of remote mode. This would eliminate the awkward implementation involving ssh.

Our implementation of bypass mode does not include many of the tricks employed by remote mode. In particular, the execution of the p4 flush command at the server site and the parallelism of the local sync activity could reap significant benefits. If the reliability and consistency problems of remote mode can be solved, its enhancements can be incorporated into bypass mode.

Many internal operations of bypass mode happen serially despite being completely independent of each other. Rewriting the bypass mode so that these tasks happen simultaneously may produce efficiency gains. For example, updating of the have list on the Server and retrieving of user data from the proxy can be done in parallel. Furthermore, multiple proxy servers could be deployed to allow large updates to distribute the actual user data updates across multiple machines.

We do not yet have a detailed understanding of the protocol used between the client, proxy, and server. Direction of future efforts would ideally be guided by a more comprehensive analysis than our empirical observations. Such information could be obtained by packet captures of transactions between the daemons, or by publication of internal transaction protocols used by Perforce.

Mtv eschews the use of labels to define releases in favor of maintaining explicit filelists. Maintaining the list of files and versions outside of Perforce avoids having to lookup release information in the Perforce Server database for local processing and facilitates the division of sync requests into manageable batches. However, using filelists scales poorly and makes communication excessively verbose. In addition, it may prevent the proxy or server from performing internal optimizations. After we update to a more recent Perforce Server version, we plan to reevaluate the benefits of converting mtv to use labels for creating and retrieving releases.

We have demonstrated dramatic improvement in client workspace update speed from our remote site by making trade-offs regarding exception handling and recoverability. Our methods provided sufficient performance gains to allow our engineers at the remote site to be productive. We believe that further refinement of our techniques will result in even greater performance gains and may mitigate the reliability concerns. We hope that our ideas will

13 Shields, Michael, "Performance Update through 2006.1," Perforce European User Conference, 2006.

inspire more robust and supportable methods of performance improvement from the Perforce Software and the Perforce community.