

PERFORCE

Using Helix for Distributed Versioning

2016.2

October 2016

Using Helix for Distributed Versioning 2016.2

October 2016

Copyright © 2015-2016 Perforce Software.

All rights reserved.

Perforce software and documentation is available from <http://www.perforce.com/>. You can download and use Perforce programs, but you can not sell or redistribute them. You can download, print, copy, edit, and redistribute the documentation, but you can not sell it, or sell any documentation derived from it. You can not modify or attempt to reverse engineer the programs.

This product is subject to U.S. export control laws and regulations including, but not limited to, the U.S. Export Administration Regulations, the International Traffic in Arms Regulation requirements, and all applicable end-use, end-user and destination restrictions. Licensee shall not permit, directly or indirectly, use of any Perforce technology in or by any U.S. embargoed country or otherwise in violation of any U.S. export control laws and regulations.

Perforce programs and documents are available from our Web site as is. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by Perforce Software.

Perforce Software assumes no responsibility or liability for any errors or inaccuracies that might appear in this book. By downloading and using our programs and documents you agree to these terms.

Perforce and Inter-File Branching are trademarks of Perforce Software.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

Any additional software included within Perforce software is listed in [License Statements on page 45](#).

Table of Contents

Chapter 1	Preface	1
	Helix documentation	1
	Syntax conventions	2
	Please give us feedback	2
Chapter 2	Introduction	3
	Centralized and distributed architecture	3
	How servers relate to each other	6
	Putting it all together	7
	Server-to-server relationships	7
	Client-to-server relationships	8
	Command line aliasing	8
Chapter 3	Installation	9
	Installing Helix Versioning Engine 16.1 on Mac OS X	9
	Installing Helix Versioning Engine 16.1 on Linux	9
	Without OS-specific packages	9
	With OS-specific packages	9
	Installing Helix Versioning Engine 16.1 on Windows	10
Chapter 4	Initializing a Server	11
	Initialize an empty server	11
	Read this first	11
	Run p4 init	12
	Directories and files	12
	Add files	12
	Prepare to fetch and push content between servers	12
	Initialize a server and populate it with files	13
	Run p4 clone	13
	P4PORT meaning before and after a clone	14
	Directories and files	14
	Get the latest changes	14
Chapter 5	Fetching and Pushing	15
	Configure security for fetching and pushing	15
	Specify what to copy	16
	Fetch a limited subset of history	16

What do fetch and push copy?	17
Attribute interoperability with 15.1	17
Fetching, pushing, and changelists	17
Fetch and push a shelved changelist	18
Track a changelist's identity from server to server	18
Workflow 1: Let Helix generate global changelist IDs	18
Workflow 2: Enter global changelist ID manually	19
Track who pushed, fetched, or unzipped a changelist	19
Fetching and pushing fixes	19
Fetching and pushing integration history	20
Configure server to limit storage of archive revisions	21
ArchiveLimits: entries	21
Per-server identities	23
When things go wrong	23
Access denial	23
History does not fit	23
Support for exclusive locking in personal servers	23
Using triggers with fetch and push	24
Chapter 6 Streams and Branching	25
List streams	25
Create streams	25
Switch between streams	27
Chapter 7 Understanding Remotes	29
Choose a remote	30
Create a remote	30
Example	30
A closer look at a remote spec	33
Specify mappings	35
Using wildcards in remote specs	35
Mapping part of the depot	36
Mapping files to different locations on the personal server	36
Excluding files and directories	36
Forward login to shared server	37
Chapter 8 Rewriting History	39
The tangent depot	39
Resolve conflicts by rewriting local history	39
Rewrite history to revise local work	40
Scenario 1: You forgot to map a file	40
Scenario 2: Combine two changes to remove "noise" from the history	40

Chapter 9	Git:Helix Command Mappings	43
	License Statements	45

This guide tells you how to use the distributed versioning features of Helix. 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 Helix concepts, or you've never used Helix before, read [Introducing Helix](#) before reading this guide.

Helix documentation

The following table lists and describes key documents for Helix users, developers, and administrators. For complete information see the following:

<http://www.perforce.com/documentation>

For specific information about...	See this documentation...
Introduction to version control concepts and workflows; Helix architecture, and related products.	Introducing Helix
Using the command-line interface to perform software version management and codeline management; working with Helix streams; jobs, reporting, scripting, and more.	Helix Versioning Engine User Guide
Basic workflows using P4V, the cross-platform Helix desktop client.	P4V User Guide
Working with personal and shared servers and understanding the distributed versioning features of the Helix Versioning engine.	Using Helix for Distributed Versioning
p4 command line (reference).	P4 Command Reference , p4 help
Installing and administering the Helix versioning engine, including user management, security settings.	Helix Versioning Engine Administrator Guide: Fundamentals
Installing and configuring Helix servers (proxies, replicas, and edge servers) in a distributed environment.	Helix Versioning Engine Administrator Guide: Multi-site Deployment
Helix plug-ins and integrations.	IDEs: Using IDE Plug-ins Defect trackers: Defect Tracking Gateway Guide Others: online help from the Helix menu or web site
Developing custom Helix applications using the Helix C/C++ API.	C/C++ API User Guide

For specific information about...	See this documentation...
Working with Helix in Ruby, Perl, Python, and PHP.	APIs for Scripting

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
<code>literal</code>	Monospace font indicates a word or other notation that must be used in the command exactly as shown.
<i>italics</i>	Italics indicate a parameter for which you must supply specific information. For example, for a <i>serverid</i> parameter, you must supply the id of the server.
<code>[-f]</code>	Square brackets indicate that the enclosed elements are optional. Omit the brackets when you compose the command. Elements that are not bracketed are required.
<code>...</code>	Ellipses (...) indicate that the preceding element can be repeated as often as needed.
<code>element1 element2</code>	A vertical bar () indicates that either <i>element1</i> or <i>element2</i> is required.

Please give us feedback

We are interested in receiving opinions on this manual from our users. In particular, we'd like to hear from users who have never used Perforce before. Does this guide teach the topic well? Please let us know what you think; we can be reached at manual@perforce.com.

If you need assistance, or wish to provide feedback about any of our products, contact support@perforce.com.

This book, *Using Helix 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 Helix commands

Centralized and distributed architecture

Before you start to work with distributed versioning, it's important to understand certain basic concepts — including distributed versioning architecture and how servers relate to one another in this architecture.

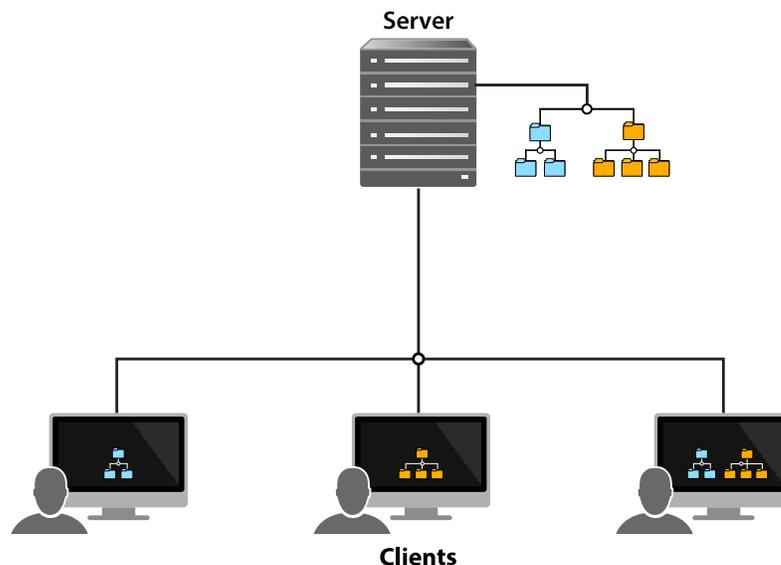
As discussed in the "[Basic Concepts](#)" chapter of *Introducing Helix*, version control systems can implement either a centralized model or a distributed model. The Helix Versioning Engine supports both of these models, as well as configurations that implement a hybrid of the two.

In a *centralized* model, users interact directly with a shared server, checking out files, working in those files, and then checking them back in to the shared server.

Note

The client is a program — the Helix command line client, P4V, and P4Connect are some examples — that users interact with. Clients, in turn, interact with servers, which also interact with each other.

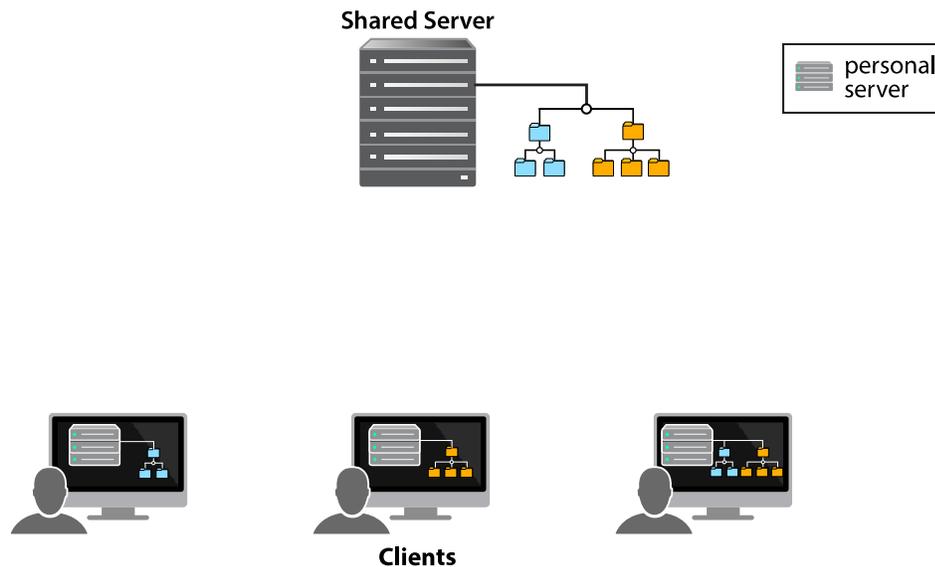
The following diagram illustrates the centralized model:



As you can see, some clients access subsets of the files stored on the shared server while others access all files stored on the server.

The *distributed* model gives users access to a repository of archived files — and changes to those files — from a server *running on their local machine*. This means that the entire history of a file is contained on each user's machine. A user can manage versioned content without interacting with any other Helix server or even connecting to a network, unless desired. A user can also rewrite and revise history to discard unwanted intermediate changes. The distributed model allows users to work experimentally, to try out changes and branch new streams, without interfering with others' work, and without the need for a network connection.

The following diagram illustrates the distributed model:

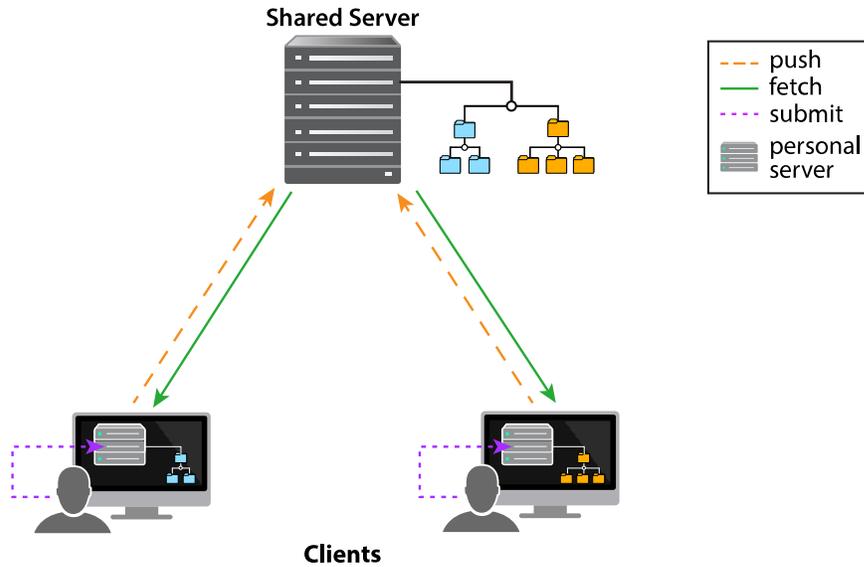


In the distributed model, a user can work on their individual server — disconnected from the network — until they're ready to copy content to a shared server, making the content available to other users.

Moreover, unlike other version control systems — such as Git — users can copy a subset of the shared server's content to the server on their own machine, rather than copying the entire shared server repository.

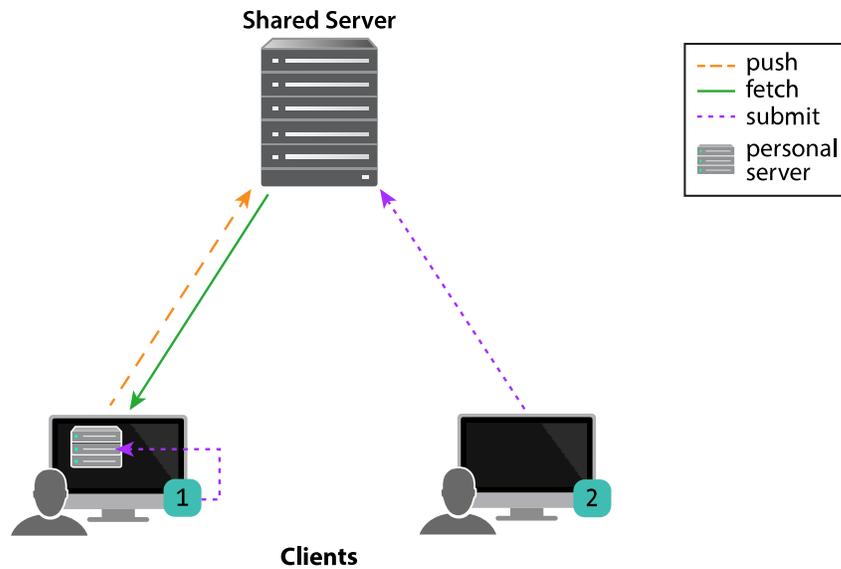
In this model, users first submit changes to their personal server and then push changes to a shared server. A different user can then fetch — that is, copy — those changes from the shared server to their personal server.

In the diagram below, each user is fetching just a subset of files: the user on the left is fetching just the blue files, while the user on the right is fetching just the orange files. Each client is submitting changes to its respective personal server and then pushing changes to and fetching changes from the shared server.



The distributed versioning model also provides a *hybrid* workflow that includes both centralized and distributed client-server relationships. The hybrid workflow allows users both to share their work with each other — by connecting their individual server to a shared server — and to interact directly with a shared server without an intervening individual server.

The following diagram illustrates a hybrid configuration:



In addition, Helix distributed versioning allows synchronization of content across multiple offices or teams. You use the **p4 fetch** and **p4 push** commands if the servers are networked or the **p4 zip** and **p4 unzip** commands if they're not. Synchronization of content across sites is covered in the "Managing Distributed Development" section of the [Helix Versioning Engine Administrator Guide: Fundamentals](#).

How servers relate to each other

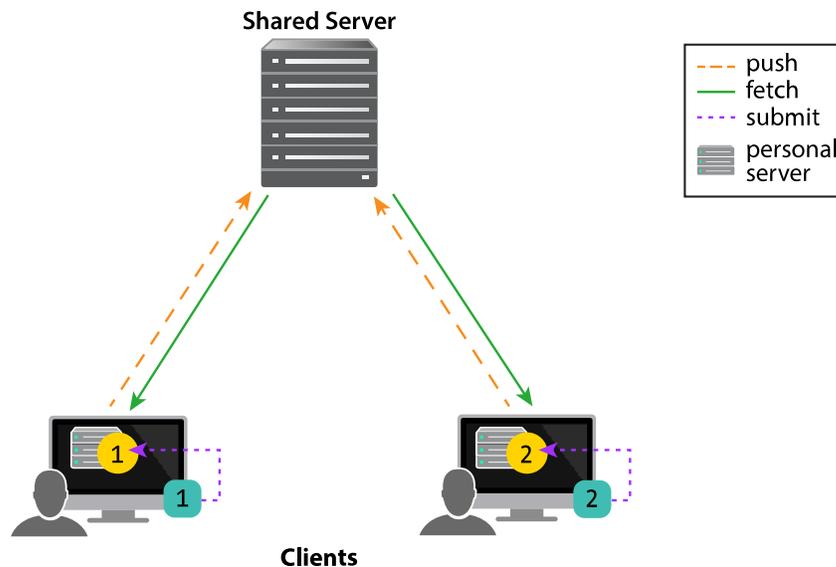
There are two ways of thinking about relationships between servers in a distributed versioning environment; understanding these distinctions is important for using server commands correctly:

1. From the point of view of intended use
2. In the context of client-server architecture

From the point of view of *intended use*, the servers are either *personal* servers or *shared* servers:

- A personal server runs on an individual user's machine; a shared server is the server in which individual users store their changes so that other users have access to these changes.
- A personal server is intended to be used by a single user, while a shared server is intended to be used by multiple users concurrently.

In the next diagram, Client 1 and Client 2 are using the shared server concurrently; both are interacting with the shared server via their respective personal servers.



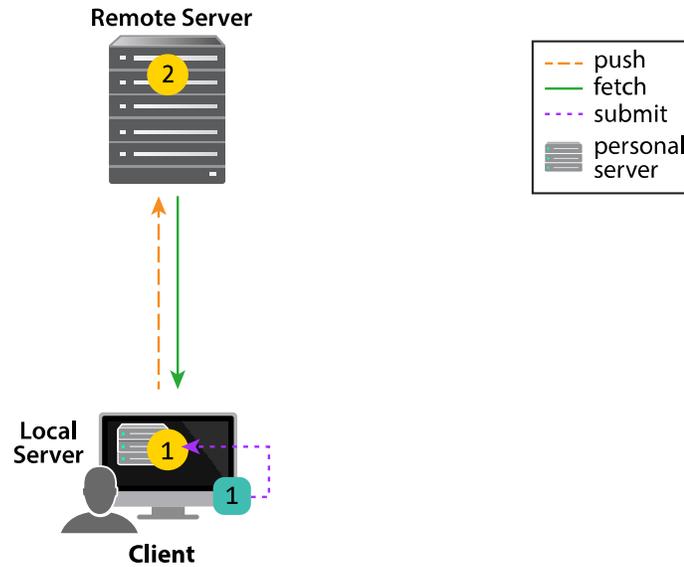
Client 1 interacts with Server 1 — a personal server — which in turn interacts with the shared server. Likewise with Client 2.

In the context of *client-server architecture*, the servers are either *local* servers or *remote* servers:

- A local server is a server running on the same machine as your client.
- A remote server is a server your local server is talking to in order to do what it needs to do.

In the next diagram, Client 1 and Server 1 are running on the same machine.

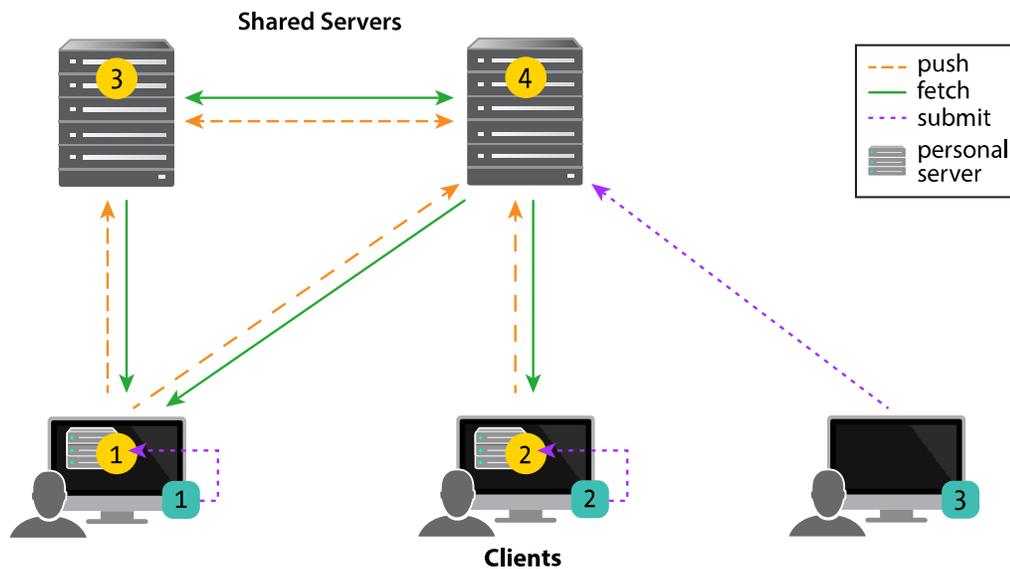
From the point of the view of Client 1, Server 1 is a local server and Server 2 is a remote server.



Throughout this guide, the name we use to refer to a server depends on which name makes sense in the context of the discussion.

Putting it all together

In the following diagram, Client 1 and Server 1 are running on the same machine and Client 2 and Server 2 are running on the same machine:



Server-to-server relationships

Server 1 and Server 2 are **personal servers**. Server 3 and Server 4 are **shared servers**.

Server 1 pushes changes to and fetches changes from Server 4. So does Server 2. Server 2 could fetch the changes Server 1 pushed, enabling the two personal servers to share content.

Shared servers 3 and 4 fetch and push changes from and to each other.

Client-to-server relationships

From the point of view of Client 1, Server 1 is a **local server** and Servers 3 and 4 are **remote servers**.

From the point of view of Client 2, Server 2 is a local server and Server 4 is a remote server.

Client 3 is interacting with Server 4 without an intervening personal server; the two servers are relating within a centralized architecture, rather than a distributed architecture.

Command line aliasing

Note

As with classic Helix server commands, you have the option of applying aliases to personal server commands, to do such things as:

- abbreviation
- creating more complex commands
- automating simple multi-command sequences
- providing alternate syntax for difficult-to-remember commands

For more information, see the "[Introduction](#)" chapter of [P4 Command Reference](#).

Installing Helix Versioning Engine 16.1 on Mac OS X

1. Open a web browser.
2. Navigate to <http://www.perforce.com/downloads>.
3. Download the `helix-versioning-engine- $\{x86,64\}$. $\{tgz,zip\}$` file.
4. Extract the Helix Versioning Engine (`p4d`) and Helix Command Line (`p4`) from the `tgz/zip` file.
5. Open a Terminal window.
6. Make the downloaded files executable:

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

7. Move the files into a common execution path:

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

Installing Helix Versioning Engine 16.1 on Linux

Without OS-specific packages

1. Open a web browser.
2. Navigate to <http://www.perforce.com/downloads>.
3. Download the `helix-versioning-engine- $\{x86,64\}$. $\{tgz,zip\}$` file.
4. Extract the Helix Versioning Engine (`p4d`) and Helix Command Line (`p4`) from the `tgz/zip` file.
5. Open a Terminal window.
6. Make the downloaded files executable:

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

7. Move the files into a common execution path:

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

With OS-specific packages

1. Open a web browser.

2. Navigate to <http://package.perforce.com>
3. Follow the instructions to configure a package repository and install OS-specific packages.

Installing Helix Versioning Engine 16.1 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. Download the Helix Command Line installer.
4. Run the installer you downloaded.
5. Accept all of the defaults.

This gives you the **p4d** executable (Helix Versioning Engine) and the **p4** executable (Helix Command Line).

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 [“Run p4 init” on page 12](#).
2. The second approach copies content from another (shared) 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, which then makes these changes available to other project users. At any given time, users can fetch the latest content from the shared server. See [“Run p4 clone” on page 13](#).

Initialize an empty server

Use this approach if you want to work in isolation on a personal server, developing and possibly branching code.

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.

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**, Helix attempts to set the case sensitivity of your personal server to match that of the shared server specified in your current **P4PORT** setting.

If you know which shared server your personal server will be fetching from and pushing to, run **p4 init -p**, passing in the address of the shared server. This tells the Helix Versioning Engine to discover the shared server’s case sensitivity and Unicode support settings and apply them to your personal server; this makes the two servers compatible.

If Helix 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**, Helix attempts to set the Unicode support of your personal server to match that of the shared server specified in your current **P4PORT** setting. If Helix 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] [-p]
```

p4 init includes a number of command-line arguments:

- To configure your personal server without Unicode support, pass the **-n** option.
- To have Helix 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 Helix.
- **.p4ignore** - A list of files Helix 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 Helix and then **p4 submit** to submit them. If you are new to Helix, see the "Managing Files and Changelists" chapter of the [Helix Versioning Engine User Guide](#).

Prepare to fetch and push content between servers

If you subsequently want to 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 5, "Fetching](#)

and [Pushing](#) on page 15 and [Chapter 7, “Understanding Remotes”](#) on page 29 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 **-p** preceding **P4PORT** is optional. If not specified, **p4 clone** uses the shared server specified by the **P4PORT** environment variable. See [“P4PORT meaning before and after a clone”](#) on page 14 for a discussion of how **P4PORT** has a different meaning before and after a clone.
- 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 shared 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, Helix will copy the remote from the shared server to the personal server and name it **origin**. For more information on remotes, see [Chapter 7, “Understanding Remotes”](#) on page 29.
- The **-f** option specifies a filespec in the shared server to use as the path to clone; this path will also be used to determine the stream setup in the personal server. You can specify the **-f** option or the **-r** option but not both.

It is optional to specify the ``-f`` string on the command line. Instead, you can simply follow `*`p4 clone`*` with `_filespec_`.

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

P4PORT meaning before and after a clone

When you clone from a shared server to create a personal server, the `P4PORT` argument you pass to the `p4 clone` command specifies the address of the shared server you wish to clone from. If you don't pass a `P4PORT` value via the `-p` option, Helix uses the value of `P4PORT` set in the current command environment to identify the address of the shared server you wish to clone from.

After a clone, `P4PORT` refers to the personal server's `P4PORT` setting in its `P4CONFIG` file.

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 personal server to files on a shared server and is required for fetching, pushing, and cloning; it describes exactly which files should be copied from a personal server to a shared server or vice-versa. It is described in detail in [Chapter 7, "Understanding Remotes" on page 29](#).

Get the latest changes

To update your personal server with the latest changes from the shared server, run `p4 fetch`. See [Chapter 5, "Fetching and Pushing" on page 15](#) for more information.

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

- To copy changelists from a personal server to a shared server
- To fetch changelists from a shared server that were pushed there by other personal servers
- To obtain and work with a subset of a shared server's entire repository.
- To copy work between two personal servers

Administrators can also use fetching and pushing to copy changelists between shared servers.

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

The **p4 fetch** command copies the specified set of files and their history from a remote server into a local server. The **p4 push** command copies the specified set of files, and their history from a local server to a remote 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 remote server, it will leave those files locked on the remote 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** command to unlock the files on the remote 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 8, "Rewriting History" on page 39](#).

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 servers, the respective servers must have authentication and access permissions configured correctly:

- The user name on the remote server must be the same as the user name on the local server. This will be the case by default unless you have specified the **RemoteUser** field in the remote server's remote spec.
- The user must exist on the remote server.
- The user must have read (fetch) and write (push) permission on the remote server.

- The `server.allowpush` and `server.allowfetch` configuration settings must be set to on (they're off by default) on both the remote server and the local server. See the command `p4 help configurables` for more information.
- The user must be logged into the remote server via `p4 login -r`.

Specify what to copy

As described in [Chapter 7, “Understanding Remotes” on page 29](#), 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 local 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 [Chapter 7, “Understanding Remotes” on page 29](#).

- 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 local server

Note that when you specify a filespec or a stream, the Helix Versioning Engine 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 remote server's repository; rather, you can fetch or push whatever subset of the repository 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.

What do fetch and push copy?

In addition to the specified set of files, the changelists that submitted those files, and integration records, fetching and pushing to a server also copies the following:

- attributes
- any fixes associated with the changelists, but only if the job that is linked by the fix is already present in the local server

Note

Zippping and unzipping files also copies attributes and fix records.

Attribute interoperability with 15.1

2015.1 DVCS servers don't support fetching and pushing of attributes. If you try to push files with attributes from a 2015.1 server to a 2016.1 server, the 2016.1 server will detect that the attribute data was not provided and not include any attributes on the pushed files.

If a 2016.1 server tries to push files with attributes to a 2015.1 server, the 2015.1 server quietly ignores the attributes data.

Fetching, pushing, and changelists

When changelists are added to the target server during a fetch or 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 target server during a fetch or 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 target server for a fetch or a push, none of the submit triggers are run in the target server. For more information about submit triggers, see the "Scripting Perforce" chapter in the [Helix Versioning Engine Administrator 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.

Fetch and push a shelved changelist

The Helix Versioning Engine allows you to fetch, push (and zip) a shelved changelist instead of one or more submitted changelists. This gives you more flexibility if your workflow typically involves shelved changelists.

Note

Both the local and the remote server must be version 2016.1 or higher to support copying a shelved changelist.

Copying a shelf always results in the creation of a new shelf in the destination server; existing shelves, even if similar, are not overwritten.

There are two key differences between copying a submitted changelist and copying a shelved changelist:

- To copy a submitted changelist, you must have write access to the changelist's files; by contrast, need to have *open* access to the shelf's files in the target server. As a reminder, *open* access means the user can open, add, edit, delete, or integrate the files.
- The resulting new shelf is owned by the user who issued the push, fetch, or zip command, even if the shelf copied was owned by a different user.

Track a changelist's identity from server to server

As described earlier, a changelist gets renumbered each time it gets fetched, pushed, or unzipped; as a result, it quickly becomes difficult to determine which changelist is which across a series of servers. Changelist 12 on one server may not be the same as changelist 12 on another server.

The Helix Versioning Engine includes a global changelist ID feature which allows you to assign to a changelist a permanent ID that remains the same from server to server. This is an opt-in feature. There are two workflows for enabling global changelist IDs. They are summarized in the following subsections:

Workflow 1: Let Helix generate global changelist IDs

The majority of Helix users will likely choose to have global changelist IDs system-generated.

To have Helix generate the IDs for you, follow these steps:

On a personal server:

1. **Run the `p4 configure` command to set `submit.identity` to whichever of the three possible formats you prefer:**
 - `uuid`: a universally-unique identifier

- **checksum**: a checksum
- **serverid**: a combination of the serverid + changelist number

This causes Helix to generate a global changelist ID and write it to the **Identity** field of the change spec for the changelist in question, each time a change is submitted. For more information, see the description of the `submit.identity` configurable in the "[Configurables](#)" chapter of the [P4 Command Reference](#).

2. Run **p4 submit** to submit the changelist. Once you've done this, the changelist ID appears in the **Identity** field of the change spec.
3. Run **p4 describe changelistnumber** to find out what changelist ID was generated.

Workflow 2: Enter global changelist ID manually

Choose this workflow if you want to customize your global changelist ID names. For example, you may want to name a changelist according to the bug it corresponds to in your bug database.

On a personal server:

1. Run **p4 submit** to submit your changelist.
2. Edit the change spec to set the value of the **Identity** field to the desired value.
3. Run the **p4 push**, **p4 fetch**, or **p4 unzip** command.

On the shared server:

1. Run **p4 describe -I changelistID** to retrieve the changelist number of the changelist that was pushed, fetched, or unzipped.

Track who pushed, fetched, or unzipped a changelist

The Helix Versioning Engine includes a feature — relevant only for users of the Helix's distributed versioning features (DVCS) — that lets you distinguish between who created a particular changelist and who pushed, fetched, or unzipped it later. This gives you more visibility into scenarios in which one user pushes, fetches, or unzips another user's work.

You use the change spec's **ImportedBy** field — via the **p4 change** command — to specify the name of the user who ran the **p4 fetch**, **p4 push**, or **p4 unzip** command that imported this changelist into the shared server.

The **ImportedBy** field is filled in at the point when Helix stores the changelist in the target shared server.

Fetching and pushing fixes

If you plan to share fixes — that is, jobs associated with changelists — between servers when fetching and pushing (as well as zipping and unzipping) you must ensure that:

- The two servers have identical job specs
- You have manually copied the jobs you plan to push, fetch, zip, or unzip from one server to the other. You do this with the `p4 job` command.

In the example below, `p_server` is pushing a job to `s_server`: You generate the job output by running `p4 job -o` and specifying the `s_server` name and port number and then pass the job form into `s_server` by running `p4 job -i`.

```
$ p4 -p s_server:1667 job -o JobName | p4 job -i
```

Fetching and pushing integration history

When you merge from one stream to another, you must have both streams mapped in the remote spec in order to push or fetch integration history.

Consider the following example:

1. **You clone from a shared server to create a personal server and the following remote spec, called `origin`:**

```
# A Helix Remote Specification.
RemoteID:      origin
Address:       p4demo:1666
Owner:        jschaffer
Options:       unlocked nocompress
Update:       2015/06/29 13:14:26
Access:       2015/06/29 13:14:57
Description:
    Created by Joe_Coder.
LastFetch:    default
LastPush:     12305
DepotMap:
    //talkhouse/main/... //depot/Talkhouse/main-dev/...
    //talkhouse/release1.0/... //depot/Talkhouse/rel1.0/...
```

2. **In the personal server, branch a development stream (`dev`), make changes to some files in that stream and submit them.**
3. **Merge changes from the `dev` stream to the `main` stream.**

4. Run **p4 push**.

You will observe that although the files were pushed to the shared server, the integration history was not.

To ensure that integration files are pushed or fetched, both the merge source and the merge target must be included in the remote spec.

1. **Modify the remote spec to add a line under `DepotMap` for development stream `//talkhouse/dev/...`:**

```
DepotMap:
  //talkhouse/main/... //depot/Talkhouse/main-dev/...
  //talkhouse/dev/... //depot/Talkhouse/jschaffer-dev/...
  //talkhouse/release1.0/... //depot/Talkhouse/rel1.0/...
```

2. Run **p4 push**.

3. **Observe that both files and integration history were pushed to the shared server.**

Configure server to limit storage of archive revisions

Recall that server files have two portions: the file data itself — known as the *archive* or *archive file* — and the file's metadata — information describing the file, such as its size and its owner.

Because digital asset archives take up substantial storage space, it would be convenient to control how many revisions of the archive you store locally when you fetch the digital asset files to your personal server. Moreover, because source code doesn't impose this same storage burden, it would be equally helpful to control the source code archive files separately from the digital assets when fetching quantities of archives.

The `ArchiveLimits:` field in the personal server's remote spec allows you to do just this. Using `ArchiveLimits:`, you specify how many revisions of a file or files archive(s) you want to store locally with a fetch. This is regulated at the level of one or more files, so if you store your digital asset files in separate subdirectories from source code files, you can impose the archive limits on just the digital asset files, leaving source code files unaffected.

`ArchiveLimits:` does not affect the fetched files' metadata; the fetch stores metadata for the entire history of the files.

If you don't set `ArchiveLimits:` the server defaults to storing all archive revisions.

Note

`ArchiveLimits:` are applied during the **p4 fetch** operation only. However, since they apply to **p4 fetch**, they also affect **p4 clone**, if they are set in the remote spec invoked by **p4 clone**.

ArchiveLimits: entries

An `ArchiveLimits:` entry consists of a sequence of one or more lines of the form `filespec depth`, where:

- **filespec** is a file or subdirectory of files
- **depth** dictates how many relative revisions of the archive files to store

The **depth** field can be a non-negative integer, or the special word **all**, which tells the server to store all revisions of the file or files specified in that line's **filespec**. Setting **depth** to **0** tells the server not to store any archives for files specified in this line's **filespec**.

The integer value **0** means that a fetch will not store any archive files, just metadata, for the files specified in the **filespec** entry on this line.

A positive integer N means that no more than N archives should be stored for each file in this section of your repo. For example, suppose you have a file whose latest revision is 17 and the **depth** setting for the **ArchiveLimits:** entry governing this file is **2**. This means that when the file is fetched, the server will store the archive for revisions 17 and 16 only.

Recall that since **ArchiveLimits:** behavior operates at the level of a **filespec**, you can separate what the server does with digital assets files from what it does with source code simply by:

1. storing the digital assets files in a distinct folder from the source code files
2. describing **ArchiveLimits:** behavior for each of these folders on separate lines

Consider the following sample **ArchiveLimits:** entry:

```
ArchiveLimits:
  //.../*.zip 1
  //.../*.iso 0
  //.../*.rpm 0
  //depot/main/.../*.zip 3
  //depot/rel*/.../*.zip all
  //depot/.../*.mp4 2
```

This would result in the server behavior summarized in the following table:

File or files	Server behavior
//depot/main/my/proj/component/MyClass.java	All revisions of the archive file are stored on the server
//artifacts/Windows/Windows10.iso	No archive is ever stored for this file
//depot/main/my/proj/builds/myProj.zip	The three most recent revisions of the archives are stored on the server
//depot/dev/my/proj/builds/myProj.zip	Only the most recent revision of the archive is stored on the server
//depot/rel1.0/my/proj/builds/myProj.zip	All revisions of archives are stored on the server

Per-server identities

There are distributed versioning scenarios in which you want to fetch and push from/to multiple shared servers and you need to use a different Helix identity for each server. You can specify the identity Helix should use for a particular shared server in the `RemoteUser` field of that shared server's spec. The `p4 fetch` and `p4 push` command then use that identity for authentication against that shared server.

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 15](#), 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 8, “Rewriting History” on page 39](#) 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.

Support for exclusive locking in personal servers

There are certain types of files that cannot be merged and therefore must only be changed by one user at a time. Examples include binary files, Microsoft Word files, and digital assets such as 3D models. In Helix, to ensure that only one user at a time writes changes to a file, you assign the file the filetype `+l`. This gives the user a global exclusive lock on the file when they open it for edit.

You can enable the support of exclusive locking in personal servers for the `p4 edit`, `p4 delete`, and `p4 revert` commands. To do this, pass the `--remote=remote` option, where `remote` specifies the shared server from which you cloned the locked file. The lock is held in the shared server. All personal servers which cloned the file from the shared server must observe this lock restriction.

Note

For exclusive locking to work, the shared server must be configured as a commit server. For instructions on how to do this, see [“Create commit and edge server”](#)

[configurations](#) in the ["Commit-edge Architecture"](#) chapter of the *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.

Once you have the locked file in the shared server:

- you can safely change the file, submit your changes, and push to the shared server; your lock releases automatically at the end of the push
- no other user is allowed to either edit that file or push it from their personal server

Exclusive locking works as follows for each of the three commands:

- For **p4 edit**, the `--remote=remote` option opens the file for edit in your personal server, and additionally — if the file is of type `+l` — takes a global exclusive lock on the file in the shared server from which you cloned the file. That global exclusive lock is retained until you push the updated file to the shared server, or until you use **p4 revert --remote=remote** to revert the file.
- For **p4 delete**, the `--remote=remote` option opens the file for delete in your personal server, and additionally — if the file is of type `+l` — takes a global exclusive lock on the file in the shared server from which you cloned the file. That global exclusive lock is retained until you push the deleted file to the shared server, or until you use **p4 revert --remote=remote filename** to revert the file.
- For **p4 revert**, the `--remote=remote` option reverts the named file in your personal server, and additionally — if the file is of type `+l` — releases the global exclusive lock on the file in the shared server from which you cloned the file.

Using triggers with fetch and push

Helix triggers are user-written programs called by a Helix Versioning Engine when certain operations are performed. You use triggers to extend or customize Helix functionality. Triggers are of different types, depending on the event that causes the trigger to execute.

The trigger types in the list below have been defined to help you customize the processing done in committing changes in a distributed versioning environment. These three types may be invoked during the execution of the **p4 push**, **p4 fetch**, or **p4 unzip** commands.

- Use push-submit triggers to customize processing during that phase of the **push**/**fetch**/**unzip** command when metadata has been transferred but files have not yet been transferred.
- Use push-content triggers to customize processing during that phase of the **push**/**fetch**/**unzip** command when files have been transferred but their contents have not yet been committed.
- Use **push-commit** triggers to do any clean up work or other post processing work after changes have been committed.

Note

Push triggers are disabled by default for the **p4 unzip** command. See the **p4 unzip** command in the [P4 Command Reference](#) for instructions on how to enable push triggers.

For more information, see ["Triggering on pushes and fetches"](#) in the ["Scripting Perforce: Triggers and Daemons"](#) chapter of the *Helix Versioning Engine Administrator Guide: Fundamentals*.

Streams are the Helix term for branches. They are variant versions of a body of code. You can read more about them in the "Streams" chapter of the [Helix Versioning Engine User Guide](#).

When using a personal server created by **p4 init** or **p4 clone**, Helix uses streams as containers for your code. Helix 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 -P main -c 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 Helix 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, **p4 switch** syncs your client workspace to the head of the new stream, and unshelves any files that were open in the default changelist 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.

The shelving process stores files in the depot from a pending changelist without submitting them. If you decide that the change you were making in a particular stream, actually belongs in a different stream, you can run **p4 switch -r stream_name** to apply the changes on the current stream to the stream specified in the switch command.

```
$ p4 switch -r stream_name
```

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 in `//stream/dev`:

```
$ p4 files //stream/dev/...
//stream/dev/dvcs_commands/remote.xml#1 - branch change 43 (text)
//stream/dev/dvcs_commands/remotes.xml#1 - branch change 43 (text)
//stream/dev/dvcs_commands/resubmit.xml#1 - branch change 43 (text)
//stream/dev/dvcs_commands/switch.xml#1 - branch change 43 (text)
//stream/dev/dvcs_commands/unsubmit.xml#1 - branch change 43 (text)
//stream/dev/dvcs_commands/unzip.xml#1 - add change 44 (text)
//stream/dev/dvcs_commands/zip.xml#1 - edit change 45 (text)
//stream/dev/dvcs_user_guide/00_preface.xml#1 - edit change 46 (text)
```

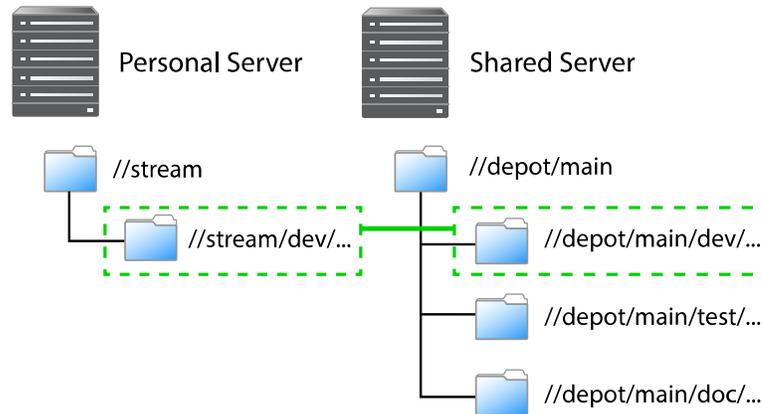
Now we open new files in `dev`:

```
$ 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 remote.xml remotes.xml switch.xml zip.xml
```

A remote describes how depot files are mapped between a personal server and 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 personal and a shared server:



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

- For fetch and clone operations, it defines the files from the remote server that you want in your personal server and specifies where you want them to reside.
- For a push operation, it defines the files from the personal server that you want in the shared 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. Over time, you can edit remote specs to account for the addition of new streams or the removal of old streams.

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, Helix 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:

- You create a remote on a shared server so that other users can clone from this server and obtain the files they need to work on a project. Note that to create a remote on a shared server, you must have an access privilege of **open** or greater. While this task typically falls in the domain of an administrator, it does not require administrator privileges.
- You — an 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 would create a remote on a shared server to dictate which subset of the shared server's repository 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 performce: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
Helix db files in '/Users/jschaffer/.p4root' will be created if missing...
Helix Versioning Engine 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 Helix Remote Specification.
#
# RemoteID:           The remote identifier.
# Address:            The P4PORT used by the shared server.
# Owner:              The user who created this remote.
# RemoteUser:        The user to use when connecting to the shared server.
# 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 shared 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.
# ArchiveLimits:      Limits on the number of files fetched (optional).

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 Helix Remote Specification.
#
# RemoteID:   The remote identifier.
# Address:    The P4PORT used by the shared 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 shared 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 Helix 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 shared 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 shared 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 personal server. The file paths on the right-hand side are on the shared server.

Remote specs give you the full power of Helix client view syntax. For details, see the section "Defining client workspaces" in the chapter [Configuring P4](#) in the [Helix Versioning Engine User 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 personal server.
2. The right-hand side specifies one or more files on the shared 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 Helix 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 depot path:

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

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

Mapping part of the depot

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

Example 7.1. Mapping part of the shared server depot to the personal server.

```
Remote Spec:
  //stream/main/... //depot/main/widget-doc/code/...
```

In this case, Helix will map only the shared server files under the code subdirectory to the personal server's `//stream/main` directory.

Mapping files to different locations on the personal server

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

Example 7.2. Multiple mappings in a single personal server

The following remote spec ensures that release notes in the remote `p4-doc` folder reside in the personal 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 personal 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 7.3. Using a remote spec to exclude files from a personal 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/...
```

Forward login to shared server

You can log into a shared server from a personal server without needing to know the shared server's P4PORT setting.

To do this, issue the following command:

```
$ p4 login -r remotespec
```

where *remotespec* is the spec corresponding to the server you want to log into.

If *RemoteUser* is specified in the remote spec, the login is performed for that user.

Helix allows you to 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 personal server's file history and a shared 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.

The tangent depot

As part of rewriting history, the Helix Versioning Engine makes use of *the tangent depot*; the tangent depot is a system-generated, read-only location in which the **p4 fetch -t** command stores conflicting changes. The **p4 fetch -t** command automatically creates the tangent depot named **tangent** if one does not already exist. This is further explained in the next section, [“Resolve conflicts by rewriting local history” on page 39](#).

For more information on the various kinds of depots, including the tangent depot, see the **p4 depot** chapter in the [P4 Command Reference](#).

Resolve conflicts by rewriting local history

If there are conflicts between a personal server's file history and a shared 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 -t**. This does the following:

1. Relocates conflicting changelists to the tangent depot.
2. Fetches the remote work from the shared server.

You then run **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** fails and you 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 -t`, which relocates User A's revision 5 to the tangent depot, 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
```

Where `NNN` is a changelist number.

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

Scenario 2: Combine two changes 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 `MMM`)

1. Unsubmit both changes:

```
$ p4 unsubmit //...@NNN,@MMM  
Change MMM 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 changelist *NNN*.
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. Delete the second change:

```
$ p4 shelve -d -c MMM  
$ p4 change -d -c MMM
```

Alternatively, to delete the second change you could run **p4 resubmit -i** and choose **d**.

Git:Helix Command Mappings

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

Git Command	Helix Command
<code>git add</code>	<code>p4 reconcile</code>
<code>git branch</code>	<code>p4 switch -l</code>
<code>git checkout --orphan <i>new_branch</i></code>	<code>p4 switch -cm <i>new_stream</i></code>
<code>git checkout <i>branch</i></code>	<code>p4 switch <i>stream</i></code>
<code>git clone <i>repository</i></code>	<code>p4 clone -p <i>host:port</i> -r <i>remote</i></code>
<code>git commit</code>	<code>p4 submit</code>
<code>git init</code>	<code>p4 init</code>
<code>git merge <i>branch</i></code>	<code>p4 merge --from <i>stream</i></code>
<code>git pull</code>	<code>p4 fetch -t -r <i>remote</i> -S <i>stream</i></code>
<code>git pull --all</code>	<code>p4 fetch -t</code>
<code>git push</code>	<code>p4 push -r <i>remote</i> -S <i>stream</i></code>
<code>git push --all</code>	<code>p4 push</code>
<code>git rebase</code>	<code>p4 unsubmit</code> followed by <code>p4 resubmit</code>
<code>git remote</code>	<code>p4 remotes</code>
<code>git remote add <i>new_remote repository</i></code>	<code>p4 remote <i>new_remote</i></code>
<code>git status</code>	<code>p4 status</code>
<code>git checkout -b <i>new-branch</i></code>	<code>p4 switch -c <i>new-branch</i></code>

For more details on Helix commands, see the [P4 Command Reference](#).

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/>).

Perforce software includes software from the Apache ZooKeeper project, developed by the Apache Software Foundation and its contributors. (<http://zookeeper.apache.org/>)

Perforce software includes software developed by the OpenLDAP Foundation (<http://www.openldap.org/>).

Perforce software includes software developed Computing Services at Carnegie Mellon University: Cyrus SASL (<http://www.cmu.edu/computing/>).

