PERFORCE

Using Perforce for Distributed Versioning

2015.1

March 2015

Using Perforce for Distributed Versioning
2015.1

March 2015

Table of Contents

About this manual

This guide tells you how to use the distributed versioning features of Perforce. Distributed versioning allows you to work disconnected from a shared central server. If you're new to version management systems, you don't know basic Perforce concepts, or you've never used Perforce before, read *Introducing Perforce* before reading this guide.

Getting started with Perforce distributed versioning

This guide assumes you are an experienced Perforce user. As such, you should understand the following concepts: changelist, depot, client workspace, sync, and submit.

Perforce documentation

This guide (*Using Perforce for Distributed Versioning*), the P4 Command Reference, and the `p4 help` command are the primary documentation sources for working with distributed versioning. For documentation on other Perforce applications, see the documentation web page at http://www.perforce.com.

For specific information about…	See this documentation
The basics of Perforce	*Introducing Perforce*
Installing and administering the Perforce service, including user management, security settings and configuring distributed environments that include proxies, replicas, and edge servers	*Perforce Server Administrator's Guide: Fundamentals* and *Perforce Server Administrator's Guide: Multi-site Deployment*
`p4` command line flags and options (reference)	*P4 Command Reference*, `p4 help`
P4V, the cross-platform Perforce Visual Client	*P4V User's Guide*
Perforce plug-ins and integrations	IDEs: Using IDE Plug-ins Defect trackers: Defect Tracking Gateway Guide Others: online help from the Perforce menu or web site
Developing custom Perforce applications using the Perforce C/C++ API	*C/C++ API User's Guide*
Working with Perforce in Ruby, Perl, Python, and PHP	*APIs for Scripting*

Please give us feedback

We are interested in hearing about your experience using this guide to learn about Perforce distributed versioning. Please let us know what you think; we can be reached at manual@perforce.com. To provide feedback on the distributed versioning features, contact support@perforce.com.

Introduction

This release of the Perforce Server introduces distributed versioning.

A distributed version control architecture gives users access to an entire repository of archived content — and changes to that content — from a personal server ("local server") on their local machine. This means that the entire history of a file is contained on each user's personal server. A user can manage versioned content without interacting with any other Perforce server unless desired. A user can also rewrite and revise history to discard unwanted intermediate information. In general, the distributed model allows users to work more experimentally, to try out changes and branch new streams, without fear of interfering with others' work.

Perforce distributed versioning functionality also provides a collaborative workflow, via shared servers ("remote servers"); these are classic Perforce servers that allow users to share their work with each other. In this model, users can work disconnected from the network and the shared servers until they're ready to copy content to or from the shared servers. Moreover, unlike other version control systems such as git, users can copy a subset of the shared server content to their personal server, rather than the entire shared server repository. In this "hub and spoke" model, the shared server becomes the center of communication for the surrounding personal servers.

This release also allows synchronization of content across multiple offices or teams. This might happen using the new `p4 fetch` and `p4 push` commands if the servers are networked or the new `p4 zip` and `p4 unzip` commands if they're not. Synchronization of content across sites is covered in the "Administering Perforce: Superuser Tasks" chapter of the *Perforce System Administrator's Guide: Fundamentals*.

This book, *Using Perforce for Distributed Versioning*, covers the following topics:

• Starting up a personal server — either empty or populated with files

• Fetching and pushing files between servers

• Branching

• Understanding remotes

• Rewriting history

• Mapping of Git commands to Perforce commands

Installation

Installing Perforce Server 15.1 on a Macintosh

1. Open a web browser.

2. Navigate to http://www.perforce.com/downloads.

3. Click the **Select** button.

4. Find the section titled **P4D: Server**.

5. Choose your operating system from the dropdown menu in that section.

6. Click the **Accept and Continue** button in that section.

7. Click the **Clients** tab.

8. Find the section titled **P4: CommandLine Client**.

9. Select your operating system from the dropdown menu in that section.

10. Click the **Accept and Continue** button in that section.

11. Open a Terminal window.

12. Make the downloaded files executable:

```
chmod +x Downloads/p4*
```

13. Move the files into a common execution path:

```
sudo mv Downloads/p4* /usr/local/bin/
```

Installing Perforce Server 15.1 on Linux

Without OS-specific packages

1. Open a web browser.

2. Navigate to http://www.perforce.com/downloads.

3. Click the **Select** button.

4. Find the section titled **P4D: Server**.

5. Click the **Accept and Continue** button in that section.

6. Click the **Clients** tab.

7. Find the section titled **P4: CommandLine Client**.

8. Select your operating system from the dropdown menu in that section.

9. Click the **Accept and Continue** button in that section.

10. Open a terminal window.

11. Make the downloaded files executable:

```
chmod +x _downloads_path_/p4*
```

12. Move the files into a common execution path:

```
sudo mv _downloads_path_/p4* /usr/local/bin/
```

With OS-specific packages

1. **Add the Perforce package repository.**

   For RPM packages, create a file called **/etc/yum.repos.d/perforce.repo** with the following content:

   ```
   [perforce]
   name=Perforce
   baseurl=http://package.perforce.com/yum/rhel/6/x86_64/
   enabled=1
   gpgcheck=1
   ```

   For the Debian package, create a file called **/etc/apt/sources.list.d/perforce.sources.list** with the following line:

   ```
   deb http://package.perforce.com/apt/ubuntu precise release
   ```

2. **Import the Perforce package signing key.**

   Run one of the following:

   a. **For RPM (run this command as root):**

   ```
   # rpm --import http://package.perforce.com/perforce.pubkey
   ```

   b. **For Debian:**

   ```
   $ wget -q http://package.perforce.com/perforce.pubkey -O- | sudo apt-key add -
   $ sudo apt-get update
   ```

For information about how to verify the authenticity of the signing key, see http://answers.perforce.com/articles/KB_Article/Public-Key-for-Installation-Packages.

3. **Install the perforce-server package.**

   a. **For CentOS and Red Hat, run (as root):**

      ```
      # yum install perforce-server
      ```

   b. **For Ubuntu, run:**

      ```
      $ sudo apt-get install perforce-server
      ```

4. **Install the perforce-cli package.**

   a. **For CentOS and Red Hat, run (as root):**

      ```
      # yum install perforce-cli
      ```

   b. **For Ubuntu, run:**

      ```
      $ sudo apt-get install perforce-cli
      ```

Installing Perforce Server 15.1 on Windows

Note	You need administrator privileges to install the server.

1. Open a web browser.

2. Navigate to http://www.perforce.com/downloads.

3. Click the **Select** button.

4. Find the section titled **P4D: Server**.

5. Choose your operating system from the dropdown menu in that section.

6. Click the **Accept and Continue** button.

7. Run the **.exe** file you downloaded

8. Accept all of the defaults.

This gives you the **p4d** executable (Perforce Server) and the **p4** executable (Perforce Client).

Initializing a Personal Server

This section describes how to start up a personal server, presenting two different approaches.

1. The first approach initializes an empty server. Choose this if you want to work in isolation on a personal server, developing and possibly branching code, and versioning locally. See <u>"Run p4 init" on page 7</u>.

2. The second approach copies content from another server to populate the newly initialized server with files and history; this is known as "cloning." This approach is best when working collectively on an existing project; users work on a set of project files that are managed on a shared server. The users make changes to the files on their personal server and then push the changes to a shared server. The shared server makes these changes available to other project users. At any given time, users can fetch the latest content from the shared server. See <u>"Run p4 clone" on page 9</u>.

Initialize an empty server

In this workflow, you invoke the `p4 init` command in your working directory to initialize a personal server and set it up with everything needed to start versioning files. Use this approach if you want to work in isolation on a personal server, developing and possibly branching code.

Read this first

In order to fetch from or push to a shared server, the case sensitivity of your personal server must match that of your shared server. When you run `p4 init`, Perforce attempts to set the case sensitivity of your personal server to match that of the shared server specified in your current `P4PORT` setting. If Perforce can't discover a shared server, the `p4 init` command will fail. You must then run this command:

```
p4 init -Cx
```

where `C0` sets the server to case-sensitive and `C1` sets it to case-insensitive; set the option to match the case sensitivity of the shared server with which you're communicating.

Similarly, in order to fetch from or push to a shared server, the Unicode support of your personal server must match that of the shared server. When you run `p4 init`, Perforce attempts to set the Unicode support of your personal server to match that of the shared server specified in your current `P4PORT` setting. If Perforce can't discover a shared server, Unicode support defaults to off. If you later want to turn Unicode support on, you can run this command:

```
p4d -xi -r /users/username/dvcsdir/.p4root
```

Run p4 init

Here is the `p4 init` command syntax:

```
p4 [-u user] [-d dir] [-c client] init [-h -q] [-c stream] [-Cx] [-xi -n]
```

`p4 init` includes a number of command-line arguments:

- To configure your personal server without Unicode support, pass the `-n` option.

- To have Perforce create the personal server's files in a directory other than the current directory, specify the directory with the `-d` option.

- Use the `-q` option to suppress informational messages.

- Use the `-c [stream]` option to create the specified stream as the mainline stream rather than the default `//stream/main`.

Directories and files

The `p4 init` command creates the following directories and files in the directory in which the command is invoked:

- `.p4root` - A directory containing the database files that will contain the metadata about files checked into Perforce.

- `.p4ignore` - A list of Perforce files Perforce shouldn't add or reconcile.

- `.p4config` - A file containing configuration parameters for the client-server connection.

In addition, the `p4 init` command does the following:

- Creates a `P4CLIENT` workspace. Note that the client option `allwrite` is set by default, making files writable without the need to check them out with `p4 edit` first. You must, however, issue a `p4 reconcile` command before shelving or submitting files.

- Creates a stream depot.

- Creates an initial stream, called `main`.

Add files

At this point, you are ready to add files to your server. You can create them, copy them and then run `p4 reconcile` — or `p4 rec` for short — to mark all of your source files to be added to Perforce and then `p4 submit` to submit them. If you are new to Perforce, see the "Managing Files and Changelists" chapter of the *P4 User's Guide*

Prepare to fetch and push content between servers

If you want to subsequently push your work to a shared server or fetch files from a shared server, you must create a remote spec with the `p4 remote` command. See Chapter 4, "Fetching and Pushing" on page 11 and Chapter 6, "Understanding Remotes" on page 19 for more information.

Initialize a server and populate it with files

This approach is best when working collectively on an existing project; users work on a set of project files that are managed on a shared server.

To start this process, users invoke the `p4 clone` command to obtain from the shared server a copy of the files associated with the project. This is a convenient way to ensure that users receive the set of files they need to participate in the project.

The user can then work on these files and periodically push changes back to the shared server from which the files were cloned. They can also periodically fetch to get the latest changes made by others to the shared server files.

Run p4 clone

Here is the `p4 clone` command syntax:

```
p4 [-u user] [-d dir] [-c client] clone [-m depth] [-v] -p port -r remote
p4 [-u user] [-d dir] [-c client] clone [-m depth] [-v] -p port -f filespec
```

`p4 clone` includes a number of command-line arguments:

- The `-d` option specifies the directory where you want to create the server's files. If you don't specify this option, the files are created in the current directory.

- The `-p` option specifies the address of the shared server you wish to clone from.

- The `-m` option performs a shallow fetch; only the last number of specified revisions of each file are fetched.

- The `-r` option specifies the remote spec installed on the remote server to use as a template for the clone and stream setup. You can obtain the name of the desired remote from the shared server administrator or run the `p4 remotes` command against the shared server to obtain a list of candidates to choose from. At the time of cloning, Perforce will copy the remote from the shared server to the personal server and name it `origin`. For more information on remotes, see Chapter 6, "Understanding Remotes" on page 19.

- The `-f` option specifies a filespec in the remote server to use as the path to clone; this path will also be used to determine the stream setup in the local server. You can specify the `-f` option or the `-r` option but not both.

- The `-v` option specifies verbose mode.

- The `-c` option lets you customize the name of the stream that `p4 clone` creates.

Directories and files

The `p4 clone` command creates all the directories and files that the `p4 init` command creates. In addition, `p4 clone` creates a remote called `origin` on the personal server. A remote is a mapping of files on a local server to files on a remote server and is required for fetching, pushing and cloning; it describes exactly which files should be copied from a local server to a remote server or vice-versa. It is described in detail in Chapter 6, "Understanding Remotes" on page 19.

Get the latest changes

To update your personal server with the latest changes from the shared server, run `p4 fetch`. See Chapter 4, "Fetching and Pushing" on page 11 for more information.

Fetching and Pushing

Fetching and pushing lie at the heart of a collaborative distributed workflow; they enable users to perform two major tasks:

1. To copy work developed on their personal (local) servers to shared (remote) servers and vice-versa.

2. To obtain and work with a subset of a shared server's entire repository.

Fetch and push also allow shared servers to exchange files and history with each other, transferring between servers both changelists and versioned file content.

Fetch and push are to the distributed versioning model what sync and submit are to classic Perforce's central server model.

The `p4 fetch` command copies the specified set of files and their history from a shared server into a personal server. The `p4 push` command copies the specified set of files, and their history from a personal server to a shared server. Both commands are atomic: either all the specified files are fetched or pushed or none of them are.

If a `p4 push` command fails after it has begun transferring files to the shared server, it will leave those files locked on the shared server. The `p4 opened` command will display '**locked**', and the files cannot be submitted by any other user. If the `p4 push` command cannot be quickly retried, you can use the `p4 unlock -r` commmand to unlock the files on the shared server.

The `p4 push` command is not allowed if there are unsubmitted changes in the server from which you're pushing; use `p4 resubmit` to resubmit those changes first, or discard the shelves with `p4 shelve -d` if they are not wanted. For more information on `p4 unsubmit` and `p4 resubmit`, see Chapter 7, "Rewriting History" on page 27.

To monitor the progress of the fetch or push, pass the `-I` option to the command:

```
p4 -I fetch
p4 -I push
```

Configure security for fetching and pushing

In order to fetch and push between a personal server and a shared server, the respective servers must have authentication and access permissions configured correctly:

• The user name must be the same on both servers.

• The user must exist on the shared server.

• The user must have read (fetch) and write (push) permission on the shared server.

• The server.allowpush and server.allowfetch configuration settings must be set to on (they're off by default) on both the shared server and the personal server. See the command `p4 help configurables` for more information.

• The user must be logged into the shared server via `p4 login`.

Specify what to copy

As described in , you typically specify which files will be pushed or fetched by listing depot paths in the `DepotMap` field of the remote spec. You can further narrow the set of files to be fetched or pushed with one of two command-line arguments: one specifying a filespec pattern and the other specifying a stream (with the `-S` option).

If a filespec or stream name is provided, and the remote spec uses differing patterns for the local and remote sides of the DepotMap, the filespec argument or stream name must specify the files using the personal server's depot syntax. Note that the filespec must always be provided using depot syntax, not client syntax nor filesystem syntax. For more information, see .

- To specify a remote you pass the `-r` option and the name of the remote to the `p4 fetch` or `p4 push` command. If `-r` is not specified, the default is `-r origin`:

```
p4 fetch -r markm-remote
```

- To specify a filespec you pass a filespec pattern to the `p4 fetch` or `p4 push` command.

```
p4 fetch //depot/projectx/...
```

- To specify a stream you pass the `-S` option to the `p4 fetch` or `p4 push` command. Note that the stream must be listed in a depot mapping in your remote spec.

```
p4 fetch -S //stream/dev
```

  where `dev` is the name of the stream on your personal server

Note that when you specify a filespec or a stream, Perforce Server cannot use the performance optimization provided by the remote spec.

Unlike other versioning engines such as Git, you do not have to fetch or push the entire contents of the shared server's repository; rather, you can fetch or push whatever subset of the respository you like. You specify this subset in the remote spec or at the command line of the fetch or push command.

Fetch a limited subset of history

If you have a server with a lot of history you may only want to fetch the latest few revisions to save on local storage. To do so, use the `-m N` option:

```
p4 fetch -m 5
```

This specifies that the server perform a shallow fetch, fetching only the last 5 revisions of each file. You can also take a slice of your history as noted above.

Fetching, pushing and changelists

When changelists are added to the personal server during a fetch or to a shared server during a push they are given new change numbers but they retain the same description, user, date, type, workspace and set of files.

When the files are added to the personal server during a fetch or to the shared server during a push, they are kept in their same changelists, as new revisions starting after the current head. The new revisions retain the same revision number, file type, action, date, timestamp, digest, and file size.

Although the changelists are new submitted changelists in the personal server for a fetch and in the shared server for a push, none of the submit triggers are run in the personal server or the shared server. For more information about submit triggers, see the "Scripting Perforce" chapter in the *Perforce Server Administrator's Guide: Fundamentals*.

If a particular changelist includes some files that match the filespec or stream restriction, and other files that do not, then only the matching files are included in the fetch or push. Note that if a remote spec is also provided, only the files that match the restriction and are mapped by the remote spec are included in the fetch or push. In other words, not all files in the changelist will necessarily be fetched or pushed. For example, consider the following DepotMap in a remote spec:

```
//stream/main/p4/... //depot/main/p4/...
```

Suppose you have a changelist with the following files:

```
//stream/main/p4/foo
//stream/jam/bar
```

Only `//stream/main/p4/foo` will be pushed or fetched, as it matches the remote spec mapping.

When things go wrong

Fetch and push have a couple of failure scenarios that require action on the part of the user or shared server administrator.

Access denial

If there are permissions or authentication problems for any of the reasons outlined in the section "Configure security for fetching and pushing" on page 11, the fetch or push will fail with a message from the shared server. The user or shared server administrator must then address the problem before the user can attempt the fetch or push again.

History does not fit

A fetch is only allowed if the files being fetched fit cleanly into the personal server, building precisely on a shared common history. If there are any conflicts or gaps, the fetch is rejected. Otherwise, the changelists from the shared server become new submitted changelists in the personal server.

If the fetch fails, this is probably because you have attempted to fetch revisions from the shared server to your personal server that are in conflict with revisions you've submitted to your personal server.

Chapter 7, "Rewriting History" on page 27 explains what to do to resolve this situation.

> **Note** As a best practice, you should generate a report of conflicts before attempting a fetch, with the `-n` command-line option.

Files with filetype modifiers +k, +l or +S

Files with the filetype modifiers `+k`, `+l`, or `+S` require special considerations.

- Files of type `+k` have their digests cleared when pushed or fetched. This means certain cross-server merge conflicts are not detected. To re-generate the digests in the target server after the push or fetch, use `p4 verify`.

- When pushing or fetching files of type `+l`, the new files are not added to the target server if the files are currently open by a pending changelist in that server. Under these circumstances, Perforce raises an error.

- When pushing or fetching files of type `+S`, old archives that exceed the specified limit are not purged by the push or fetch command.

Using triggers with fetch and push

Perforce triggers are user-written programs called by a Perforce server when certain operations are performed. You use triggers to extend or customize Perforce functionality. Triggers are of different types, depending on the event that causes the trigger to execute. Command type triggers allow some action to be taken before or after a given command.

You may use command type triggers with `p4 push` and `p4 fetch` commands. The command field of the trigger definition contains the location of the trigger and any arguments you want to pass to it. For example:

```
myTrig command post-user-push "/usr/bin/perl /usr/bin/test.pl"
```

For more information, see "Triggering before or after commands" in the Scripting Perforce: Triggers and Daemons chapter of the *Perforce Server Administrator's Guide: Fundamentals*.

Streams and Branching

Streams are Perforce's term for branches. They are variant versions of a body of code. You can read more about them in the "Codelines, Branching and Streams" chapter of the *P4 User's Guide*.

When using a personal server created by `p4 init` or `p4 clone`, Perforce uses streams as containers for your code. Perforce will create a stream named 'main' to contain the content created or cloned. If, in working with your personal server, you need to create new streams—also known as branching— you can do so with the `p4 switch` command. You can then use merge and copy as normal to move individual changes between streams.

> **Note**
>
> Although you can switch between streams on a shared server, you cannot use `p4 switch` to create new streams on shared servers.

List streams

To display the current stream, issue `p4 switch` with no options.

```
$ p4 switch
main
```

main is the default stream created by the `p4 clone` command.

Pass the `-l` option to p4 switch to list all known streams.

```
$ p4 switch -l
main *
```

The asterisk indicates the current stream. As we haven't yet created any other streams, main is the only one listed and is the current stream.

Create streams

`p4 switch -c [stream]` creates a new stream and populates it with a copy of all the files in the current stream.

```
$ p4 switch -c dev
dev
```

A quick comparison reveals that the two streams contain identical files:

```
$ p4 diff2 //stream/main/... //stream/dev/...
==== //stream/main/a/test1.txt#1 (text) - //stream/dev/a/test1.txt#1 (text) ==== identical
==== //stream/main/a/test2.txt#1 (text) - //stream/dev/a/test2.txt#1 (text) ==== identical
```

The **-P** *parent* option specifies that **p4 switch -c** should create a new stream with the specified stream as its parent, rather than the default of the current stream; thus the new stream will be populated with the files from the specified parent stream, rather than the files from the current stream.

```
$ p4 switch -c -P main child_of_main
child_of_main
```

As the following output demonstrates, **//stream/main** is the parent of **//stream/child_of_main**:

```
$ p4 stream -o //stream/child_of_main
# A Perforce Stream Specification.
#
#  Stream:      The stream field is unique and specifies the depot path.
#  Update:      The date the specification was last changed.
#  Access:      The date the specification was originally created.
#  Owner:       The user who created this stream.
#  Name:        A short title which may be updated.
#  Parent:      The parent of this stream, or 'none' if Type is mainline.
#  Type:        Type of stream provides clues for commands run
#               between stream and parent.  Five types include 'mainline',
#               'release', 'development' (default), 'virtual' and 'task'.
#  Description: A short description of the stream (optional).
#  Options:     Stream Options:
#                       allsubmit/ownersubmit [un]locked
#                       [no]toparent [no]fromparent mergedown/mergeany
#  Paths:       Identify paths in the stream and how they are to be
#               generated in resulting clients of this stream.
#               Path types are share/isolate/import/import+/exclude.
#  Remapped:    Remap a stream path in the resulting client view.
#  Ignored:     Ignore a stream path in the resulting client view.
#
# Use 'p4 help stream' to see more about stream specifications and command.

Stream: //stream/child_of_main

Update: 2015/02/06 10:57:04

Access: 2015/02/06 10:57:04

Owner: jschaffer

Name: //stream/child_of_main (created by switch command)

Parent: //stream/main

Type: development

Options: allsubmit unlocked toparent fromparent mergeany

Paths:
 share ...
```

Switch between streams

To switch between streams issue this command:

```
p4 switch other_stream
```

When switching to a different stream, the `p4 switch` command first runs `p4 reconcile` to determine which files have been modified in the current stream. It then shelves any changed files for safekeeping.

After switching to a new stream, switch syncs your client workspace to the head of the new stream, and unshelves any files that were open and any changelists pending the last time you used that stream.

Note	You cannot switch to a new stream if files are open in a numbered changelist. If files are open in the default changelist, they will be shelved and reverted prior to switching to the new stream, and will be automatically unshelved when switching back to this stream.

To switch to a different stream and bring changed files with you from the current stream, pass the `-r` option:

```
$ p4 switch -r
```

Here's the list of all of our streams:

```
$ p4 switch -l
child_of_main
dev *
gui
main
```

Here's the stream we're currently in:

```
$ p4 switch
dev
```

Here are the files currently open in **//stream/dev**:

```
$ p4 files //stream/dev/...
//stream/dev/asciidoc/branching#1 - branch change 44 (text)
//stream/dev/asciidoc/fetching_and_pushing#1 - branch change 44 (text)
//stream/dev/asciidoc/initializing_a_server#1 - branch change 44 (text)
//stream/dev/asciidoc/intro#1 - branch change 44 (text)
//stream/dev/asciidoc/remote_spec#1 - branch change 44 (text)
//stream/dev/asciidoc/understanding_remotes#1 - branch change 44 (text)
//stream/dev/dvcs_commands/clone.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/fetch.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/init.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/push.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/remote.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/remotes.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/resubmit.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/switch.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/unsubmit.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/unzip.xml#1 - branch change 44 (text)
//stream/dev/dvcs_commands/zip.xml#1 - branch change 44 (text)
//stream/dev/dvcs_user_guide/00_preface.xml#1 - branch change 44 (text)
//stream/dev/dvcs_user_guide/git_to_perforce.xml#1 - branch change 44 (text)
//stream/dev/resubmit.xml#1 - branch change 44 (text)
```

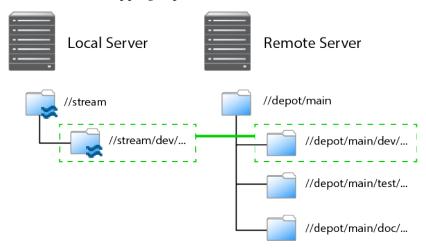Now we open new files in **dev**:

```
$ p4 add a b c
```

If we then issue the following command, we switch to the **gui** stream but bring over the content that
was changed in **dev**.

```
$ p4 switch -r gui
$ ls
a b c
```

Understanding Remotes

A remote describes how depot files are mapped between a local server and a remote server. (Local servers are also known as personal servers and a remote server is also known as a shared server.) A remote spec — which describes a remote — is created by the user and has a unique name. A remote is used with the `p4 push`, `p4 fetch`, and `p4 clone` commands to describe source and target directories. The following picture illustrates mapping depot files between a local and a remote server.

As depicted in the figure above, a remote holds file mappings between depot paths on the remote server and depot paths on the local server.

- For fetch and clone operations, it defines the files from the remote server that you want in your local server and specifies where you want them to reside.

- For a push operation, it defines the files from the local server that you want in the remote server and specifies where you want them to reside.

Remotes provide a convenient way to give you the exact files you need to work on a particular project. You can simply clone from a shared server, specifying the remote id of the remote that maps the desired files. These files are then copied to your personal server. Once they've cloned, you can use `p4 fetch` to refresh the files initially obtained with the `p4 clone` command.

Using remotes allows you to fetch a subset of all the files on the shared server. This is in contrast to other distributed versioning systems, such as Git, which require that you fetch all files.

Note that when you clone a set of files from a shared server by specifying a remote, Perforce creates a new remote named `origin` and copies the remote into your local system. Future invocations of `p4 fetch` do not need to pass in `-r` remote, as `origin` is now assumed to be the remote.

There are two different scenarios in which remotes are created:

- An administrator creates a remote on a shared server so that users can clone from this server and obtain the files they need to work on a project.

- You, the individual user, create one or more remotes on your personal server so that you can eventually push your work to and fetch files from one or more shared servers.

You can create a remote both on a shared server and on a personal server. An administrator would create a remote on a shared server to dictate which subset of the shared server's respository a personal

server retrieves when it clones from the shared server. After cloning, you use the **origin** remote on your personal server. You can then either edit the **origin** remote or create a different remote to control which streams the personal server fetches and pushes when using that remote.

Choose a remote

How you choose a remote depends on whether you're doing your initial clone or your daily fetching and pushing.

If you're cloning, run the **p4 remotes** command on the shared server from which you're cloning and choose the remote you want to work with. To look at the details of each remote, run **p4 remote -o**. Alternatively, you can obtain the id of the remote from a shared server administrator or project leader.

If you want the content of just one depot path, pass the filespec of the path by running **p4 clone -f**.

In a typical use case, you've cloned from a shared server and the remote has been copied to your personal server and named **origin**. Because **origin** is the default remote, you don't have to pass a remote id during subsequent fetches and pushes.

In the more complicated case, you're pushing to or fetching from multiple shared servers, in which case you would run **p4 remotes** on your personal server and choose from among the remotes based on which shared server you're fetching from or pushing to. Again, you can use **p4 remote -o** to get the details of each remote.

Create a remote

Remotes are described by *remote specifications* or *remote specs* for short. To create a remote, run the **p4 remote** command. This puts the remote specification or *spec* into a temporary file and invokes the editor configured by the environment variable **P4EDITOR**. You then edit the file to specify depot mappings and other information. Saving the file creates the remote spec.

To modify the remote, invoke **p4 remote** with the **remoteID** of the remote you want to modify; make changes in the editor to the remote spec and then save the file.

Example

In the following example, we get a list of remotes from a shared server, clone from the shared server using one of those remotes, show the resulting remote in the personal server — with the **p4 remotes** command — and then demonstrate that the path listed in the remote spec corresponds to the path passed to the clone command.

1. First, we query a shared server for a list of remotes:

```
$ p4 -p perforce:1666 remotes
bpendleton-dev  'To clone bpendleton's dev branch, use this remote spec. '
h_dev localhost:1666 'Created by hmackiernan. '
markm-remote2  'Created by markm. '
mw-dvcs localhost:1666 '[dvcs] Map main server components.  Created by mwittenberg. '
p4-client localhost:1666 'Created by cmclouth. '
```

2. Then we choose a remote and pass it to the clone command:

```
$ p4 clone -p perforce:1666 -r markm-remote2
Perforce db files in '/Users/jschaffer/.p4root' will be created if missing...
Perforce server info:
 Server initialized and ready to use.
Remote origin saved.
main

Changes were successfully fetched.
Remote origin saved.
Server jschaffer-dvcs-1422657971 saved.
```

3. Next we run **p4 remotes** against the personal server to show that we now have a remote called "origin," which is the renamed remote we cloned from the shared server:

```
$ p4 remotes
origin perforce:1666 'Description '
```

4. Next, we write the contents of the remote we passed to **p4 clone** to standard output to show the depot paths it specified in the **DepotMap** field:

```
$ p4 -p perforce:1666 remote -o markm-remote2
# A Perforce Remote Specification.
#
#  RemoteID:    The remote identifier.
#  Address:     The P4PORT used by the remote server.
#  Owner:       The user who created this remote.
#  Options:     Remote options: [un]locked, [no]compress.
#  Update:      The date this specification was last modified.
#  Access:      The date of the last 'push/fetch' on this remote.
#  Description: A short description of the remote server (optional).
#  LastFetch:   The last changelist that was fetched.
#  LastPush:    The last changelist that was pushed.
#  DepotMap:    Lines to map local files to remote files.

RemoteID: markm-remote2

Owner: markm

Options: unlocked compress

Update: 2014/12/11 11:15:15

Description:
 Created by markm.

LastFetch: default

LastPush: default

DepotMap:
 //depot/main/p4/msgs/... //depot/main/p4/msgs/...
```

5. Finally, we write the contents of the origin remote spec to standard out to demonstrate that the depot paths it specifies in the DepotMap field are identical to those of markm-remote2:

```
$ p4 remote -o origin
# A Perforce Remote Specification.
#
#  RemoteID:    The remote identifier.
#  Address:     The P4PORT used by the remote server.
#  Owner:       The user who created this remote.
#  Options:     Remote options: [un]locked, [no]compress.
#  Update:      The date this specification was last modified.
#  Access:      The date of the last 'push/fetch' on this remote.
#  Description: A short description of the remote server (optional).
#  LastFetch:   The last changelist that was fetched.
#  LastPush:    The last changelist that was pushed.
#  DepotMap:    Lines to map local files to remote files.

RemoteID: origin

Address: perforce:1666

Owner: jschaffer

Options: unlocked nocompress

Update: 2015/01/30 14:46:51

Description:
 Description

LastFetch: 996270

LastPush: 4024

DepotMap:
 //depot/main/p4/msgs/... //depot/main/p4/msgs/...
```

Notice that the `LastFetch` and `LastPush` values have changed to non-zero numbers to reflect the highest changelist numbers most recently fetched and pushed.

A closer look at a remote spec

The following is a sample remote spec, describing a remote named `server-main-darwin`:

```
# A Perforce Remote Specification.

RemoteID:       server-main-darwin

Owner:  bruno

Options:        unlocked compress

Update: 2014/11/21 08:21:32

Description:
        A fairly complete set of the mainline code for the widget, with the
        test harness limited to the darwin platform. Fetch or clone from
        this remote spec if you want to build and work with the mainline
        widget code on a darwin machine.

LastFetch:      default

LastPush:       default

DepotMap:
        //stream/main/widget/... //depot/main/widget/...
        //stream/main/widget-test/server/... //depot/main/widget-test/server/...
        //stream/main/widget-test/bin/... //depot/main/widget-test/bin/...
        -//stream/main/widget-test/bin/arch/... //depot/main/widget-test/bin/arch/...
        //stream/main/widget-test/bin/arch/darwin90x86_64/... //depot/main/widget-test/bin/arch/
darwin90x86_64/...
        //stream/main/widget-doc/code/... //depot/main/widget-doc/code/...
```

The following table describes the remote spec in more detail:

Entry	Meaning
RemoteID	The remote identifier.
Address	The P4PORT used by the remote server.
Owner	The user who created this remote.
Options ([un]locked, [no]compress)	The unlocked option setting means people other than the owner can update the spec. The compress option setting means that when files are fetched or pushed they're compressed, as a performance optimization. You would only set this option to uncompress if you were fetching or pushing binary files that were already in a compressed format.
Update	The date this specification was last modified.
Access	The date of the last push or fetch on this remote.
Description	A short description of the remote server (optional).

Entry	Meaning
`LastFetch`	The last changelist that was fetched. If set to default, means no fetches have yet occurred.
`LastPush`	The last changelist that was pushed. If set to default, means no pushes have yet occurred.
`DepotMap`	The lines to map local files to remote files. The file paths on the left-hand side are on the local server. The file paths on the right-hand side are on the remote server.

Remote specs give you the full power of Perforce client view syntax. For details, see the section "Defining client workspaces" in the chapter [Configuring P4](#) in the *[P4 User's Guide](#)*. Below is some basic information about creating a remote spec.

Specify mappings

Remote specs consist of one or more mappings. Each mapping has two parts:

1. The left-hand side specifies one or more files on the local server.

2. The right-hand side specifies one or more files on the remote server.

Although the two sides don't have to name identical paths, they can.

Enclose paths with spaces in quotation marks.

Using wildcards in remote specs

To map groups of files in remote specs, you use Perforce wildcards (`*`, `...`). Any wildcard used on the remote side of a mapping must be matched with an identical wildcard in the mapping's local side. You can use the following wildcards to specify mappings in your remote spec:

Wildcard	Description
`*`	Matches anything except slashes. Matches only within a single directory. Case sensitivity depends on your platform.
`...`	Matches anything including slashes. Matches recursively (everything in and below the specified directory).

Now consider another remote spec's simple DepotPath:

```
//stream/main/... //depot/main/...
```

All files in the remote server's depot path are mapped to the corresponding locations on the local server. For example, the remote server file **//depot/main/widget-test/server.txt** is mapped to the local server file **//stream/main/widget-test/servert.txt**.

Mapping part of the depot

If you are interested only in a subset of the depot files on the remote server, map only that portion. Reducing the scope of the local server's files also ensures that your commands do not inadvertently affect the entire depot. To restrict the local server scope, map only part of the remote server depot to the local server.

Example 6.1. Mapping part of the remote server depot to the local server.

```
Remote Spec:

//stream/main/... //depot/main/widget-doc/code/...
```

In this case, Perforce server will map only the remote server files under the code subdirectory to the local server's **//stream/main** directory.

Mapping files to different locations on the local server

Remote specs can consist of multiple mappings; these map portions of the remote server file tree to different parts of the local server. If there is a conflict in the mappings, later mappings have precedence over earlier ones.

Example 6.2. Multiple mappings in a single local server

The following remote spec ensures that release notes in the remote p4-doc folder reside in the local server in a top-level folder called doc.

```
Remote Spec:

  //stream/main/src/... //depot/main/p4/...
  //stream/main/doc/... //depot/main/p4-doc/relnotes/...
```

Excluding files and directories

Exclusionary mappings enable you to exclude files and directories from being mapped to a local server. To exclude a file or directory, precede the mapping with a minus sign (-). Whitespace is not allowed between the minus sign and the mapping.

Example 6.3. Using a remote spec to exclude files from a local server.

Suppose you're working on a game project and you don't need the art files to be local:

```
Remote Spec:
  //stream/main/... //my_game/...
  -//stream/main/art/... //my_game/art/...
```

Rewriting History

Perforce allows you rewrite the history of the changes in your server. There are two reasons why you would want to rewrite history:

1. To resolve conflicts between a local server's file history and a remote server's file history that arise when fetching or pushing.

2. To revise local work: correcting mistakes, clarifying intent, and streamlining the local commit history by consolidating intermediate changes.

Resolve conflicts by rewriting local history

If there are conflicts between a local server's file history and a remote server's file history, a fetch will fail and report the conflict. This happens when you've changed some files in your personal server at the same time that someone else has changed those files in the shared server.

In this situation, you run `p4 fetch -u`. This does the following:

1. Unsubmits and creates a shelved changelist for the current local changes.

2. Fetches the remote work from the shared server.

The user then runs `p4 resubmit -m` to resubmit and automatically merge the conflicting local changes.

If your conflict(s) involved the same line or lines then `p4 resubmit -m` will fail and you will need to:

1. Run `p4 resolve` to resolve the conflict(s).

2. Run `p4 resubmit -Rm` to resume the resubmit.

Consider the following example:

1. User A clones from a shared server, bringing down revision 4 of `//stream/main/foo.c` (`//stream/main/foo.c#4`).

2. User A edits `foo.c` and then submits it, creating `//stream/main/foo.c#5`.

3. In the meantime, User B, has made two edits to `//stream/main/foo.c` and pushed them to the shared server. The shared server is now at revision 6 (`//stream/main/foo.c#6`).

4. User A attempts to push their change to the shared server, but the push fails because the file version history doesn't fit.

5. User A must now run `fetch -u`, which unsubmits and shelves User A's revision 5, and fetches revisions 5 and 6 from the shared server.

6. User A now runs `resubmit -m`. User A's change, originally numbered 5, is submitted as revision 7.

7. User A pushes their change to the shared server. The push succeeds.

Rewrite history to revise local work

This section examines two scenarios in which you might want to revise local work by rewriting history.

Scenario 1: You forgot to map a file

Suppose you wrote a new class in C++: **src/module/UserUtils.cpp** and it uses the header file **inc/UserUtils.h**. You then issue this command:

```
$ p4 submit UserUtils.cpp
```

Your build script complains about the missing include file **UserUtils.h**. To fix this, you would issue the following commands:

```
p4 unsubmit UserUtils.cpp
p4 resubmit -e
```

Now **UserUtils.cpp** is open. You would then run:

```
$ p4 add -c NNN UserUtils.h
$ p4 resubmit -Re
```

Now the permanent history shows that your change contains both **UserUtils.cpp** and **UserUtils.h**.

Scenario 2: Combine two changes together to remove "noise" from the history

Suppose you add a feature in change *NNN*. A reviewer finds a problem with it, so you make another change to fix the problem. Then you realize that the second change is just adding "noise" to the history.

To fix this, you would do the following:

(We assume your first change is *NNN* and your second change is *NNN+1*)

1. Unsubmit both changes:

    ```
    p4 unsubmit //…@NNN
    Change NNN+1 unsubmitted and shelved
    Change NNN unsubmitted and shelved
    ```

2. Start the partially-interactive resubmit process:

    ```
    `p4 resubmit -e`
    ```

    Now change *NNN* is open for edit.

3. Make the change you originally made in the second change.

4. Update the change description:

```
p4 change NNN
```

5. Resume the resubmit process:

```
`p4 resubmit -Re`
```

Now the second change is open for edit but you don't need it. You can demonstrate this to yourself by running **p4 resolve**, **p4 diff** and **p4 revert -a** to see that nothing is changed by the second change.

6. Get rid of the second change:

```
p4 shelve -d -c NNN+1
p4 change -d -c NNN+1
```

Alternatively, to get rid of the second change you could run **p4 resubmit -i** and choose "d".

Git:Perforce Command Mappings

The following table maps Git commands to their corresponding Perforce commands:

Git Command	Perforce Command
`git add`	`p4 reconcile`
`git branch`	`p4 switch -l`
`git checkout --orphan` *`new_branch`*	`p4 switch -cm` *`new_stream`*
`git checkout` *`branch`*	`p4 switch` *`stream`*
`git clone` *`repository`*	`p4 clone -p` *`host:port`* `-r` *`remote`*
`git commit`	`p4 submit`
`git init`	`p4 init`
`git merge` *`branch`*	`p4 merge --from` *`stream`*
`git pull`	`p4 fetch -u -r` *`remote`* `-S` *`stream`*
`git pull --all`	`p4 fetch -u`
`git push`	`p4 push -r` *`remote`* `-S` *`stream`*
`git push --all`	`p4 push`
`git rebase`	`p4 unsubmit` followed by `p4 resubmit`
`git remote`	`p4 remotes`
`git remote add` *`new_remote repository`*	`p4 remote` *`new_remote`*
`git status`	`p4 status`

For more details on Perforce commands, see the *P4 Command Reference*.

Appendix   # License Statements

Perforce software includes software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).