Scripting with Perforce

J. Bowles
S. Vance

April 2005

Abstract

This paper follows the arc that starts with **context**, which is the environment in which a script will run. It then continues by showing examples of specific **techniques**, giving strong support to a body of example scripts that are written in Perl/ Python/ Ruby/ P4Perl/ P4Ruby. It then shows how to create tools that support higher-level **purpose** to implement specific policies or address organizational needs.

1 Introduction

Software development and handoff activities include repetitive steps, which are sometimes automated and sometimes repeated by hand. Often, the reason given for a manual step is, *we didn't have the time to automate that part.*

This invites the question, *once you have the time, where do you start?*

AUTOMATING QUERIES AND SYSTEM LEVEL TASKS NEED NOT BE DAUNTING.

2 Terminology

There are three perspectives from which we can approach Perforce scripting tasks:

context *Where it's done.* Server-side triggers are such a context; workspace-oriented tools are another; [post-submit] review daemons are a third.

techniques *How it's done.* Methods that can be applied to address particular technical concerns such as language choice, efficient data access, performance, maintainability and extensibility.

purpose *Why it's done.* Business, system or process goals that motivate the creation of the script. Examples are tools to verify that code is reviewed/authorized prior to checkin (or prior to release).

Note that the **purpose** might restrict the choices of **context**. E.g., a backup script might need to be run in the *context* of the server machine for security reasons.

2.1 Examples of "context"

The *context* of a script is where it is run and how it is installed[1]. The following is an incomplete list of *contexts* in which one creates (or runs) Perforce scripts:

- Client commands run in workspace. An *overnight build script* runs Perforce commands, in the context of a certain user and a certain workspace name. (It might, as part of initialization, create that workspace definition.)
- Trigger scripts run on server. Examples related to this context can be found in [Vance2005].
- Review scripts ("post-submit") run in workspace.
- Maintenance commands run on server. For example, a script that runs a backup using the command, `p4d -r .  -jc`, needs to be invoked directly on the server machine as a user that has *write* permission on the database.
- *p4win* Tools menu. The *p4win* menu is another place in which one might run a script or *need something done*. In this context, the script / program can receive a small number of arguments (e.g., the selected file, user, etc.), cannot do any user interaction to speak of, and needs to realize that anything printed out to *standard output* might vanish when the enclosing window goes away.

2.2 Examples of "techniques"

The *techniques used* reflect the programming choices: which language to use, how to retrieve data efficiently, how to gauge performance, coding for long-term maintenance needs and extensibility.

This section lists several guidelines that are helpful techniques.

2.2.1 Use the right language

Compiled languages such as *C++* and *Java* are useful, in that they are more resistant to change in the environment due to compile-time binding and less subject to break when a new package or compiler/interpreter is installed[2].

[1]Issues such as user permissions, are part of the *context* also.

[2]Compiled languages are better at preventing accidental and purposeful modification of the behavior. There is a trade-off between ease of development and security in most cases.

However, such mechanisms / structure might prevent a build engineer from making a fast change to a script to continue a compile, or to release a product. Compiled languages might need too many files/libraries to guarantee a quick tinker-and-rerun cycle.

The wealth of scripting languages, which have [somewhat] smaller footprints and support fast development of new scripts, makes them attractive for these needs. Scripting languages[3] support loop constructs, string operations, and modification from any text editor. The more recent languages, such as Python and Ruby, have object-oriented support that rivals the most sophisticated "traditional" languages.

2.2.2 Not all languages are created equal.

The language might be too primitive, using arcane constructs for looping (DOS *.bat / .cmd* files) or for string handling (Unix *.sh* scripts). Also, the language environment might be too combersome, requiring editing within an IDE or compiling to run[4].

Using language-specific advantages There are different reasons to choose one language over another: business needs, features of a language, availability on various platforms, and cost to implement and maintain.

Certain languages, such as Python and Ruby, have a binary data support called *marshal*. This retrieves data from a Perforce server, already parsed into usable fields and columns.

As another example, Ruby has excellent primitives for set operations. As a result, once the contents of two labels has been retrieved into a Ruby *list*, computing the difference between the labels is fast[5]:

```
# case 1:  files in the first label but not the second
filesOnlyInLabel1 = label1filenames - label2filenames
filesOnlyInLabel1.each { |fname|
     puts "Only in #{label1}:  #{fname}"
}
```

2.2.3 Use tagged output

Whenever possible, scripts should not bother parsing Perforce data. There are output formats that deliver already-parsed data to an application[6]:

[3]The Unix shell, Perl, Python, Ruby.

[4]Various *C / C++ / Java* solutions. A compiler expert will point out that the issue is not the compilation itself, but the complexity introduced when the additional steps are added. Also, compiling to machine code is rarely necessary for these scripts, since they are rarely CPU-intensive enough to matter. In times when that's the case, optimizing the algorithm is often more fruitful than switching to a compiled language.

[5]The full program can be found as "difflabel.rb" on the Perforce Public Depot.
This script is *//guest/jeff_bowles/scripts/difflabel.rb* ; there are Python, Perl, P4Perl, and P4Ruby counterparts there, also.

[6]Filenames with embedded spaces require no extra programming, using this technique.

Type of Output	Example ’p4’ Option
ascii name / value pairs	`p4 -Ztag` ***cmd***
markshal-encoded arrays of hashes (Python)	`p4 -G` ***cmd***
markshal-encoded arrays of hashes (Ruby)	`p4 -R` ***cmd***

[Goldstone2005] provides a Python program that uses `p4 -G` output to create a Unix-like filter for reformatting Perforce output.

2.2.4 Automatically started scripts should not depend on “current user” .

Scripts might be started automatically, at a certain time, to create a checkpoint or run a build. *Such scripts must set the user name, workspace name, and any other “context / environment” information and not inherit it from the current environment.*

One approach is to set such things from the script itself, as seen in the first lines of these examples[7]:

Unix shell version	Windows version
export P4USER=george **export** P4PASSWD=banana cd /home/perforce/database **p4 admin checkpoint** ls -l checkpoint.* journal*	**set** P4USER=george **set** P4PASSWD=banana d: cd \perforcedb **p4 admin checkpoint** dir checkpoint.* journal*

An experienced programmer will realize that such information might be placed in a common script/module that all automatic scripts reference.

2.2.5 Understand basic installation and redirection.

Continuing with the example in the previous section, it is helpful to capture or redirect output for debugging and auditing[8].

Example: capturing output Using the standard Unix shell and the standard Windows command-line, the sequence to redirect standard output is: `cmd > outfile.txt` .

The three-line sequence, given below, is a simple example of this.

[7]Note that this script doesn’t directly create the checkpoint, but connects to the Perforce server and asks it to make a checkpoint. This avoids certain common errors, because the script relies on the running server for configuration information instead of looking it up. (As an example, the running server knows whether journalling is enabled and the like.)

[8]The system-level commands might discard the output or email it somewhere, and it is more helpful to send the output to your own log/records.

```
p4 files //depot/main/...@label1 > contents1.txt
p4 files //depot/main/...@label2 > contents2.txt
diff contents1.txt contents2.txt
```

Example: capturing a form's contents Any command that brings up an editor, such as `p4 label` or `p4 client`, has an alternate form: `p4 label -o` or `p4 client -o`. Often, a script will capture the output of a form, massage it slightly, and write it back to the database:

```
p4 client -o > client.txt
...run 'awk / sed / perl' to modify ''client.txt''
p4 client -i  < client.txt
```

Example: logging output from a script/trigger The command, `cmd > outfile.txt 2>&1`, captures the standard output to *outfile.txt*[9].

The script from the previous section, *d:\perforcedb\mkckpt.cmd*, will generate output that should be saved:

```
d:\perforcedb\mkckpt.cmd > d:\p4chpt.out 2>&1
```

To install a program onto Windows or Unix so that it's invoked every day at 4 AM, it's important to find the operating system tools for such things. On Unix and Linux, it is *cron* ; on Windows, it is *at*.

After copying this script to the Perforce root directory (or a "scripts directory"), the command to run this script on Windows, twice a week at 4 AM, would be:

```
at 4am /every:monday,wednesday "d:\perforcedb\mkckpt.cmd > d:\p4chpt.out 2>&1"
```

2.2.6 Performance: Count the trips to the well

A *Perl* script, that calls `p4 describe 18291` several times, can be optimized on two levels:

1. Don't call `p4 describe 18291` so much. Recode the script to avoid such things.
2. Write a routine called "getchangeinfo" that remembers previous results, and returns those results if called again with the same request. (There needs to be a way to invalidate the cache as necessary.)

Note that the second strategy could be implemented in a library of routines that all scripts use.

This optimizes database queries and also network traffic; even small queries have network overhead.

This strategy generalizes, to a point: during the debugging of a new script, it can be useful to save the output of Perforce commands into a file. This saves overhead on the server and makes debugging faster. The following example is written in Perl:

[9] The archane syntax, identical in the Unix shell and the Microsoft command interpreter, is an instruction to redirect output to *outfile.txt* and then to redirect file number 2 (standard error) to the same place as file number 1 (standard output). *Order is significant – give the log filename as the first redirection!*

```
$dataf = "/tmp/test18291.txt";
if (! -f "$dataf")  {
    system("p4 describe  18291 > $dataf");
}
open(IN, "<$dataf") || die "Cannot open $dataf\n";
```

For production needs, any temporary files need to be cleaned up, eventually, as part of the script's exit strategy.

2.2.7 Performance: Group arguments together.

The command, `p4 files x y z`, results in three requests, one for each argument. For simple or short-running commands, this is of no concern.

In the case of commands that affect files or take time, such as `p4 sync //depot/main/...@label1 //depot/main/...@label2`, the results reflect this architecture[10].

In this example, a third – temporary – label is helpful. It could be used to craft the list of what files to include from `label1` and `label2`, so that the argument to `p4 sync` is simpler.

Similar strategies exist for temporary clients. Build engineers sometimes create them to guarantee a clean workspace definition before a build, or to simplify the arguments[11] of `p4 obliterate`.

2.2.8 Performance: Loop through results

Database programmers have to wrestle with database performance. One rule is that *one query is better than two, or twelve, or a hundred.*

For example, `p4 fstat ...` is one database query. It is somewhat expensive, in that it collects a good amount of information about each file, but it's one query.

A less-efficient approach is to run `p4` on each filename or argument, which increases network traffic (more requests made) and increases the number of queries (from one query to one-per-file).

As an example, *p4unknown.rb* computes the list of all files in the workspace and compares against what is mapped from the server. The Perforce-specific parts are below:

[10] In this case, `p4 sync //depot/main/...@label1` would put one set of files in place, and then `p4 sync //depot/main/...@label2` would put a second of files/revisions into place.

[11] and improve the running time

```
#---------------------------------------------------------
# first call to P4:  'p4 client -o'
#---------------------------------------------------------
puts "Step 1:  Get the client name."
cl_spec = p4.run("client", "-o")[0]
cl_name = cl_spec['Client']
cl_root = cl_spec['Root']

puts "Ran user-client, output was 'client=#{cl_name}'"
#---------------------------------------------------------
# second call to P4:  'p4 fstat //myclient/...'
#---------------------------------------------------------
puts "Step 2:  Get the list of Perforce-known files."
ret = p4.run("fstat", "//#{cl_name}/...")
```

In this case, the command `p4 fstat //myclient/...` makes it possible to write this script with two calls to Perforce.

Even if it becomes necessary to create temporary labels or client workspace definitions[12], this optimizes traffic to the server.

2.2.9 Use //clientname/... syntax when appropriate.

The previous section uses `p4 fstat //myclient/...` to refer to the files mapped onto workspace *myclient*. The results were dramatic, reducing a complex task to a small number of queries.

This intermediate syntax is an easy way to address only the client-mapped files.

2.2.10 Know how to document performance.

All the scripts referenced in this paper have a comment section at the top. The comment block for *p4unknown.rb* is:

```
# Task:  determine which files need to be "p4 add'ed."
#
# num of calls to 'p4':  2
# status:  tested on Darwin Mac OS X using "p4 -R"
```

Note that the number of calls to `p4` *does not depend on the number of arguments or number of files.* If it does, recode the script.

In addition, WALK THROUGH THE PROGRAM AS IF YOU'RE A FIRST-SEMESTER STUDENT, ASKING YOURSELF WHERE THE PROGRAM WILL SPEND THE MOST TIME. Often, a lookup of a filename in a

[12]From time to time.

list of 100,000 files can be sped up with a simple hash or change in how the list is stored. (For example, a string comparison of files that start with *//depot/main/* against a large list of similar files can waste CPU cycles. Hashing into a 'dict' or 'hash' object, based on the filename (the last component of the full name) might prove fruitful, for example.)

2.3 Examples of "purpose"

The *purpose* is the highest-level need that's being addressed.

Even seemingly "high level" needs, such as automating build scripts, serve a higher purpose: *supporting development "workflow"* needs.

2.3.1 The build script

Every development group eventually creates an automated script to build and stage the current product. Using the terminology from the previous section:

purpose ...to create a "development heart-beat" that tells development and testing groups, once a day, what the *health* of the development code line is;

context ...is that of a script (or scripts) running on a stable machine that houses a Perforce client workspace;

techniques ...include choice of language, guaranteeing reproducibility, and installation so that it is done every night like a night watchman.

Note that the context ("IS wants it run on the server machine") might limit the techniques available. E.g., *language X is not a choice because it isn't cleared for the security level of the server machine*.

2.3.2 The backup script

Similarly, every Perforce installation will need to checkpoint the database to create a tidy version of the *metadata* for later filesystem backups. In this case:

purpose ...to create a *metadata checkpoint* that be a component of a larger backup scheme.

context ...is that of a script (or scripts) running on the Perforce server machine. In this case, that is the right place to run a checkpoint and [possibly] delete old checkpoints or archive to other volumes[13].

techniques ...choice of language, inspecting the code to find critical sections during which a machine failure could be catastrophic (such as removing a needed checkpoint before authenticating its successor).

[13]This is something that requires write access to the Perforce server directories, which implies running on the Perforce server machine or having write access to its disks. The former is cleaner, from a security aspect.

2.3.3 The security enhancements

(What follows is a small example, but one that hints at the options available.)

purpose ... Giving permissions to `user *` might create a security problem. It is useful or necessary to enforce the rule, **no permissions can be given to *user* ***.

context ... The Perforce 2004.2 triggers provide a way to examine the user-provided forms before the *commit* to the database.

- Any such trigger will need to run on the server, and will therefore run with the system-level permissions of the user who started the server.
- It is possible for the trigger to run, but to provide the user-provided form as contents of a file. The trigger, thereby, needn't call `p4` at all.

technique Using a scripting language such as Python or Ruby, use a trigger script that's invoked at state "in" for form "protect". The trigger script can parse the filename that's provided, and will call `p4` zero times.

The example script follows[14].

```
# Task: form trigger that refuses "p4 protect" entries that
#     give permissions to "user *"
# num of calls to 'p4': 0
# status: tested on Darwin Mac OS X using python 2.3
#
# Appropriate 'p4 triggers' line follows:
#     example1 protect in "python  /path/to/trigparanoid.py  --formfile %formfile%
2>&1"

import getopt
import sys
import re

defaultFormFile = None

[options,args] = getopt.getopt(sys.argv[1:], '',
        [ 'formfile=', 'ff='])

for [opt,arg] in options:
    if  opt == "--formfile" or opt == '--ff':      defaultFormFile = arg

if defaultFormFile is None:
    print "--formfile XXXX must be given on command-line"
    sys.exit(1)

errorList = []

fd = open(defaultFormFile, 'r')
```

[14] A Ruby version is available from the authors.

```
if fd is None:
    print "Cannot open file %s" % defaultFormFile
    sys.exit(1)

protect_re = re.compile('^\s+(\S+)\s+(\S+)\s+(\S+)\s+(\S+)\s+(.*)')

for ln in fd.readlines():
    m = protect_re.match(ln)
    if m is None :    continue
    (perm, entityType, entity, ipAddr, pathName) = m.groups()
    print perm, entityType, entity, ipAddr, pathName

    if entityType == "user" and entity == "*" and pathName[0] != '-':
        errorList.append("Cannot add reference to 'user *'. Sorry.\n")

fd.close()

if len(errorList) > 0:
    for e in errorList:
        print e
    sys.exit(1)
else:
    sys.exit(0)
```

2.3.4 Integrating a bug report to a Perforce changelist

Many approaches can relate a *bug report* to a Perforce submission of work (*changelist*). Examples include the *p4 jobs* database, the *defect tracker integration* ([p4DTI]) mechanism that integrates third-party bug databases to Perforce's jobs, and commercial plug-ins that do this.

In each case, the problem does not reduce to a single Perforce script. Still, the larger categories exist:

purpose . . . relate the bug reports (or feature requests) to current development checkins, to support queries/reports for management and for downstream needs such as QA / Testing, release notes / documentation, and later maintenance of the product.

context This depends on the tools available. The *p4 jobs* database uses standard Perforce "client workspace" commands, but might be augmented with triggers or post-update scripts that do something else; the *p4DTI* approach is that stragegy, but with post-update scripts that update a Bugzilla (or other) database.

techniques [p4DTI] is a good example, in that it is a Python program that mirrors the `p4 jobs` database into a third-party bug database using the `p4 logger` hooks.

2.3.5 Enhancing the “jobs” database to include state transition enforcement

Most QA departments rely on bug database rules to prevent bug reports from moving from a *current status* to a *new status* that does not make sense. (E.g., a bug is filed and immediately closed, without a *review* step.)

Although the default `p4 job` behavior does not include validating the proposed *Status* of a *job* when it is updated, that functionality can be added in a script that runs in the **context** of a *form trigger*.

purpose . . . Install a “jobs” database mechanism that allows updates to the *Status* field of a job, only when the transition from the old value to the proposed value is allowed.

context If run as a *form trigger* on the “job” form, this will be run on the server as part of the update of a job. It can refuse to allow the update, if the job is attempting an invalid change of its state.

techniques By invoking `p4 job` to get the current values, the trigger script will have ASCII versions of current and proposed job data. It can compare the two *Status* fields, refusing invalid transitions.

2.3.6 Installing triggers to enhance access control of forms

It is now possible to install a trigger that is run when a Perforce form is deleted. This makes it possible to notice a deletion, or possibly to refuse a deletion. This can provide for a stronger access control (“only members of group XXX can delete a label”) or reporting mechanisms. [Vance2005] gives more details on this strategy.

purpose . . . Limit deletion of forms, to implement a formal access control of *jobs* and *labels*.

context If run as a *form trigger*, this can be run on the server as part of the deletion of a *job* or *label*. It can refuse to allow the deletion, as appropriate.

techniques By invoking `p4 group`, the trigger script can decide whether the user deleting the form is in a group that is allowed to delete the form.

3 Summary

For any scripting task, the first step is to examine the large questions. The terms, *purpose, context, techniques*, correspond to the questions, *why do this, where will I do this, how will I get it done*.

References

[Goldstone2005] Goldstone, John, *Using P4G.py From The Command Line*, Proceedings of the 2005 Perforce User’s Conference. Las Vegas.

[p4DTI] Brooksby, Richard, *Perforce Defect Tracking Integration Project*, retrieved from *http://www.ravenbrook.com/project/p4dti/*

[Vance2005] Vance, Steve, *Writing Triggers in Perforce*, Proceedings of the 2005 Perforce User's Conference. Las Vegas.