

Commit Trigger Example

A configurable number of revisions

Richard E. Baum
February 26, 2005

Abstract

A brief discussion and walk through of a commit trigger that allows a configurable number of revisions per binary file. The trigger intelligently replaces file revisions with small stub files that notify users of what happened to the original file. It fixes up MD5 hash values to prevent subsequent error warnings by “p4 verify” runs.

1 Introduction

For binary files, Perforce offers a choice of storing either every revision of a file or only the last one. There are problems with this approach, though. If the files are very large it may be desirable to store just the history of the revisions but not all of the revisions themselves. Similarly, if storing just the last revision, a submission to the repository wipes out the previous versioned file being stored and eliminates the opportunity for rollback. Therefore, it would be convenient to be able to store a user-configurable number of revisions per file.

This paper discusses a possible solution to the problem as implemented via a Bourne shell script. While this may not be your scripting language of choice, the concepts exemplified herein are fairly straightforward. Implementation in another programming language is left as an exercise for the user.

2 Limitations

There are a few limitations to this script. A 2004.2 or later Perforce server is required. Earlier server releases do not support post-submit commit triggers. There are issues related to quoting special characters in file paths. These cause problems in filenames in general. The characters known to cause problems are as follows: % () # \$ * Compatibility testing for your choice of special characters should be performed before implementation.

Computations are based on the filesystem dates of files, which in general is the same as the dates that the file revisions were submitted to Perforce. Modification of the versioned file tree outside of Perforce will affect this and should be undertaken with extreme care.

Purged revisions that are the source revision of lazy copies will cause the lazy copy revisions to have the stub contents and to therefore fail a subsequent verify with a “BAD!” message. This problem can be alleviated with a “p4 obliterate -z” of the file prior to the revision replacement or “p4 verify -v” of the lazy copy afterwards.

3 Input

The script takes two command line arguments as its inputs: the changelist number and the number of revisions to keep for matching files.

4 Script internal setup

Within the script there are a number of configuration options and settings. These control basic functionality and are as follows:

Variable	Description
MSG	Message for small stub used to replace large files
DEPOT_PATH	Depot path that the trigger is to match. The trigger is fired when a file in the changelist matches the trigger path. Without this check it will then run for all files in the changelist regardless of whether they match the trigger rule individually or not.
ROOT	P4ROOT location -- Absolute path of the P4ROOT
LOG	Where to log program output
BLOCKS	Number of disk blocks a file must use for it to count as a large file
P4	Path to the Perforce client executable
PORT	Perforce port to use for server connections
CLIENT	Perforce client specification to use

5 Perforce setup

The Perforce trigger specification needs a line with the setup information, including the input values described above. For example:

```
rev_purge commit //depot/bin/... "/path/purge_trigger.sh %changelist% 10"
```

6 The script

First, variables are set up and values that are passed in are converted to local variables:

```
MSG="This file was removed at `date` to save disk space!"
DEPOT_PATH="//depot/bin/..."
ROOT=/perforce/purge
LOG=/perforce/purge/purge.log
BLOCKS=10
P4=/usr/team/reb/test/p4
PORT=7890
CLIENT=reb_play
CHANGE=$1
REVS2KEEP=$2
date >> $LOG
P4="$P4 -p $PORT -c $CLIENT"
```

Then, the files within the changelist are processed. Each is checked to see if it matches the storage criteria for processing. If it does, the revisions are sorted by date and those that exceed the configured number of disk blocks are counted. Large revisions that exceed the configured number are then converted to small stub files. The MD5 hash values are then fixed up, to alleviate future “p4 verify” problems. The meat of the script is as follows:

```
# For each file strip off hash version number & other non-filename parts.
# Dereference spaces with "%" to keep file paths a single "for" argument
for FILE in ` $P4 files $DEPOT_PATH@=$CHANGE | \
  sed -e 's/\(.*\)\#[0-9]* - .*$/\1/' -e 's/ /%/g' `
do
  # Undo the dereference and remove leading //
  FILE=`echo $FILE | sed -e 's/%/ /g' -e 's|^//|'|`"
  #
  # If stored as a ,d file it's a file/revision file. Otherwise ignore.
  if [ -d $ROOT/${FILE},d ]
  then
    # Go through each file in the revision dir in time order.
    THISREV=0
    for REV in `ls -lt $ROOT/${FILE},d/1.*.gz`
    do
      # Figure out if it's big or small, count, process...
      SIZE=`ls -s $REV | sed -e 's/\(.*[0-9]*\)\ .*/\1/'`
      if [ $SIZE -gt $BLOCKS ]
      then
        let THISREV=$THISREV+1
        if [ $THISREV -gt $REVS2KEEP ]
        then
          # Do the dirty work - delete the big file!
          echo "$MSG" | gzip -9 > $REV
          #
          # Calculate version
          VER=`echo $REV | sed -e 's|/. *1\.\([0-9]*\)\.gz|\1|'`
          #
          # Fix up the MD5 hash
          p4 verify -v "//$FILE#$VER,$VER"
          echo "$REV BIG $THISREV DELETED" >> $LOG
        else
          echo "$REV BIG $THISREV of $REVS2KEEP" >> $LOG
        fi
      fi
    done
  fi
done
#
# set the exit status!
exit 0
#
# END
```

7 Sample run:

For our example, prior to running the script for the first time, there are 11 revisions of a file in the versioned file tree. The directory listing looks like this:

```
-rw-r--r-- 1 reb team 4288726 Feb 26 12:43 1.1.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:45 1.2.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:46 1.3.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:47 1.4.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:48 1.5.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:52 1.6.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:53 1.7.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:53 1.8.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:53 1.9.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:54 1.10.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:56 1.11.gz
```

The trigger is then put into place. Subsequent submission of revision 12 of the file results in the following log entries being produced by the script:

```
Sat Feb 26 12:57:51 PST 2005
/perforce/purge/depot/bin/binfile,d/1.12.gz BIG 1 of 10
/perforce/purge/depot/bin/binfile,d/1.11.gz BIG 2 of 10
/perforce/purge/depot/bin/binfile,d/1.10.gz BIG 3 of 10
/perforce/purge/depot/bin/binfile,d/1.9.gz BIG 4 of 10
/perforce/purge/depot/bin/binfile,d/1.8.gz BIG 5 of 10
/perforce/purge/depot/bin/binfile,d/1.7.gz BIG 6 of 10
/perforce/purge/depot/bin/binfile,d/1.6.gz BIG 7 of 10
/perforce/purge/depot/bin/binfile,d/1.5.gz BIG 8 of 10
/perforce/purge/depot/bin/binfile,d/1.4.gz BIG 9 of 10
/perforce/purge/depot/bin/binfile,d/1.3.gz BIG 10 of 10
/perforce/purge/depot/bin/binfile,d/1.2.gz BIG 11 DELETED
/perforce/purge/depot/bin/binfile,d/1.1.gz BIG 12 DELETED
```

The versioned file tree now looks like this, with the first two file revisions being truncated and only 10 “large” file revisions remaining:

```
-rw-r--r-- 1 reb team 108 Feb 26 13:08 1.1.gz
-rw-r--r-- 1 reb team 108 Feb 26 13:08 1.2.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:46 1.3.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:47 1.4.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:48 1.5.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:52 1.6.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:53 1.7.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:53 1.8.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:53 1.9.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:54 1.10.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:56 1.11.gz
-rw-r--r-- 1 reb team 4288726 Feb 26 12:57 1.12.gz
```

A listing of one of the stub file revisions shows the new file content:

```
p4 print //depot/bin/p4d#2
//depot/bin/p4d#2 - edit change 2 (xbinary)
This file was automatically removed at Sat Feb 26 13:08:27 PST 2005 in
order to save disk space!
```

Since all 12 submitted revisions were of the same file, the MD5 hash values were originally the same. After the purge, however, a “p4 verify” of the file shows different values for the first two revisions, and no error. The verify output looks like this:

```
//depot/bin/p4d#12 - edit change 12 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#11 - edit change 11 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#10 - edit change 10 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#9 - edit change 9 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#8 - edit change 8 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#7 - edit change 7 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#6 - edit change 6 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#5 - edit change 5 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#4 - edit change 4 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#3 - edit change 3 (xbinary) 3C8753F93DBF76656509830DC53
//depot/bin/p4d#2 - edit change 2 (xbinary) 83A3332ECF00DD2AA3C1FFE0021
//depot/bin/p4d#1 - add change 1 (xbinary) 83A3332ECF00DD2AA3C1FFE0021
```

In this example the two stub files were changed at the same time, by the same changelist, so the new MD5 hash values are identical. The message value, however, has the date and time embedded. Subsequent file replacement at a different date and time will result in different MD5 hash values for replaced stub files.

8 Future improvements

It would be preferable to have an option to archive to-be-deleted files to an offline storage device instead of just deleting them. This is trivial to implement without notifying Perforce. Simply code a move of the file instead of deleting it. Notifying Perforce of the change, however, would require a Perforce journal patch to be applied to the server for each file being moved. Generating the data necessary to make such a change to the Perforce db.rev table would add a great deal of complexity to this otherwise fairly simple script.

9 Conclusions

The code fragments above are intended as a guide to one of the more unusual possible uses of Perforce triggers. Since this script deletes files from the versioned file tree it should be implemented and used with extreme care. The limitations discussed above should be seriously considered prior to any production use of this code or any derivative of it. The complete and unedited script can be obtained by requesting a copy from the Perforce Technical Support department.