PERFORCE

Helix Versioning Engine Administrator Guide: Fundamentals

2016.2

November 2016

Helix Versioning Engine Administrator Guide: Fundamentals
2016.2

November 2016

Table of Contents

Preface

This guide (previously titled *Perforce Server Administrator Guide: Fundamentals*) describes the installation, configuration, and management of the Helix versioning engine (Perforce server). This guide covers tasks typically performed by a system administrator (for instance, installing and configuring the software and ensuring uptime and data integrity), as well as tasks performed by a Perforce administrator, such as setting up Perforce users, configuring Perforce depot access controls, resetting Perforce user passwords, and so on.

This guide focuses on the installation, configuration, and management of a single Perforce server. For information on the installation, configuration, and management of multiple distributed servers as well as of proxies and brokers, see *Helix Versioning Engine Administrator Guide: Multi-site Deployment*

Because Perforce requires no special system permissions, a Perforce administrator does not typically require root-level access. Depending on your site's needs, your Perforce administrator need not be your system administrator.

Both the UNIX and Windows versions of the Perforce service are administered from the command line. To familiarize yourself with the Perforce Command-Line Client, see the *P4 Command Reference*.

About this manual

This manual includes the following chapters:

Chapter	Contents
Chapter 1, "Overview" on page 1	Discusses the basic client-server architecture for connected and disconnected clients. Describes the basic administration workflows for installing, configuring, and managing the Perforce server.
Chapter 2, "Installing and Upgrading the Server" on page 7	Describes how to install the Perforce service or upgrade an existing installation
Chapter 3, "Configuring the Server" on page 33	Explains basic configuration options for enabling DVCS, accepting client requests, case sensitivity, logging, and P4V settings.
Chapter 4, "Working with Depots" on page 53	Explains how you work with each type of depot to organize and archive your work.
Chapter 5, "Securing the Server" on page 65	Explains how you encrypt client-server communication and how you authenticate and authorize users.
Chapter 6, "Backup and Recovery" on page 105	Explains how you back up and recover versioned data and Perforce meta data.
Chapter 7, "Monitoring the Server" on page 119	Describes how you can monitor the Perforce server and its use of system resources: disk space, processes, commands, locked files, user file access, and logging.

Chapter	Contents
Chapter 8, "Managing the Server and Its Resources" on page 129	Provides information about managing code sharing, distributed development, users, changelists, disk space, processes, and Windows deployments.
Chapter 9, "Tuning Perforce for Performance" on page 147	Outlines some of the factors that can affect the performance of a Perforce server, provides a few tips on diagnosing network-related difficulties, and offers some suggestions on decreasing server load for larger installations.
Chapter 10, "Customizing Perforce: Job Specifications" on page 169	Explains how jobs enable users to link changelists to enhancement requests, problem reports, and other user-defined tasks. Describes the template that defines jobs and explores the use of third party defect tracking systems.
Chapter 11, "Using triggers to customize behavior" on page 179	Explains how you use different kids of scripts to customize the behavior of server processing.
Perforce Server (p4d) Reference on page 231	Provides complete reference information for the command used to start and configure the Perforce server.
Moving a Perforce server to a new machine on page 239	Explains how you move an existing Perforce server from one machine to another .
License Statements on page 251	Provides license information.

What's new in this guide for the 2016.2 release

This section provides a list of changes to this guide for the Perforce Server 2016.2 release. For a list of all new functionality and major bug fixes in Perforce Server 2016.2, see the Perforce Server 2016.2 Release Notes.

Major changes

Support for delegation of control in protections table

Information has been added to explain how you can use the **`p4 protect`** command's new **`owner`** permission to give a user or group the ability to run the **`p4 protect`** command for a particular path. See "Authorizing access" on page 83 for more information.

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*
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
`literal`	Monospace font indicates a word or other notation that must be used in the command exactly as shown.

Notation	Meaning
italics	Italics indicate a parameter for which you must supply specific information. For example, for a *serverid* parameter, you must supply the id of the server.
`[-f]`	Square brackets indicate that the enclosed elements are optional. Omit the brackets when you compose the command. Elements that are not bracketed are required.
…	Ellipses (…) indicate that the preceding element can be repeated as often as needed.
element1 | *element2*	A vertical bar (|) indicates that either *element1* or *element2* 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.

Chapter 1

Overview

This chapter explains the scope of this book and describes the basic architecture that an administrator can install, configure, monitor, and manage. Many of the issues that are covered in this book: monitoring, management, tuning, jobs, and scripting are relevant to more complex architectures; for this reason, this book remains the foundation of Perforce administration even if you are setting up more complex architectures.

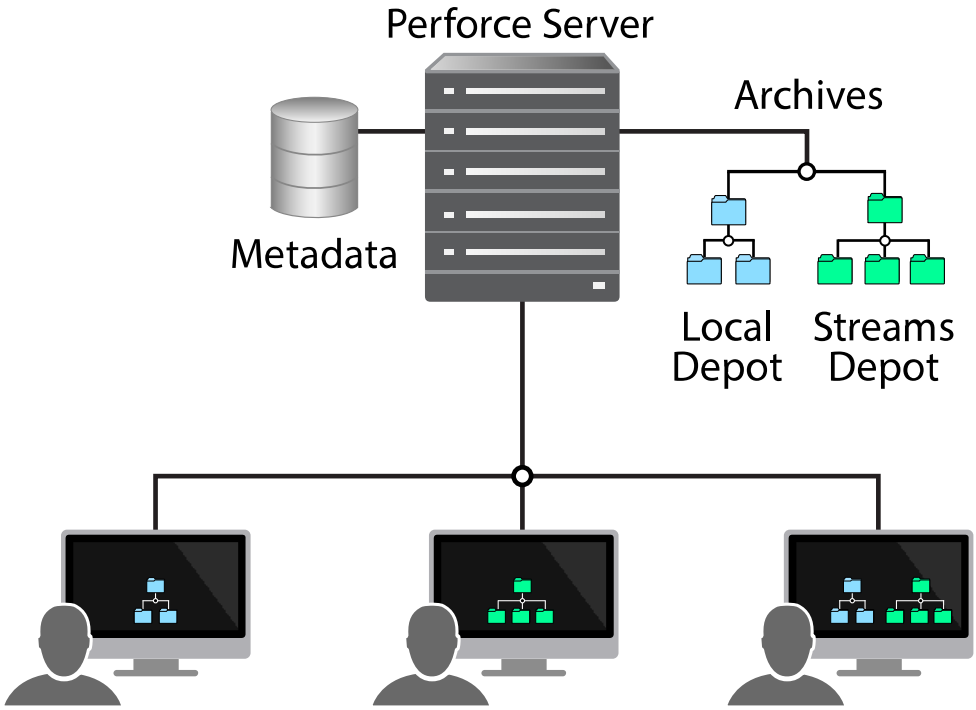
It is strongly recommended that you read *Introducing Helix* before you read this book.

Basic architecture

The simplest Perforce (Helix versioning engine) configuration consists of a client application and server application communicating over a TCP/IP connection. The server application manages a single repository that consists of one or more depots. A client application communicates with the server to allow the user to view trees of versioned files and repository metadata (file history and other information). Clients also manage local workspaces that contain a subset of the files in the repository. Users can view, check out, and modify these local files and submit changes back to the repository. Archived user files are stored on the server either in local type depots or in stream type depots.

The following figure illustrates this basic architecture. Multiple users connect to the server and view files stored either in a streams type depot or a local type depot using workspaces (local directories) on their own machines.

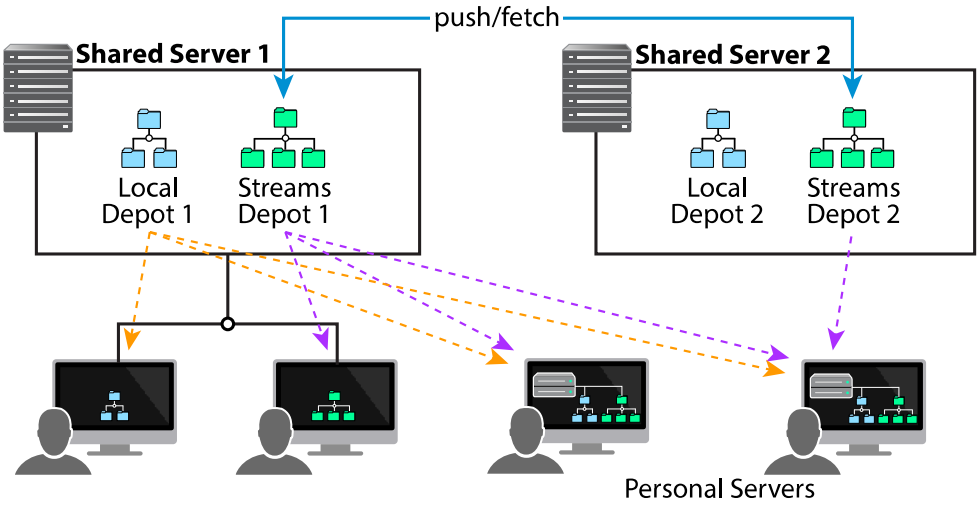
Figure 1.1. Single Server



Administrators support this architecture by installing and configuring the server, setting up users and security, monitoring performance, managing the resources used by the server, and customizing the behavior of the server if needed.

Users can also work disconnected from the server: they use a personal server to manage their work locally and share their work with others via a shared server. This option expands the basic architecture, as shown in the next figure:

Figure 1.2. Shared servers



Using this distributed versioning architecture, users can either connect directly to a shared server or work disconnected from the server, sharing their files with others by pushing or fetching content from their personal server to the shared server. Equally, an administrator can move content directly from one shared server to another by pushing and fetching content. Content can be moved across disparate networks, from one shared server to another shared server, by zipping and unzipping.

Administrators support this architecture by installing and configuring the shared server, setting up users and security, monitoring performance, managing the resources used by the server, and customizing the behavior of the server if needed. Personal servers are automatically set up when the user executes the **`p4 init`** or **`p4 clone`** command to create (and populate) their workspace and depot.

The administrator can expand this basic architecture either to resolve issues of geographical distribution, or scaling, or both.

- *Helix Versioning Engine Administrator Guide: Multi-site Deployment* explores some of the ways this architecture can be expanded for geographic distribution and performance.

Basic workflow

This book is roughly organized according to the administrator workflow. This section summarizes the basic workflow for setting up, configuring, and managing the Perforce server.

1. Set up the environment in which you will install the Perforce server.

   Review installation pre-requisites in "Planning the installation" on page 7.

2. Download and install the Perforce server.

   See Chapter 2, "Installing and Upgrading the Server" on page 7.

3. Start the server.

   See the appropriate section on starting the server in Chapter 2, “Installing and Upgrading the Server” on page 7.

4. Execute the **`p4 protect`** command to restrict access to the server.

   See “When should protections be set?” on page 84.

5. Configure the server.

   Basic configuration includes enabling distributed versioning if needed, defining depots, defining case sensitivity and unicode, managing client requests, configuring logging,and configuring P4V settings. See Chapter 3, “Configuring the Server” on page 33.

6. Define additional depots if needed.

   See Chapter 4, “Working with Depots” on page 53.

7. Add users if they are not automatically added on login.

   See “Creating standard users” on page 134.

8. Secure the server: set up secure client-server connection. Set up authorization and authentication.

   See Chapter 5, “Securing the Server” on page 65.

9. Back up the server.

   See Chapter 5, “Securing the Server” on page 65.

10. Monitor server performance and resource use.

    See Chapter 7, “Monitoring the Server” on page 119.

11. Manage the server and its resources: changelists, users, code sharing, disk space, and processes.

    See Chapter 8, “Managing the Server and Its Resources” on page 129.

12. Tune the server to improve performance.

    See Chapter 9, “Tuning Perforce for Performance” on page 147.

13. Customize Perforce by extending job definitions.

    See Chapter 10, “Customizing Perforce: Job Specifications” on page 169.

14. Customize Perforce using trigger scripts.

    See Chapter 11, “Using triggers to customize behavior” on page 179.

Administrative access

Perforce security depends on the security level that is set and on how authentication and access privileges are configured; these are described in Chapter 5, “Securing the Server” on page 65. Access levels relevant for the administrator are **`admin`** and **`super`**:

- `admin` grants permission to run Perforce commands that affect metadata, but not server operation. A user with admin access can edit, delete, or add files, and can use the **`p4 obliterate`** command.
- `super` grants permission to run all Perforce commands, allows the creation of depots and triggers, permits the definition of protections, and enables user management.

Users of type `operator` are allowed to run commands that affect server operation, but not metadata.

All server commands documented in the *P4 Command Reference* indicate the access level needed to execute that command.

Until you define a Perforce superuser, every Perforce user is a Perforce superuser and can run any Perforce command on any file. After you start a new Perforce service, use the following command:

```
$ p4 protect
```

as soon as possible to define a Perforce superuser.

Naming Perforce objects

As you work with Perforce, you will be creating a variety of objects: clients, depots, branches, jobs, labels, and so on. This section provides some guidelines you can use when naming these objects.

Object	Name
Branches	A good idea to name them, perhaps using a convention to indicate the relationship of the branch to other branches or to your workflow.
Client	Depends on usage, but some common naming conventions include: • *`user.machineTag.product`* • *`user.machineTag.product.branch`* Whether you use *`product`* or *`product.branch`* depends on whether your workspace gets re-purposed from stream to stream (in which case you use just *product*), or whether you have multiple workspaces, one for each branch (in which case you use *`product.branch`*, effectively tying the workspace name to the branch). A client may not have the same name as a depot.
Depot	Depot names are part of an organizations hierarchy for all your digital assets. Take care in choosing names and in planning the directory structure. It is best to keep the names short. A client may not have the same name as a depot.
Jobs	Use names that match whatever your external defect tracker issues look like. For example `PRJ-1234` for JIRA issues.

Object	Name
Labels	Site-dependent, varies with your code management and versioning needs. For example: `R-3.2.0`.
Machine Tags	The host name, or something simple and descriptive. For example `Win7VM`, `P4MBPro` (for Perforce MacBook Pro).
User	The OS user.

Chapter 2

Installing and Upgrading the Server

This chapter describes how to install the Perforce service or upgrade an existing installation. It contains information about the following topics:

- Pre-requisites for installation
- Where to obtain installation files
- Installing on UNIX (or Mac OS X)
- Installing on Windows
- Default location of installed files
- Upgrading your installation
- License requirements

Many of the examples in this book are based on the UNIX version of the Perforce service. In most cases, the examples apply equally to both Windows and UNIX installations. The material for UNIX also applies to Mac OS X.

Warning If you are upgrading an existing installation to Release 2013.3 or later, see the notes in "Upgrading the Perforce service" on page 28 before proceeding.

Install architecture

The chapter Chapter 1, "Overview" on page 1 describes the two deployment options that are covered in this book. This chapter focuses on the installation of the server for connected clients. See the "Install" chapter of *Using Helix for Distributed Versioning* for information on how to install a server that supports clients who want to work disconnected.

Planning the installation

The following sections describe some of the issues you need to think about before installing and configuring the server.

Network

Perforce can run over any TCP/IP network. For remote users or distributed configurations, Perforce offers options like proxies and the commit/edge architecture that can enhance performance over a WAN. Compression in the network layer can also help. For additional information about network and performance tuning, see Chapter 9, "Tuning Perforce for Performance" on page 147.

CPU

CPU resource consumption can be adversely affected by compression, lockless reads, or a badly designed protections table. In general, there is a trade-off between speed and the number of cores. A minimum of 2.4 GHZ and 8 cores is recommended. With greater speed, fewer cores will do: for example, a 3.2 GHZ and 4-core processor will also work.

For additional details, see "CPU" on page 148.

Memory

There are a couple of guidelines you can follow to anticipate memory needs:

- Multiply the number of licensed users by 64MB.
- Allocate 1.5 kilobytes of RAM per file in the depot.

In general Perforce performs well on machines that have large memory footprints that can be used for file system cache. I/O to even the fastest disk will be slower than reading from the file cache. These guidelines only apply for a single server.

For additional information about memory and performance tuning, see Chapter 9, "Tuning Perforce for Performance" on page 147.

Disk space allocation

Perforce disk space usage is a function of three variables:

- Number and size of client workspaces
- Size of server database
- Size of server's archive of all versioned files

All three variables depend on the nature of your data and how heavily you use Perforce.

The client file space required is the size of the files that your users will need in their client workspaces at any one time.

The server's database size can be calculated with a fair level of accuracy; as a rough estimate, it requires 0.5 kilobytes per user per file. (For instance, a system with 10,000 files and 50 users requires 250 MB of disk space for the database). The database can be expected to grow over time as histories of the individual files grow.

The size of the server's archive of versioned files depends on the sizes of the original files stored and grows as revisions are added. A good guideline is to allocate sufficient space in your `P4ROOT` directory to hold three times the size of your users' present collection of versioned files, plus an additional 0.5KB per user per file to hold the database files that store the list of depot files, file status, and file revision histories.

The `db.have` file holds the list of files opened in client workspaces. This file tends to grow more rapidly than other files in the database. If you are experiencing issues related to the size of your `db.have` file and are unable to quickly switch to a server with adequate support for large files, deleting unused client workspace specifications and reducing the scope of client workspace views can help alleviate the problem.

Filesystem

File size and disk I/O are the key issues here. For more information, see "File systems" on page 148.

Filesystem performance

Perforce is judicious with regards to its use of disk I/O; its metadata is well-keyed, and accesses are mostly sequential scans of limited subsets of the data. The most disk-intensive activity is file check-in, where the Perforce server must write and rename files in the archive. Server performance depends heavily on the operating system's filesystem implementation, and in particular, on whether directory updates are synchronous. Server performance is also highly dependent upon the capabilities of the underlying hardware's I/O subsystem.

Although Perforce does not recommend any specific hardware configuration or filesystem, Linux servers are generally fastest (owing to Linux's asynchronous directory updating), but they may have poor recovery if power is cut at the wrong time.

Performance in systems where database and versioned files are stored on NFS-mounted volumes is typically dependent on the implementation of NFS in question or the underlying storage hardware. Perforce has been tested and is supported using implementations that support the `flock` protocol.

Under Linux and FreeBSD, database updates over NFS can be an issue because file locking is relatively slow; if the journal is NFS-mounted on these platforms, all operations will be slower. In general (but in particular on Linux and FreeBSD), we recommend that the Perforce database, depot, and journal files be stored on disks local to the machine running the Perforce server process or that they be stored on a low-latency SAN device.

These issues affect only the Perforce server process (**p4d**). Perforce applications, (such as **p4**, the Perforce Command-Line Client) have always been able to work with client workspaces on NFS-mounted drives (for instance, workspaces in users' home directories).

Separate physical drives for server root and journal

Whether installing on UNIX or Windows, it is advisable to have your `P4ROOT` directory (that is, the directory containing your database and versioned files) on a different physical drive than your journal file.

By storing the journal on a separate drive, you can be reasonably certain that, if a disk failure corrupts the drive containing `P4ROOT`, such a failure will *not* affect your journal file. You can then use the journal file to restore any lost or damaged metadata. Separating the live journal from the **db.*** files can also improve performance.

Further details are available in Chapter 6, "Backup and Recovery" on page 105 and in "Journal and archive location" on page 150.

Protections and passwords

Until you define a Perforce superuser, every Perforce user is a Perforce superuser and can run any Perforce command on any file. After you start a new Perforce service, use:

```
$ p4 protect
```

as soon as possible to define a Perforce superuser. To learn more about how **p4 protect** works, see "Authorizing access" on page 83.

Without passwords, any user is able to impersonate any other Perforce user, either with the **-u** flag or by setting **P4USER** to an existing Perforce user name. Use of Perforce passwords prevents such impersonation. See the *Helix Versioning Engine User Guide* for details.

To set (or reset) a user's password, either use **p4 passwd username** (as a Perforce superuser), and enter the new password for the user, or invoke **p4 user -f username** (also while as a Perforce superuser) and enter the new password into the user specification form.

The security-conscious Perforce superuser also uses **p4 protect** to ensure that no access higher than **list** is granted to unprivileged users, **p4 configure** to set the **security** level to a level that requires that all users have strong passwords, and **p4 group** to assign all users to groups (and, optionally, to require regular changes of passwords for users on a per-group basis, to set a minimum required password length for all users on the site, and to lock out users for predefined amounts of time after repeated failed login attempts).

Note

An alternate way to reduce security risk during initial setup or during a maintenance interval is to start the Perforce server using **localhost:*port*** syntax. For example:

```
$ p4d localhost:2019
```

This forces the server to ignore non-local connection requests.

For complete information about security, see Chapter 5, "Securing the Server" on page 65.

Getting Perforce

Perforce requires at least two executables: the Perforce service (**p4d** on Unix, **p4s.exe** on Windows), and at least one Perforce application (such as **p4** on UNIX, or **p4.exe** on Windows).

The Perforce service and applications are available from the Downloads page on the Perforce web site:

http://www.perforce.com/downloads/complete_list

Go to the web page, select the files for your platform, and save the files to disk. In addition to plain binaries, installers for Windows are also available at the above site. You are encouraged to use them.

Many components are also available as Linux packages. See instructions in the next section for installing OS-specific packages for select Linux distributions.

Linux package-based installation

The Perforce service is available in two distribution package formats: Debian (**.deb**) for Ubuntu systems, and RPM (**.rpm**) for CentOS and RedHat Enterprise Linux (RHEL).

Using distribution packages greatly simplifies the installation, update, and removal of software, as the tools that manage these packages are aware of the dependencies for each package.

You can install packages for the Perforce service on the following Linux (Intel x86_64) platforms:

- Ubuntu 12.04 LTS
- Ubuntu 14.04 LTS
- Ubuntu 16.04 LTS
- CentOS or Red Hat 6.x
- CentOS or Red Hat 7.x

During the course of the installation, you will be asked to make choices about case sensitivity and Unicode settings. Please read the following sections now to understand the consequences of your selections:

- "Case sensitivity and multi-platform development" on page 39
- "Setting up and managing Unicode installations" on page 40

Make sure, before you start the install, that you have root level access to the server that will host your Perforce service.

Installation

1. **Configure the Perforce package repository.**

   As root, run one of the following:

   a. **For Ubuntu 12.04:**

   Create the file `/etc/apt/sources.list.d/perforce.list` with the following content:

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

   b. **For Ubuntu 14.04:**

   Create the file `/etc/apt/sources.list.d/perforce.list` with the following content:

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

   c. **For Ubuntu 16.04:**

   Create the file `/etc/apt/sources.list.d/perforce.list` with the following content:

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

   d. **For CentOS/RHEL 6:**

   Create the file `/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
```

e. **For CentOS/RHEL 7:**

Create the file `/etc/yum.repos.d/perforce.repo`, with the following content:

```
[perforce]
name=Perforce
baseurl=http://package.perforce.com/yum/rhel/7/x86_64/
enabled=1
gpgcheck=1
```

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

Run one of the following:

a. **For Ubuntu:**

```
$ wget -qO - https://package.perforce.com/perforce.pubkey | sudo apt-key add -
```

b. **For CentOS/RHEL (run this command as root):**

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

For information about verifying the authenticity of the signing key, see: https://www.perforce.com/perforce-packages

3. **Install the appropriate Perforce service package.**

The Perforce service is divided into multiple packages, so you can install just the components you need. The component package names are:

- `helix-p4d`
- `helix-p4dctl`
- `helix-proxy`
- `helix-broker`
- `helix-cli`

The `helix-p4d` package installs the main component of a Perforce service, **`p4d`**, as well as the command line interface, the service controller, and a configuration script to set them up.

At minimum, you need to install the `helix-p4d` package. To install a different package, substitute its name for `helix-p4d` in the commands below.

Run one of the following:

a. **For Ubuntu:**

```
$ sudo apt-get update
$ sudo apt-get install helix-p4d
```

b. **For CentOS/RHEL (run this command as root):**

```
# yum install helix-p4d
```

The files contained in the package are installed, and status information describing the main elements that have been installed is displayed.

4. **Run the post-installation configuration script.**

   If you installed the `helix-p4d` package, and if installation was successful, proceed on to "Post-installation configuration" on page 13.

Post-installation configuration

After the `helix-p4d` package has been installed, additional configuration is required. Perform the following steps:

1. **Use the `configure-helix-p4d.sh` script to configure a Perforce service.**

   Note

   The **`configure-helix-p4d.sh`** script can be used in a few different ways. The steps below outline the most straightforward configuration using interactive mode, but you can review the options by running:

   ```
   $ sudo /opt/perforce/sbin/configure-helix-p4d.sh -h
   ```

   Run in interactive mode:

   ```
   $ sudo /opt/perforce/sbin/configure-helix-p4d.sh
   ```

   In interactive mode, the configuration script begins by displaying a summary of default settings and those which have optionally been set with a command line argument.

2. **Provide information to the configuration script.**

   After the summary, the configuration script prompts for information it needs to set up your Perforce service.

Note

If you already have a Perforce service configured, and you supply its `service name`, then the configuration script only prompts for settings that you can change on an existing service.

At each prompt, you can accept the proposed default value by pressing **Enter**, or you can specify your own value.

The list below contains details about the options for each prompt:

a. **The Service Name:**

The name used when managing this service with **`p4dctl`**, for instance when starting and stopping the service.

This name is also used to set the Perforce `serverid` attribute on the underlying **`p4d`** instance, to distinguish it from others that may be in your overall installation.

b. **The Server Root (`P4ROOT`):**

The directory where versioned files and metadata should be stored.

c. **The Unicode Mode for the server:**

This is off by default.

Warning

If you turn Unicode mode on, you will not be able to turn it off. Be sure you are familiar with Unicode functionality when selecting this mode. See "Setting up and managing Unicode installations" on page 40 for information.

d. **The Case Sensitivity for the server:**

This is on by default.

See "Case sensitivity and multi-platform development" on page 39 for information.

e. **The Server Address (`P4PORT`):**

This specifies the host and port where the Perforce service should listen, and whether to communicate in plaintext or over SSL. For more information, see "Communicating port information" on page 19.

f. **Superuser login:**

The desired userid for a new user to be created with `super` level privileges.

For more information about superusers, see "Access levels" on page 85.

g. **Superuser password:**

The desired password to be set for the new superuser.

Due to the unlimited privileges granted to this user, a strong password is required.

After you answer all prompts, the script begins configuration according to your choices. As it runs, the script displays information about the configuration taking place.

After the configuration has completed successfully, a summary is displayed with details about what was done, and where settings are stored.

You can now connect to the service, or you can manage the service using the **p4dctl** utility. For more information, see Perforce Server Control (p4dctl) on page 243.

Updating

Important

The package update commands with **apt-get** or **yum** do not complete the process of updating your Perforce service. Packages for Linux simplify only certain steps of that process.

Updating packages without completing the rest of the update process leaves your Perforce service in a precarious state. Make sure to read and understand the entire process before updating any packages.

1. **Review the general update process.**

   a. See "Upgrading the Perforce service" on page 28 for details on the **general process** for how to update a Perforce service, on any platform. You should read and thoroughly understand this section before continuing.

   b. Packages for Linux help you accomplish only **specific steps** from the **general process**. If you are attempting to update your Perforce service using packages, you should still follow the **general process** linked above, but with the package specific modifications below:

      i. You may be able to stop, checkpoint, and start your Perforce service using **p4dctl**:

      ```
      $ sudo -u perforce p4dctl [stop|checkpoint|start] servicename
      ```

      ii. You do not need to manually retrieve the new component binaries (such as **p4d**) from the Perforce website; the package update commands with **apt-get** or **yum** accomplish this step.

      Platform-specific package update commands are below.

      iii. You still need to upgrade the Perforce service database to use the new versions of components delivered by the packages.

      As a convenience, 2016.1 and newer packages attempt to present tailored instructions and commands on-screen for upgrading those Perforce service databases that are discovered automatically.

2. **Determine if an updated package is available.**

Note

To update a different package, substitute its name for `helix-p4d` in the commands below.

Run one of the following:

a. **For Ubuntu:**

```
$ sudo apt-get update
$ sudo apt-cache madison helix-p4d
```

b. **For CentOS/RHEL (run this command as root):**

```
# yum --showduplicates list helix-p4d
```

3. **Install an updated package.**

Note

To update a different package, substitute its name for `helix-p4d` in the commands below.

The command to update is the same used to install initially.

Run one of the following:

a. **For Ubuntu:**

```
$ sudo apt-get update
$ sudo apt-get install helix-p4d
```

b. **For CentOS/RHEL (run this command as root):**

```
# yum install helix-p4d
```

Important

Failure to complete all update steps in the **general process** referenced above could result in continued downtime for your Perforce service.

UNIX non-package installation

Although you can install **p4** and **p4d** in any directory, on UNIX, the Perforce applications typically reside in `/usr/local/bin`, and the Perforce service is usually located either in `/usr/local/bin` or in its own server root directory. You can install Perforce applications on any machine that has TCP/IP access to the **p4d** host.

To limit access to the Perforce service's files, ensure that the **p4d** executable is owned and run by a Perforce user account that has been created for the purpose of running the Perforce service.

For an example Unix installation see:

http://answers.perforce.com/articles/KB_Article/Example-Unix-Installation

Note

To maximize performance, configure the server root (`P4ROOT`) to reside on a local disk and not an NFS-mounted volume. Perforce's file-locking semantics work with NFS mounts on Solaris 2.5.1 and later; some issues still remain regarding file locking on noncommercial implementations of NFS (for instance, Linux and FreeBSD). It is best to place metadata and journal data on separate drives

These issues affect only the Perforce server process (**p4d**). Perforce applications (such as **p4**, the Perforce Command-Line Client) have always been able to work with client workspaces on NFS-mounted drives, such as client workspaces located in users' home directories.

To start using Perforce:

1. Download the **p4** and **p4d** applications for your platform from the Perforce web site.
2. Make the downloaded **p4** and **p4d** files executable.
3. Create a server root directory to hold the Perforce database and versioned files.
4. Tell the Perforce service what port to listen to by specifying a TCP/IP port to **p4d**.
5. Start the Perforce service (**p4d**).
6. Set the **p4d** port and address for Perforce applications by setting the `P4PORT` environment variable.

Downloading the files and making them executable

On UNIX (or Mac OS X), you must make the **p4** and **p4d** binaries executable. After you download the software, use the **chmod** command to make them executable, as follows:

```
$ chmod +x p4
$ chmod +x p4d
```

Creating a Perforce server root directory

The Perforce service stores all user-submitted files and system-generated metadata in files and subdirectories beneath its own root directory. This directory is called the *server root*.

To specify a server root, either set the environment variable `P4ROOT` to point to the server root, or use the -r ***server_root*** flag when invoking **p4d**. Perforce applications never use the `P4ROOT` directory or environment variable; **p4d** is the only process that uses the `P4ROOT` variable.

Because all Perforce files are stored by default beneath the server root, the contents of the server root can grow over time. See "Disk space allocation" on page 8 for information about diskspace requirements.

The Perforce service requires no privileged access; there is no need to run **p4d** as `root` or any other privileged user. For more information, see "Running p4d as an unprivileged user" on page 21.

The server root can be located anywhere, but the account that runs **p4d** must have `read`, `write`, and `execute` permissions on the server root and all directories beneath it. For security purposes, set the `umask(1)` file-creation-mode mask of the account that runs **p4d** to a value that denies other users access to the server root directory.

Telling Perforce applications which port to connect to

The **p4d** service and Perforce applications communicate with each other using TCP/IP. When **p4d** starts, it listens (by default) for plaintext connections on port `1666`. Perforce applications like **p4** assume (also by default) that the corresponding **p4d** is located on a host named `perforce`, listening on port `1666`, and that communications are performed in plaintext.

If **p4d** is to listen on a different host or port and/or use a different protocol, either specify the configuration with the -p *`protocol`*:*`host`*:*`port`* flag when you start **p4d** (as in, **`p4d -p ssl:perforce:1818`**), or by the contents of the `P4PORT` environment variable.

Plaintext communications are specified with `tcp:`*`host`*`:`*`port`* and SSL encryption is specified with `ssl:`*`port`*. (To use SSL, you must also supply or generate an x509 certificate and private key, and store them in a secure location on your server. See "Using SSL to encrypt connections to a Perforce server" on page 66 for details.)

The preferred syntax for specifying the port is the following:

`protocol``:`*`host`*`:`*`port`*

There are situations, for example if you are using multiple network cards, where you might want to specify the port on which to listen using syntax like the following:

```
P4PORT=ssl::1666
```

The use of the double colon directs the server to bind to all available network addresses and to listen on port 1666. This can be useful if the host has multiple network addresses.

Note

To enable IPv6 support, specify the wildcard address with two colons when starting **p4d**. For example:

```
$ p4d -p tcp64:[::]:1818
```

starts a Perforce service that listens for plaintext connections, on both IPv6 and IPv4 transports, on port 1818. Similarly,

```
$ p4d -p ssl64:[::]:1818
```

starts a Perforce service that requires SSL and listens on IPv6 and IPv4, and

```
$ p4d -p ssl6:[::]:1818
```

starts a Perforce service that requires SSL connections, and listens for IPv6 connections exclusively.

See "IPv6 support and mixed networks" on page 19 for more information about IPv6 and IPv4 transports.

Unlike `P4ROOT`, the environment variable `P4PORT` is used by both the Perforce service and the Perforce applications, so it must be set both on the machine that hosts the Perforce service and on individual user workstations.

Communicating port information

Perforce applications need to know on what machine the **p4d** service is listening, on which TCP/IP port **p4d** is listening, and whether to communicate in plaintext or over SSL.

Set each Perforce user's `P4PORT` environment variable to *`protocol:host:port`*, where protocol is the communications protocol (beginning with `ssl:` for SSL, or `tcp:` for plaintext), *`host`* is the name of the machine on which **p4d** is running, and *`port`* is the number of the port on which **p4d** is listening. For example:

P4PORT	Behavior
`tcp:dogs:3435`	Perforce applications connect in plaintext to the Perforce service on host `dogs` listening on port `3435`.
`tcp64:dogs:3435`	Perforce applications connect in plaintext to the Perforce service on host `dogs` listening on port `3435`. The application first attempts to connect over an IPv6 connection; if that fails, the application attempts to connect via IPv4.
`ssl:example.org:1818`	Perforce applications connect via SSL to the Perforce service on host `example.org` listening on port `1818`.
`<not set>`	Perforce applications connect to the Perforce service on a host named or aliased `perforce` listening on port `1666`. Plaintext communications are assumed.

If you have enabled SSL, users are shown the server's fingerprint the first time they attempt to connect to the service. If the fingerprint is accurate, users can use the **p4 trust** command (either **p4 trust -y**, or **p4 -p ssl:*host*:*port* trust -i *fingerprint***) to install the fingerprint into a file (pointed to by the `P4TRUST` environment variable) that holds a list of known/trusted Perforce servers and their respective fingerprints. If `P4TRUST` is unset, this file is `.p4trust` in the user's home directory.

IPv6 support and mixed networks

As of Release 2013.1, Perforce supports connectivity over IPv6 networks as well as over IPv4 networks.

Behavior and performance of networked services is contingent not merely upon the networking capabilities of the machine that hosts the service, nor only on the operating systems used by the end users, but also on your specific LAN and WAN infrastructure (and the state of IPv6 support for every router between the end user and the Perforce versioning service).

To illustrate just one possible scenario, a user working from home; even if they have an IPv6-based home network, their ISP or VPN provider may not fully support IPv6. We have consequently provided several variations on `P4PORT` to provide maximum flexibility and backwards compatibility for administrators and users during the transition from IPv4 to IPv6.

P4PORT protocol value	Behavior in IPv4/IPv6 or mixed networks
`<not set>`	Use `tcp4:` behavior, but if the address is numeric and contains two or more colons, assume `tcp6:` If the `net.rfc3484` configurable is set, allow the OS to determine which transport is used.
`tcp:`	Use `tcp4:` behavior, but if the address is numeric and contains two or more colons, assume `tcp6:` If the `net.rfc3484` configurable is set, allow the OS to determine which transport is used.
`tcp4:`	Listen on/connect to an IPv4 address/port only.
`tcp6:`	Listen on/connect to an IPv6 address/port only.
`tcp46:`	Attempt to listen/connect to an IPv4 address. If this fails, try IPv6.
`tcp64:`	Attempt to listen/connect to an IPv6 address. If this fails, try IPv4.
`ssl:`	Use `ssl4:` behavior, but if the address is numeric and contains two or more colons, assume `ssl6:` If the `net.rfc3484` configurable is set, allow the OS to determine which transport is used.
`ssl4:`	Listen on/connect to an IPv4 address/port only, using SSL encryption
`ssl6:`	Listen on/connect to an IPv6 address/port only, using SSL encryption.
`ssl46:`	Listen on/connect to an IPv4 address/port. If that fails, try IPv6. After connecting, require SSL encryption.
`ssl64:`	Listen on/connect to an IPv6 address/port. If that fails, try IPv4. After connecting, require SSL encryption.

In mixed environments it is good practice to set the `net.rfc3484` configurable to 1:

```
$ p4 configure set net.rfc3484=1
```

Doing so ensures RFC3484-compliant behavior for users who do not explicitly specify the protocol value; that is, if the client-side configurable `net.rfc3484` is set to `1`, and `P4PORT` is set to `example.com:1666`, or `tcp:example.com:1666`, or `ssl:example.com:1666`, the user's operating system will automatically determine, for any given connection, whether to use IPv4 or IPv6 transport.

In multi-server environments, `net.rfc3484`, when set server-side, also controls the behavior of of host resolution when initiating server-to-server (or proxy, broker, etc.) communications.

Running p4d as an unprivileged user

Perforce does not require privileged access. For security reasons, do not run **`p4d`** as `root` or otherwise grant the owner of the **`p4d`** process `root`-level privileges.

Create an unprivileged UNIX user (for example, `perforce`) to manage **`p4d`** and (optionally) a UNIX group for it (for example, `p4admin`). Use the **`umask(1)`** command to ensure that the server root (`P4ROOT`) and all files and directories created beneath it are writable only by the UNIX user `perforce`, and (optionally) readable by members of the UNIX group `p4admin`.

Under this configuration, the Perforce service (**`p4d`**), running as UNIX user `perforce`, can write to files in the server root, but no users are able to read or overwrite its files. To grant access to the files created by **`p4d`** (that is, the depot files, checkpoints, journals, and so on) to trusted users, you can add the trusted users to the UNIX group `p4admin`.

Running from inetd on UNIX

Under a normal installation, the Perforce service runs on UNIX as a background process that waits for connections from users. To have **`p4d`** start up only when connections are made to it, using `inetd` and **`p4d -i`**, add the following line to `/etc/inetd.conf`:

```
p4dservice stream tcp nowait username /usr/local/bin/p4d p4d -i -r p4droot
```

and then add the following line to `/etc/services`:

```
p4dservice nnnn /tcp
```

where:

- *`p4dservice`* is the service name you choose for this Perforce server
- `/usr/local/bin` is the directory holding your **`p4d`** binary
- *`p4droot`* is the root directory (`P4DROOT`) to use for this Perforce server (for example, `/usr/local/p4d`)
- *`username`* is the UNIX user name to use for running this Perforce server
- *`nnnn`* is the port number for this Perforce server to use

The "extra" **`p4d`** on the `/etc/inetd.conf` line must be present; **`inetd`** passes this to the OS as `argv[0]`. The first argument, then, is the `-i` flag, which causes **`p4d`** not to run as a background process, but rather to serve the single client connected to it on stdin/stdout. (This is the convention used for services started by **`inetd`**.)

This method is an alternative to running **`p4d`** from a startup script. It can also be useful for providing special services; for example, at Perforce, we have a number of test servers running on UNIX, each defined as an `inetd` service with its own port number.

There are caveats with this method:

- `inetd` may disallow excessive connections, so a script that invokes several thousand **`p4`** commands, each of which spawns an instance of **`p4d`** via `inetd` can cause `inetd` to temporarily disable the service. Depending on your system, you might need to configure `inetd` to ignore or raise this limit.
- There is no easy way to disable the server, since the **`p4d`** executable is run each time; disabling the server requires modifying `/etc/inetd.conf` and restarting `inetd`.
- To use Perforce with this license, you will need to request a server license that does not specify a port. Contact Perforce licensing for more information.

Note	For information about using `systemd` to launch services and daemons at boot time see http://answers.perforce.com/articles/KB/10832.

Starting the Perforce service

After you set **`p4d`**'s `P4PORT` and `P4ROOT` environment variables, start the service by running **`p4d`** in the background with the command:

```
$ p4d &
```

Although the example shown is sufficient to run **`p4d`**, you can specify other flags that control such things as error logging, checkpointing, and journaling.

Example 2.1. Starting the Perforce service

You can override `P4PORT` by starting **`p4d`** with the `-p` flag (in this example, listen to port 1818 on IPv6 and IPv4 transports), and `P4ROOT` by starting **`p4d`** with the `-r` flag. Similarly, you can specify a journal file with the `-J` flag, and an error log file with the `-L` flag. A startup command that overrides the environment variables might look like this:

```
$ p4d -r /usr/local/p4root -J /var/log/journal -L /var/log/p4err -p tcp64:[::]:1818 &
```

The `-r`, `-J`, and `-L` flags (and others) are discussed in Chapter 6, "Backup and Recovery" on page 105. To enable SSL support, see "Using SSL to encrypt connections to a Perforce server" on page 66. A complete list of flags is provided in the Perforce Server (p4d) Reference on page 231.

For information about the files that have been installed, see "Installed files" on page 28.

Stopping the Perforce service

To shut down the Perforce service, use the command:

```
$ p4 admin stop
```

Only a Perforce superuser can use **p4 admin stop**.

Restarting a running Perforce service

To restart a running Perforce service (for example, to read a new license file), use the command:

```
$ p4 admin restart
```

Only a Perforce superuser can use **p4 admin restart**. On UNIX platforms, you can also use **kill -HUP** to restart the service.

Windows installation

To install Perforce on Windows, use the Perforce installer (**perforce.exe**) from the Downloads page of the Perforce web site:

http://www.perforce.com/downloads/complete_list

Use the Perforce installer program to install or upgrade the Perforce service, Perforce proxy, broker, or the Perforce Command-Line Client. Other Perforce applications on Windows, such as the Perforce Visual Client (P4V), as well as third-party plug-ins, may be downloaded and installed separately.

For an example of how to install Perforce on Windows, see:

http://answers.perforce.com/articles/KB_Article/Example-Windows-Installation

Note	If you have Administrator privileges, it is usually best to install Perforce as a service. If you don't, install it as a server.

Windows services and servers

In this manual, the terms *Perforce Service* and **p4d** are used interchangeably to refer to "the process which provides versioning services to Perforce applications" unless the distinction between a Windows server process or a service process is relevant.

The Perforce versioning service (**p4d**) can be configured to run as a Windows service (**p4s.exe**) process that starts at boot time, or as a server (**p4d.exe**) process that you invoke manually from a command prompt. To run a task as a Windows server, the user must be logged in because shortcuts in a user's **startup** folder cannot be run until that user logs in.

The Perforce service (**p4s.exe**) and the Perforce server (**p4d.exe**) executables are copies of each other; they are identical apart from their filenames. When run, the executables use the first three characters of the name with which they were invoked (either **p4s** or **p4d**) to determine their behavior. (For example, invoking copies of **p4d.exe** named **p4smyservice.exe** or **p4dmyserver.exe** invoke a service and a server, respectively.)

By default, the Perforce installer configures Perforce as a Windows service..

Note	On Windows, directory permissions are set securely by default; when Perforce runs as a Windows server, the server root is accessible only to the user who invoked

> **p4d.exe** from the command prompt. When Perforce is installed as a service, the files are owned by the `LocalSystem` account, and are accessible only to those with `Administrator` access.
>
> To allow the Perforce service to run under a regular user account, make sure that the user has read/write access to the registry key and that the user has access to the directory structure under `P4ROOT`. For additional information see the following article:
>
> http://kbportal.perforce.com/article/3925

Installing the Perforce service on a network drive

By default, the Perforce service runs under the local `System` account. Because the `System` account has no network access, a real userid and password are required in order to make the Perforce service work if the metadata and depot files are stored on a network drive. The Perforce service is then configured with the supplied data and run as the specified user instead of `System`.

If you are installing your server root on a network drive, the Perforce installer (**`perforce.exe`**) requests a valid combination of userid and password at the time of installation. This user must have administrator privileges.

Although the Perforce service runs reliably using a network drive as the server root, there is still a marked performance penalty due to increased network traffic and slower file access. Consequently, Perforce recommends that the depot files and Perforce database reside on a drive local to the machine on which the Perforce service is running.

Starting and stopping the Perforce service

If you install Perforce as a service under Windows, the service starts whenever the machine boots. Use the **Services** applet in the **Control Panel** to control the Perforce service's behavior.

To stop a Perforce service, use the command:

```
$ p4 admin stop
```

Only a Perforce superuser can use **`p4 admin stop`**.

For older revisions of Perforce, shut down services manually by using the **Services** applet in the **Control Panel**.

For information about the files that have been installed, see "Installed files" on page 28.

Multiple Perforce services under Windows

By default, the Perforce installer for Windows installs a single Perforce server as a single service. If you want to host more than one Perforce installation on the same machine (for instance, one for production and one for testing), you can either manually start Perforce servers from the command line, or use the Perforce-supplied utility **`svcinst.exe`**, to configure additional Perforce services.

Warning	Setting up multiple services to increase the number of users you support without purchasing more user licenses is a violation of the terms of your Perforce End User License Agreement.

Understanding the precedence of environment variables in determining Perforce configuration is useful when configuring multiple Perforce services on the same machine. Before you begin, read and understand "Windows configuration parameter precedence" on page 26.

To set up a second Perforce service:

1. Create a new directory for the Perforce service.
2. Copy the server executable, service executable, and your license file into this directory.
3. Create the new Perforce service using the **`svcinst.exe`** utility, as described in the example below. (The **`svcinst.exe`** utility comes with the Perforce installer, and can be found in your Perforce server root.)
4. Set up the environment variables and start the new service.

We recommend that you install your first Perforce service using the Perforce installer. This first service is called `Perforce` and its server root directory contains files that are required by any other Perforce services you create on the machine.

Example 2.2. Adding a second Perforce service.

You want to create a second Perforce service with a root in `C:\p4root2` and a service name of `Perforce2`. The **`svcinst`** executable is in the server root of the first Perforce installation you installed in `C:\perforce`.

Verify that your **`p4d.exe`** executable is at Release 99.1/10994 or greater:

```
C:\> p4d -V
```

(If you are running an older release, you must first download a more recent release from http://www.perforce.com and upgrade your server before continuing.)

Create a `P4ROOT` directory for the new service:

```
C:\> mkdir c:\p4root2
```

Copy the server executables, both **`p4d.exe`** (the server) and **`p4s.exe`** (the service), and your license file into the new directory:

```
C:\> copy c:\perforce\p4d.exe c:\p4root2
C:\> copy c:\perforce\p4d.exe c:\p4root2\p4s.exe
C:\> copy c:\perforce\license c:\p4root2\license
```

Use Perforce's **svcinst.exe** (the service installer) to create the Perforce2 service:

```
C:\> svcinst create -n Perforce2 -e c:\p4root2\p4s.exe -a
```

After you create the Perforce2 service, set the service parameters for the Perforce2 service:

```
C:\> p4 set -S Perforce2 P4ROOT=c:\p4root2
C:\> p4 set -S Perforce2 P4PORT=1667
C:\> p4 set -S Perforce2 P4LOG=log2
C:\> p4 set -S Perforce2 P4JOURNAL=journal2
```

Finally, use the Perforce service installer to start the Perforce2 service:

```
$ svcinst start -n Perforce2.
```

The second service is now running, and both services will start automatically the next time you reboot.

Windows configuration parameter precedence

Under Windows, Perforce configuration parameters can be set in many different ways. When a Perforce application (such as **p4** or P4V), or a Perforce server program (**p4d**) starts up, it reads its configuration parameters according to the following precedence:

1. For Perforce applications or a Perforce server (**p4d**), command-line flags have the highest precedence.
2. For a Perforce server (**p4d**), persistent configurables set with **p4 configure**.
3. The P4CONFIG file, if P4CONFIG is set.
4. User environment variables.
5. System environment variables.
6. The Windows user registry (or OS X user preferences) (set by **p4 set**).
7. The Windows system registry (or OS X system preferences) (set by **p4 set -s**).

When a Perforce service (**p4s**) starts up, it reads its configuration parameters from the environment according to the following precedence:

1. Persistent configurables set with **p4 configure** have the highest precedence.
2. Windows service parameters (set by **p4 set -S servicename**).
3. System environment variables.
4. The Windows system registry (or OS X user preferences) (set by **p4 set -s**).

User environment variables can be set with any of the following:

- The MS-DOS **set** command
- The **AUTOEXEC.BAT** file
- The **User Variables** tab under the **System Properties** dialog box in the Control Panel

System environment variables can be set with:

- The **System Variables** tab under the **System Properties** dialog box in the Control Panel.

Starting and stopping the Perforce server

The server executable, **p4d.exe**, is normally found in your P4ROOT directory. To start the server, first make sure your current P4ROOT, P4PORT, P4LOG, and P4JOURNAL settings are correct; then run: **%P4ROOT%\p4d**

To start a server with settings different from those set by P4ROOT, P4PORT, P4LOG, or P4JOURNAL, use **p4d** command-line flags. For example:

```
C:\> C:\test\p4d -r c:\test -p 1999 -L c:\test\log -J c:\test\journal
```

starts a Perforce server process with a root directory of **c:\test**, listening to port **1999**, logging errors to **c:\test\log**, and with a journal file of **c:\test\journal**. The **p4d** command-line flags are case-sensitive.

To stop the Perforce server, use the command:

```
C:\> p4 admin stop
```

For information about the files that have been installed, see "Installed files" on page 28.

Support for long file names

Support for long file names is enabled by default in Perforce server versions 2015.2 or newer. For older versions of the Perforce server, you can enable long filename support on the server with the **filesys.windows.lfn** configurable.

Note	The server root or client root cannot be a long path.

Set **filesys.windows.lfn** to **1** to support filenames longer than 260 characters on Windows platforms. A file name length of up to 32,767 characters is allowed. Each component of the path is limited to 255 characters.

To set on the server, use a command like the following:

```
C:\> p4 configure set filesys.windows.lfn=1
```

Depending on the depth of your workspace path, you might also need to set this configurable on the client and/or proxy (which acts as a client). To set the configurable for a proxy, use a command like the following:

```
C:\> p4 set -S "Perforce Proxy" P4DEBUG=filesys.windows.lfn=1
```

Installed files

Installation adds three types of files to the Perforce server host:

- Database files
- The Journal file
- The Perforce binary

The database files and the Journal file are placed in the root directory of the Perforce server. Eventually, as users and administrators work with Perforce other files are added to the Perforce root directory (`P4ROOT`): user's archived files, checkpoint file, and log files.

The Perforce binary is also installed as shown in the table below

Operating system	Location
Linux download	Wherever the administrator puts it. Usually `/usr/local/bin/p4d` or `/opt/perforce/bin/p4d`
SDP for CentOS	`/opt/perforce/bin/p4d`
SDP for Ubuntu	`/opt/perforce/bin/p4d`
Windows download	Where the administrator puts it. By default it is downloaded to the following directory: `C:\Program Files\Perforce\Server\p4d`
Mac OS X	Where the administrator puts it. Usually `/usr/bin/p4d` or `/user/local/bin/p4d`

Upgrading the Perforce service

You *must* back up your Perforce installation (see "Backup procedures" on page 111) as part of any upgrade process.

Warning

Before you upgrade the Perforce service, always read the release notes associated with your upgraded installation.

In order to upgrade from 2013.2 (or earlier) to 2013.3 (or later), you *must* restore the database from a checkpoint. See "Checkpoints for database tree rebalancing" on page 167 for an overview of the process and "Upgrading p4d - between 2013.2 and 2013.3" on page 30 for instructions specific to this upgrade.

In replicated and distributed environments (see *Helix Versioning Engine Administrator Guide: Multi-site Deployment*), all replicas must be at the same release level as the master. Any functionality that requires an upgrade for the master requires an upgrade for the replica, and vice versa.

Using old Perforce applications after an upgrade

Although older Perforce applications generally work with newer versions of Perforce, some features in new server releases require upgrades to Perforce applications. In general, users with older applications are able to use features available from the Perforce versioning service at the user application's release level, but are not able to use the new features offered by subsequent upgrades to the service.

Licensing and upgrades

To upgrade Perforce to a newer version, your Perforce license file must be current. Expired licenses do not work with upgraded versions of Perforce.

Upgrading p4d

Follow the instructions in this section if both your old and new versions are 2013.3 or later, or if both old and new versions are 2013.2 or earlier.

Read the *Release Notes* for complete information on upgrade procedures.

Warning	In order to upgrade from 2013.2 (or earlier) to 2013.3 (or later), you *must* restore the database from a checkpoint. See "Checkpoints for database tree rebalancing" on page 167 for an overview of the process, and "Upgrading p4d - between 2013.2 and 2013.3" on page 30 instructions specific to this upgrade.

In general, Perforce upgrades require that you:

1. Stop the Perforce service (**`p4 admin stop`**).
2. Make a checkpoint and back up your old installation. (see "Backup procedures" on page 111)
3. Verify your files, see "Verifying files during server upgrades" on page 31 for more information.
4. Run the **`p4d -xv`** and **`p4d -xx`** commands to ensure that **`db.*`** files are OK before the upgrade.
5. Replace the **`p4d`** executable with the upgraded version.

   On UNIX, replace the old version of **`p4d`** with the new version downloaded from the Perforce website. On Windows, use the Perforce installer (**`perforce.exe`**); the installer automatically replaces the executable.
6. Some upgrades (installations with more than 1000 changelists, or upgrades with certain database changes) may require that you manually upgrade the database by running:

   ```
   p4d -r server_root -J journal_file -xu
   ```

   This command may take considerable time to complete. You must have sufficient disk space to complete the upgrade.

7. Restart the Perforce service with your site's usual parameters.

If you have any questions or difficulties during an upgrade, contact Perforce technical support.

Upgrading p4d - between 2013.2 and 2013.3

Follow the instructions in this section if your old version is 2013.2 or earlier *and* your new version is 2013.3 or later.

Perforce 2013.3 contains major changes to the Perforce database implementation. These changes allow for increased concurrency and scalability, and increase the size limit for the **db.*** database files to 16TB.

Although the **db.*** database file format has changed, the checkpoint and journal file formats are identical. In order to upgrade from 2013.2 (or earlier) to 2013.3 (or later), you *must* restore the database from a checkpoint. To do this:

1. Stop the Perforce service (**p4 admin stop**).
2. Make a checkpoint and back up your old installation. (see "Backup procedures" on page 111)
3. If a file called **tiny.db** exists in your old server root, you must back it up separately by running the following command with the old **p4d**:

   ```
   p4d -xf 857 > tiny.ckp
   ```

4. Remove the old **db.*** files, or preferably, move them to a safe location in the event that the upgrade fails.

   ```
   mv your_root_dir /db.* /tmp
   ```

   There must be no **db.*** files in the **P4ROOT** directory when you rebuild a database from a checkpoint. Although the old **db.*** files will not be used again, it's good practice not to delete them until you're certain your upgrade was successful.

5. Remove the **rdb.lbr** file, if it exists.

   The **rdb.lbr** file keeps track of files that need to be transferred to the (local) replica, and may become out of date while the upgrade is underway. Note that this file only exists if your Perforce service was configured as a replica.

6. Replace the old (2013.2 or earlier) **p4d** executable with the new (2013.3 or later) **p4d** executable.

   Do *not* run **p4d -xu** after replacing **p4d** at this time. In this upgrade scenario, you are not upgrading an existing database, you have removed it completely and will rebuild it from the checkpoint that you just took.

7. Use the upgraded **p4d** to replay the checkpoint and rebuild the new database tables:

   ```
   p4d -r $P4ROOT -jr checkpoint_file
   ```

8. If your site uses localized server messages from a message file obtained through Perforce technical support, retrieve the original **message.txt** file and re-create **db.message** in the new database format by running the following command with the new **p4d**:

```
p4d -jr /fullpath/message.txt
```

See "Localizing server error messages" on page 43 for more information.

9. If you created a `tiny.ckp` file as part of your backup process, restore `tiny.db` by running the following command with the new **`p4d`**:

```
$ p4d -xf 857 tiny.ckp
```

10. Run **`p4d -xu`** against the Perforce database to update the database schema:

```
$ p4d -r $P4ROOT -J myJournal -xu
```

11. Restart the Perforce service and resume operations.

Verifying files by signature

Perforce administrators can use the **`p4 verify filenames`** command to validate stored MD5 digests of each revision of the named files. The signatures created when users store files in the depot can later be used to confirm proper recovery in case of a crash: if the signatures of the recovered files match the previously saved signatures, the files were recovered accurately. If a new signature does not match the signature in the Perforce database for that file revision, Perforce displays the characters `BAD!` after the signature.

It is good practice to run **`p4 verify`** before performing your nightly system backups, and to proceed with the backup only if **`p4 verify`** reports no corruption.

For large installations, **`p4 verify`** can take some time to run. The server is also under heavy load while files are being verified, which can impact the performance of other Perforce commands. Administrators of large sites might want to perform **`p4 verify`** on a weekly basis, rather than a nightly basis.

If you ever see a `BAD!` signature during a **`p4 verify`** command, your database or versioned files might be corrupt, and you should contact Perforce Technical Support.

Verifying files during server upgrades

It is good practice to use **`p4 verify`** as follows before and after server upgrades:

1. Before the upgrade, run:

```
$ p4 verify -q //...
```

   to verify the integrity of your server before the upgrade.

2. Take a checkpoint and copy the checkpoint and your versioned files to a safe place.

3. Perform the server upgrade.
4. After the upgrade, run:

```
$ p4 verify -q //...
```

   to verify the integrity of your new system.

Release and license information

The Perforce versioning service is licensed according to how many standard users it supports. There are three types of Perforce users: **standard** users, **operator** users, and **service** users.

- A **standard** user is a traditional user of Perforce.

  Standard users are the default, and each standard user consumes one Perforce license.

- An **operator** user is intended for human or automated system administrators.

  An **operator** user does not require a Perforce license.

- A **service** user is used for server-to-server authentication, whether in the context of remote depots (see "Remote depots and distributed development" on page 58) or in distributed environments.

  Service users do not require licenses, but are restricted to automated inter-server communication processes in replicated and multi-server environments.

Licensing information is contained in a file called **license** in the server root directory. The **license** file is a plain text file supplied by Perforce Software. Without the **license** file, the service limits itself to either 20 users and 20 client workspaces (and unlimited files), or to an unlimited number of users and workspaces (but with a limit of 1000 files).

You can update an existing license file without stopping Perforce by using the **p4 license** command. See "Adding new licensed users" on page 137 for details.

- If the service is running, any user can use **p4 info** to view basic licensing information. Administrators can use **p4 license -u** to obtain more detailed information about how many users and files are in use.
- If the service is down, you can also obtain licensing information by running **p4d -V** from the server root directory where the **license** file resides, or by specifying the server root directory either on the command line (**p4d -V -r server_root**) or in the P4ROOT environment variable.

The server version is also displayed when you invoke **p4d -V** or **p4 -V**.

Chapter 3

Configuring the Server

The Perforce service is highly configurable and this is accomplished through the setting of server, client, and proxy configurables. Available configurables number in the hundreds, and it is probably best to set them as you continue to work with the server. This chapter limits itself to describing the configurables you might initially want to configure before you begin working with the server.

The following areas are covered:

- Enabling distributed versioning
- Using p4 typemap to determine a file's type and to implement site-wide exclusive locking
- Defining additional depots
- Managing client requests
- Managing case sensitivity and Unicode installations
- Configuring logging
- Configuring P4V settings

For complete information about using the **`p4 configure`** command and all available server, client, and proxy configurables, see *P4 Command Reference* and **`p4 help configurables.`**

Enabling distributed versioning

If you need to enable the transfer of files between a user's local repository and the shared repository, you must set the following configurables: `server.allowfetch` and `server.allowpush`.

Defining filetypes with p4 typemap

Perforce uses the `filesys.binaryscan` configurable to determine how many bytes to examine when determining if a file is of type `text` or `binary`. By default, `filesys.binaryscan` is 65536; if the high bit is clear in the first 65536 bytes, Perforce assumes it to be `text`; otherwise, it is assumed to be `binary`. Files compressed in the `.zip` format (including `.jar` files) are also automatically detected and assigned the type `ubinary`.

Although this default behavior can be overridden by the use of the `-t` ***`filetype`*** flag, it's easy for users to overlook this consideration, particularly in cases where files' types are usually (but not always) detected correctly. Certain file formats, such as RTF (Rich Text Format) and Adobe PDF (Portable Document Format), can start with a series of comment fields or other textual data. If these comments are sufficiently long, such files can be erroneously detected by Perforce as being of type `text`.

The **`p4 typemap`** command solves this problem by enabling system administrators to set up a table that links Perforce file types with filename specifications. If an entry in the typemap table matches a file being added, it overrides the file type that would otherwise be assigned by the Perforce application. For example, to treat all PDF and RTF files as `binary`, use **`p4 typemap`** to modify the typemap table as follows:

```
Typemap:
        binary //....pdf
        binary //....rtf
```

The first three periods ("`...`") in the specification are a Perforce wildcard specifying that all files beneath the root directory are to be included in the mapping. The fourth period and the file extension specify that the specification applies to files ending in `.pdf` (or `.rtf`).

The following table lists recommended Perforce file types and modifiers for common file extensions.

File type	Perforce file type	Description
`.asp`	`text`	Active server page file
`.avi`	`binary+F`	Video for Windows file
`.bmp`	`binary`	Windows bitmap file
`.btr`	`binary`	Btrieve database file
`.cnf`	`text`	Conference link file
`.css`	`text`	Cascading style sheet file
`.doc`	`binary`	Microsoft Word document
`.dot`	`binary`	Microsoft Word template
`.exp`	`binary+w`	Export file (Microsoft Visual C++)
`.gif`	`binary+F`	GIF graphic file
`.gz`	`binary+F`	Gzip compressed file
`.htm`	`text`	HTML file
`.html`	`text`	HTML file
`.ico`	`binary`	Icon file
`.inc`	`text`	Active Server include file
`.ini`	`text+w`	Initial application settings file
`.jpg`	`binary`	JPEG graphic file
`.js`	`text`	JavaScript language source code file
`.lib`	`binary+w`	Library file (several programming languages)
`.log`	`text+w`	Log file

File type	Perforce file type	Description
`.mpg`	`binary+F`	MPEG video file
`.pdf`	`binary`	Adobe PDF file
`.pdm`	`text+w`	Sybase Power Designer file
`.ppt`	`binary`	Microsoft PowerPoint file
`.prefab`	`binary`	Unity3D file
`.xls`	`binary`	Microsoft Excel file

Use the following **`p4 typemap`** table to map all of the file extensions to the Perforce file types recommended in the preceding table.

```
# Perforce File Type Mapping Specifications.
#
#  TypeMap:     a list of filetype mappings; one per line.
#               Each line has two elements:
#               Filetype: The filetype to use on 'p4 add'.
#               Path:     File pattern which will use this filetype.
# See 'p4 help typemap' for more information.
TypeMap:

        text //....asp
        binary+F //....avi
        binary //....bmp
        binary //....btr
        text //....cnf
        text //....css
        binary //....doc
        binary //....dot
        binary+w //....exp
        binary+F //....gif
        binary+F //....gz
        text //....htm
        text //....html
        binary //....ico
        text //....inc
        text+w //....ini
        binary //....jpg
        text //....js
        binary+w //....lib
        text+w //....log
        binary+F //....mpg
        binary //....pdf
        text+w //....pdm
        binary //....ppt
        binary //....xls
```

If a file type requires the use of more than one file type modifier, specify the modifiers consecutively. For example, `binary+lFS10` refers to a `binary` file with exclusive-open (`l`), stored in full (`F`) rather than compressed, and for which only the most recent ten revisions are stored (`S10`).

For more information, see the **`p4 typemap`** page in the *P4 Command Reference*.

Implementing site-wide exclusive locking with p4 typemap

By default, Perforce supports concurrent development, but environments in which only one person is expected to have a file open for edit at a time can implement site-wide exclusive locking by using the `+l` (exclusive open) modifier as a partial filetype. If you use the following typemap, the `+l` modifier is automatically applied to all newly added files in the depot:

```
Typemap:
        +l //depot/...
```

If you use this typemap, any files your users add to the depot after you update your typemap automatically have the `+l` modifier applied, and may only be opened for edit by one user at a time. The typemap table applies only to new additions to the depot; after you update the typemap table for site-wide exclusive open, files previously submitted without `+l` must be opened for edit with **`p4 edit -t+l filename`** and resubmitted. Similarly, users with files already open for edit must update their filetypes with **`p4 reopen -t+l filename`**.

Defining depots

By default, the standard depot `Depot` is created in the server when the server starts up. Depending on your user's needs, you can change its name and you can create additional depots to serve your needs:

- Additional standard depots allow you to organize user's work in relevant categories.
- Stream depots are dedicated to the organization and management of streams.
- Remote depots are used to facilitate the sharing of code.
- A spec depot is used to track changes to user-edited forms such as workspace specifications, jobs, branch mappings, and so on.
- Archive depots are used to offline storage of infrequently needed content.
- Unload depots are used to offline storage of infrequently needed metadata.

Please see Chapter 4, "Working with Depots" on page 53 for more information.

Managing client requests

The following sections describe configuration options that relate to handling client requests.

Using P4PORT to control access to the server

Under most circumstances, your Perforce server's `P4PORT` setting consists of a port number. Users must know the IP address (or be able to resolve it from a hostname) of the Perforce server in order to connect to it.

The value of `P4PORT` however, can also include an IP address or hostname that resolves to an IP address. You can set `P4PORT` to configure the following possibilities:

- `P4PORT=`*`portnumber`*

  In this case, the server listens on the specified port for every IP address associated with this host.

- `P4PORT=`*`ipaddress`*`|`*`hostname`*`:`*`portnumber`*

  In this case, the server listens on the specified port for the specified IP address or host name, and it ignores requests to any other IP address.

- `P4PORT=localhost:`*`portnumber`*

  In this case, the server listens on the specified port for requests that originate from users on this host. This forces the Perforce server to ignore all non-local connection requests.

`P4PORT` might also specify a protocol (*`protocol:address:port`*), which further restricts possible connections to those using the specified protocol. For complete information, see the description of the `P4PORT` variable in the *P4 Command Reference*.

Requiring minimum client revisions

The Perforce versioning service offers a mechanism to control which revisions of client applications are able to connect to Perforce.

To require a minimum revision, set the configurables `minClient` to the appropriate revision, and (optionally) set `minClientMessage` to the error message displayed when users of older applications connect to the server.

For example:

```
$ p4 configure set minClient=2010.2
$ p4 configure set minClientMessage="Please upgrade to 2010.2 or higher"
```

Rejecting client connection requests

You can set the `rejectList` configurable to block one or more client programs from accessing the Perforce server. By default, all clients may access the server.

The simple version of the syntax for setting `rejectList` is the following:

```
rejectList = progName [[,progName]...]
```

The syntax of ***progName*** is the following:

```
progName[,version=versionName]
```

For example, the following command blocks requests from all command line clients.

```
$ p4 configure set "rejectList = p4"
```

The following command blocks requests from command line clients version 13.1 and 13.2.

```
$ p4 configure set "rejectList = p4, version=13.1, p4, version=13.2"
```

You may not use any wild card character in the program name parameter.

The comma is used as the default separator for the elements in `rejectList`. If the version number for the program you are excluding includes a comma, you must use a slightly more complicated syntax to define `rejectList`:

```
rejectList = separator=char  progName [[char progName]...]
```

If you define ***char*** to be **#**, the previous command line would look like this:

```
$ p4 configure set "rejectList = separator=# p4, version=13,1# p4# version=13,2"
```

The rejected connection is never logged; the log will not include information about the connection attempt.

You can specify a version using a build number; for example:

```
$ p4 configure set "rejectList = p4, version=1221235"
```

Or you can use platform information; for example:

```
$ p4 configure set "rejectList = p4, version=DARWIN90X86_64"
```

Or you can block for either condition; for example:

```
$ p4 configure set "rejectList = p4, version=1221235, p4, version=DARWIN90X86_64"
```

Note the use of quotation marks for strings that include spaces!

Important

If you accidentally lock out key clients needed to access the server, use the following command to unset the configurable:

```
$ p4d -r P4ROOT '-cunset rejectList'
```

Disabling user metrics collection prompt

P4V users have the option of enabling user metrics collection. By default no data is collected. The first time a user connects to the server, a prompt is displayed asking if the user wants to send Perforce anonymous user data. Such data includes information about system hardware, non-default user preferences, and so on. The user can subsequently change collection preference using the Preferences menu.

If you do not want users to see the prompt, you can set a property on the server as follows:

```
$ p4 property -a -n P4.DataAnalyticsPrompt -v off
```

This prevents the user from seeing the prompt. However, this is an incomplete solution, because if the user connects to a server that does not have the property set, they will see the prompt and might choose to send the data. To fully disable this feature, you will need to have IT shut down any outgoing POST requests to udc.perforce.com.

Case sensitivity and multi-platform development

Very early (pre-97.2) releases of the Perforce server treated all filenames, pathnames, and database entity names with case significance, whether the server was running on UNIX or Windows.

For example, `//depot/main/file.c` and `//depot/MAIN/FILE.C` were treated as two completely different files. This caused problems where users on UNIX were connecting to a Perforce server running on Windows because the filesystem underlying the server could not store files with the case-variant names submitted by UNIX users.

In release 97.3, the behavior was changed, and only the UNIX server supports case-sensitive names. However, there are still some case-sensitivity problems that users can encounter when sharing development projects across UNIX and Windows.

If you are running a pre-97.2 server on Windows, please contact support@perforce.com to discuss upgrading your server and database.

For current releases of the server:

- The Perforce server on UNIX supports case-sensitive names.
- The Perforce server on Windows ignores case differences.
- Case is always ignored in keyword-based job searches, regardless of platform.

The following table summarizes these rules.

Case-sensitive	UNIX server	Windows server
Pathnames and filenames	Yes	No
Database entities (workspaces, labels, and so on.)	Yes	No
Job search keywords	No	No

To find out what platform your Perforce server runs on, use **`p4 info`**.

Perforce server on UNIX

If your Perforce server is on UNIX, and you have users on both UNIX and Windows, your UNIX users must be very careful not to submit files whose names differ only by case. Although the UNIX server can support these files, when Windows users sync their workspaces, they'll find files overwriting each other.

Conversely, Windows users will have to be careful to use case consistently in filenames and pathnames when adding new files. They might not realize that files added as `//depot/main/one.c` and `//depot/MAIN/two.c` will appear in two different directories when synced to a UNIX user's workspace.

The UNIX Perforce server always respects case in client names, label names, branch view names, and so on. Windows users connecting to a UNIX server should be aware that the lowercased workstation names are used as the default names for new client workspaces. For example, if a new user creates a client workspace on a Windows machine named `ROCKET`, his client workspace is named `rocket` by default. If he later sets `P4CLIENT` to `ROCKET` (or `Rocket`, Perforce will tell him his workspace is undefined. He must set `P4CLIENT` to `rocket` (or unset it) to use the client workspace he defined.

Perforce server on Windows

If your Perforce server is running on Windows, your UNIX users must be aware that their Perforce server will store case-variant files in the same namespace.

For example, users who try something like this:

```
C:\> p4 add dir/file1
C:\> p4 add dir/file2
C:\> p4 add DIR/file3
```

should be aware that all three files will be stored in the same depot directory. The depot pathnames and filenames assigned to the Windows server will be those first referenced. (In this case, the depot pathname would be `dir`, and not `DIR`.)

Setting up and managing Unicode installations

The following sections describe the benefits of running the Perforce server in Unicode mode and explain how you enable this mode.

Warning	Converting a server to Unicode mode is a one-way operation! You cannot restore a Unicode server to its previous state.

Overview

The Perforce server can be run in Unicode mode to convert certain elements from their unicode representation on the server, to the particular character set used on clients and triggers that communicate with the server. The following elements are converted:

- File names or directory names that contain Unicode characters
- Perforce identifiers (for example, user names) and specifications (for example, changelist descriptions or jobs) that contain Unicode characters

  If you need to manage textual files that contain Unicode characters, but do not need the features listed above, you do not need to run your server in Unicode mode. For such installations, assign the Perforce `utf16` file type to textual files that contain Unicode characters.

- `unicode` files and metadata. These are converted to the character set configured on the user's machine.

  The Perforce server also verifies that the unicode files and metadata contain valid UTF-8 characters.

Normally, setting the server in Unicode mode should automatically configure the appropriate rendering for each client, independently of the platform where it runs. However, there are some cases in which you might also have to configure the client. The following subsections describe how you set up the server and the client if needed, and offer some troubleshooting tips.

In addition to affecting the client, Unicode settings also affect trigger scripts that communicate with the server. You should check your trigger's use of the elements noted above (file names, Perforce identifiers, etc.) and make sure that these are consistent with the character set used by the server.

Note	All p4d error and info logs are in UTF8 for a server in unicode mode. You need an UTF8 console or editor to properly render this log information.

Setting up a server for Unicode

How you configure a Unicode-mode server and the workstations that access it, depends on whether you are starting a server for the first time or whether you are converting an existing non-unicode server to unicode mode. The following sections explain each use case.

Note	The Perforce service limits the lengths of strings used to index job descriptions, to specify filenames and view mappings, and to identify client workspaces, labels, and other objects. The most common limit is 2,048 bytes. Because no basic Unicode character expands to more than three bytes, you can ensure that no name exceeds the Perforce limit by limiting the length of object names and view specifications to 682 characters for Unicode-mode servers.

Configuring a new server for Unicode

To configure a new server for Unicode, start the server using the following command:

```
$ p4d -xi -r server_root [other options]
```

This command verifies that all existing metadata is valid UTF8, and then sets the protected counter `unicode` to indicate that the server now runs in Unicode mode. If you stop and restart the server, it remains in Unicode mode. Once you have placed the server in this mode, you cannot change it to non-unicode mode.

When a client connects to the server, it attempts to discover what the server's setting is, and it sets the `P4_port_CHARSET` variable to reflect that setting. If the server is not in unicode mode, the variable is set to `none`. If the server is set to Unicode, the variable is set to `auto`. Likewise, the client sets the `P4CHARSET` variable to `auto`. The client then examines its environment to figure out what character set it needs to select.

The `P4_port_CHARSET` variable is stored in a file called `.p4enviro`. By default, this file is stored in the user's home directory. To change the file location, the user must set the `P4ENVIRO` variable to the desired path.

Configuring an existing server for Unicode

To convert an existing server to Unicode mode, perform the following steps:

1. Stop the server by issuing the **`p4 admin stop`** command.
2. Create a server checkpoint, as described in Chapter 6, "Backup and Recovery" on page 105.
3. Convert the server to Unicode mode by invoking the server (**`p4d`**) and specifying the `-xi` flag, for example:

   ```
   p4d -xi -r server_root
   ```

   The server verifies that its existing metadata contains only valid UTF-8 characters, then creates and sets a protected configurable called `unicode` that is used as a flag to ensure that the next time you start the server, it runs in Unicode mode. After validating metadata and setting the configurable, **`p4d`** exits and displays the following message:

   ```
   Server switched to Unicode mode.
   ```

   If the server detects invalid characters in its metadata, it displays error messages like the following:

   ```
   Table db.job has 7 rows with invalid UTF8.
   ```

   In case of such errors, contact Perforce Technical Support for instructions on locating and correcting the invalid characters.

4. Restart **p4d**, specifying server root and port as you normally do. The server now runs in Unicode mode.

When a client connects to the server, it attempts to discover what the server's setting is, and it sets the `P4_port_CHARSET` variable to reflect that setting. If the server is not in Unicode mode, the variable is set to `none`. If the server is set to Unicode, the variable is set to `auto`. Likewise, the client sets the `P4CHARSET` variable to `auto`. The client then examines its environment to figure out what character set it needs to select.

The default location of the `P4_port_CHARSET` variable depends on your operating system:

- On UNIX or on the Mac, the `P4_port_CHARSET` variable is stored in a file called `.p4enviro`. By default, this file is stored in the user's home directory. To change the file location, the user must set the `P4ENVIRO` variable to the desired path.
- On Windows, the `P4_port_CHARSET` variable is stored in the registry. To store it in a file, use the **p4 set P4ENVIRO** command and specify the path of the file where you want to store the value.

Localizing server error messages

By default, the Perforce server informational and error messages are in English. You can localize server messages. To ensure best results, contact Perforce Technical Support. The following overview explains the localization process.

To localize Perforce server messages:

1. Obtain the message file from Perforce Technical Support.
2. Edit the message file, translating messages to the target language. Each message includes a two-character language code. Change the language code from `en` (English) to the code for the target language. Do not translate any of the key parameters or named parameters (which are specified between percent signs and single quotes, for example, `%depot%`). You can change the order in which the parameters appear in the message.

   Original English:

   ```
   @en@ 0 @db.message@ @en@ 822220833 @Depot '%depot%' unknown - use 'depot' to create it.@
   ```

   Correct translation to Portuguese (note reordered parameters):

   ```
   @pt@ 0 @db.message@ @pt@ 822220833 @Depot '%depot' inexistente - use o comando 'depot' para
    criar-lo.@
   ```

   Although you are free to use any two-letter language code to designate the target language (so long as it's not "en," you might want to use a standard convention, such as the one described here:

   http://www.w3schools.com/tags/ref_language_codes.asp

   Many messages use Perforce command names. It is important to distinguish the word as a command name from the word as a description. For example:

```
@Depot '%depot%' unknown - use 'depot' to create it.@
```

In this case, **depot** and **%depot%** should not be translated.

3. Load the translated messages into the server by issuing the following command:

```
$ p4d -jr /fullpath/message.txt
```

This command creates a **db.message** file in the server root. The Perforce service uses this database file when it displays error messages. The Perforce proxy can also use this **db.message** file; see the section on localizing **P4P** in *Helix Versioning Engine Administrator Guide: Multi-site Deployment*

4. The character set of the resulting translation needs to be UTF-8 for unicode mode servers. That file should not have a leading Byte-order-mark (BOM).

If the target server is not in Unicode mode, the translation file does not need to be in UTF-8. In this case you might want multiple instances of the translated messages in multiple character sets. You can effect this by combining the language code field with a character set name. For example, **@ru_koi8-r@** to indicate Russian with a **koi8-r** encoding versus **@ru_iso8859-5@** to indicate Russian with an ISQ encoding.

5. You can load translated message files into a p4d server by recovering them with the server's journal recovery command:

```
$ p4d -r server_root -jr translated_message_file
```

To view localized messages, set the **P4LANGUAGE** environment variable on user workstations to the language code you assigned to the messages in the translated message file. For example, to have your messages returned in Portuguese, set **P4LANGUAGE** to **pt**.

To view localized messages using P4V, you must set the **LANG** environment variable to the language code that you use in the messages file.

Configuring clients for Unicode

When you set up a server to work in unicode mode, the client determines what character set to use by examining the current environment and, generally, you should have nothing more to do to get a correct translation. For example a UNIX client examines the **LANG** or **LOCALE** variables to determine the appropriate character set. However, there might be situations when you need to override the selection made by the client:

- The automatically selected setting is producing bad translations.

  See "Troubleshooting user workstations in Unicode installations" on page 47 for more information.

- You want to use separate workspaces (clients) and each of these needs to use a different character set. In this case, you must set a different **P4CHARSET** value for each client.

- The files you check out need to be accessed by applications for which byte order is important.

  See "Unicode character sets and Byte Order Markers (BOMs)" on page 45 for more information.

- You need to set `P4CHARSET` to an `utf16` or `utf32` setting.

  See "Controlling translation of server output" on page 46 for more information.

- The file is checked out using Perforce client applications that handle Unicode environments in different ways.

  See "Using other Perforce client applications" on page 46 for more information.

In each of these cases, you will need to explicitly set `P4CHARSET` to an appropriate value or take some other action. To get a list of the possible values for `P4CHARSET`, use the command:

```
$ p4 help P4CHARSET
```

Warning

Do not submit a file using a `P4CHARSET` that is different than the one you used to sync it; the file is translated in a way that is likely to be incorrect. That is to say, do not change the value of `P4CHARSET` while files are checked out.

Unicode character sets and Byte Order Markers (BOMs)

Byte order markers (BOMs) are used in Unicode files to specify the order in which multi-byte characters are stored and to identify the file content as Unicode. Not all extended-character file formats use BOMs.

To ensure that such files are translated correctly by the Perforce server when the files are synced or submitted, you must set `P4CHARSET` to the character set that corresponds to the format used on your workstation by the applications that access them, such as text editors or IDEs. Typically the formats are listed when you save the file using the **Save As...** menu option.

The following table lists valid settings for `P4CHARSET` for specifying byte order properties of Unicode files.

Client Unicode format	BOM?	Big or Little-Endian	Set P4CHARSET to	Remarks
UTF-8	No	(N/A)	`utf8`	Suppresses Perforce server UTF-8 validation
	Yes		`utf8-bom`	
	No		`utf8unchecked`	
	Yes		`utf8unchecked-bom`	

Client Unicode format	BOM?	Big or Little-Endian	Set P4CHARSET to	Remarks
UTF-16	Yes	Per client	`utf16`	Synced with a BOM according to the client platform byte order
	Yes	Little	`utf16le`	Best choice for Windows Unicode files
	Yes	Big	`utf16be`	
	No	Per client	`utf16-nobom`	
	No	Little	`utf16le-nobom`	
	No	Big	`utf16be-nobom`	
UTF-32	Yes	Per client	`utf32`	Synced with a BOM according to the client platform byte order
	Yes	Little	`utf32le`	
	Yes	Big	`utf32be`	
	No	Per client	`utf32-nobom`	
	No	Little	`utf32le-nobom`	
	No	Big	`utf32be-nobom`	

If you set `P4CHARSET` to a UTF-8 setting, the Perforce server does not translate text files when you sync or submit them. Perforce does verify that such files contain valid UTF-8 data.

Controlling translation of server output

If you set `P4CHARSET` to any `utf16` or `utf32` setting, you must set the `P4COMMANDCHARSET` to a non-`utf16` or non-`utf32` character set in which you want server output displayed. "Server output" includes informational and error messages, diff output, and information returned by reporting commands.

To specify `P4COMMANDCHARSET` on a per-command basis, use the `-Q` flag. For example, to display all filenames in the depot, as translated using the `winansi` code page, issue the following command:

```
C:\> p4 -Q winansi files //...
```

Using other Perforce client applications

If you are using other Perforce client applications, note how they handle Unicode environments:

- **P4V (Perforce Visual Client):** the first time you connect to a Unicode-mode server, you are prompted to choose the character encoding. Thereafter, P4V retains your selection in association with the connection. P4V also has a global default setting for Charset. If you set this, it will be used instead of asking you to provide a charset.
- **P4Eclipse** will ask for a charset when connecting to a Unicode-mode server.
- **P4Web:** when you invoke P4Web, you can specify the character encoding on the command line using the `-C` flag. P4Web uses this flag when it sends commands to a Unicode-mode server. This approach means that each instance of P4Web can handle a single character encoding and that browser machines must have compatible fonts installed.
- **P4Merge:** To configure the character encoding used by P4Merge, choose P4Merge's **File > Character Encoding...** menu option. When launched from P4V, P4Merge uses P4V's `P4CHARSET` instead of the one defined in it's preferences.
- **IDE SCC plug-in:** the first time you connect to a Unicode-mode server, you are prompted to choose the character encoding. Thereafter, the plug-in retains your selection in association with the connection.
- **P4GT** and **P4EXP** use environmental settings and will fail with a Unicode-mode server.

Troubleshooting user workstations in Unicode installations

To prevent file corruption, it is essential that you configure your workstation correctly. The following section describes common problems and provides solutions.

- "Cannot Translate" error message

  This message is displayed if your workstation is configured with a character set that does not include characters that are being sent to it by the Perforce server. Your workstation cannot display unmapped characters. For example, if `P4CHARSET` is set to `shiftjis` and your depot contains files named using characters from the Japanese EUC character set that do not have mappings in shift-JIS, you see the "Cannot translate" error message when you list the files by issuing the **`p4 files`** command.

  To ensure correct translation, do not use unmappable characters in Perforce user specifications, client specifications, jobs, or file names.

- Strange display of file content

  If you attempt to display an extended-character text file and see odd-looking text, your workstation might lack the font required to display the characters in the file. Typical symptoms of this problem include the display of question marks or boxes in place of characters. To solve this problem, install the required font.

Configuring logging

You might want to address the following issues in setting up logging. For information on setting up structured logging, see "Logging and structured log files" on page 125.

Logging errors

Use the `-L` flag to **`p4d`** or the environment variable `P4LOG` to specify Perforce's error output file. If no error output file is defined, errors are dumped to the **`p4d`** process' standard error. Although **`p4d`** tries to ensure that all error messages reach the user, if an error occurs and the user application disconnects before the error is received, **`p4d`** also logs these errors to its error output.

Perforce also supports trace flags used for debugging. See "Setting server trace and tracking flags" on page 123 for details.

Logging file access

If your site requires that user access to files be tracked, use the `-A` flag to **`p4d`** or the environment variable `P4AUDIT` to activate auditing and specify the Perforce's audit log file. When auditing is active, every time a user accesses a file, a record is stored in the audit log file. This option can consume considerable disk space on an active installation.

See "Auditing user file access" on page 125 for details.

Configuring P4V settings

Not every site (nor every user at every site) requires the full suite of functionality in P4V, the Perforce Visual Client. By using the **`p4 property`** command, it is possible for an administrator to control which P4V features are available for a given site, group, or user. Properties relate to performance, features, or enabling the rich comparison of Microsoft .docx files. Performance and feature-related properties set at the server level override local P4V settings.

Configuring performance-related properties

If a user connects to a new Perforce service, performance-related properties are reloaded for the Perforce service to which the user has most recently connected.

Property	Meaning
`P4V.Performance.FetchCount`	Number of changelists, jobs, branch mappings, or labels to fetch at any one time.
`P4V.Performance.OpenedLimit`	Limits the number of files to check in the 'opened' call during a rollback operation. Default value is 1000. If the number of files to roll back exceeds the configured value, a popup informs the user that no opened check will be performed, and asks if the user wants to complete the operation.
`P4V.Performance.MaxFiles`	Maximum number of files displayed per changelist.
`P4V.Performance.MaxPreviewSize`	Maximum size of files to preview, in kilobytes.

Property	Meaning
`P4V.Performance.ServerRefresh`	Number of time between display refreshes, in minutes.

Configuring feature-related properties

You can use the following properties to enable or disable P4V features. These properties are read once, upon P4V startup, from the first service to which the user connects. Features that are deactivated by setting these properties to `Off` are unavailable in P4V and do not display in P4V's Preferences page:

Property	Meaning
`P4V.Features.Integration`	If `Off`, users cannot integrate.
`P4V.Features.Labeling`	If `Off`, the labels tab does not appear.
`P4V.Features.Jobs`	If `Off`, jobs support is disabled. Jobs do not appear in changelists, etc.
`P4V.Features.RevisionGraph`	If `Off`, the Revision Graph is disabled.
`P4V.Features.Timelapse`	If `Off`, Time-Lapse View is disabled.
`P4V.Features.CustomTools`	If `Off`, the Manage Custom Tools dialog is disabled.
`P4V.Features.Administration`	If `Off`, the Administration menu option is not displayed.
`P4V.Features.ConnectionWizard`	If `Off`, P4V does not attempt to use the New Connection Wizard.
`P4V.Features.Workspaces`	If `Off`, users cannot edit or display their own (or other users') workspaces.
`P4V.Features.DashBoard`	If `Off`, the Dashboard is not displayed.
`P4V.Features.P4Applets`	If `Off`, Perforce applets are disabled in P4V, and the menu option to re-enable them is no longer accessible.
`P4V.Features.Streams`	If `Off`, streams-related icons, menus, and the Stream Graph do not appear.
`P4V.Features.Parallel`	If `Force`, parallel sync and submit is enabled regardless of user preference. If `Off`, parallel sync cannot be enabled by the user. P4V defaults to using four threads.

For example, the administrator of a site that does not use Perforce's built-in defect tracking can disable access to jobs from within P4V by running:

```
$ p4 property -a -n P4V.Features.Jobs -v Off
```

A new property is added/updated (`-a`), it is named (`-n`) `P4V.Features.Jobs`, and it is assigned the value (`-v`) of `Off`.

If one group of users within the organization has a need to use the jobs functionality of P4V, the feature can be selectively (and centrally) re-enabled for those users with:

```
$ p4 property -a -n P4V.Features.Jobs -v On -g jobusers
```

The jobs feature of P4V is re-enabled by setting its value to `On`, but only for users in the `jobusers` group.

Configuring Swarm connections

In order for P4V to connect to a Swarm server, it must know where the server is installed. Because Swarm is a web application, a URL can specify its location.

The Swarm or P4V administrator uses the `P4.Swarm.URL[.serverid]` property to specify the location of a Swarm server.

- To identify the location of a single Swarm server, use either the `P4.Swarm.URL` or the `P4.Swarm.URL[.serverid]` syntax, depending on whether the server has a serverid. For example, the following command specifies that the location of the server given by `10.5.40.145:1666` is `https://my_swarm_server.com`.

  ```
  $ p4 -p "10.5.40.145:1666" property -a -n P4.Swarm.URL -v "https://my_swarm_server.com"
  ```

- To identify the location of several Swarm server instances, use the `P4.Swarm.URL[.serverid]` syntax, and specify the server id for each Swarm server each time you invoke the **`p4 property`** command. For example:

  ```
  $ p4 -p "10.5.40.145:1666" property -a -n P4.Swarm.URL.svr1 -v "https://my_swarm_server1.com"
  $ p4 -p "10.5.40.145:1667" property -a -n P4.Swarm.URL.svr2 -v "https://my_swarm_server2.com"
  ```

  Using the server id format is only necessary if you are using an authentication server (and multiple **`p4d`** instances are funneling through it) or if you are deploying multiple instances of Swarm against replicas or edge servers.

When P4V attempts to connect to a server that has no serverid, it checks to see if the property `P4.Swarm.URL` is set, and it uses that URL to access Swarm. If the property is not set, P4V does not attempt to talk to Swarm.

When P4V attempts to connect to a server that has a serverid,

1. P4V asks the server for its server id and gets, for example, `svr1`.
2. P4V checks the setting of `p4.Swarm.URL.svr1`, and it uses that URL to talk to Swarm
3. If `p4.Swarm.URL.svr1` is not set, P4V checks the value of `p4.Swarm.URL` and uses that value to access the Swarm server.

4. If `p4.Swarm.URL` is not set, P4V does not attempt to talk to Swarm.

If there is a value both for `p4.Swarm.URL` and for `p4.Swarm.URL.myserverid` when P4V attempts to connect to a Swarm server, the serverid match takes precedence over the generic match.

The user issuing the **`p4 property`** command must have an account on the specified Swarm server.

You can use the **`p4 property`** command to list the current properties of the Swarm server; for example:

```
$ p4 -p "10.5.40.145:1666" property -l -A
   P4.Swarm.Timeout = 10 (any) #1
   P4.Swarm.URL.master-1666 = https://my_swarm_server1.com
```

Enabling .docx diffs

You can use the `P4.Combine.URL` property to enable the rich comparison of Microsoft Word `.docx` files through P4Merge. You must deploy P4Combine as part of a Commons deployment to use this feature.

To enable the feature, set the `P4.Combine.URL` server property to the P4Combine Web Service URL. P4V will then display a rich compare of `.docx` files in the P4Merge window, using HTML5 to show differences for text, images, formats, styles, tables, headers, footers, and other objects.

Windows configuration parameter precedence

Under Windows, Perforce configuration parameters can be set in many different ways. When a Perforce application (such as **`p4`** or P4V), or a Perforce server program (**`p4d`**) starts up, it reads its configuration parameters according to the following precedence:

1. For Perforce applications or a Perforce server (**`p4d`**), command-line flags have the highest precedence.
2. For a Perforce server (**`p4d`**), persistent configurables set with **`p4 configure`**.
3. The `P4CONFIG` file, if `P4CONFIG` is set.
4. User environment variables.
5. System environment variables.
6. The Windows user registry (or OS X user preferences) (set by **`p4 set`**).
7. The Windows system registry (or OS X system preferences) (set by **`p4 set -s`**).

When a Perforce service (**`p4s`**) starts up, it reads its configuration parameters from the environment according to the following precedence:

1. Persistent configurables set with **`p4 configure`** have the highest precedence.
2. Windows service parameters (set by **`p4 set -S servicename`**).
3. System environment variables.

4. The Windows system registry (or OS X user preferences) (set by **`p4 set -s`**).

User environment variables can be set with any of the following:

- The MS-DOS **`set`** command
- The **`AUTOEXEC.BAT`** file
- The **User Variables** tab under the **System Properties** dialog box in the Control Panel

System environment variables can be set with:

- The **System Variables** tab under the **System Properties** dialog box in the Control Panel.

Chapter 4

Working with Depots

All the versioned files that users work with are stored in a shared repository called a *depot*: files are checked out of the depot for modification and checked back into the depot to archive changes and to share changes with other users.

By default, a depot named `Depot` of type `local` is created in the server when the server starts up. In addition to the default local depot, you can create additional depots of various types:

- Additional `local` depots allow you to organize users' work in relevant categories. You might, for example, want to separate HR source docs from development source docs.
- Stream depots are dedicated to the organization and management of streams.
- A spec depot is used to track changes to user-edited forms such as workspace specifications, jobs, branch mappings, and so on.
- Archive depots are used to offline storage of infrequently needed content.
- Unload depots are used to offline storage of infrequently needed metadata.
- Remote depots are used to facilitate the sharing of code.
- A tangent depot is generated by Perforce and used internally to store conflicting changes during fetch operations. The only action the administrator might want to take with respect to the tangent depot is to rename it if its default name of `tangent` is unacceptable.

This chapter includes general information about working with depots of different types. The **`p4 depot`** command, used to create any type of depot, is described in *P4 Command Reference*.

Overview

New depots are defined with the command **`p4 depot depotname`**. Depots can be defined as `local`, `stream`, `remote`, `unload`, `archive`, or `spec` depots.

Perforce servers can host multiple depots, and Perforce client applications can access files from multiple depots. These other depots can exist on the Perforce server normally accessed by the Perforce client, or they can reside within other, *remote*, Perforce servers.

Naming depots

The name of a depot may not be the same as the name of a branch, client workspace, or label.

Listing depots

To list all depots known to the current Perforce server, use the **`p4 depots`** command.

Deleting depots

To delete a depot, use **`p4 depot -d depotname`**.

To delete a depot, it must be empty; you must first obliterate all files in the depot with **p4 obliterate**.

For `local` and `spec` depots, **p4 obliterate** deletes the versioned files as well as all their associated metadata. For `remote` depots, **p4 obliterate** erases *only* the locally held client and label records; the files and metadata still residing on the remote server remain intact.

Before you use **p4 obliterate**, and *especially* if you're about to use it to obliterate all files in a depot, read and understand the warnings in "Reclaiming disk space by obliterating files" on page 142.

In a distributed environment, the unload depot may have different contents on each edge server. Since the commit server does not verify that the unload depot is empty on every edge server, you must specify **p4 depot -d -f** in order to delete the unload depot from the commit server. For more information, see *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.

Moving depots in a production environment

Follow these steps to move a depot in a production environment:

1. Shut down the server where the depot resides.
2. Move the versioned file tree to its new location.
3. Restart the server so that it listens only on localhost (or on some port other than the one you normally use). For example:

```
$ p4d -p 127.0.0.1:1666 flags_you_normally_use
```

4. Change the map field using the **p4 depot depotname** command.
5. Shut down the server using a command like the following:

```
$ p4d -p 127.0.0.1:1666 admin stop
```

6. Restart the server normally.

Standard depots

Standard or `local`-type depots reside on local, remote, or shared servers. Local-type depots reside on the Perforce server normally accessed by the user's Perforce application. When using local depots, a Perforce application communicates with the Perforce server specified by the user's `P4PORT` environment variable or equivalent setting.

To define a new local depot (that is, to create a new depot in the current Perforce server namespace), call **p4 depot** with the new depot name, and edit only the `Map:` field in the resulting form.

For example, to create a new depot called `book` with the files stored in the local Perforce server namespace in a root subdirectory called `book` (that is, `$P4ROOT/book`), enter the command **p4 depot book**, and fill in the resulting form as follows:

```
Depot:      book
Type:       local
Address:    local
Suffix:     .p4s
Map:        book/...
```

The `Address:` and `Suffix:` fields do not apply to local depots and are ignored.

By default, the `Map:` field on a local depot points to a depot directory matching the depot name, relative to the server root (`P4ROOT`) setting for your server. To store a depot's versioned files on another volume or drive, specify an absolute path in the `Map:` field. This path need not be under `P4ROOT`. Absolute paths in the `Map:` field on Windows must be specified with forward slashes (for instance, `d:/newdepot/`) in the **`p4 depot`** form.

Stream depots

Stream depots contain *streams*, a type of branch that includes hierarchy and policy. Like local depots, stream depots reside on the Perforce server. When creating a stream depot, you must provide the following information: name, owner, date, type, and stream depth. For additional information, see "Working with Stream Depots" in the description of the **`p4 depot`** command.

If you are using the distributed versioning architecture, the personal server uses a stream-type depot.

Spec depot

The spec depot is used to track changes to user-edited forms such as client workspace specifications, jobs, branch mappings, and so on. There can be only one `spec` depot per server. (If you already have a spec depot, attempting to create another one results in an error message.)

In order to retrieve change histories of user-edited forms, you must enable versioned specifications. After you have enabled versioned specs by creating the spec depot, all user-generated forms (such as client workspace specifications, jobs, branch mappings, and so on) are automatically archived as text files in the spec depot. Filenames within the spec depot are automatically generated by the server, and are represented in Perforce syntax as follows:

```
//specdepotname/formtype/[objectname[suffix]]
```

Some ***`formtype`***s (for example, the `protect`, `triggers`, and `typemap` forms) are unique to the server, and do not have corresponding ***`objectname`***s.

Note

As of Release 2011.1, the first line of every saved form stored in the spec depot is a comment line that identifies the user who most recently changed the form:

```
# The form data below was edited by username
```

Creating the spec depot

To create a spec depot named `//spec`, enter **`p4 depot spec`**, and fill in the resulting form as follows:

```
Depot:      spec
Type:       spec
Address:    local
Map:        spec/...
SpecMap:    //spec/...
Suffix:     .p4s
```

The `Address:` field does not apply to spec depots and is ignored.

Using a `Suffix:` is optional, but specifying a file extension for objects in the spec depot simplifies usability for users of applications such as P4V, because users can associate the suffix used for Perforce specifications with their preferred text editor. The default suffix for these files is `.p4s`.

For example, if you create a spec depot named `spec`, and use the default suffix of `.p4s`, your users can see the history of changes to `job000123` by using the command:

```
$ p4 filelog //spec/job/job000123.p4s
```

or by using P4V to review changes to `job000123.p4s` in whatever editor is associated with the `.p4s` file extension on their workstation.

The default `SpecMap:` of `//spec/...` indicates that all specs are to be versioned.

Populating the spec depot with current forms

After you create a spec depot, you can populate it using the **`p4 admin updatespecdepot`** command. This command causes the Perforce Server to archive stored forms (specifically, `client`, `depot`, `branch`, `label`, `typemap`, `group`, `user`, and `job` forms) into the spec depot.

To archive all current forms, use the `-a` flag:

```
$ p4 admin updatespecdepot -a
```

To populate the spec depot with only one type of form (for instance, extremely large sites might elect to update only one table at a time), use the `-s` flag and specify the form ***`type`*** on the command line. For example:

```
$ p4 admin updatespecdepot -s job
```

In either case, only those forms that have not yet been archived are added to the spec depot; after the spec depot is created, you only need to use **`p4 admin updatespecdepot`** once.

Controlling which specs are versioned

By default, all specs (`//spec/...`) are versioned. You can use the `SpecMap:` field to control which specs are versioned by adding lines in depot syntax that include (or exclude) paths in the spec depot.

For example, you can exclude the protections table from versioning by configuring your spec depot's `SpecMap:` field as follows:

```
SpecMap:
    //spec/...
    -//spec/protect/...
```

In an environment such as a build farm, in which large numbers of temporary client workspaces and/or labels are created, you can configure the spec depot to exclude them, while keeping track of other changes to client workspaces and labels. For example, a spec depot configured with the following spec mapping:

```
SpecMap:
    //spec/...
    -//spec/client/build_ws_*
    -//spec/label/temp_label_*
```

will no longer track changes to client workspaces whose names begin with `build_ws_`, nor will it track changes to labels whose names begin with `temp_label_`.

Note that adding or changing the `SpecMap:` field only affects future updates to the spec depot; files already stored in the spec depot are unaffected.

Large sites and old filesystems

Use the `spec.hashbuckets` configurable to define the number of buckets (subdirectories) into which files in the spec depot are hashed. By default, `spec.hashbuckets` is 99; for each type of object, directories associated with objects in the spec depot are allocated between 99 subdirectories.

To disable hashing, set `spec.hashbuckets` to 0, as follows:

```
$ p4 configure set spec.hashbuckets=0
```

With hashing disabled, for each subdirectory for each spec type, one sub-subdirectory is created for each object, and all of these sub-subdirectories are stored in one single subdirectory. Disabling hashing may subject your installation to filesystem-imposed limitations on the maximum number of subdirectories in any one directory (for example, the 32K limit imposed by older `ext2`, `ext3`, and `ufs` filesystems).

Archive depots

Archive depots are used for near-line or offline storage of infrequently-accessed content. For details, see "Reclaiming disk space by archiving files" on page 141.

Unload depot

The unload depot is analogous to the archive depot, but provides a place to store infrequently-accessed metadata (specifically, metadata concerning client workspaces and labels) rather than old versioned files. There can be only one `unload` depot per server. For details, see "Unloading infrequently-used metadata" on page 163.

Remote depots and distributed development

Perforce is designed to cope with the latencies of large networks and inherently supports users with client workspaces at remote sites. A single Perforce installation is ready, out of the box, to support a shared development project, regardless of the geographic distribution of its contributors.

Partitioning joint development projects into separate Perforce installations does not improve throughput, and usually only complicates administration. If your site is engaged in distributed development (that is, developers in multiple sites working on the same body of code), it is better to set up a distributed Perforce installation. For information on setting up and monitoring a distributed Perforce configuration, see the *Helix Versioning Engine Administrator Guide: Multi-site Deployment* manual.

If, however, your organization regularly imports or exports material from other organizations, you might want to consider using Perforce's remote depot functionality to streamline your code drop procedures.

When using remote depots, the user's client application uses the Perforce server specified by the user's `P4PORT` environment variable or equivalent setting as a means to access a second, *remote*, Perforce server. The local Perforce server communicates with the remote Perforce server to access a subset of its files.

Remote depots are designed to support shared *code*, not shared *development*. They enable independent organizations with separate Perforce installations to integrate changes between Perforce installations. Briefly:

- A "remote depot" is a depot on your Perforce server of type `remote`. It acts as a pointer to a depot of type "local" that resides on a second Perforce server.
- A user of a remote depot is typically a build engineer or handoff administrator responsible for integrating software between separate organizations.
- Control over what files are available to a user of a remote depot resides with the administrator of the remote server, *not* the users of the local server.
- See "Restricting access to remote depots" on page 61 for security requirements.

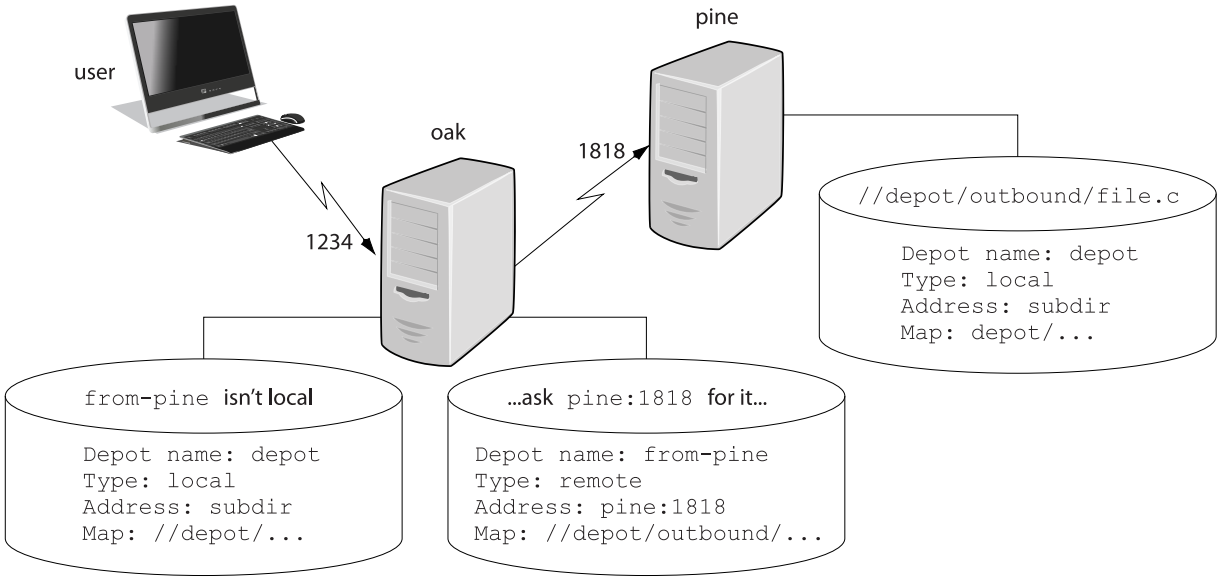
For additional information about the options you have to share code, see "Distributed development using Fetch and Push" on page 131.

How remote depots work

The following diagram illustrates how Perforce applications use a user's default Perforce server to access files in a depot hosted on another Perforce server.

In this example, an administrator of a Perforce server at **`oak:1234`** is retrieving a file from a remote server at **`pine:1818`**.

```
P4PORT=oak:1234
p4 integ //from-pine/file.c //depot/codedrops/file.c
```



Although it is possible to permit individual developers to sync files from remote depots into their client workspaces, this is generally an inefficient use of resources.

The preferred technique for using remote depots is for your organization's build or handoff administrator to integrate files from a remote depot into an area of your local depot. After the integration, your developers can access copies of the files from the local depot into which the files were integrated.

To accept a code drop from a remote depot, create a branch in a local depot from files in a remote depot, and then integrate changes from the remote depot into the local branch. This integration is a one-way operation; you cannot make changes in the local branch and integrate them back into the remote depot. The copies of the files integrated into your Perforce installation become the responsibility of your site's development team; the files on the depot remain under the control of the development team at the other Perforce installation.

Restrictions on remote depots

Remote depots facilitate the sharing of code between organizations (as opposed to the sharing of development within a single organization). Consequently, access to remote depots is restricted to read-only operations, and server metadata (information about client workspaces, changelists, labels, and so on) cannot be accessed using remote depots.

Using remote depots for code drops

Performing a code drop requires coordination between two organizations, namely the site receiving the code drop and the site providing the code drop. In most cases, the following three things must be configured:

- The Perforce administrator at the site receiving the code drop must create a remote depot on his or her Perforce server that points to the site providing the code drop.

  This is described in "Defining remote depots" on page 60.

- The Perforce administrator at the site providing the code drop should configure his or her Perforce server to allow the recipient site's remote depot to access the providing site's Perforce server.

  This is described in "Restricting access to remote depots" on page 61.

- The configuration manager or integration manager at the receiving site must integrate the desired files from the remote depot into a local depot under his or her control.

  This is described in "Receiving a code drop" on page 63.

Defining remote depots

To define a new remote depot:

1. Create the depot with **`p4 depot depotname`**.
2. Set the **`Type:`** to **`remote`**.
3. Direct your Perforce server to contact the remote Perforce server by providing the remote server's name and listening port in the **`Address:`** field.

   A remote server's host and port are specified in the **`Address:`** field just as though it were a **`P4PORT`** setting.
4. Set the **`Map:`** field to map into the desired portion of the remote server's namespace.

   For remote depots, the mapping contains a subdirectory relative to the remote depot namespace. For example, **`//depot/outbound/...`** maps to the **`outbound`** subdirectory of the depot named **`depot`** hosted on the remote server.

   The **`Map:`** field must contain a single line pointing to this subdirectory, specified in depot syntax, and containing the "**`...`**" wildcard on its right side.

   If you are unfamiliar with client views and mappings, see the *Helix Versioning Engine User Guide* for general information about how Perforce mappings work.
5. The **`Suffix:`** field does not apply to remote depots; ignore this field.

In order for anyone on your site to access files in the remote depot, the administrator of the remote server must grant **`read`** access to user **`remote`** to the depots and subdirectories within the depots specified in the **`Map:`** field.

Example 4.1. Defining a remote depot

Lisa is coordinating a project and wants to provide a set of libraries to her developers from a third-party development shop. The third-party development shop uses a Perforce server on host **`pine`** that listens on port **`1818`**. Their policy is to place releases of their libraries on their server's single depot **`depot`** under the subdirectory **`outbound`**.

Lisa creates a new depot from which she can access the code drop; she'll call this depot `from-pine`; she'd type **`p4 depot from-pine`** and fill in the form as follows:

```
Depot:      from-pine
Type:       remote
Address:    pine:1818
Map:        //depot/outbound/...
```

This creates a remote depot called `from-pine` on Lisa's Perforce server; this depot (`//from-pine`) maps to the third party's `depot`'s namespace under its `outbound` subdirectory.

Restricting access to remote depots

Remote depots are accessed either by a virtual user named `remote`, or (if configured) by the service user of the accessing server's **`p4d`**. Service users (including the virtual `remote` user) do not consume Perforce licenses.

Note

Perforce Servers at release 2010.2 authenticate as `remote` to older Perforce servers, and either as `remote` (if no service user is configured), or as the service user (if configured) to Perforce Servers at release 2010.2 and above.

By default, all files on a Perforce server can be accessed remotely. To limit or eliminate remote access to a particular server, use **`p4 protect`** to set permissions for user `remote` (or the remote site's service user) on that server. Perforce recommends that administrators deny access to user `remote` across all files and all depots by adding the following permission line in the **`p4 protect`** table:

```
list user remote * -//...
```

Because remote depots can only be used for `read` access, it is not necessary to remove `write` or `super` access to user `remote` (or the service user). Keep in mind that the virtual user remote does not have access to anything unless that access is granted explicitly in the protection table.

Note

As of Release 2010.2, it remains good practice to deny access to user `remote`. If the Perforce Servers at partner sites are configured to use service users, you can use their service users to further restrict which portions of your server are available for code drops.

As of Release 2010.2, it remains good practice to deny access to user `remote`. If the Perforce Servers at partner sites are configured to use service users, you can use their service users to further restrict which portions of your server are available for code drops.

Example security configuration

Using the two organizations described in "Receiving a code drop" on page 63, a basic set of security considerations for each site would include:

On the local (`oak`) site:

- Deny access to `//from-pine` to all users. Developers at the `oak` site have no need to access files on the `pine` server by means of the remote depot mechanism.

- Grant `read` access to `//from-pine` to your integration or build managers. The only user at the `oak` site who requires access the `//from-pine` remote depot is the user (in this example, `adm`) who performs the integration from the remote depot to the local depot.

  The `oak` administrator adds the following lines to the **`p4 protect`** table:

  ```
  list user * * -//from-pine/...
  read user adm * //from-pine/...
  ```

On the remote (`pine`) site, access to code residing on `pine` is entirely the responsibility of the `pine` server's administrator. At a minimum, this administrator should:

- Preemptively deny access to user `remote` across all depots from all IP addresses:

  ```
  list user remote * -//...
  ```

  Adding these lines to the **`p4 protect`** table is sound practice for any Perforce installation whether its administrator intends to use remote depots or not.

- **If both servers are at Release 2010.2 or higher:** contact the `oak` site's administrator and obtain the name of the `oak` site's service user.

  In this example, the `oak` site's service user is `service-oak`. When a user of the `oak` server accesses a remote depot hosted on `pine`, the `oak` server will authenticate with the `pine` server as a user named `service-oak`.

  As administrator of the `pine` site, you must:

  - Create a service user on your site named `service-oak`. (see "Service users" on page 134). This user's name must match the name of the receiving site's service user.
  - Assign this user a strong password.
  - Inform the `oak` administrator of this password.

    The administrator of the `oak` site must:

  - Use the password set by the pine administrator to obtain a ticket valid for `pine` for the user `service-oak` (that is, run **`p4 login service-oak`** against the `pine` server).
  - Place the ticket somewhere where the `oak` server's **`p4d`** process can access it. (For example, the `.p4tickets` file in the server's root directory, with `P4TICKETS` set to point to the location of the ticket file.)
  - Configure `oak` to work with the `pine` service user, either by starting `oak`'s **`p4d`** process with the `-u service-oak` flag, or configure the server with **`p4 configure set serviceUser=service-oak`**.)
  - Grant `read` access to user `remote` (or the `oak` site's service user) to only those areas of the `pine` server into which code drops are to be placed. Further restrict access to requests originating from the IP address of the Perforce server that is authorized to receive the code drop.

In this example, outgoing code drops reside in `//depot/outbound/...` on the `pine` server. If `oak`'s IP address is `192.168.41.2`, the `pine` site's protections table looks like:

```
list user remote * -//...
read user remote 192.168.41.2 //depot/outbound/...
```

- **If both sites are at Release 2010.2 or higher**, and the `oak` server is configured to use `service-oak` as its service user, the `pine` site's protections table looks like:

```
list user remote * -//...
list user service-oak * -//...
read user service-oak 192.168.41.2 //depot/outbound/...
```

Only Perforce Servers at IP address 192.168.41.2 that have valid tickets for the `pine` site's `service-oak` user, are permitted to access the `pine` server through remote depots, and only `//depot/outbound/...` is accessible.

Receiving a code drop

To perform a handoff or code drop between two Perforce installations:

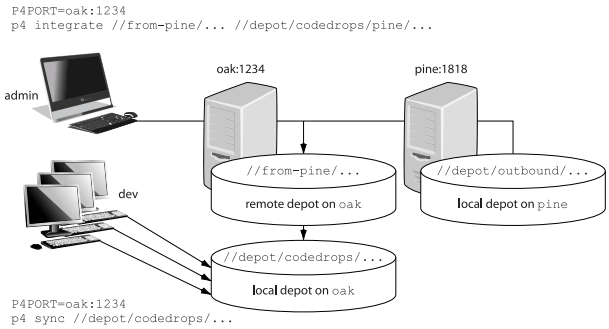
1. Developers on `pine:1818` complete work on a body of code for delivery.

2. The build or release manager on `pine:1818` branches the deliverable code into an area of `pine:1818` intended for outbound code drops. In this example, the released code is branched to `//depot/outbound/...`

3. A Perforce administrator at `oak:1234` configures a remote depot called `//from-pine` on the `oak` server. This remote depot contains a `Map:` field that directs the `oak` server to the `//depot/outbound` area of `pine:1818`.

4. Upon notification of the release's availability, a build or release manager at `oak:1234` performs the code drop by integrating files in the `//from-pine/...` remote depot into a suitable area of the local depot, such as `//depot/codedrops/pine`.

5. Developers at `oak:1234` can now use the `pine` organization's code, now hosted locally under `//depot/codedrops/pine`. Should patches be required to `pine`'s code, `oak` developers can make such patches under `//depot/codedrops/pine`. The `pine` group retains control over its code.


P4PORT=oak:1234
p4 integrate //from-pine/... //depot/codedrops/pine/...
admin
oak:1234
pine:1818
//from-pine/...
remote depot on oak
//depot/outbound/...
local depot on pine
dev
//depot/codedrops/...
local depot on oak
P4PORT=oak:1234
p4 sync //depot/codedrops/...

Chapter 5

Securing the Server

You can set up secure communication between clients and servers as well as between servers.

- Communication between clients and servers can be secured using the SSL protocol, which you specify when connecting to the server. See "Using SSL to encrypt connections to a Perforce server" on page 66 for information on how you secure client-server communication.

  Communication between clients and servers can also be secured using a firewall. For more information, see "Using firewalls" on page 69.

- User authentication can be done using passwords or tickets, and the strength of the password can be defined by an administrator. Users can be authenticated against an Active Directory or LDAP server, or against an internal Helix user database. See "Authentication options" on page 69 for information about how you can authenticate users.

- Access is defined using a protections that determine which Perforce commands can be run, on which files, by whom, and from which host. See "Authorizing access" on page 83 to find out how you define protections.

- Communication between servers in a distributed environment can be secured using a trust file and by setting permissions for the service users that own the different servers in the environment. For more information, see *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.

Before you can configure access and authentication, you must create users as described in "Managing users" on page 134.

Securing the server: workflow

The following workflow summarizes the steps required to secure the server and authenticate users. The suggested order might vary, depending on the authentication method used and on whether users are automatically created.

1. Set up SSL if needed.
2. Set up a firewall if needed.
3. Set up protections for users and user groups.
4. Review available authentication options and server security levels.
5. Set the security level for the server.
6. Define the authentication to be used for existing users and new users.
7. Create authentication triggers if you are planning to use a non-standard LDAP server.
8. Enable and configure LDAP authentication if you are planning to authenticate users against an LDAP or Active Directory server.

For information about basic security considerations when setting up a Perforce server, see

http://kbportal.perforce.com/article/2484

Using SSL to encrypt connections to a Perforce server

The following sections explain how you set up encrypted communications between a client and a Perforce server.

For any given Perforce server, proxy, or broker, SSL encryption is an all-or-nothing option: If a Perforce server is configured to use SSL (presumably for security reasons), all Perforce applications must be configured to use SSL. Conversely, if a Perforce server is configured to accept plaintext connections (either for performance reasons or for backwards compatibility), all client applications must connect in plaintext. It is possible however, if you have an intermediary (such as a proxy or a broker) between the client and the Perforce server, that one leg of the communication is encrypted and the following is not. For more information, see "Using SSL in a mixed environment" on page 69.

Note	TLSv1.1 is currently supported; SSL 3.0 is not.

Server and client setup

By default, a `P4PORT` setting that does not specify a protocol is assumed to be in plaintext. It is good practice to configure Perforce applications to explicitly specify the protocol, either `tcp:`*`host`*`:`*`port`* for plaintext, or `ssl:`*`host`*`:`*`port`* for encrypted connections.

The first time a user connects to an SSL-enabled server, their Perforce applications will inform them of the fingerprint of the server's key.

If the user can independently verify that the fingerprint is accurate, they should add the server to their `P4TRUST` file (either by using the **`p4 trust`** command, by following the prompts in P4V or other Perforce applications, or by manually adding the fingerprint to the file).

Key and certificate management

When configured to accept SSL connections, all server processes (**`p4d`**, **`p4p`**, **`p4broker`**), require a valid certificate and key pair on startup. These files are stored in the directory specified by the `P4SSLDIR` environment variable. In order for an SSL-enabled server process to start, the following additional conditions must be met:

- `P4SSLDIR` must be set to a valid directory.
- The `P4SSLDIR` directory must be owned by the same userid as the one running the Perforce server, proxy, or broker process. The `P4SSLDIR` directory must not be readable by any other user. On UNIX, for example, the directory's permissions must be set to 0700 (`drwx------`) or 0500 (`dr-x------`).
- Two files, named `privatekey.txt` and `certificate.txt`, must exist in `P4SSLDIR`.

  These files correspond to the PEM-encoded private key and certificate used for the SSL connection. They must be owned by the userid that runs the Perforce server, proxy, and broker process, and must also have their permissions set such as to make them unreadable by other users. On UNIX, for example, the files' permissions must be set to 0600 (`-rw-------`) or 0400 (`-r--------`).

You can supply your own private key and certificate, or you can use **p4d -Gc** to generate a self-signed key and certificate pair.

- To generate a fingerprint from your server's private key and certificate, run **p4d -Gf** . (P4SSLDIR must be configured with the correct file names and permissions, and the current date must be valid for the certificate.)

  After you have communicated this fingerprint to your end users, your end users can then compare the fingerprint the server offers with the fingerprint you have provided. If the two fingerprints match, users can use **p4 trust** to add the fingerprint to their P4TRUST files.

Key and certificate generation

To generate a certificate and private key for your server:

1. Set P4SSLDIR to a valid directory in a secure location. The directory specified by P4SSLDIR must be secure: owned by the same userid as the one generating the key pair, and it must not be readable by any other user.
2. Optionally, create a file named config.txt in your P4SSLDIR directory before running **p4d -Gc**, and format the file as follows:

```
# C: Country Name - 2 letter code (default: US)
C =

# ST: State or Province Name - full name (default: CA)
ST =

# L: Locality or City Name (default: Alameda)
L =

# O: Organization or Company Name (default: Perforce Autogen Cert)
O =

# OU = Organization Unit - division or unit
OU =

# CN: Common Name (usually the DNS name of the server)
# (default: the current server's DNS name)
CN =

# EX: number of days from today for certificate expiration
# (default: 730, e.g. 2 years)
EX =

# UNITS: unit multiplier for expiration (defaults to "days")
# Valid values: "secs", "mins", "hours"
UNITS =
```

3. Generate the certificate and key pair with the following command:

   p4d -Gc

If `P4SSLDIR` (and optionally, `config.txt`) has been correctly configured, and if no existing private key or certificate is found, two files, named `privatekey.txt` and `certificate.txt`, are created in `P4SSLDIR`.

If a `config.txt` file is not present, the following default values are assumed, and a certificate is created that expires in 730 days (two years, excluding leap years).

```
C=US
ST=CA
L=Alameda
O=Perforce Autogen Cert
OU=
CN=the-DNS-name-of-your-server
EX=730
UNITS=days
```

4. Generate a fingerprint for your server's key and certificate pair.

   `p4d -Gf`

   This command displays the fingerprint of the server's public key, and then exits.

   ```
   Fingerprint: CA:BE:5B:77:14:1B:2E:97:FO:5F:31:6E:33:6F:OE:1A:E9:DA:EF:E2
   ```

   Record your server's fingerprint for your own records and communicate it to your users via an out-of-band communications channel.

   If a Perforce application reports a different fingerprint (and you have not recently installed a new certificate and key pair), your users should consider such changes as evidence of a potential man-in-the-middle threat.

Note

Because Perforce Servers can use self-signed certificates, you may also use third-party tools such as OpenSSL or PuTTY to generate the key pairs, or supply your own key pair. The **`p4d -Gf`** command accepts user-supplied credentials.

If you are supplying your own key, your `privatekey.txt` and `certificate.txt` files in `P4SSLDIR` must be PEM-encoded, with the private key file stripped of passphrase protection.

Whether you supply your own key and certificate pair or generate one with **`p4d -Gc`**, it is *imperative* that these files are stored in a secure location that is readable only by the **`p4d`** binary.

Secondary cipher suite

By default, Perforce's SSL support is based on the AES256-SHA cipher suite. To use CAMELLIA256-SHA, set the `ssl.secondary.suite` tunable to `1`.

Using SSL in a mixed environment

In a mixed environment, each link between Perforce servers, proxies, or brokers may be configured to be in either plaintext or SSL, independent of the encryption choice for any other link. Consider the following examples:

- During a migration from cleartext to SSL, a Perforce Broker may be configured to accept plaintext connections from older Perforce applications, and to forward those requests (encrypted by SSL) to a Perforce Server that requires SSL connections.
- A Perforce Broker could be configured to `listen` on `tcp:old-server:1666`, and redirect all requests to a `target` of `ssl:new-server:1667`. Users of new Perforce applications could use SSL to connect directly to the upgraded Perforce Server (by setting `P4PORT` to `ssl:new-server:1667`), while users of older Perforce applications could continue to use plaintext when connecting to a Perforce Broker (by setting `P4PORT` to `old-server:1666`). After migration is complete, the broker at `old-server:1666` could be deactivated (or reconfigured to require SSL connections), and any remaining legacy processes or scripts still attempting to connect via plaintext could be upgraded manually.

The Perforce Proxy and the Perforce Broker support the `-Gc` and `-Gf` flags, and use the `P4SSLDIR` environment variable. You generate certificate and key pairs for these processes (and confirm fingerprints) as you would with a single Perforce Server. In order for two servers to communicate over SSL, the administrator of the downstream server (typically a replica server, Proxy, or Broker process) must also use the **`p4 trust`** command to generate a `P4TRUST` file for the service user associated with the downstream server.

When migrating from a non-SSL environment to an SSL-based environment, it is your responsibility to securely communicate the new server's fingerprint to your users.

Using firewalls

If available, remote clients can use a Virtual Private Network (VPN) or a Secure Shell (SSH) tunnel to access services on the inside trusted network.

For additional information about using an SSH tunnel to connect to a Perforce server, see the following Knowledge Base article:

http://answers.perforce.com/articles/KB/2433

Authentication options

This section introduces the options you have in authenticating users who log in to Perforce. It focuses on authenticating against Active Directory and LDAP servers without using authentication triggers.

Overview

User authentication can take place using one of three options:

- Against an Active Directory or LDAP server that is accessed according to an LDAP specification. Enabling this option disables trigger-based authentication.

This section focuses on this option. It notes the advantages of using this option, it explains how you create an LDAP configuration, it gives instructions on how you activate and test this configuration, and it provides reference information on the commands and configurables you use to implement this option.

- Against Perforce's internal user database, **`db.user`**.

  This option allows plain-text password-based authentication. It is described in "Authenticating using passwords and tickets" on page 72.

- Against an authentication server, using an authentication trigger.

  These types of triggers are useful if you need to authenticate users against a non-standard authentication server. Authentication triggers fire when the **`p4 login`** or **`p4 passwd`** commands execute. This option is described in the section "Triggering to use external authentication" on page 215.

The authentication server you choose is used for user definitions, user authentication (passwords), group definitions, license details, and ticket generation.

Authentication is configured on a per-user basis (except for trigger-based authentication): for each user, you can specify what method should be used for authentication. Some options are mutually exclusive: enabling configuration-based LDAP authentication turns off trigger-based authentication. However, you can have some users authenticate using LDAP, while others authenticate against Perforce's internal user database. For more information, see "Defining authentication for users" on page 71.

When logging in using either authentication method, Perforce encrypts the password before passing it to the specified authentication agent.

Server security levels

The authentication option you choose is partly determined by the security level set for the server. Perforce superusers can configure server-wide password usage requirements, password strength enforcement, and supported methods of user/server authentication by setting the **`security`** configurable. To set or change the **`security`** configurable, issue the command:

```
$ p4 configure set security=seclevel
```

where ***`seclevel`*** is **`0`**, **`1`**, **`2`**, **`3`**, or **`4`**.

Security level settings do not apply if you are using an external authentication manager such as LDAP or Active Directory. In this case, the server either behaves as though **`security=3`** (or greater) was set, or is placed completely under the control of the external authentication system.

The following table explains the effect of each security level:

Security level	Server behavior
`0` (or unset)	The default security level **`0`** does not require passwords and does not enforce password strength.

Security level	Server behavior
	Users with passwords can use either their `P4PASSWD` setting or the **`p4 login`** command for ticket-based authentication.
`1`	Ensures that all users have passwords. (Users of old Perforce applications can still enter weak passwords.) Users with passwords can use either their `P4PASSWD` setting or the **`p4 login`** command for ticket-based authentication.
`2`	Ensures that all users have strong passwords. Very old Perforce applications continue to work, but users must change their password to a strong password and upgrade to 2003.2 or later.
`3`	Requires that all users have strong passwords, and requires the use of ticket-based (**`p4 login`**) authentication. If you have scripts that rely on passwords, use **`p4 login`** to create a ticket valid for the user running the script, or use **`p4 login -p`** to display the value of a ticket that can be passed to Perforce commands as though it were a password (that is, either from the command line, or by setting `P4PASSWD` to the value of the valid ticket). Setting passwords with the **`p4 user`** form or the **`p4 passwd -O`** ***`oldpass`*** **`-P`** ***`newpass`*** command is prohibited.
`4`	In multi-server and replicated environments this level ensures that only authenticated service users (subject to all of the restrictions of level 3) can connect to this server. The following checks are also made: • The request must come from a replica with a valid serverid. • The serverid must identify a valid server spec. • If the server spec has a user field, the request must come from that service user. • If the server spec has filters, these are used in preference to whatever filters might have been specified by the replica.

Note Use the `dm.password.minlength` configurable to enforce a minimum password length at levels `1` - `3`.

Defining authentication for users

Authentication is defined by the setting of the `AuthMethod` field of the user spec and also by configurables that affect user authentication.

The `AuthMethod` field of the user specification, created with the **`p4 user`** command, specifies the authentication method to be used for that user.

- `ldap` indicates that the user is to be authenticated against the LDAP directory defined by an active LDAP configuration. User access can be further restricted to those users who belong to a particular LDAP group.

  All authentication triggers are disabled when LDAP authentication is enabled.

- `perforce` indicates that the user is to be authenticated by an authentication trigger script if such a script exists, or against Perforce's internal user database. This is the default setting.

A superuser must edit the user spec with the **`p4 user -f`** command to change the default value to `ldap` if desired.

The `auth.default.method` configurable defines the default value for the `AuthMethod` on *new* users. Possible values are `perforce` or `ldap`.

By default, Perforce creates a new user record in its database whenever a previously unknown user invokes any command that can update the repository or its metadata. For greatest security, it is recommended that you turn this feature off using the `dm.user.noautocreate` configurable with the **`p4 configure`** command.

If you select the `ldap` configurable, only superusers are allowed to create new users (using the **`p4 user`** command). To have new users automatically created upon login, you must set `auth.ldap.userautocreate` to 1.

If you need more control over which LDAP users are allowed access to Perforce, you can use the group-related fields of the LDAP configuration to implement a basic authorization step that filters out non-Perforce users. For example, specifying a filter like the following limits access to LDAP users who belong to the LDAP group with the common name `perforce`.

```
Base DN: ou=groups,dc=example,dc=org
LDAP query: (&(cn=perforce)(memberUid=%user%))
```

In this case, only users who provide the proper credentials and who are members of the specified group are authenticated. For more information about the `auth.default.method` configurable, see the description of the **`p4 configure`** command and the "Configurables" appendix in the *P4 Command Reference*.

Note	If a user is set to use LDAP-configuration based authentication, the user may not update their password with the **`p4 passwd`** command.

Authenticating using passwords and tickets

Perforce supports two methods of authentication: password-based and ticket-based. Although it might be more accurate to say that you can use password-only authentication or authentication that uses passwords *and* associated tickets.

- Password-only authentication is based on plain-text passwords that do not expire and that are passed around when the user executes a command.

- Ticket-based authentication is based on tickets that are issued for a given amount of time and are generated after the user has logged in with a valid password. After log in, the ticket is used to authenticate the user (rather than the password being passed around).

Warning

Although ticket-based authentication is more secure than password-based authentication, it does not encrypt network traffic between client workstations and the Perforce server.

To encrypt network traffic between client workstations and the Perforce server, configure your installation to use SSL. See "Using SSL to encrypt connections to a Perforce server" on page 66.

Password-based authentication

Plain-text password-based authentication is stateless; after a password is correctly set, access is granted for indefinite time periods. Passwords may be up to 1024 characters in length. To enforce password strength and existence requirements, set the server security level. See "Server security levels" on page 70 for details. Plain-text password based authentication is supported only at security levels `0`, `1`, and `2`.

The default minimum password length is eight characters. Minimum password length is configurable by setting the `dm.password.minlength` configurable. For example, to require passwords to be at least 16 characters in length, a superuser can run:

```
$ p4 configure set dm.password.minlength=16
```

To require users to change their passwords after a specified interval, assign your users to at least one group and set the `PasswordTimeout:` value for that group. For users in multiple groups, the largest defined `PasswordTimeout` (including `unlimited`, but ignoring `unset`) value applies.

The **`p4 admin resetpassword`** command forces specified users with existing passwords to change their passwords before they can run another command. (This command works only for users whose `authMethod` is set to `perforce`. However, you can use it in a mixed environment, that is an environment in which both Perforce-based and LDAP-based authentication are enabled.)

Password strength requirements

Certain combinations of server security level and Perforce applications require users to set "strong" passwords. A password is considered strong if it is at least `dm.password.minlength` characters long (by default, eight characters), and at least two of the following are true:

- The password contains uppercase letters.
- The password contains lowercase letters.
- The password contains non-alphabetic characters.

For example, the passwords `a1b2c3d4`, `A1B2C3D4`, `aBcDeFgH` are considered strong in an environment in which `dm.password.minlength` is 8, and `security` is configurable to at least `2`.

You can configure a minimum password length requirement on a site-wide basis by setting the `dm.password.minlength` configurable. For example, to require passwords to be at least 16 characters in length, a superuser can run:

```
$ p4 configure set dm.password.minlength=16
```

Passwords may be up to 1,024 characters in length. The default minimum password length is eight characters.

Managing and resetting user passwords

Perforce superusers can manually set a Perforce user's password with:

```
$ p4 passwd username
```

When prompted, enter a new password for the user.

To force a user with an existing password to reset his or her own password the next time they use Perforce, use the following command:

```
$ p4 admin resetpassword -u username
```

You can force all users with passwords (including the superuser that invokes this command) to reset their passwords by using the command:

```
$ p4 admin resetpassword -a
```

Running **`p4 admin resetpassword -a`** resets only the passwords of users who already exist (and who have passwords). If you create new user accounts with default passwords, you can further configure your installation to require that all newly-created users reset their passwords before issuing their first command. To do this, set the `dm.user.resetpassword` configurable as follows:

```
$ p4 configure set dm.user.resetpassword=1
```

Ticket-based authentication

Ticket-based authentication is based on time-limited tickets that enable users to connect to Perforce servers. Perforce creates a ticket for a user when they log in to Perforce using the **`p4 login -a`** command. Perforce applications store tickets in the file specified by the `P4TICKETS` environment variable. If this variable is not set, tickets are stored in `%USERPROFILE%\p4tickets.txt` on Windows, and in `$HOME/.p4tickets` on UNIX and other operating systems.

By default, tickets have a finite lifespan, after which they cease to be valid. By default, tickets are valid for 12 hours (43200 seconds). To set different ticket lifespans for groups of users, edit the `Timeout:` field in the **`p4 group`** form for each group. The timeout value for a user in multiple groups is the largest

timeout value (including `unlimited`, but ignoring `unset`) for all groups of which a user is a member. To create a ticket that does not expire, set the `Timeout:` field to `unlimited`.

Although tickets are not passwords, Perforce servers accept valid tickets wherever users can specify Perforce passwords (except when logging in with the **`p4 login`** command). This behavior provides the security advantages of ticket-based authentication with the ease of scripting afforded by password authentication. Ticket-based authentication is supported at all server security levels, and is required at security level `3` and `4`.

If password aging is in effect, tickets expire when their passwords expire.

Login process for the user

Users are authenticated in one of two ways:

- The user logs in explicitly using the **`p4 login`** command.

  The user enters a p4 command, and the command requires that the user be authenticated. If the user is not already authenticated, the command will prompt for login. If the login is successful, the original command continues.

To log in to Perforce, the user obtains a ticket from the server by using the **`p4 login`** command:

```
$ p4 login
```

The user is prompted for a password, and a ticket is created for the user in the file specified by `P4TICKETS`. The user can extend the ticket's lifespan by calling **`p4 login`** while already logged in; this extends the ticket's lifespan by 1/3 of its initial timeout setting, subject to a maximum of the user's initial timeout setting.

The Perforce service rate-limits the user's ability to run **`p4 login`** after multiple failed login attempts. To alter this behavior, set `dm.user.loginattempts` to the maximum allowable failed login attempts before the service imposes a 10-second delay on subsequent login attempts.

By default, Perforce tickets are valid for the user's IP address only. If the user has a shared home directory that is used on more than one machine, the user can log in to Perforce from both machines by using **`p4 login -a`** to create a ticket in the home directory that is valid from all IP addresses.

Tickets can be used by multiple clients on the same machine so long as they use the same user and port.

Note

The `auth.csv` log is used to log the results of **`p4 login`** attempts. If the login failed, the reason for this is included in the log. Additional information provided by the authentication method is included in the log entries.

Login process for the server

The server uses the following process to login a user:

1. The user logs in, specifying a name and password.

2. The server checks to see if LDAP integration has been enabled for the server.
   - If LDAP integration has been enabled, the server checks the user record as described in Step 3.
   - If LDAP integration has not been enabled, the server passes the user's credentials to an authentication script if one exists, or it validates credentials using the `db.user` table; it then issues a ticket if validation succeeds.
3. The server checks the user record to see which authentication method to use: `ldap` or `perforce`.
   - If `ldap`, the server cycles through available LDAP configurations to find the user. If the user is found and the password is valid, a ticket is issued for the user.
   - If `perforce`, the server validates the user against the `db.user` table and issues a ticket if the user exists and credentials are valid.

Logging out of Perforce

To log out of Perforce from one machine by removing your ticket, use the command:

```
$ p4 logout
```

The entry in your ticket file is removed. If you have valid tickets for the same Perforce server, but those tickets exist on other machines, those tickets remain present (and you remain logged in) on those other machines.

If you are logged in to Perforce from more than one machine, you can log out of Perforce from all machines from which you were logged in by using the command:

```
$ p4 logout -a
```

All of your Perforce tickets are invalidated and you are logged out.

Determining ticket status

To see if your current ticket (that is, for your IP address, user name, and `P4PORT` setting) is still valid, use the command:

```
$ p4 login -s
```

If your ticket is valid, the length of time for which it will remain valid is displayed.

To display all tickets you currently have, use the command:

```
$ p4 tickets
```

The contents of your ticket file are displayed.

Invalidating a user's ticket

As a super user, you can use the `-a` flag of the **`p4 logout`** command to invalidate a user's ticket. The following command invalidates Joe's ticket.

```
$ p4 logout -a joe
```

LDAP Authentication

The following sections explain how you can authenticate against Active Directory and LDAP servers. It describes how you do the following:

- Create an LDAP configuration
- Set LDAP-related configurables
- Authorize access using LDAP groups
- Test and enable LDAP configurations
- Get information about LDAP servers
- Use LDAP with SSO triggers

Authenticating against Active Directory and LDAP servers

LDAP, Lightweight Directory Access Protocol, is supported by many directory services; chief among these is Active Directory and OpenLDAP. Perforce offers two ways of authenticating against Active Directory or LDAP servers: using an authentication trigger or using an LDAP specification. The latter method offers a number of advantages: it is easier to use, no external scripts are required, it allows users who are not in the LDAP directory to be authenticated against Perforce's internal user database, and it is more secure.

Note

Create at least one account with `super` access that uses perforce authentication. This will allow you to login if by some chance you lose AD/LDAP connectivity.

SASL authentication is supported; SAML is not.

The steps required to set up configuration-based LDAP authentication are described in the following sections. Throughout this section, information relating to LDAP authentication applies equally to using Active Directory. In broad strokes, the configuration process include the following steps:

- Use the **`p4 ldap`** command to create an LDAP configuration specification for each LDAP or Active Directory server that you want to use for authentication.
- Define authentication-related configurables to enable authentication, to specify the order in which multiple LDAP servers are to be searched, and to provide additional information about how LDAP authentication is to be implemented.
- Set the `AuthMethod` field of the user specification for existing users to specify how they are to be authenticated.

- Test the LDAP configurations you have defined to make sure searches are conducted as you expect.
- If this is the first time you have enabled LDAP authentication, restart the server.

Note

You must restart the Perforce server whenever you enable or disable LDAP authentication:

- You enable LDAP authentication the first time you enable an LDAP configuration by setting the `auth.ldap.order.N` configurable.
- You disable LDAP authentication by removing or disabling all existing LDAP configurations. You remove an LDAP configuration by using the `-d` option to the **`p4 ldap`** command. You disable all LDAP configurations by having no `auth.ldap.order.N` configurables set.
- LDAP implies at least security level `3`.

Creating an LDAP configuration

An *LDAP configuration* specifies an Active Directory or other LDAP server against which the Perforce server can authenticate users. You use the **`p4 ldap`** command to create configurations.

To define an LDAP configuration specification, you provide values that specify the host and port of the Active Directory or LDAP service, bind method information, and security parameters. Here is a sample LDAP configuration using the search bind method:

```
Name:              sleepy
Host:              openLdap.example.com
Port:              389
Options:           getattrs
Encryption:        tls
BindMethod:        search
SearchBaseDN:      ou=employees,dc=example,dc=com
SearchFilter:      (cn=%user%)
SearchScope:       subtree
GroupSearchScope:  subtree
```

You can choose among the following bind methods: SASL, simple, and search.

- **SASL**: One complication of the non-SASL bind methods is that the administrator needs to know about the structure of the directory. Most LDAP and Active Directory servers have the option of binding using SASL, which only requires a username and password to authenticate a user.

  If the LDAP server supports SASL DIGEST-MD5 (Active Directory does), this method defers the user search to the LDAP server and does not require a distinguished name to be discovered before the bind is attempted. This method is recommended for Active Directory. Look how simple this is:

  ```
  BindMethod: sasl
  ```

If your LDAP server has multiple realms (or domains in Active Directory), you might need to specify which one the LDAP configuration should be using. In this case, you'll need to set the `SaslRealm` field too; for example:

```
BindMethod:  sasl
SaslRealm:   example
```

Active Directory supports SASL out of the box, and most LDAP servers support SASL.

- **Simple**: This method is suitable for simple directory layouts. It uses a pattern and the user's username to produce a distinguished name that the Perforce server attempts to bind against, validating the user's password. The name given is set on the Simple Pattern field. For example:

  ```
  BindMethod: simple
  SimplePattern: uid=%user%,ou=users,dc=example,dc=com
  ```

  This pattern is expanded when a user is logging in. For example, if the user is `jsmith`, the Perforce server would attempt to bind against the DN shown below, using the password the user provided.

  ```
  uid=jsmith,ou=users,dc=example,dc=com
  ```

  This bind method only works in environments where the user's username is part of their DN and all of the users you want to authenticate are in the same organizational unit (OU).

- **Search**: This method performs a search for the user's record in the directory, overcoming the restrictions of the simple bind method Instead of a DN pattern, an LDAP search query is provided to identify the user's record. The `%user%` placeholder is also used with this method. A starting point and scope for the search are provided, allowing control over how much of the directory is searched. The search relies on a known base DN and an LDAP search query; you provide these using the `SearchBaseDN`, `SearchFilter`, and `SearchScope` fields of the LDAP configuration specification. This method might also require the full distinguished name and password of a known read-only entity in the directory. You supply these using the `SearchBindDN` and `SearchPasswd` fields of the LDAP configuration. Here are two sample search queries:

  ```
  BindMethod:    search
  SearchBaseDN:  ou=users,dc=example,dc=com
  SearchFilter:  (&(objectClass=inetOrgPerson) (uid=%user%))
  SearchScope:   subtree
  SearchBindDN:  uid=read-only, dc=example, dc=com
  SearchPasswd:  ******
  ```

```
BindMethod:    search
SearchBaseDN:  ou=users,dc=example,dc=com
SearchFilter:  (&(objectClass=user) (sAMAccountName=%user%))
SearchScope:   subtree
SearchBindDN:  uid=read-only, dc=example, dc=com
SearchPasswd:  ******
```

See the *P4 Command Reference* for information about the **p4 ldap** command and the LDAP specification. The LDAP spec also allows you to fine tune the behavior of LDAP integration. In particular, three options allows you to control the following behavior:

- Set the **downcase** option to specify that user names should be downcased from the directory on an LDAP sync.
- Set the **getattrs** option to specify that the Fullname and Email fields should be populated for autocreated users; the information is taken from the LDAP server.
- Set the **realminusername** option to specify that the realm should be taken for the SASL user name if it is in UNC or UPN format
- Test your LDAP configuration using a command like the following:

```
$ p4 ldap -t myuser myldapconfig
```

After you create the configuration, you must enable it using the **auth.ldap.order.N** configurable. For example:

```
$ p4 configure set auth.ldap.order.1=sleepy
```

(You must restart the server to enable LDAP.)

The configuration is now active and can be used for authentication. You might also have to define additional configurables to define the authentication process. These are described in "Defining LDAP-related configurables" on page 80.

You might need to create multiple LDAP configurations if you are using multiple directory servers for failover or user management. In this case, you will need to create an LDAP configuration for each LDAP server; you must also use the **auth.ldap.order.*N*** configurable to specify the order in which they should be searched. Configurables are keyed on their name, therefore you may not have two LDAP configurations using the same order number for the same Perforce server.

Defining LDAP-related configurables

To use LDAP authentication, you must set a number of authentication-related configurables:

- **auth.ldap.order.*N*** - enables an LDAP server and specifies the order in which it should be searched.

- `auth.default.method` - specifies whether new users should be authenticated by Perforce or using LDAP. `dm.user.noautocreate` is implied at 2 for `auth.default.method=ldap`
- `auth.ldap.userautocreate` - specifies whether new users should be automatically created on login when using LDAP authentication. This requires `auth.default.method=ldap`.

  You can set the `getattrs` Options field of the LDAP configuration to have the `FullName` and `Email` fields populated from the directory.
- `dm.user.noautocreate` - further controls the behavior of user autocreation.
- `auth.ldap.timeout` - time to wait before giving up on a connection attempt.
- `auth.ldap.cafile` - the path to a file used for certification when the LDAP server uses SSL or TLS.
- `auth.ldap.ssllevel` - level of SSL certificate validation.
- `auth.ldap.ssllevel` - helps you manage LDAP searches with paged results by setting limits to page size.

For example, the following commands define the define the search order for active directories and the default authentication method for new users to be `perforce`:

```
$ p4 configure set auth.ldap.order.1=sleepy
$ p4 configure set auth.ldap.order.2=dopey
$ p4 configure set auth.ldap.order.5=sneezy
$ p4 configure set auth.default.method=perforce
```

For additional information about authentication-related configurables, see the "Configurables" appendix in the *P4 Command Reference*.

Authorization using LDAP groups

You use bind methods to configure user authentication, but you don't want to give everyone in your organization the ability to log in to your Perforce server, especially if everyone is in the same directory. Rather, you should create a group object in the directory that contains only authorized users. The new LDAP integration provides support for checking group membership.

LDAP groups work just like the search bind method, where an LDAP search query determines whether a user is a member of an allowed group and whether a search base and scope are also provided. For example, if there is a group in the LDAP directory named `perforce`, whose users are allowed to access Perforce servers, you might have a configuration like this:

```
GroupBaseDN:    ou=groups, dc=example, dc=com
GroupSearchFilter:   (&(objectClass=posixGroup) (cn=perforce) (memberUid=%user%))
GroupSearchScope:    subtree
```

Group objects in Active Directory are slightly different from those in OpenLDAP: rather than containing a list of member's user names, they contain a list of the member's full DNs. These are not typically easy to match; however, back references are added to the member's User objects, which can be matched. Therefore, when using group authorization against Active Directory, you will probably

need to search for the user's User object and check that it contains a `memberOf` reference to the group. For example:

```
GroupBaseDN:    ou=users, dc=example, dc=com
SearchFilter:   (&(objectClass=user) (sAMAccountName=%user%)
                    (memberOf=cn=perforce,ou=groups,dc=example,dc=com))
SearchScope:    subtree
```

Testing and enabling LDAP configurations

Before you enable LDAP configurations, you should create at least one account with `super` access that uses `perforce` authentication. This will allow you to login if by some chance you lose AD/LDAP connectivity.

Having created an LDAP configuration, you must test and enable the configuration. The ability to test your LDAP configurations allows you to make sure everything is working properly without impacting existing users, even if they are already using an authentication trigger to authenticate against LDAP. Once the LDAP configuration proves successful, you can switch users to the new mechanism without having to recreate them. The following steps illustrate the process of testing and activating a configuration.

1. Test the configuration using the `-t` flag on the **`p4 ldap`** command; for example:

   ```
   $ p4 ldap -t Cleopatra sleepy
   ```

   You will be prompted for the user's password. If the password is correct, the command completes successfully.

   The amount of information returned by testing depends on the bind method used:

   - A simple bind returns only pass/fail feedback.
   - A search-based bind returns information about whether the user's credentials are bad and whether the user could be found.
   - SASL binds usually provide more diagnostics than simple binds, but results can vary.

2. Define the `auth.ldap.order.`*`N`* to tell Perforce to in what order to use this configuration; for example:

   ```
   $ p4 configure set auth.ldap.order.1=sleepy
   ```

   You must set this configurable even if you are only using one configuration.

3. Check active configurations by running the following command:

   ```
   $ p4 ldaps -A
   ```

4. Restart the server:

```
$ p4 admin restart
```

5. Check that the server is running in LDAP authentication mode by running the following command:

```
$ p4 -ztag info
```

   Then check to see that `ldapAuth` is enabled.

6. Create additional LDAP servers if needed, and repeat steps 1, 2, 3 for each. Of course, if you add more configurations, you will need to assign a different priority to each.

7. Migrate users to LDAP authentication by setting the `authMethod` to `ldap` for each user to be authenticated by LDAP.

In addition to testing authentication against a single LDAP server, you can test against multiple servers using the **`p4 ldaps -t`** command. For more information, see the description of the see the description of the **`p4 ldaps -t`** command in the *P4 Command Reference*.

Getting information about LDAP servers

You can use two commands to get information about LDAP servers:

- The **`p4 ldap -o`** command displays information about a single server.
- The **`p4 ldaps`** command lists all defined servers or, using the **`-A`** option, lists only enabled servers in order of priority.

For more information, see the description of the two commands in *P4 Command Reference*.

Using LDAP with single sign-on triggers

You have the option of using **`auth-check-sso`** type triggers when LDAP authentication is enabled. In this case, users authenticated by LDAP can define a client-side SSO script instead of being prompted for a password. If the trigger succeeds, the active LDAP configurations are used to confirm that the user exists in at least one LDAP server. The user must also pass the group authorization check if it is configured. Triggers of type **`auth-check-sso`** will not be called for users who do not authenticate against LDAP.

For information about SSO triggers, see "Triggering to use external authentication" on page 215. For information about group authorization, see the next section.

Authorizing access

Perforce provides a protection scheme to prevent unauthorized or inadvertent access to files in the depot. The protections determine which Perforce commands can be run, on which files, by whom, and from which host. You configure protections with the **`p4 protect`** command.

Note

Protections apply to files in the depot only. They do not apply to forms: changelists, workspace views, and so on.

When should protections be set?

Run **`p4 protect`** immediately after installing Perforce for the first time. Before the first call to **`p4 protect`**, every Perforce user is a superuser and thus can access and change anything in the depot. The first time a user runs **`p4 protect`**, a protections table is created that gives superuser access to the user from all IP addresses, and lowers all other users' access level to **`write`** permission on all files from all IP addresses.

The Perforce protections table is stored in the **`db.protect`** file in the server root directory; if **`p4 protect`** is first run by an unauthorized user, the depot can be brought back to its unprotected state by removing this file.

Setting protections with p4 protect

The **`p4 protect`** form contains a single form field called **`Protections:`** that consists of multiple lines. Each line in **`Protections:`** contains subfields, and the table looks like this:

Example 5.1. A sample protections table

```
Protections:
    read    user    emily   *                   //depot/elm_proj/...
    write   group   devgrp  *                   //...
    write   user    *       192.168.41.0/24     -//...
    write   user    *       [2001:db8:1:2::]/64 -//...
    write   user    joe     *                   -//...
    write   user    lisag   *                   -//depot/...
    write   user    lisag   *                   //depot/doc/...
    super   user    edk     *                   //...
```

(The five fields might not line up vertically on your screen; they are aligned here for readability.)

Note

If your site makes use of the Perforce Proxy or broker, prepend **`proxy-`** to the addresses in the host field to make the lines apply to users of the proxy or broker. For detailed information, see the material on "P4P and protections" in *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.

The permission lines' five fields

Each line specifies a particular permission level and/or access right, as defined by five fields:

Field	Meaning
`Access Level`	Which access level (**`list`**, **`read`**, **`open`**, **`write`**, **`review`**, **`owner`**, **`admin`**, or **`super`**) or specific right (**`=read`**, **`=open`**, **`=write`**, or **`=branch`**) is being granted or denied. • Each permission level includes all the permissions above it (except for **`review`**).

Field	Meaning
	• Each permission right (denoted by an `=`) only includes the specific right and not all of the lesser rights. In general, one typically grants an access level to a user or group, after which, if finer-grained control is required, one or more specific rights may then be denied.
`User/Group`	Does this protection apply to a **`user`** or a **`group`**?
`Name`	The user or group whose protection level is being defined. This field can contain the * wildcard. A * by itself grants this protection to everyone, **`*e`** grants this protection to every user (or group) whose username ends with an **`e`**.
`Host`	The TCP/IP address of the host being granted access. This must be provided as the numeric address of either one specific host (for instance, **`192.168.41.2`** or [2001:db8:195:1:2::1234]) or a subnet expressed in CIDR notation. The host field can also contain the * wildcard. A * by itself means that this protection is being granted for all hosts. The wildcard can be used as in any string, so **`192.168.41.*`** is equivalent to **`192.168.41.0/24`**. You cannot combine the * wildcard with CIDR notation, except at the start of a line when controlling proxy matching. If you are using IPv6 with the * wildcard, you must enclose the address with square brackets. **`[2001:db8:1:2:*]`** is equivalent to **`[2001:db8:1:2::]/64`**. Best practice is to use CIDR notation, surround IPv6 addresses with brackets, and to avoid the * wildcard. For more about controlling access to a Perforce server via the Perforce Proxy, see the relevant chapter of *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.
`Files`	A file specification representing the files in the depot on which permissions are being granted. Perforce wildcards can be used in the specification. "**`//...`**" means all files in all depots. If a depot is excluded, the user denied access will no longer see the depot in the output of **`p4 depots`**. Nor will the depot show up, for this user, in the default branch, client, and label views.

Access levels

The access level is described by the first value on each line. The permission levels and access rights are described in the following table:

Level	Meaning
`list`	Permission is granted to run Perforce commands that display file metadata, such as **`p4 filelog`**. No permission is granted to view or change the contents of the files.

Level	Meaning
`read`	The user can run those Perforce commands that are needed to read files, such as **`p4 client`** and **`p4 sync`**. The `read` permission includes `list` access.
`=read`	If this right is denied, users cannot use **`p4 print`**, **`p4 diff`**, or **`p4 sync`** on files.
`open`	Grants permission to read files from the depot into the client workspace, and gives permission to open and edit those files. This permission does not permit the user to write the files back to the depot. The `open` level is similar to `write`, except that with `open` permission, users are not permitted to run **`p4 submit`** or **`p4 lock`**. The `open` permission includes `read` and `list` access.
`=open`	If this right is denied, users cannot open files with **`p4 add`**, **`p4 edit`**, **`p4 delete`**, or **`p4 integrate`**.
`write`	Permission is granted to run those commands that edit, delete, or add files. The `write` permission includes `read`, `list`, and `open` access. This permission allows use of all Perforce commands except `protect`, `depot`, `obliterate`, and `verify`.
`=write`	If this right is denied, users cannot submit open files.
`=branch`	If this right is denied, users may not use files as a source for **`p4 integrate`**.
`review`	Provides list and read access, plus use of the **`p4 review`** command. This is a special permission granted to review scripts.
`owner`	Allows access to the **`p4 protect`** command to the specified user or group, for the specified path. See "Delegate management of parts of the protections table" on page 88 for details.
`admin`	For Perforce administrators; grants permission to run Perforce commands that affect metadata, but not server operation. Provides `write` and `review` access plus the added ability to override other users' branch mappings, client specifications, jobs, labels, and change descriptions, as well as to update the typemap table, verify and obliterate files, and customize job specifications.
`super`	For Perforce superusers; grants permission to run all Perforce commands. Provides `write`, `review`, and `admin` access plus the added ability to create depots and triggers, edit protections and user groups, delete users, reset passwords, and shut down the server.

Each Perforce command is associated with a particular minimum access level. For example, to run **`p4 sync`** or **`p4 print`** on a particular file, the user must have been granted at least `read` access on that file. For a full list of the minimum access levels required to run each Perforce command, see "How protections are implemented" on page 95.

The specific rights of `=read`, `=open`, `=write`, and `=branch` can be used to override the automatic inclusion of lower access levels. This makes it possible to deny individual rights without having to then re-grant lesser rights.

For example, if you want administrators to have the ability to run administrative commands, but to deny them the ability to make changes in certain parts of the depot, you could set up a permissions table as follows:

```
admin      user      joe       *            //...
=write     user      joe       *            -//depot/build/...
=open      user      joe       *            -//depot/build/...
```

In this example, user `joe` can perform administrative functions, and this permission applies to all depots in the system. Because the `admin` permission level also implies the granting of all lower access levels, `joe` can also write, open, read and list files anywhere in the system, including `//depot/build/`. To protect the build area, the `=write` and `=open` exclusionary lines are added to the table. User `joe` is prevented from opening any files for edit in the build area. He is also prevented from submitting any changes in this area he may already have open. He can continue to create and modify files, but only if those files are outside of the protected `//depot/build/...` area.

Default protections

Before **`p4 protect`** is invoked, every user has superuser privileges. When **`p4 protect`** is first run, two permissions are set by default. The default protections table looks like this:

```
write        user        *        *        //...
super        user        edk      *        //...
```

This indicates that `write` access is granted to all users, on all hosts, to all files. Additionally, the user who first invoked **`p4 protect`** (in this case, `edk`) is granted superuser privileges.

Which users should receive which permissions?

The simplest method of granting permissions is to give `write` permission to all users who don't need to manage the Perforce system and `super` access to those who do, but there are times when this simple solution isn't sufficient.

`Read` access to particular files should be granted to users who never need to edit those files. For example, an engineer might have `write` permission for source files, but have only `read` access to the documentation, and managers not working with code might be granted `read` access to all files.

Because `open` access enables local editing of files, but does not permit these files to be written to the depot, `open` access is granted only in unusual circumstances. You might choose `open` access over `write` access when users are testing their changes locally but when these changes should not be seen by other users. For instance, bug testers might need to change code in order to test theories as to why particular bugs occur, but these changes are not to be written to the depot. Perhaps a codeline has been frozen, and local changes are to be submitted to the depot only after careful review by the development team. In these cases, `open` access is granted until the code changes have been approved, after which time the protection level is upgraded to `write` and the changes submitted. `open` access is also useful

with shelves. Using `open` is enough to shelve changes but not submit them and can be useful for code reviews.

Interpreting multiple permission lines

The access rights granted to any user are defined by the union of mappings in the protection lines that match her user name and client IP address. (This behavior is slightly different when exclusionary protections are provided and is described in the next section.)

Example 5.2. Multiple permission lines

Lisa, whose Perforce username is `lisag`, is using a workstation with the IP address `195.42.39.17`. The protections file reads as follows:

```
read      user    *       195.42.39.17    //...
write     user    lisag   195.42.39.17    //depot/elm_proj/doc/...
read      user    lisag   *               //...
super     user    edk     *               //...
```

The union of the first three permissions applies to Lisa. Her username is `lisag`, and she's currently using a client workspace on the host specified in lines 1 and 2. Thus, she can write files located in the depot's `elm_proj/doc` subdirectory but can only read other files. Lisa tries the following:

She types **`p4 edit depot/elm_proj/doc/elm-help.1`**, and is successful.

She types **`p4 edit //depot/elm_proj/READ.ME`**, and is told that she doesn't have the proper permission. She is trying to write to a file to which has only `read` access. She types **`p4 sync depot/elm_proj/READ.ME`**, and this command succeeds, because only `read` access is needed, and this is granted to her on line 1.

Lisa later switches to another machine with IP address `195.42.39.13`. She types **`p4 edit //depot/elm_proj/doc/elm-help.1`**, and the command fails; when she's using this host, only the third permission applies to her, and she only has read privileges.

Delegate management of parts of the protections table

It is possible to delegate management of parts of the protections table to non-super users or groups, by creating an entry with the mode `owner`. These entries must have a unique path, without wildcards except for a trailing ellipsis (…).

Users with `super` or that have been granted `owner` for a path may run the **`p4 protect`** command specifying the granted path as an argument, accessing the sub-protections table for that path.

The server appends any entries in this table to the effective protections table directly below the 'owner' entry; if an 'owner' entry is removed, so are any entries in the sub-protections table for that path. Neither 'owner' nor 'super' entries may be added to a sub-protections table, and any other entries' paths must be within the scope of the sub-protections table's path.

If a path argument is specified, and an 'owner' entry with the same path exists, then the sub-protections table for that path will be accessed instead of the main protections table.

Suppose super user Bruno issues the following commands:

```
# Create a user called Sally
$ p4 user -f sally

# Create a depot called stats
$ p4 depot stats

# Edit the protections table
$ p4 protect
```

The last command opens the protections table in an editor. Let's suppose the protections table contains the following lines:

```
Protections:
    write user * * //...
    super user bruno * //...
```

Suppose Bruno wants to delegate control of the sub-protections table for the path `//stats/dev/...` to Sally. He edits the protections table to append the necessary line to the protections table, which now looks like this:

```
Protections:
    write user * * //...
    super user bruno * //...
    owner user sally * //stats/dev/...
```

Exclusionary protections

A user can be denied access to particular files by prefacing the fifth field in a permission line with a minus sign (`-`). This is useful for giving most users access to a particular set of files, while denying access to the same files to only a few users.

To use exclusionary mappings properly, it is necessary to understand some of their peculiarities:

- When an exclusionary protection is included in the protections table, the order of the protections is relevant: the exclusionary protection is used to remove any matching protections above it in the table.
- No matter what access level is provided in an exclusionary protection, all access levels for the matching files and IP addresses are denied. The access levels provided in exclusionary protections are irrelevant. See "How protections are implemented" on page 95 for a more detailed explanation.
- Without exclusionary mappings, the order of items in the protections table is not important.

Example 5.3. Exclusionary protections

An administrator has used **`p4 protect`** to set up protections as follows:

```
write      user      *          *      //...
read       user      emily      *      //depot/elm_proj/...
super      user      joe        *      -//...
list       user      lisag      *      -//...
write      user      lisag      *      //depot/elm_proj/doc/...
```

The first permission looks like it grants write access to all users to all files in all depots, but this is overruled by later exclusionary protections for certain users.

The third permission denies Joe permission to access any file from any host. No subsequent lines grant Joe any further permissions; thus, Joe has been effectively denied any file access.

The fourth permission denies Lisa all access to all files on all hosts, but the fifth permission gives her back **write** access on all files within a specific directory. If the fourth and fifth lines were switched, Lisa would be unable to run any Perforce command.

Displaying protections for a user, group, or path.

Use the **p4 protects** command to display the lines from the protections table that apply to a user, group, or set of files.

With no options, **p4 protects** displays the lines in the protections table that apply to the current user. If a ***file*** argument is provided, only those lines in the protection table that apply to the named files are displayed. Using the **-m** flag displays a one-word summary of the maximum applicable access level, ignoring exclusionary mappings.

Perforce superusers can use **p4 protects -a** to see all lines for all users, or **p4 protects -u user**, -g ***group***, or -h ***host*** flags to see lines for a specific user, group, or host IP address.

Use the **-s** option to display protection information from a protect table referenced by the file revision specified with the ***spec*** argument. For example, the following command returns information about the user sam in the third revision of the protections table:

```
C:\> p4 -u super protects -s //spec/protect.p4s#3 -u sam
write user* * //...
```

This is useful when users lose access privileges at a given point in time and you want to check what changes were made to the protection table just before that date.

Note	To use this option, you must define a spec depot for protect forms; this automatically saves revisions to the protect specification every time you edit the protection table. See the description of the **p4 depot** command in the *P4 Command Reference* for information on how to create a spec depot.

Granting access to groups of users

Perforce *groups* simplify maintenance of the protections table. The names of users with identical access requirements can be stored in a single group; the group name can then be entered in the table, and all the users in that group receive the specified permissions.

Groups are maintained with **p4 group**, and their protections are assigned with **p4 protect**. Only Perforce superusers can use these commands. (Perforce administrators can use **p4 group -A** to administer a group, but only if the group does not already exist.)

For information about groups and LDAP, see "Synchronizing Perforce groups with LDAP groups" on page 92.

Creating and editing groups

If **p4 group groupname** is called with a nonexistent *groupname*, a new group named *groupname* is created. Calling **p4 group** with an existing *groupname* allows editing of the user list for this group.

To add users to a group, add user names in the **Users:** field of the form generated by the **p4 group groupname** command. User names are entered under the **Users:** field header; each user name must be typed on its own line, indented. A single user can be listed in any number of groups. Group owners are not necessarily members of a group; if a group owner is to be a member of the group, the userid must also be added to the **Users:** field.

Groups can contain other groups as well as individual users. To add all users in a previously defined group to the group you're working with, include the group name in the **Subgroups:** field of the **p4 group** form. User and group names occupy separate namespaces, so groups and users can have the same names.

Adding nonexistent users to group definitions does not actually create the users, nor does it consume licenses; use the **p4 user** command to create users.

Groups and protections

To use a group with the **p4 protect** form, specify a group name instead of a user name in any line in the protections table and set the value of the second field on the line to **group** instead of **user**. All the users in that group are granted the specified access.

Example 5.4. Granting access to Perforce groups

This protections table grants **list** access to all members of the group **devgrp**, and **super** access to user **edk**:

```
list       group       devgrp      *       //...
super      user        edk         *       //...
```

According to the following three permission lines, group ac1 will have write access to **//ac1/...** while giving the group read-only access to **//ac1/ac1_dev/...**.

```
write       group       ac1      *       //ac1/...
list        group       ac1      *       -//ac1/ac1_dev/...
read        group       ac1      *       //ac1/ac1_dev/...
```

If a user belongs to multiple groups, one permission can override another. For instance, if you use exclusionary mappings to deny access to an area of the depot to members of **group1**, but grant access

to the same area of the depot to members of `group2`, a user who is a member of both `group1` and `group2` is either granted or denied access based on whichever line appears last in the protections table. The actual permissions granted to a specific user can be determined by replacing the names of all groups to which a particular user belongs with the user's name within the protections table and applying the rules described earlier in this chapter.

Synchronizing Perforce groups with LDAP groups

You can configure Perforce to automatically synchronize the contents of a given Perforce user group with that of an LDAP group. Protections are still assigned based on the identity of the Perforce group (using the **`p4 protect`** command), but which users are included in the Perforce group is determined by the membership of the LDAP group.

Synchronization can happen once or at specified intervals. See the description of the **`p4 ldapsync`** command in the *P4 Command Reference* for additional information.

Before you configure group synchronization, you need to define an LDAP configuration.

Note	If the LDAP server requires login for read-only queries, the LDAP configuration must contain valid bind credentials in the LDAP spec's `SearchBindDN` and `SearchPasswd` fields.

To configure group synchronization, you must do the following:

1. Define the following fields in the Perforce `group` spec:

   - `LdapConfig`: The name of an LDAP configuration created using the **`p4 ldap`** command.

     The LDAP configuration provides the hostname, port, and encryption for the LDAP connection; it also specifies how authentication is to be done using the `SearchBindDN`, `SearchPasswd`, and `GroupSearchBaseDN` fields.

   - `LdapSearchQuery`: The search query to identify the group member records.

   - `LdapUserAttribute`: The attribute that contains the group member's user id. This user name is added to the Perforce group.

2. Define a group owner for the Perforce group. The owner does not have to be a member of the corresponding LDAP group.

3. Use the **`p4 ldapsync`** command, specifying which Perforce group(s) should be synchronized, to test the anticipated results using a command like the following.

   ```
   $ p4 ldapsync -g -n my-perforce-group1 my-perforce-group2
   ```

   `p4 ldapsync` uses the context provided by the LDAP configuration to execute the search query and collect all the defined attributes from the results that are returned. The resultant list becomes the members list of the group.

4. If you are satisfied with the preview results, run **`p4 ldapsync`** again (without `-n`) to synchronize the groups.

To schedule synchronization to occur at regular intervals, you must make the **p4 ldapsync** command run at startup time and specify the value of the interval. For details see the description of the **p4 ldapsync** command in *P4 Command Reference*.

The following examples, included in "Synchronizing with Active Directory" on page 93 and "Synchronizing with OpenLDAP" on page 94, demonstrate two ways in which you can define group synchronization. These examples illustrate how configurations depend on how references to users and groups are stored on different servers:

- OpenLDAP stores a list of memberUid's in its group records; these can often be used directly as Perforce user names.
- Active Directory stores a list of member's full DN's in its group records, and the full DN of each group a user belongs to in its user records; in this case, you need to look for the user records that contain the back reference to the group instead of finding the group record directly.

Note the difference in the GroupBaseDn in the LDAP spec. In Active Directory, we're looking for users who are in the group; in OpenLDAP, we're looking for groups that contain users. This affects the path we're searching under.

In the following examples, both servers have user under the DN `ou=users,dc=example,dc=com`. We will be creating a Perforce group called `development` that is populated from the LDAP group `cn=development,ou=groups,dc=example,dc=com.`

Synchronizing with Active Directory

We begin with a sample LDAP configuration named `my-ad-example` defined as follows:

```
Name:              my-ad-example
Host:              ad.example.com
Port:              389
Encryption:        tls
BindMethod:        search
SearchBaseDN:      ou=users,dc=example,dc=com
SearchFilter:      (&(objectClass=user)(sAMAccountName=%user%))
SearchBindDN:      cn=read-only,ou=users,dc=example,dc=com
SearchPasswd:      password
SearchScope:       subtree
GroupBaseDN:       ou=users,dc=example,dc=com
GroupSearchScope:  subtree
```

The group spec created by the command **p4 group development**, would then look like this:

```
Group:             development
LdapConfig:        my-ad-example
LdapSearchQuery:   (&(objectClass=user)(memberOf=cn=development,ou=groups,
                                                    dc=example,dc=com))
LdapUserAttribute: sAMAccountName
Owners:            super
```

Synchronizing with OpenLDAP

We begin with a sample LDAP configuration named `my-openldap-example` defined as follows:

```
Name:              my-openldap-example
Host:              openldap.example.com
Port:              389
Encryption:        tls
BindMethod:        search
SearchBaseDN:      ou=users,dc=example,dc=com
SearchFilter:      (&(objectClass=inetOrgPerson)(uid=%user%))
SearchBindDN:      cn=read-only,ou=users,dc=example,dc=com
SearchPasswd:      password
SearchScope:       subtree
GroupBaseDN:       ou=groups,dc=example,dc=com
GroupSearchScope:  subtree
```

The group spec created by the command **`p4 group development`**, would then look like this:

```
Group:             development
LdapConfig:        my-openldap-example
LdapSearchQuery:   (&(objectClass=posixGroup)(cn=development))
LdapUserAttribute: memberUid
Owners:            super
```

Deleting groups

To delete a group, invoke

```
$ p4 group -d groupname
```

Alternately, invoke **`p4 group groupname`** and delete all users, subgroups, and owners from the group in the resulting editor form. The group will be deleted when the form is closed.

Comments in protection tables

Protection tables can be difficult to interpret and debug. Including comments can make this work much easier.

- You can append comments at the end of a line using the `##` symbols:

```
write user *   10.1.1.1   //depot/test/...  ## robinson crusoe
```

- Or you can write a comment line by prefixing the line with the `##` symbols:

```
## robinson crusoe
write user *   10.1.1.1   //depot/test/...
```

Warning

Comments you have created using the P4Admin tool are not compatible with comments created using the 2016.1 version of **`p4 protect`**. You can use the following command to convert a file containing comments created with P4Admin into a file containing **`p4 protect`** type comments:

```
$ p4 protect --convert-p4admin-comments -o
```

Then save the resulting file.

Once you have converted the comments, you must continue to define and manage protections using **`p4 protect`** and can no longer use P4Admin to do so because this tool is unable to parse **`p4 protect`** comments.

How protections are implemented

This section describes the algorithm that Perforce follows to implement its protection scheme. Protections can be used properly without reading this section; the material here is provided to explain the logic behind the behavior described above.

Users' access to files is determined by the following steps:

1. The command is looked up in the command access level table shown in "Access Levels Required by Perforce Commands" on page 96 to determine the minimum access level needed to run that command. In our example, **`p4 print`** is the command, and the minimum access level required to run that command is **`read`**.

2. Perforce makes the first of two passes through the protections table. Both passes move up the protections table, bottom to top, looking for the first relevant line.

   The first pass determines whether the user is permitted to know if the file exists. This search simply looks for the first line encountered that matches the user name, host IP address, and file argument. If the first matching line found is an inclusionary protection, the user has permission to at least list the file, and Perforce proceeds to the second pass. Otherwise, if the first matching protection found is an exclusionary mapping, or if the top of the protections table is reached without a matching protection being found, the user has no permission to even list the file, and will receive a message such as `File not on client`.

   ### Example 5.5. Interpreting the order of mappings in the protections table

   Suppose the protections table is as follows:

   ```
   write      user       *       *      //...
   read       user       edk     *      -//...
   read       user       edk     *      //depot/elm_proj/...
   ```

   If Ed runs **`p4 print //depot/file.c`**, Perforce examines the protections table bottom to top, and first encounters the last line. The files specified there don't match the file that Ed wants to print, so this line is irrelevant. The second-to-last line is examined next; this line matches Ed's user name,

his IP address, and the file he wants to print; since this line is an exclusionary mapping, Ed isn't allowed to list the file.

3. If the first pass is successful, Perforce makes a second pass at the protections table; this pass is the same as the first, except that access level is now taken into account.

   If an inclusionary protection line is the first line encountered that matches the user name, IP address, and file argument, *and* has an access level greater than or equal to the access level required by the given command, the user is given permission to run the command.

   If an exclusionary mapping is the first line encountered that matches according to the above criteria, or if the top of the protections table is reached without finding a matching protection, the user has no permission to run the command, and receives a message such as:

```
You don't have permission for this operation
```

Access Levels Required by Perforce Commands

The following table lists the minimum access level required to run each command. For example, because **`p4 add`** requires at least `open` access, you can run **`p4 add`** if you have `open`, `write`, `admin`, or `super` access

Command	Access Level	Notes
`add`	`open`	
`admin`	`super`	
`annotate`	`read`	
`archive`	`admin`	
`attribute`	`write`	The `-f` flag to set the attributes of submitted files requires `admin` access.
`branch`	`open`	The `-f` flag to override existing metadata or other users' data requires `admin` access.
`branches`	`list`	
`cachepurge`	`super`	
`change`	`open`	The `-o` flag (display a change on standard output) requires only `list` access. The `-f` flag to override existing metadata or other users' data requires `admin` access.
`changes`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.

Command	Access Level	Notes
`clean`	`read`	
`client`	`list`	The `-f` flag to override existing metadata or other users' data requires `admin` access.
`clients`	`list`	
`clone`	`read`	On the remote server.
`configure`	`super`	
`copy`	`list`	`list` access to the source files; `open` access to the destination files.
`counter`	`review`	`list` access to at least one file in any depot is required to view an existing counter's value; `review` access is required to change a counter's value or create a new counter.
`counters`	`list`	
`cstat`	`list`	
`dbschema`	`super`	
`dbstat`	`super`	
`dbverify`	`super`	
`delete`	`open`	
`depot`	`super`	The `-o` flag to this command, which allows the form to be read but not edited, requires only `list` access.
`depots`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`describe`	`read`	The `-s` flag to this command, which does not display file content, requires only `list` access.
`diff`	`read`	
`diff2`	`read`	
`dirs`	`list`	
`diskspace`	`super`	

Command	Access Level	Notes
`edit`	`open`	
`export`	`super`	
`fetch`	`admin`	
`filelog`	`list`	
`files`	`list`	
`fix`	`open`	
`fixes`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`flush`	`list`	
`fstat`	`list`	
`grep`	`read`	
`group`	`super`	The `-o` flag to this command, which allows the form to be read but not edited, requires only `list` access. The `-a` flag to this command requires only `list` access, provided that the user is also listed as a group owner. The `-A` flag requires `admin` access.
`groups`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`have`	`list`	
`help`	`none`	
`ignores`	`N/A`	
`info`	`none`	
`init`	`N/A`	
`integrate`	`open`	The user must have `open` access on the target files and `read` access on the source files.
`integrated`	`list`	
`interchanges`	`list`	

Command	Access Level	Notes
`istat`	`list`	
`job`	`open`	The `-o` flag to this command, which allows the form to be read but not edited, requires only `list` access. The `-f` flag to override existing metadata or other users' data requires `admin` access.
`jobs`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`jobspec`	`admin`	The `-o` flag to this command, which allows the form to be read but not edited, requires only `list` access.
`journalcopy`	`super`	
`journaldbchecksums`	`super`	
`journals`	`super` or `operator`	
`key`	`review`	`list` access to at least one file in any depot is required to view an existing key's value; `review` access is required to change a key's value or create a new key.
`key`	`list`	`admin` access is required if the `dm.keys.hide` configurable is set to `2`.
`keys`	`list`	`admin` access is required if the `dm.keys.hide` configurable is set to `1` or `2`.
`label`	`open`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot. The `-f` flag to override existing metadata or other users' data requires `admin` access.
`labels`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`labelsync`	`open`	
`ldap`	`super`	
`ldaps`	`super`	

Command	Access Level	Notes
`ldapsync`	`super`	
`license`	`super`	The `-u` flag, which displays license usage, requires only `admin` access.
`list`	`open`	
`lock`	`write`	
`lockstat`	`super`	
`logappend`	`list`	
`logger`	`review`	
`login`	`list`	
`logout`	`list`	
`logparse`	`super`	
`logrotate`	`super`	
`logschema`	`super`	
`logstat`	`super`	
`logtail`	`super`	
`merge`	`open`	
`monitor`	`list`	`super` access is required to terminate or clear processes, or to view arguments.
`move`	`open`	
`obliterate`	`admin`	
`opened`	`list`	
`passwd`	`list`	
`ping`	`admin`	
`populate`	`open`	
`print`	`read`	

Command	Access Level	Notes
`property`	`list, admin`	`list` to read, `admin` to add/delete new properties, or show a property setting and sequence number for all users and groups.
`protect`	`super`	
`protects`	`list`	`super` access is required to use the `-a`, `-g`, and `-u` flags.
`proxy`	`none`	Must be connected to a Perforce Proxy.
`prune`	`write`	For stream owner.
`pull`	`super`	
`push`	`read` or `write`	`read` on the local server or `write` on the remote server.
`reconcile`	`open`	
`reload`	`open`	`admin` access is required to use **`p4 reload -f`** to reload other users' workspaces and labels.
`remote`	`open` or `list` or `admin`	`open` or `list` to use the `-o` option or `admin` to use the `-f` option.
`remotes`	`list`	
`rename`	`read` or `write`	`read` for *`fromFile`* or `write` for *`toFile`*.
`renameuser`	`super`	
`reopen`	`open`	
`replicate`	`super`	
`resolve`	`open`	
`resolved`	`open`	
`restore`	`admin`	
`resubmit`	`write` or `admin`	`write` or `admin` for `-i` option.
`revert`	`list`	
`review`	`review`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.

Command	Access Level	Notes
`reviews`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`server`	`super`	
`serverid`	`list`	`super` access is required to set the server ID.
`servers`	`list`	
`set`	`none`	
`shelve`	`open`	`admin` access is required to forcibly delete shelved files with **`p4 shelve -f -d`**
`sizes`	`list`	
`status`	`open`	
`stream`	`open`	
`streams`	`list`	
`submit`	`write`	
`switch`	`open` or `list` or `write`	`open` to use the `-c` or `-r` options, or `list` to use the `-L`, or `write` for default switching.
`sync`	`read`	
`tag`	`list`	
`tickets`	`none`	
`triggers`	`super`	
`trust`	`none`	
`typemap`	`admin`	The `-o` flag to this command, which allows the form to be read but not edited, requires only `list` access.
`unload`	`open`	`admin` access is required to use **`p4 unload -f`** to unload other users' workspaces and labels.
`unlock`	`open`	The `-f` flag to override existing metadata or other users' data requires `admin` access.
`unshelve`	`open`	
`unsubmit`	`admin`	

Command	Access Level	Notes
`unzip`	`admin`	
`update`	`list`	
`user`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot. The `-f` flag (which is used to create or edit users) requires `super` access.
`users`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot. If the `run.users.authorize` configurable is set to 1, you must also authenticate yourself to the server before you can run **`p4 users`**.
`verify`	`admin`	
`where`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`workspace`	`list`	
`workspaces`	`list`	
`zip`	`super`	

Commands that list files, such as **`p4 describe`**, list only those files to which the user has at least `list` access.

Some commands (for example, **`p4 change`**, when you edit a previously submitted changelist) take a `-f` flag that can only be used by Perforce superusers. See "Forcing operations with the -f flag" on page 129 for details.

Chapter 6

Backup and Recovery

The Perforce service stores two kinds of data: *versioned files* and *metadata*.

- *Versioned files* are files submitted by Perforce users. Versioned files are stored in directory trees called *depots*.

  There is one subdirectory under the server's root directory for each depot in your Perforce installation. The versioned files for a given depot are stored in a tree of directories beneath this subdirectory.

- *Database files* store *metadata*, including changelists, opened files, client workspace specifications, branch mappings, and other data concerning the history and present state of the versioned files.

  Database files appear as **db.*** files in the top level of the server root directory. Each **db.*** file contains a single, binary-encoded database table.

This chapter describes the commands and processes you use to backup and recover your Perforce server. For information about backup and recovery of distributed systems, see *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.

Backup and recovery concepts

Disk space shortages, hardware failures, and system crashes can corrupt any of the Perforce server's files. That's why the entire Perforce root directory structure (your versioned files and your database) must be backed up regularly.

The versioned files are stored in subdirectories beneath your Perforce server root, and can be restored directly from backups without any loss of integrity.

The files that constitute the Perforce database, on the other hand, are not guaranteed to be in a state of transactional integrity if archived by a conventional backup program. Restoring the **db.*** files from regular system backups can result in an inconsistent database. The only way to guarantee the integrity of the database after it's been damaged is to reconstruct the **db.*** files from Perforce checkpoint and journal files:

- A *checkpoint* is a snapshot or copy of the database at a particular moment in time.
- A *journal* is a log of updates to the database since the last snapshot was taken.

The checkpoint file is often much smaller than the original database, and it can be made smaller still by compressing it. The journal file, on the other hand, can grow quite large; it is truncated whenever a checkpoint is made, and the older journal is renamed. The older journal files can then be backed up offline, freeing up more space locally.

Both the checkpoint and journal are text files, and have the same format. A checkpoint and (if available) its subsequent journal can restore the Perforce database.

Warning

Checkpoints and journals archive only the Perforce database files, **not** the versioned files stored in the depot directories!

You must always back up the depot files (your versioned file tree) with the standard OS backup commands after checkpointing.

Because the information stored in the Perforce database is as irreplaceable as your versioned files, checkpointing and journaling are an integral part of administering Perforce, and must be part of your regular backup cycle.

Checkpoint files

A *checkpoint* is a file that contains all information necessary to re-create the metadata in the Perforce database. When you create a checkpoint, the Perforce database is locked, enabling you to take an internally consistent snapshot of that database.

Versioned files are backed up separately from checkpoints. This means that a checkpoint does *not* contain the contents of versioned files, and as such, **you cannot restore any versioned files from a checkpoint.** You can, however, restore all changelists, labels, jobs, and so on, from a checkpoint.

To guarantee database integrity upon restoration, the checkpoint must be as old as, or older than, the versioned files in the depot. This means that the database must be checkpointed, and the checkpoint generation must be complete, before the backup of the versioned files starts.

Regular checkpointing is important to keep the journal from getting too long. Making a checkpoint immediately before backing up your system is good practice.

Creating a checkpoint

Checkpoints are not created automatically; someone or something must run the checkpoint command on the Perforce server machine. To create a checkpoint, invoke the **p4d** program with the **-jc** (journal-create) flag:

```
$ p4d -r server_root -jc
```

You can create a checkpoint while the Perforce service (**p4d**) is running. The checkpoint is created in your server root directory (that is, `P4ROOT` if no *`server_root`* is specified).

To make the checkpoint, **p4d** locks the database and then dumps its contents to a file named `checkpoint.`*`n`* in the `P4ROOT` directory, where *n* is a sequence number.

Before unlocking the database, **p4d** also copies (on UNIX where the journal is uncompressed, renames) the journal file to a file named `journal.`*`n`*`-1` in the `P4ROOT` directory (regardless of the directory in which the current journal is stored), and then truncates the current journal. The MD5 checksum of the checkpoint is written to a separate file, `checkpoint.`*`n`*`.md5`, and the `lastCheckpointAction` counter is updated to reflect successful completion.

Note

When verifying the MD5 signature of a compressed checkpoint, the checkpoint must first be uncompressed into a form that reflects the line ending convention native to the system that produced the checkpoint. (That is, a compressed checkpoint generated by a Windows server should have CR/LF line endings, and a compressed checkpoint generated on a UNIX system should have LF line endings.)

This guarantees that the last checkpoint (`checkpoint.`*`n`*) combined with the current journal (`journal`) always reflects the full contents of the database at the time the checkpoint was created.

The sequence numbers reflect the roll-forward nature of the journal. To restore databases to older checkpoints, match the sequence numbers. That is, you can restore the state of the Perforce server as it was when `checkpoint.6` was taken by restoring `checkpoint.5` and then loading `journal.5` which contains all the changes made between `checkpoint.5` and `checkpoint.6`. In most cases, you're only interested in restoring the current database, which is reflected by the highest-numbered `checkpoint.`*`n`* rolled forward with the changes in the current `journal`.

To specify a prefix or directory location for the checkpoint and journal, use the `-jc` option. For example, you might create a checkpoint with:

```
$ p4d -jc prefix
```

In this case, your checkpoint and journal files are named *`prefix`*`.ckp.`*`n`* and *`prefix`*`.jnl.`*`n`* respectively, where *`prefix`* is as specified on the command line and *`n`* is a sequence number. If no *`prefix`* is specified, the default filenames `checkpoint.`*`n`* and `journal.`*`n`* are used. You can store checkpoints and journals in the directory of your choice by specifying the directory as part of the prefix. For example

```
$ p4 -r . -J /where/my/journal/lives/journal -z -jc
               /Users/bruges/server151/checkpoints/mybackup
```

returns

```
Checkpointing to /Users/bruges/server151/checkpoints/mybackup.ckp.299.gz…
MD5 (/Users/bruges/server151/checkpoints/mybackup.ckp.299) = 5D7D8E548D080B16ECB66AD6CE0F2E5D
Rotating journal to /Users/bruges/server151/checkpoints/mybackup.jnl.298.gz…
```

You can also specify the prefix for a server with:

```
$ p4 configure set journalPrefix=prefix
```

When the `journalPrefix` configurable is set, the configured `prefix` takes precedence over the default filenames. This behavior is particularly useful in multi-server and replicated environments.

To create a checkpoint without being logged in to the machine running the Perforce service, use the command:

```
$ p4 admin checkpoint [-z | -Z] [prefix]
```

Running **`p4 admin checkpoint`** is equivalent to **`p4d -jc`** except that using **`p4 admin checkpoint`** requires that you be connected to the server. You must be a Perforce superuser to use **`p4 admin`**.

You can set up an automated program to create your checkpoints on a regular schedule. Be sure to always check the program's output to ensure that checkpoint creation was started. Compare the checkpoint's actual MD5 checksum with that recorded in the `.md5` file, and back up the `.md5` file along with the checkpoint. After successful creation, a checkpoint file can be compressed, archived, or moved

onto another disk. At that time or shortly thereafter, back up the versioned files stored in the depot subdirectories.

To restore from a backup, *the checkpoint must be at least as old as the files in the depots*, that is, the versioned files can be newer than the checkpoint, but not the other way around. As you might expect, the shorter this time gap, the better.

If the checkpoint command itself fails, contact Perforce technical support immediately. Checkpoint failure is usually a symptom of a resource problem (disk space, permissions, and so on) that can put your database at risk if not handled correctly.

Note	You can verify the integrity of a checkpoint using the **`p4d -jv`** command.

Journal files

The *journal* is the running transaction log that keeps track of all database modifications since the last checkpoint. It's the bridge between two checkpoints.

If you have Monday's checkpoint and the journal that was collected from then until Wednesday, those two files (Monday's checkpoint plus the accumulated journal) contain the same information as a checkpoint made Wednesday. If a disk crash were to cause corruption in your Perforce database on Wednesday at noon, for instance, you could still restore the database even though Wednesday's checkpoint hadn't yet been made.

Warning	By default, the current journal filename is **`journal`**, and the file resides in the **`P4ROOT`** directory. However, if a disk failure corrupts that root directory, your journal file will be inaccessible too. We strongly recommend that you set up your system so that the journal is written to a filesystem other than the **`P4ROOT`** filesystem. To do this, specify the name of the journal file in the environment variable **`P4JOURNAL`** or use the ***`-J filename`*** flag when starting **`p4d`**.

To restore your database, you only need to keep the most recent journal file accessible, but it doesn't hurt to archive old journals with old checkpoints, should you ever need to restore to an older checkpoint.

Journaling is automatically enabled on all Windows and UNIX platforms. If **`P4JOURNAL`** is left unset (and no location is specified on the command line), the default location for the journal is **`$P4ROOT/journal`**.

The journal file grows until a checkpoint is created; you'll need make regular checkpoints to control the size of the journal file. An extremely large current journal is a sign that a checkpoint is needed.

Every checkpoint after your first checkpoint starts a new journal file and renames the old one. The old **`journal`** is renamed to **`journal.`*`n`***, where ***`n`*** is a sequence number, and a new **`journal`** file is created.

By default, the journal is written to the file **`journal`** in the server root directory (**`P4ROOT`**. Because there is no sure protection against disk crashes, the journal file and the Perforce server root should be located

on different filesystems, ideally on different physical drives. The name and location of the journal can be changed by specifying the name of the journal file in the environment variable `P4JOURNAL` or by providing the -J *`filename`* flag to **`p4d`**.

Warning	If you create a journal file with the **`-J`** *`filename`* flag, make sure that subsequent checkpoints use the same file, or the journal will not be properly renamed.

Whether you use `P4JOURNAL` or the -J *`journalfile`* option to **`p4d`**, the journal filename can be provided either as an absolute path, or as a path relative to the server root.

Example 6.1. Specifying journal files

Starting the service with:

```
$ p4d -r $P4ROOT -p 1666 -J /usr/local/perforce/journalfile
Perforce Server starting...
```

requires that you either checkpoint with:

```
$ p4d -r $P4ROOT -J /usr/local/perforce/journalfile -jc
Checkpointing to checkpoint.19...
Saving journal to journal.18...
Truncating /usr/local/perforce/journalfile...
```

or set `P4JOURNAL` to `/usr/local/perforce/journalfile` and use the following command:

```
$ p4d -r $P4ROOT -jc
Checkpointing to checkpoint.19...
MD5(checkpoint.19)=48769A82387B04987568309823E784C9
Rotating /usr/local/perforce/journalfile to journal.18
```

If your `P4JOURNAL` environment variable (or command-line specification) doesn't match the setting used when you started the Perforce service, the checkpoint is still created, but the journal is neither saved nor truncated. This is highly undesirable!

Checkpoint and journal history

You can use the **`p4 journals`** command to display the history of checkpoint and journal activity for the server. This history includes information about the following events: the server takes a checkpoint, journal rotation, journal replay, checkpoint scheduling. For detailed information about command output and options, see the description of the **`p4 journals`** command in the *P4 Command Reference*.

Verifying journal integrity

You can verify the integrity of a checkpoint using the **`p4d -jv`** command.

Automating maintenance work after journal rotation

To configure Perforce to run trigger scripts when journals are rotated, use the `journal-rotate` and `journal-rotate-lock` type triggers. Journal-rotate triggers are executed after the journal is rotated on a running server, but only if journals are rotated with the **`p4 admin journal`** or **`p4 admin checkpoint`** commands. Journals are not rotated if you invoke the **`p4d -jc`** or **`p4d --jj`** commands.

Journal-rotate triggers allow you to run maintenance routines on servers after the journal has been rotated, either while the database tables are still locked or after the locks have been released. These triggers are intended to be used on replicas or edge servers where journal rotation is triggered by journal records. The server must be running for these triggers to be invoked.

See "Triggering on journal rotation" on page 204 for more information.

Disabling journaling

To disable journaling, stop the service, remove the existing journal file (if it exists), set the environment variable `P4JOURNAL` to `off`, and restart **`p4d`** without the `-J` flag.

Versioned files

Your checkpoint and journal files are used to reconstruct the Perforce database files only. Your versioned files are stored in directories under the Perforce server root, and must be backed up separately.

Versioned file formats

Versioned files are stored in subdirectories beneath your server root. Text files are stored in RCS format, with filenames of the form ***`filename`*`,v`**. There is generally one RCS-format (`,v`) file per text file. Binary files are stored in full in their own directories named ***`filename`*`,d`**. Depending on the Perforce file type selected by the user storing the file, there can be one or more archived binary files in each ***`filename`*`,d`** directory. If more than one file resides in a ***`filename`*`,d`** directory, each file in the directory refers to a different revision of the binary file, and is named **`1.`*`n`***, where ***`n`*** is the revision number.

Perforce also supports the AppleSingle file format for Macintosh. These files are stored in full and compressed, just like other binary files. They are stored in the Mac's AppleSingle file format; if need be, the files can be copied directly from the server root, uncompressed, and used as-is on a Macintosh.

Because Perforce uses compression in the depot file tree, do not assume compressibility of the data when sizing backup media. Both text and binary files are either compressed by **`p4d`** (and are denoted by the `.gz` suffix) before storage, or they are stored uncompressed. At most installations, if any binary files in the depot subdirectories are being stored uncompressed, they were probably incompressible to begin with. (For example, many image, music, and video file formats are incompressible.)

Backing up after checkpointing

In order to ensure that the versioned files reflect all the information in the database after a post-crash restoration, the **`db.*`** files must be restored from a checkpoint that is at least as old as (or older than) your versioned files. For this reason, create the checkpoint before backing up the versioned files in the depot directory or directories.

Although your versioned files can be newer than the data stored in your checkpoint, it is in your best interest to keep this difference to a minimum; in general, you'll want your backup script to back up your versioned files immediately after successfully completing a checkpoint.

Backup procedures

To back up your Perforce installation, perform the following steps as part of your nightly backup procedure.

1. Verify the integrity of your server:

```
$ p4 verify //...
```

You might want to use the `-q` (quiet) option with **`p4 verify`**. If called with the `-q` option, **`p4 verify`** produces output only when errors are detected.

By running **`p4 verify`** before the backup, you verify that the archive data on the server is correct and has not been damaged since the files were submitted.

Regular use of **`p4 verify`** is good practice not only because it enables you to spot any corruption before a backup, but also because it gives you the ability, following a crash, to determine whether or not the files restored from your backups are in good condition.

Note

For large installations, **`p4 verify`** might take some time to run. Furthermore, the server is under heavy load when **`p4 verify`** is verifying files, which can impact the performance of other Perforce commands. Administrators of large sites might choose to perform **`p4 verify`** on a weekly basis, rather than a nightly basis.

For more about the **`p4 verify`** command, see "Verifying files by signature" on page 31.

2. Make a checkpoint by invoking **`p4d`** with the `-jc` (journal-create) flag, or by using the **`p4 admin`** command. Use one of:

```
$ p4d -jc
```

or:

```
$ p4 admin checkpoint
```

Because **`p4d`** locks the entire database when making the checkpoint, you do not generally have to stop the Perforce service during any part of the backup procedure.

Note

If your site is very large (gigabytes of `db.*` files), creating a checkpoint might take a considerable length of time.

> Under such circumstances, you might want to defer checkpoint creation and journal truncation until times of low system activity. You might, for instance, archive only the `journal` file in your nightly backup and only create checkpoints and roll the journal file on a weekly basis.

3. Ensure that the checkpoint has been created successfully before backing up any files. (After a disk crash, the last thing you want to discover is that the checkpoints you've been backing up for the past three weeks were incomplete!)

   You can tell that the checkpoint command has completed successfully by examining the error code returned from **`p4d -jc`** (or **`p4d admin checkpoint`**) or by observing the truncation of the current journal file. You can also use the command **`p4d -jv`** to verify the integrity of a checkpoint.

4. Confirm that the checkpoint was correctly written to disk by comparing the MD5 checksum of the checkpoint with the `.md5` file created by the checkpoint process.

   The checksum in the `.md5` file corresponds to the checksum of the file as it existed before any compression was applied, and assumes UNIX-style line endings even if the service is hosted on Windows.

   If your checkpoint file was created with the `-z` compression option, you might need to decompress it and account for line ending differences. On Windows, after decompressing a checkpoint, Windows line endings must be re-added before calculating the `.md5` sum.

5. Once the checkpoint has been created successfully, back up the checkpoint file, its `.md5` file, the rotated journal file, and your versioned files. (In most cases, you don't actually need to back up the journal, but it is usually good practice to do so. If the checkpoint is `n`, the rotated journal is `journal.n-1`.)

Note

> There are rare instances (for instance, users obliterating files during backup, or submitting files on Windows servers during the file backup portion of the process) in which your versioned file tree can change during the interval between the time the checkpoint was taken and the time at which the versioned files are backed up by the backup utility.
>
> Most sites are unaffected by these issues. Having Perforce available on a 24/7 basis is generally a benefit worth this minor risk, especially if backups are being performed at times of low system activity.
>
> If, however, the reliability of every backup is of paramount importance, consider stopping the Perforce service before checkpointing, and restart it only after the backup process has completed. Doing so will eliminate any risk of the system state changing during the backup process.
>
> You never need to back up the **`db.*`** files. Your latest checkpoint and journal contain all the information necessary to re-create them. More significantly, a database restored from **`db.*`** files is not guaranteed to be in a state of transactional integrity. A database restored from a checkpoint is.

Note

On Windows, if you make your system backup while the Perforce service is running, you must ensure that your backup program doesn't attempt to back up the `db.*` files.

If you try to back up the `db.*` files with a running server, Windows locks them while the backup program backs them up. During this brief period, Perforce is unable to access the files; if a user attempts to perform an operation that would update the file, the server can fail.

If your backup software doesn't allow you to exclude the `db.*` files from the backup process, stop the server with **`p4 admin stop`** before backing up, and restart the service after the backup process is complete.

6. If you have used the **`p4 serverid`** command to identify your server with a `server.id` file, the `server.id` file (which exists in the server's root directory) must be backed up.

Recovery procedures

If the database files become corrupted or lost either because of disk errors or because of a hardware failure such as a disk crash, the database can be re-created with your stored checkpoint and journal.

There are many ways in which systems can fail. Although this guide cannot address all failure scenarios, it can at least provide a general guideline for recovery from the two most common situations, specifically:

- corruption of your Perforce database only, without damage to your versioned files
- corruption to both your database and versioned files.

The recovery procedures for each failure are slightly different and are discussed separately in the following two sections.

If you suspect corruption in either your database or versioned files, contact Perforce technical support.

Database corruption, versioned files unaffected

If only your database has been corrupted, (that is, your `db.*` files were on a drive that crashed, but you were using symbolic links to store your versioned files on a separate physical drive), you need only re-create your database.

You *will* need:

- The last checkpoint file, which should be available from the latest `P4ROOT` directory backup. If, when you backed up the checkpoint, you also backed up its corresponding `.md5` file, you can confirm that the checkpoint was restored correctly by comparing its checksum with the contents of the restored `.md5` file.
- The current journal file, which should be on a separate filesystem from your `P4ROOT` directory, and which should therefore have been unaffected by any damage to the filesystem where your `P4ROOT` directory was held.

You will *not* need:

- Your backup of your versioned files; if they weren't affected by the crash, they're already up to date

To recover the database

1. Stop the current instance of **p4d**:

   ```
   $ p4 admin stop
   ```

   (You must be a Perforce superuser to use **p4 admin**.)

2. Rename (or move) the database (db.*) files:

   ```
   $ mv your_root_dir /db.* /tmp
   ```

   There can be no db.* files in the P4ROOT directory when you start recovery from a checkpoint. Although the old db.* files are never used during recovery, it's good practice not to delete them until you're certain your restoration was successful.

3. Verify the integrity of your checkpoint using a command like the following:

   ```
   $ p4d -jv my_checkpoint_file
   ```

   The command tests the following:

   - Can the checkpoint be read from start to finish?
   - If it's zipped can it be successfully unzipped?
   - If it has an MD5 file with its MD5, does it match?
   - Does it have the expected header and trailer?

     Use the -z flag with the -jv flag to verify the integrity of compressed checkpoints.

4. Invoke **p4d** with the -jr (journal-restore) flag, specifying your most recent checkpoint and current journal. If you explicitly specify the server root (P4ROOT), the -r $P4ROOT argument must precede the -jr flag. Also, because the **p4d** process changes its working directory to the server root upon startup, any relative paths for the *checkpoint_file* and *journal_file* must be specified relative to the P4ROOT directory:

   ```
   $ p4d -r $P4ROOT -jr checkpoint_file journal_file
   ```

   This recovers the database as it existed when the last checkpoint was taken, and then applies the changes recorded in the journal file since the checkpoint was taken.

Note

If you're using the `-z` (compress) option to compress your checkpoints upon creation, you'll have to restore the uncompressed journal file separately from the compressed checkpoint.

That is, instead of using:

```
$ p4d -r $P4ROOT -jr checkpoint_file journal_file
```

you'll use two commands:

```
$ p4d -r $P4ROOT -z -jr checkpoint_file.gz
$ p4d -r $P4ROOT -jr journal_file
```

You must explicitly specify the `.gz` extension yourself when using the `-z` flag, and ensure that the `-r $P4ROOT` argument precedes the `-jr` flag.

Check your system

Your restoration is complete. See "Ensuring system integrity after any restoration" on page 117 to make sure your restoration was successful.

Your system state

The database recovered from your most recent checkpoint, after you've applied the accumulated changes stored in the current journal file, is up to date as of the time of failure.

After recovery, both your database and your versioned files should reflect all changes made up to the time of the crash, and no data should have been lost. If restoration was successful, the `lastCheckpointAction` counter will indicate "checkpoint completed".

Both database and versioned files lost or damaged

If both your database and your versioned files were corrupted, you need to restore both the database and your versioned files, and you'll need to ensure that the versioned files are no older than the restored database.

You *will* need:

- The last checkpoint file, which should be available from the latest `P4ROOT` directory backup. If, when you backed up the checkpoint, you also backed up its corresponding `.md5` file, you can confirm that the checkpoint was restored correctly by comparing its checksum with the contents of the restored `.md5` file.
- Your versioned files, which should be available from the latest `P4ROOT` directory backup.

You will *not* need:

- Your current journal file.

The journal contains a record of changes to the metadata and versioned files that occurred between the last backup and the crash. Because you'll be restoring a set of versioned files from a backup taken *before* that crash, the checkpoint alone contains the metadata useful for the recovery, and the information in the journal is of limited or no use.

To recover the database

1. Stop the current instance of **p4d**:

   ```
   $ p4 admin stop
   ```

   (You must be a Perforce superuser to use **p4 admin**.)

2. Rename (or move) the corrupt database (`db.*`) files:

   ```
   $ mv your_root_dir /db.* /tmp
   ```

   The corrupt `db.*` files aren't actually used in the restoration process, but it's safe practice not to delete them until you're certain your restoration was successful.

3. Compare the MD5 checksum of your most recent checkpoint with the checksum generated at the time of its creation, as stored in its corresponding `.md5` file.

   The `.md5` file written at the time of checkpointing holds the checksum of the file as it existed before any compression was applied, and assumes UNIX-style line endings even if the service is hosted on Windows. (If your checkpoint file was created with the `-z` compression option, you may need to decompress them and account for line ending differences.)

4. Invoke **p4d** with the `-jr` (journal-restore) flag, specifying *only* your most recent checkpoint:

   ```
   $ p4d -r $P4ROOT -jr checkpoint_file
   ```

   This recovers the database as it existed when the last checkpoint was taken, but does not apply any of the changes in the journal file. (The `-r $P4ROOT` argument must precede the `-jr` flag. Also, because the **p4d** process changes its working directory to the server root upon startup, any relative paths for the *checkpoint_file* must be specified relative to the `P4ROOT` directory.)

   The database recovery without the roll-forward of changes in the journal file brings the database up to date as of the time of your last backup. In this scenario, you do not want to apply the changes in the journal file, because the versioned files you restored reflect only the depot as it existed as of the last checkpoint.

To recover your versioned files

1. After you recover the database, you then need to restore the versioned files according to your system's restoration procedures (for instance, the UNIX **restore(1)** command) to ensure that they are as new as the database.

Check your system

Your restoration is complete. See "Ensuring system integrity after any restoration" on page 117 to make sure your restoration was successful.

Files submitted to the depot between the time of the last system backup and the disk crash will not be present in the restored depot.

Note

Although "new" files (submitted to the depot but not yet backed up) do not appear in the depot after restoration, it's possible (indeed, highly probable!) that one or more of your users will have up-to-date copies of such files present in their client workspaces.

Your users can find such files by using the following Perforce command to examine how files in their client workspaces differ from those in the depot. If they run…

```
$ p4 diff -se
```

…they'll be provided with a list of files in their workspace that differ from the files Perforce believes them to have. After verifying that these files are indeed the files you want to restore, you may want to have one of your users open these files for `edit` and submit the files to the depot in a changelist.

Your system state

After recovery, your depot directories might not contain the newest versioned files. That is, files submitted after the last system backup but before the disk crash might have been lost.

- In most cases, the latest revisions of such files can be restored from the copies still residing in your users' client workspaces.
- In a case where *only* your versioned files (but *not* the database, which might have resided on a separate disk and been unaffected by the crash) were lost, you might also be able to make a separate copy of your database and apply your journal to it in order to examine recent changelists to track down which files were submitted between the last backup and the disk crash.

In either case, contact Perforce Technical Support for further assistance.

Ensuring system integrity after any restoration

After any restoration, use the command:

```
$ p4 counter lastCheckpointAction
```

to confirm that the `lastCheckpointAction` counter has been updated to reflect the date and time of the checkpoint completion.

You should also run **`p4 verify`** to ensure that the versioned files are at least as new as the database:

```
$ p4 verify -q //...
```

This command verifies the integrity of the versioned files. The `-q` (quiet) option tells the command to produce output only on error conditions. Ideally, this command should produce no output.

If any versioned files are reported as `MISSING` by the **`p4 verify`** command, you'll know that there is information in the database concerning files that didn't get restored. The usual cause is that you restored from a checkpoint and journal made after the backup of your versioned files (that is, that your backup of the versioned files was older than the database).

If (as recommended) you've been using **`p4 verify`** as part of your backup routine, you can run **`p4 verify`** after restoration to reassure yourself that the restoration was successful.

If you have any difficulties restoring your system after a crash, contact Perforce Technical Support for assistance.

Chapter 7

Monitoring the Server

This chapter describes how you use **p4d** commands to monitor the server and its resources:

- You use the **p4 diskspace** command to monitor diskspace usage.
- You use the **p4 monitor** command to monitor processes.
- You set server trace flags with the **p4d** startup command to diagnose problems.
- You examine logs to track commands that exceed predetermined thresholds of resource usage.
- You use the **p4 monitor** command to display information about locked files.
- You use the P4AUDIT environment variable to enable the auditing of file access.
- You use a variety of logging commands to log information and manage log files.

Monitoring disk space usage

Use the **p4 diskspace** command to monitor diskspace usage. By default, **p4 diskspace** displays the amount of free space, diskspace used, and total capacity of any filesystem used by Perforce.

By default, the Perforce Server rejects commands when free space on the filesystems housing the P4ROOT, P4JOURNAL, P4LOG, or TEMP fall below 10 megabytes. To change this behavior, set the filesys.P4ROOT.min (and corresponding) configurables to your desired limits:

Configurable	Default Value	Meaning
filesys.P4ROOT.min	10M	Minimum diskspace required on server root filesystem before server rejects commands.
filesys.P4JOURNAL.min	10M	Minimum diskspace required on server journal filesystem before server rejects commands.
filesys.P4LOG.min	10M	Minimum diskspace required on server log filesystem before server rejects commands.
filesys.TEMP.min	10M	Minimum diskspace required for temporary operations before server rejects commands.
filesys.depot.min	10M	Minimum diskspace required for any depot before server rejects commands. (If there is less than filesys.depot.min diskspace available for any one depot, commands are rejected for transactions involving all depots.)

If the user account that runs the Perforce Server process is subject to disk quotas, the Server observes these quotas with respect to the filesys.*.min configurables, regardless of how much physical free space remains on the filesystem(s) in question. The next section explains the options you have in reconfiguring default values.

Specifying values for filesys configurables

In specifying `filesys.*.min` values, you have the option of specifying an absolute number or a percentage indicating a portion of the current space. So, there are six possible numeric formats you can use, as shown in the following table:

Format	Meaning
nnn	A plain number, used as is.
*nnn*K	A number in kilobytes For example, the following command sets `filesys.P4TEMP.min` to 100 kilobytes. `$ p4 configure set filesys.P4TEMP.min=100K`
*nnn*M	A number in megabytes For example, the following command sets `filesys.P4ROOT.min` to 10 megabytes. `$ p4 configure set filesys.P4ROOT.min=10M`
*nnn*G	A number in gigabytes. For example, the following command sets `filesys.P4JOURNAL.min` to 1 gigabytes. `$ p4 configure set filesys.P4JOURNAL.min=1G`
*nnn*T	A number in terabytes.
nnn%	A number as a percentage of the current space. For example, the following command means that at least ten percent of the total disk space must be free and available for `P4ROOT`. `$ p4 configure set filesys.P4ROOT.min=10%`

Determining available disk space

To estimate how much disk space is currently occupied by specific files in a depot, use the **`p4 sizes`** command with a block size corresponding to that used by your storage solution. For example, the command:

```
$ p4 sizes -a -s -b 512 //depot/...
```

shows the sum (`-s`) of all revisions (`-a`) in `//depot/...`, as calculated with a block size of 512 bytes.

```
//depot/... 34161 files 277439099 bytes 5429111 blocks
```

The data reported by **`p4 sizes`** actually reflects the diskspace required when files are synced to a client workspace, but can provide a useful estimate of server-side diskspace consumption.

Monitoring processes

Use the **`p4 monitor`** command to observe and control Perforce-related processes running on your Perforce server machine.

The following sections explain how you enable process monitoring and list running processes,.

Enabling process monitoring

Server process monitoring requires minimal system resources, but you must enable process monitoring for **`p4 monitor`** to work. To monitor all active commands, set the `monitor` configurable as follows:

```
$ p4 configure set monitor=1
```

Additional settings offer more options:

- `0`: Server process monitoring off. (Default)
- `2`: monitor both active commands and idle connections.
- `5`: monitor both active commands and idle connections, including a list of the files locked by the command for more than one second.
- `10`: monitor both active commands and idle connections, including a list of the files locked by the command for more than one second, with lock wait times included in the lock information.
- `25`: monitor both active commands and idle connections, including a list of the files locked by the command for any duration, with lock wait times included in the lock information.

How you set up monitoring levels `5`, `10`, and `25`, depends on the platform where the server is running. See the description of the **`p4 monitor`** command in *P4 Command Reference* for more information.

Enabling idle processes monitoring

By default, `IDLE` processes (often associated with custom applications based on the Perforce API) are not included in the output of **`p4 monitor`**. To include idle processes in the default output of **`p4 monitor`**, use monitoring level `2`.

```
$ p4 configure set monitor=2
```

To display idle processes, use the command:

```
$ p4 monitor show -s I
```

Listing running processes

To list the processes monitored by the Perforce server, use the command:

```
$ p4 monitor show
```

To restrict the display to processes currently in the running state, use the command:

```
$ p4 monitor show -s R
```

By default, each line of **`p4 monitor`** output looks like this:

```
pid status owner hh:mm:ss command [args]
```

where ***`pid`*** is the UNIX process ID (or Windows thread ID), ***`status`*** is `R` or `T` depending on whether the process is running or marked for termination, ***`owner`*** is the Perforce user name of the user who invoked the command, ***`hh:mm:ss`*** is the time elapsed since the command was called, and ***`command`*** and ***`args`*** are the command and arguments as received by the Perforce server. For example:

```
$ p4 monitor show
74612 R qatool     00:00:47 job
78143 R edk        00:00:01 filelog
78207 R p4admin    00:00:00 monitor
```

To show the arguments with which the command was called, use the `-a` (arguments) flag:

```
$ p4 monitor show -a
74612 R qatool     00:00:48 job job004836
78143 R edk        00:00:02 filelog //depot/main/src/proj/file1.c //dep
78208 R p4admin    00:00:00 monitor show -a
```

To obtain more information about user environment, use the `-e` flag. The `-e` flag produces output of the form:

```
pid client IP-address status owner workspace hh:mm:ss command [args]
```

where *client* is the Perforce application (and version string or API protocol level), *IP-address* is the IP address of the user's Perforce application, and *workspace* is the name of the calling user's current client workspace setting. For example:

```
$ p4 monitor show -e
74612 p4/2011.1 192.168.10.2   R qatool    buildenvir 00:00:47 job
78143           192.168.10.4   R edk       eds_elm    00:00:01 filelog
78207 p4/2011.1 192.168.10.10  R p4admin   p4server   00:00:00 monitor
```

By default, all user names and (if applicable) client workspace names are truncated at 10 characters, and lines are truncated at 80 characters. To disable truncation, use the `-l` (long-form) option:

```
$ p4 monitor show -a -l
74612 R qatool     00:00:50 job job004836
78143 R edk        00:00:04 filelog //depot/main/src/proj/file1.c //dep
ot/main/src/proj/file1.mpg
78209 R p4admin    00:00:00 monitor show -a -l
```

Only Perforce administrators and superusers can use the `-a`, `-l`, and `-e` options.

Setting server trace and tracking flags

To modify the behavior of command tracing or performance tracking, specify the appropriate `-v` *subsystem=value* flag to the **p4d** startup command. Use `P4LOG` or the `-L` `` `logfile` `` flag to specify the log file. For example:

```
$ p4d -r /usr/perforce -v server=2 -p 1666 -L /usr/perforce/logfile
```

Before you activate logging, make sure that you have adequate disk space.

Note

When running Perforce as a Windows service, use the **p4 set** command to set `P4DEBUG` as a registry variable. You can also set these trace flags when running **p4d.exe** as a server process from the command line.

Setting server debug levels on a Perforce server (**p4d**) has no effect on the debug level of a Perforce Proxy (**p4p**) process, and vice versa.

Higher levels of the Perforce server command tracing and tracking flags are typically recommended only for system administrators working with Perforce Technical Support to diagnose or investigate problems.

Command tracing

The server command trace flags and their meanings are as follows.

Trace flag	Meaning
`server=0`	Disable server command logging.
`server=1`	Logs server commands to the server log file. (As of release 2011.1, this is the default setting.)
`server=2`	In addition to data logged at level 1, logs server command completion and basic information on CPU time used. Time elapsed is reported in seconds. On UNIX, CPU usage (system and user time) is reported in milliseconds, as per `getrusage()`.
`server=3`	In addition to data logged at level 2, adds usage information for compute phases of **`p4 sync`** and **`p4 flush`** (**`p4 sync -k`**) commands.

For command tracing, output appears in the specified log file, showing the date, time, username, IP address, and command for each request processed by the server.

Performance tracking

The Perforce Server produces diagnostic output in the server log whenever user commands exceed certain predetermined thresholds of resource usage. Performance tracking is enabled by default, and if `P4DEBUG` is unset (or the tracking flag is not specified on the command line), the tracking level is computed based on the number of users in the license file.

Tracking flag	Meaning
`track=0`	Turn off tracking.
`track=1`	Track all commands.
`track=2`	Track excess usage for a server with less than 10 users.
`track=3`	Track excess usage for a server with less than 100 users.
`track=4`	Track excess usage for a server with less than 1000 users.
`track=5`	Track excess usage for a server with more than 1000 users.

The precise format of the tracking output is undocumented and subject to change.

Showing information about locked files

You can use the `-L` option of the **`p4 monitor`** to show information about locked files. The information is collected only for the duration of the **`p4 monitor`** command and is not persisted. See the description of the **`p4 monitor`** command for more information about how to set up this kind of monitoring.

The following sample output to the **`p4 monitor show -L`** command, shows the information displayed about locked files:

```
8764 R user 00:00:00 edit
      [server.locks/clients/88,d/ws4(W),db.locks(R),db.rev(R)]
8766 R user 00:00:00 edit
      [server.locks/clients/89,d/ws5(W),db.locks(R),db.rev(R)]
8768 R user 00:00:00 monitor
```

Following pid, status, owner, and time information, output shows two edit commands that have various files locked, including the client workspace lock in exclusive mode for the workspaces **ws4** and **ws5**, and **db.locks** and **db.rev** tables in read-only mode.

Auditing user file access

The Perforce Server is capable of logging individual file accesses to an audit logfile. Auditing is disabled by default, and is only enabled if **P4AUDIT** is set to point to the location of the audit log file, or if the server is started with the **-A** ***auditlog*** flag.

When auditing is enabled, the server adds a line to the audit log file every time file content is transferred from the server to the client. On an active server, the audit log file will grow very quickly.

Lines in the audit log appear in the form:

```
date time user@client clientIP command file#rev
```

For example:

```
$ tail -2 auditlog
2011/05/09 09:52:45 karl@nail 192.168.0.12 diff //depot/src/x.c#1
2011/05/09 09:54:13 jim@stone 127.0.0.1 sync //depot/inc/file.h#1
```

If a command is run on the machine that runs the Perforce Server, the ***clientIP*** is shown as **127.0.0.1**.

If you are auditing server activity in a replicated environment, each of your build farm or forwarding replica servers must have its own **P4AUDIT** log set.

Logging and structured log files

The Perforce Server can be configured to write log files in a structured (**.csv**) format. Structured log files contain more detail than conventional log files, are easier to parse, and the Perforce Server offers additional commands to help customize your site's logging configuration. This section summarizes the commands you use to manage logging, describes the use of structured logs, and explains how log files are rotated.

Note	All **p4d** error and info logs are in UTF8 for a server in unicode mode. You need an UTF8 console or editor to properly render this log information.

Logging commands

You can use the following commands to work with logs.

Command	Meaning
`p4 logappend`	If the user log is enabled, write an entry to `user.csv`
`p4 logparse`	Parse a structured log file and return the logged data in tagged format
`p4 logrotate`	Rotate a named logfile, or, if no name is specified, all server logs. This command applies only to structured logs; it does not rotate the unstructured `P4LOG` or `P4AUDIT` logs.
`p4 logstat`	Report the file size of the journal (`P4JOURNAL`, error log (`P4LOG`), audit log (`P4AUDIT`), or the named structured log file.
`p4 logtail`	Output the last block of the error log (`P4LOG`)
`p4 logschema`	Return a description of the specified log record type.

Enabling structured logging

To enable structured logging, set the **`serverlog.file.`***`N`* configurable to the name of the file. Valid names for structured log files and the information logged are shown in the following table. You can use a file path in conjunction with the file name.

Warning You must use one of the file names specified in the table. If you use an arbitrary name, no data will be logged to the file you specify.

Filename	Description
`all.csv`	All loggable events (commands, errors, audit, etc…)
`audit.csv`	Audit events (audit, purge)
`auth.csv`	The results of **`p4 login`** attempts. If the login failed, the reason for this is included in the log. Additional information provided by the authentication method is also included.
`commands.csv`	Command events (command start, compute, and end)
`errors.csv`	Error events (errors-failed, errors-fatal)
`events.csv`	Server events (startup, shutdown, checkpoint, journal rotation, etc.)
`integrity.csv`	Major events that occur during replica integrity checking.
`route.csv`	Log the full network route of authenticated client connections. Errors related to `net.mimcheck` are also logged against the related hop.

Filename	Description
`track.csv`	Command tracking (track-usage, track-rpc, track-db)
`user.csv`	User events; one record every time a user runs **`p4 logappend`**.

Files do not have to be set in consecutive order; for example, this is fine:

```
$ p4 configure set serverlog.file.1=audit.csv
$ p4 configure set serverlog.file.2=auth.csv
$ p4 configure set serverlog.file.4=track.csv
```

Note

Enabling all structured logging files can consume considerable diskspace. See "Structured logfile rotation" on page 127 for information on how to manage the size of the log file and the number of log rotations.

The value you specify for *`N`* may not exceed 500.

Structured logfile rotation

Each of the configured `serverlog.file.N` files has its own corresponding `serverlog.maxmb.N` and `serverlog.retain.N` configurables. For each configured server log type, these configurables control the maximum size (in megabytes) of the logfile before rotation, and the number of rotated server logs retained by the server.

Structured log files are automatically rotated on checkpoint, journal creation, overflow of associated `serverlog.maxmb.N` limit (if configured), and the **`p4 logrotate`** command. You can disable log rotation after journal rotation by setting the configurable `dm.rotatelogwithjnl` to `0`. Disabling this behavior can help when you're doing frequent journal rotations and you want the log rotated on a different schedule.

You can use the `serverlog.counter.N` configurable to create a counter that tracks the number of times a structured log file has been rotated. For example, the following command creates a rotation counter called `myErrorsCount`:

```
$ p4 configure set serverlog.counter.3=myErrorsCount
```

Each time the `errors.csv` log file is rotated, the counter is increased by one. In addition, the name of the log file is changed to specify the pre-incremented counter value. That is, if the counter `myErrors` is 7, the `errors.csv` file is named `errors-6.csv`.

You can create a counter for each file described in the preceding table. Do not use system reserved counter names for your counter: `change`, `maxCommitChange`, `job`, `journal`, `traits`, `upgrade`.

The **`p4 logtail`** command returns the current value of the counter when you logtail that log. It also returns the current size of the log at the end of the output (along with the ending offset in the log). The size and offset are identical if **`p4 logtail`** reads to the end of the log. Security monitoring tools can

use counters and the **p4 logtail** command in the process of scanning log files to monitor suspicious activity.

Chapter 8

Managing the Server and Its Resources

This chapter describes common management, maintenance, and troubleshooting tasks:

- Managing the sharing of code
- Managing distributed development
- Managing users
- Managing changelists
- Backing up a workspace
- Managing disk space
- Managing processes
- Scripted client deployment
- Troubleshooting Windows installations

These are all tasks that go beyond the initial configuration of the server.

Forcing operations with the -f flag

Certain commands support the `-f` flag, which enables Perforce administrators and superusers to force certain operations unavailable to ordinary users. Perforce administrators can use this flag with **`p4 branch`**, **`p4 change`**, **`p4 client`**, **`p4 job`**, **`p4 label`**, and **`p4 unlock`**. Perforce superusers can also use it to override the **`p4 user`** command. The usages and meanings of this flag are as follows.

Command	Syntax	Function
`p4 branch`	**`p4 branch -f branchname`**	Allows the modification date to be changed while editing the branch mapping
	`p4 branch -f -d branchname`	Deletes the branch, ignoring ownership
`p4 change`	**`p4 change -f [`*`changelist#`*`]`**	Allows the modification date to be changed while editing the changelist specification
	`p4 change -f changelist#`	Allows the description field and username in a committed changelist to be edited
	`p4 change -f -d changelist#`	Deletes empty, committed changelists
`p4 client`	**`p4 client -f clientname`**	Allows the modification date to be changed while editing the client specification
	`p4 client -f -d clientname`	Deletes the client, ignoring ownership, even if the client has opened files

Command	Syntax	Function
p4 job	**p4 job -f [*jobname*]**	Allows the manual update of read-only fields
p4 label	**p4 label -f labelname**	Allows the modification date to be changed while editing the label specification
	p4 label -f -d labelname	Deletes the label, ignoring ownership
p4 unlock	**p4 unlock -c changelist -f file**	Releases a lock (set with **p4 lock**) on an open file in a pending numbered changelist, ignoring ownership
p4 user	**p4 user -f username**	Allows the update of all fields, ignoring ownership This command requires **super** access.
	p4 user -f -d username	Deletes the user, ignoring ownership This command requires **super** access.

Managing the sharing of code

Users have three options in how they share code:

- **Using distributed development**

  This method allows users to share code and development. Using this option, users connect to a shared server and use the p4 push and p4 fetch commands to copy files to and from the shared server. Integration with the shared server is bi-directional and both file contents and history is shared. See "Distributed development using Fetch and Push" on page 131 for more information about this option.

- **Using the p4 zip and p4 unzip commands**

  This option allows users to share code. In addition to file contents, users can see the associated changelists, fixes, file attributes and integration history. See "Code drops without connectivity" on page 133 for additional information about this option.

- **Using remote depots**

  This option enables independent organizations with separate Perforce installations to integrate changes between installations. Code integration is only one way, and metadata information cannot be accessed. This option allows code drops to expose only files and file content. This might be preferable for security reasons.

  For additional information about this option, see Chapter 4, "Working with Depots" on page 53.

Managing distributed development

This section explains the work you need to do to support code sharing between distributed sites. This functionality is similar to using remote depots to do code drops, except that you can move file history in addition to files.

Distributed development using Fetch and Push

The following sections describe how you use the **`p4 fetch`** and **`p4 push`** commands to share code easily between distributed sites.

Consider the scenario described below.

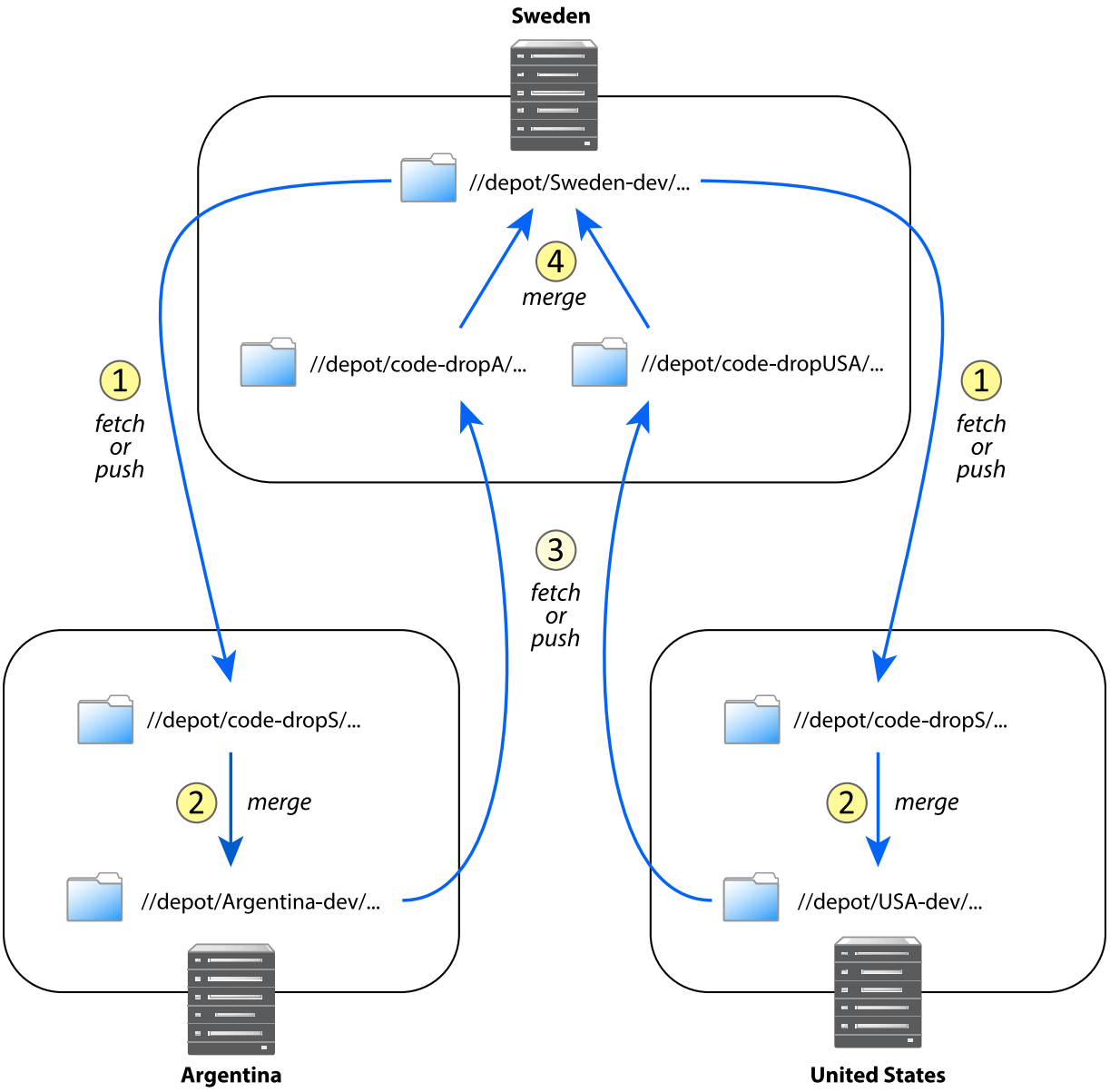
The gaming company Ukko Productions has offices in Sweden, Argentina and the United States. Each site is responsible for a different part of the gaming code; each does development on the section of code or "component" for which it is responsible. This work happens on the office's Perforce server, in a directory of the Perforce depot called **`dev`**. **`dev`** will contain locally submitted changes.

Let's suppose Sweden is working on a widget which is used by the developers in Argentina and the United States. First, Sweden makes the widget code available to Argentina and the United States by dropping the code — using the **`p4 push`** into drop directories on the servers in Argentina and the United States (see "1" in the figure below). (Alternatively, the Argentina and United States developers could use the **`p4 fetch`** to copy Sweden's code into their drop directories.) The Argentina and United States development teams can then merge the Sweden widget code into their respective **`dev`** directories using **`p4 merge`** (See "2" in the figure below). They can then customize the widget for their own purposes, without sharing these customizations with the Sweden developers.

If developers in the US and Argentina have a subset of changes they do want to share with Sweden, they use **`p4 push`** to copy this code into a special drop location on the Sweden server — one location for Argentina and one for the United States. (See "3" in the figure below). (Alternatively, Sweden could use the **`p4 fetch`** to obtain the code and drop it into the appropriate locations.) The Sweden developers can then merge the Argentina and United States code into their **`dev`** directory using **`p4 merge`** (See "4" in the figure below).

Then the cycle repeats.

This scenario is illustrated in the following drawing:



The next section explains how you must define remote specs to be able to implement this scenario.

Configuring the remote specifications

In order for the **`p4 push`** and **`p4 fetch`** commands to work properly, each of the three servers — Argentina's, the United States' and Sweden's — must have properly configured remote specifications. Remote specifications determine which remote servers a local server can fetch from or push to and which files will be fetched and pushed. (For more information about remotes and remote specifications, see the "Understanding Remotes" section of "Using Perforce for Distributed Versioning.")

Because the Argentina developers are fetching from or pushing to Sweden's server, their server's remote spec would look as follows:

```
RemoteID: ServerSweden
Address: ServerSweden:1666
DepotMap:
  //depot/code-dropA/... //depot/Sweden-dev/...
  //depot/Argentina-dev/... //depot/code-dropS/...
```

Because the United States developers are fetching from or pushing to Sweden's server, their server's remote spec would look as follows:

```
RemoteID: ServerSweden
Address: ServerSweden:1666
DepotMap:
  //depot/code-dropUSA/... //depot/Sweden-dev/...
  //depot/USA-dev/... //depot/code-dropS/...
```

Because the Sweden developers are fetching from or pushing to Argentina, their server's remote spec would look as follows:

```
RemoteID: ServerArgentina
Address: ServerArgentina:1666
DepotMap:
  //depot/code-dropS/... //depot/Argentina-dev/...
  //depot/Sweden-dev/... //depot/code-dropA/...
```

Because the Sweden developers are also fetching from or pushing to the United States, their server would have a second remote spec that would look as follows:

```
RemoteID: ServerUnitedStates
Address: ServerUnitedStates:1666
DepotMap:
  //depot/code-dropS/... //depot/USA-dev/...
  //depot/Sweden-dev/... //depot/code-dropUSA/...
```

Code drops without connectivity

Perforce Server provides a pair of commands that enable you to move files and their associated change history between servers when there is no connectivity between the servers; they are **`p4 zip`** and its companion command **`p4 unzip`**.

The **`p4 zip`** takes the specified list of files and the changelists which submitted those files and writes them to the specified zip file. It lets you bundle up any depot path from a server — from a subset to all the files on the server — into a zip file. You can also bundle by changelist number, capturing any number of changes through history.

You can then use the **`p4 unzip`** to unzip the content of the zip file into any Perforce server or servers.

Managing users

This section describes the three types of Perforce users and explains how you can create users, add new licensed users, rename users, delete users, and manage the files of deleted users.

For information about authenticating users and granting them access, please see Chapter 5, "Securing the Server" on page 65.

User types

There are three types of Perforce users: `standard` users, `operator` users, and `service` users.

- A `standard` user is a traditional user of Perforce.

  Standard users are the default, and each standard user consumes one Perforce license.

- An `operator` user is intended for human or automated system administrators.

  An `operator` user does not require a Perforce license.

- A `service` user is used for server-to-server authentication, whether in the context of remote depots (see "Remote depots and distributed development" on page 58) or in distributed environments (see *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.)

  Service users do not require licenses, but are restricted to automated inter-server communication processes in replicated and multi-server environments.

The following sections describe these types and how they need to be managed.

Important	Once you set the user type, you cannot change it.

Creating standard users

By default, Perforce creates a new user record in its database whenever a command is issued by a user who does not exist. Perforce superusers can also use the `-f` (force) flag to create a new user as follows:

```
$ p4 user -f username
```

Fill in the form fields with the information for the user you want to create.

The **`p4 user`** command also has an option (`-i`) to take its input from the standard input instead of the forms editor. To quickly create a large number of users, write a script that reads user data, generates output in the format used by the **`p4 user`** form, and then pipes each generated form to **`p4 user -i -f`**.

Service users

Creating a `service` user for each Perforce service you install can simplify the task of interpreting your server logs, and also improve security by requiring that any remote Perforce services with which yours is configured to communicate have valid login tickets for your installation. Service users do not consume Perforce licenses.

A service user can run only the following commands:

p4 dbschema	**p4 export**	**p4 info**	**p4 login**
p4 logout	**p4 logparse**	**p4 logschema**	**p4 logstat**
p4 logtail	**p4 passwd**	**p4 servers**	**p4 user**

To create a service user, run the command:

```
$ p4 user -f service1
```

The standard user form is displayed. Enter a new line to set the new user's `Type:` to be `service`; for example:

```
User:      service1
Email:     services@example.com
FullName:  Service User for remote depots
Type:      service
```

By default, the output of **`p4 users`** omits service users. To include service users, run **`p4 users -a`**.

Tickets and timeouts for service users

A newly-created service user that is not a member of any groups is subject to the default ticket timeout of 12 hours. To avoid issues that arise when a service user's ticket ceases to be valid, create a group for your service users that features an extremely long timeout, or set the value to `unlimited`. On the master server, issue the following command:

```
$ p4 group service_users
```

Add `service1` to the list of `Users:` in the group, and set the `Timeout:` and `PasswordTimeout:` values to a large value or to `unlimited`.

```
Group:            service_users
Timeout:          unlimited
PasswordTimeout:  unlimited
Subgroups:
Owners:
Users:
        service1
```

Permissions for service users

On your server, use **`p4 protect`** to grant the service user `super` permission. Service users are tightly restricted in the commands they can run, so granting them `super` permission is safe. If you are

only using the service user for remote depots and code drops, you may further reduce this user's permissions as described in "Restricting access to remote depots" on page 61.

Operator users

Organizations whose system administrators do not use Perforce's versioning capabilities might be able to economize on licensing costs by using the `operator` user type.

The `operator` user type is intended for system administrators who, even though they have `super` or `admin` privileges, are responsible for the maintenance of the Perforce Server, rather than the development of software or other assets on the server.

An `operator` user does not require a Perforce license, and can run only the following commands:

`p4 admin stop`	**`p4 admin restart`**	**`p4 admin checkpoint`**
`p4 admin journal`	**`p4 dbstat`**	**`p4 dbverify`**
`p4 depots`	**`p4 diskspace`**	**`p4 configure`**
`p4 counter` (including `-f`)	**`p4 counters`**	**`p4 info`**
`p4 journaldbchecksums`	**`p4 jobs`** (including `-R`)	**`p4 login`**
`p4 logout`	**`p4 logappend`**	**`p4 logparse`**
`p4 logrotate`	**`p4 logschema`**	**`p4 logstat`**
`p4 logtail`	**`p4 lockstat`**	**`p4 monitor`**
`p4 passwd`	**`p4 ping`**	**`p4 serverid`**
`p4 verify`	**`p4 user`**	

Preventing automatic creation of users

By default, Perforce creates a new user record in its database whenever a user invokes any command that can update the depot or its metadata. You can control this behavior by setting the `dm.user.noautocreate` configurable with the **`p4 configure`** command:

Value	Meaning
`0`	A user record is created whenever any new user invokes a command that updates the depot or its metadata (default).
`1`	New users must create their own user records by explicitly running **`p4 user`**.
`2`	Only the Perforce superuser can create a new user with **`p4 user`**.

For example:

```
$ p4 configure set dm.user.noautocreate=1
```

changes the server's behavior to require that new users first create their own accounts before attempting to modify data on the server.

Adding new licensed users

Perforce licenses are controlled by a text file called `license`. This file resides in the server root directory.

To add or update a license file, stop the Perforce Server, copy the `license` file into the server root directory, and restart the Perforce Server.

You can update an existing `license` without shutting down the Perforce Server, use **`p4 license -i`** to read the new license file from the standard input.

Most new license files obtained from Perforce can be installed with **`p4 license`**, except for when the server IP address has changed. If the server IP address has changed, or if no `license` file currently exists, restart the Server with **`p4 admin restart`**.

Renaming users

You can use the **`p4 renameuser`** command to rename users. The command renames the user and modifies associated artifacts to reflect the change: the user record, groups that include the user, properties that apply to the user, and so on. For detailed information see the description of the **`p4 renameuser`** command in the *P4 Command Reference*. In general, the user name is not changed in descriptive text fields such as change descriptions. It is only changed where the name appears as the owner or user field of the database record.

For best results, follow these guidelines:

- Before you use this command, check to see that the new user name does not already exist. Using an existing name might result in the merging of data for the existing and the renamed user despite the best efforts of the system to prevent such merges.
- The user issuing this command should not be the user being renamed.
- The user being renamed should not be using the server when this command executes. After the command completes, the user should log out and then log back in.
- The **`p4 renameuser`** command does not process unloaded workspaces: all the user's workspaces should be reloaded (or deleted) first.

  A distributed installation might contain local workspaces or local labels owned by the user; these workspaces and labels, which are bound to Edge Servers, should be deleted or moved to the Commit Server first.
- Files of type +k which contain the **`$Author$`** tag that were submitted by the user will have incorrect digests following this command. Use **`p4 verify -v`** to recompute the digest value after the rename.

Deleting obsolete users

Each standard user on the system consumes one Perforce license. A Perforce administrator can free up licenses by deleting users with the following command:

```
$ p4 user -d -f username
```

Before you delete a user, you must first revert (or submit) any files a user has open in a changelist. If you attempt to delete a user with open files, Perforce displays an error message to that effect.

Deleting a user frees a Perforce license but does not automatically update the group and protections tables. Use **`p4 group`** and **`p4 protect`** to delete the user from these tables.

Reverting files left open by obsolete users

If files have been left open by a nonexistent or obsolete user (for instance, a departing employee), a Perforce administrator can revert the files by deleting the client workspace specification in which the files were opened.

As an example, if the output of **`p4 opened`** includes:

```
//depot/main/code/file.c#8 - edit default change (txt) by jim@stlouis
```

you can delete the **`stlouis`** client workspace specification with:

```
$ p4 client -d -f stlouis
```

Deleting a client workspace specification automatically reverts all files opened in that workspace, deletes pending changelists associated with the workspace, and any pending fix records associated with the workspace. Deleting a client workspace specification does *not* affect any files in the workspace actually used by the workspace's owner; the files can still be accessed by other employees.

Deleting changelists and editing changelist descriptions

Perforce administrators can use the **`-f`** (force) flag with **`p4 change`** to change the description, date, or user name of a submitted changelist. The syntax is **`p4 change -f changenumber`**. This command presents the standard changelist form, but also enables superusers to edit the changelist's time, description, date, and associated user name.

You can also use the **`-f`** flag to delete any submitted changelists that have been emptied of files with **`p4 obliterate`**. The full syntax is **`p4 change -d -f changenumber`**.

Example 8.1. Updating changelist 123 and deleting changelist 124

Use **`p4 change`** with the **`-f`** (force) flag:

```
$ p4 change -f 123
$ p4 change -d -f 124
```

The `User:` and `Description:` fields for change `123` are edited, and change `124` is deleted.

Managing shelves

It's a good idea to check periodically for stale or abandoned shelves. Based on the last time a shelf was accessed, you might decide to delete the shelf.

The command **`p4 -Ztag change -o`** displays, in addition to other information, the access time for shelved files. You can use this information to determine if a shelved file has been abandoned and needs to be removed.

```
p4 -Ztag change -o 38
... Change 38
... Date 2015/10/01 16:54:47
... Client edge-one
... User markm
... Status pending
... Description shelve file

... Files0 //depot/new/code/dma/dmajob.cc
... Type public
... extraTag0 IsPromoted
... extraTagType0 int
... IsPromoted 1
... extraTag1 shelveAccess
... extraTagType1 date
... shelveAccess 2015/10/08 15:53:12
```

Note

When a shelf is viewed or modified, its access time is updated if its last access time was longer than the limit specified by the value of `dm.shelve.accessupdate`

Backing up a workspace

You can use the `-o` flag to the **`p4 unload`** command to unload a client, label, or task stream to a flat file on the client rather than to a file in the unload depot. This can be useful for seeding a client into another database or for creating a private backup of the client. The flat file uses standard journal format. The client, label, or task stream remains fully loaded after the command is run.

Managing disk space

You can manage disk space by minimizing the amount of space taken up by journal files and checkpoints and by relocating files. The following sections describe the strategies available for minimizing disk space use.

Diskspace Requirements

By default, the Perforce Server rejects commands when free space on the filesystems housing the `P4ROOT`, `P4JOURNAL`, `P4LOG`, or `TEMP` fall below 10 megabytes. To change this behavior, set the `filesys.P4ROOT.min` (and corresponding) configurables to your desired limits:

Configurable	Default Value	Meaning
`filesys.P4ROOT.min`	10M	Minimum diskspace required on server root filesystem before server rejects commands.
`filesys.P4JOURNAL.min`	10M	Minimum diskspace required on server journal filesystem before server rejects commands.
`filesys.P4LOG.min`	10M	Minimum diskspace required on server log filesystem before server rejects commands.
`filesys.TEMP.min`	10M	Minimum diskspace required for temporary operations before server rejects commands.
`filesys.depot.min`	10M	Minimum diskspace required for any depot before server rejects commands. (If there is less than `filesys.depot.min` diskspace available for any one depot, commands are rejected for transactions involving all depots.)

You can use the following abbreviations to specify size:

`t` or `T` for tebibytes
`g` or `G` for gibibytes
`m` or `M` for mebibytes
`k` or `K` for kibibytes

You can also use a percentage to specify the relative amount of free diskspace required. For example, setting `filesys.P4JOURNAL.min` to 5% means that at least 5% of total diskspace must be free for the server to continue to accept commands.

Saving disk space

All of Perforce's versioned files reside in subdirectories beneath the server root, as do the database files, and (by default) the checkpoints and journals. If you are running low on disk space, consider the following approaches to limit disk space usage:

- Configure Perforce to store the journal file on a separate physical disk. Use the `P4JOURNAL` environment variable or **`p4d -J`** to specify the location of the journal file.
- Keep the journal file short by taking checkpoints on a daily basis.
- Compress checkpoints, or use the `-z` option to tell **`p4d`** to compress checkpoints on the fly.

- Use the -jc ***prefix*** option with the **p4d** command to write the checkpoint to a different disk. Alternately, use the default checkpoint files, but back up your checkpoints to a different drive and then delete the copied checkpoints from the root directory. Moving checkpoints to separate drives is good practice not only in terms of diskspace, but also because old checkpoints are needed when recovering from a hardware failure, and if your checkpoint and journal files reside on the same disk as your depot, a hardware failure could leave you without the ability to restore your database.
- On UNIX systems, you can relocate some or all of the depot directories to other disks by using symbolic links. If you use symbolic links to shift depot files to other volumes, create the links only after you stop the Perforce service.
- If your installation's database files have grown to more than 10 times the size of a checkpoint, you might be able to reduce the size of the files by re-creating them from a checkpoint. See "Checkpoints for database tree rebalancing" on page 167.
- Use the **p4 diskspace** and **p4 sizes** commands to monitor the amount of disk space currently consumed by your entire installation, or by selected portions of your installation. See "Monitoring disk space usage" on page 119.
- If you have large binary files that are no longer accessed frequently, consider creating an archive depot and using the **p4 archive** command to transfer these files to bulk, near-line, or off-line storage. See "Reclaiming disk space by archiving files" on page 141.

Reclaiming disk space by archiving files

Over time, a Perforce server accumulates many revisions of files from old projects that are no longer in active use. Because **p4 delete** merely marks files as deleted in their head revisions, it cannot be used to free up disk space on the server.

Archive depots are a solution to this problem. You use archive depots to move infrequently-accessed files to bulk storage. To create one, mount a suitable filesystem, and use the **p4 archive** (and related **p4 restore**) commands to populate an archive depot located on this storage.

Note	Archive depots are *not* a backup mechanism. Archive depots are merely a means by which you can free up diskspace by reallocating infrequently-accessed files to bulk storage, as opposed to **p4 obliterate**, which removes file data and history.

Archiving is restricted to files that meet all of the following criteria:

- By default, files must be stored in full (`+F`) or compressed (`+C`) format. To archive text files (or other files stored as deltas), use **p4 archive -t**, but be aware that the archiving of RCS deltas is computationally expensive.
- Files must not be copied or branched from other revisions
- Files must not be copied or branched to other revisions
- Files must already exist in a local depot.

To create an archive depot and archive files to it:

1. Create a new depot with **p4 depot** and set the depot's **Type:** to **archive**. Set the archive depot's **Map:** to point to a filesystem for near-line or detachable storage.
2. Mount the volume to which the archive depot is to store its files.
3. Use **p4 archive** to transfer the files from a local depot to the archive depot.
4. (Optionally), unmount the volume to which the archive files were written.

Disk space is freed up on the (presumably high-performance) storage used for your local depot, and users can no longer access the contents of the archived files, but all file history is preserved.

To restore files from an archive depot:

1. Mount the volume on which the archive depot's files are stored.
2. Use the **p4 verify -A** command to verify files before you restore them.
3. Use **p4 restore** to transfer the files from the archive depot to a local depot.
4. (Optionally), unmount the volume to which the archive files were restored.

To purge data from an archive depot

1. Mount the volume on which the archive depot's files are stored.
2. Use **p4 archive -p** to purge the archives of the specified files in the archive depot.

   On completion, the action for affected revisions is set to **purge**, and the purged revisions can no longer be restored. The data is permanently lost.
3. (Optionally), unmount the volume from which the archive files were purged.

Reclaiming disk space by obliterating files

The purpose of a version management system is to enable your organization to maintain a history of what operations were performed on which files. The **p4 obliterate** command defeats this purpose; as such, it is intended only to be used to remove files that never belonged in the depot in the first place, and not as part of a normal software development process. Consider using **p4 archive** and **p4 restore** instead.

Note also that **p4 obliterate** is computationally expensive; obliterating files requires that the entire body of metadata be scanned per file argument. Avoid using **p4 obliterate** during peak usage periods.

Warning	Use **p4 obliterate** with caution. This is the one of only two commands in Perforce that actually remove file data. (The other command that removes file data is the archive-purging option for **p4 archive**)

Occasionally, users accidentally add files (or entire directory trees) to the wrong areas of the depot by means of an inadvertent branch or submit. There may also be situations that require that projects not

only be removed from a depot, but the history of development work be removed with it. These are the situations in which **p4 obliterate** can be useful.

Perforce administrators can use **p4 obliterate filename** to remove all traces of a file from a depot, making the file indistinguishable from one that never existed.

Warning

Do not use operating system commands (**erase**, **rm**, and their equivalents) to remove files from the Perforce server root by hand.

By default, **p4 obliterate filename** does nothing; it merely reports on what it would do. To actually destroy the files, use **p4 obliterate -y filename**.

To destroy only one revision of a file, specify only the desired revision number on the command line. For instance, to destroy revision 5 of a file, use:

```
$ p4 obliterate -y file#5
```

Revision ranges are also acceptable. To destroy revisions 5 through 7 of a file, use:

```
$ p4 obliterate -y file#5,7
```

Warning

If you intend to obliterate a revision range, be certain you've specified it properly. If you fail to specify a revision range, **all** revisions of the file are obliterated.

The safest way to use **p4 obliterate** is to use it **without** the **-y** flag until you are certain the files and revisions are correctly specified.

Managing processes

The following sections describe the circumstances under which you might want to pause or terminate a process, and explain why you might need to do some clean-up work after a process has terminated.

Pausing, resuming, and terminating processes

To pause and resume long-running processes (such as **p4 verify** or **p4 pull**), a Perforce superuser can use the commands **p4 monitor pause** and **p4 monitor resume**. If a process on a Perforce Server consumes excessive resources, it can also be marked for termination with **p4 monitor terminate**.

Once marked for termination, the process is terminated by the Perforce server within 50,000 scan rows or lines of output. Only processes that have been running for at least ten seconds can be marked for termination.

Users of terminated processes are notified with the following message:

```
Command has been canceled, terminating request
```

Processes that involve the use of interactive forms (such as **p4 job** or **p4 user**) can also be marked for termination, but data entered by the user into the form is preserved. Some commands, such as **p4 obliterate**, cannot be terminated.

Clearing entries in the process table

Under some circumstances (for example, a Windows machine is rebooted while certain Perforce commands are running), entries may remain in the process table even after the process has terminated.

Perforce superusers can remove these erroneous entries from the process table altogether with **p4 monitor clear *dip***, where *dip* is the erroneous process ID. To clear all processes from the table (running or not), use **p4 monitor clear all**.

Running processes removed from the process table with **p4 monitor clear** continue to run to completion.

Managing the database tables

Use the **p4 dbstat** command to display statistics on the internal state of the Perforce database. For example,

```
$ p4 dbstat -a
```

You can also specify the name of a database file in your server's root directory. This command is typically used in conjunction with Perforce technical support to estimate disk seeks due to sequential database scans.

Options allow you to display the following:

- statistics for all tables
- a page count, free pages, and percent free data for the specified table
- a histogram showing distances between leaf pages
- a report on the file sizes of database tables

Warning

Because **p4 dbstat** blocks write access to the database while it scans the tables, use this command with care. You will most often use this command when working with Perforce technical support.

Scripted client deployment on Windows

The Perforce installer supports scripted installation, enabling you to accelerate a deployment of Perforce across a large number of desktops.

Scripted installations are controlled by a configuration file that comes with the scrip table version of the Perforce installer. You can edit this file to preconfigure Perforce environment variables (such as

`P4PORT`) for your environment, to automatically select Perforce applications in use at your site, and more.

To learn more about how to automate a deployment of Perforce, see "Automated Deployment of Perforce" in the Perforce knowledge base:

http://answers.perforce.com/articles/KB_Article/Automated-Deployment-of-Perforce

Perforce technical support personnel are available to answer any questions or concerns you have about automating your Perforce deployment.

Troubleshooting Windows installations

The following sections explain how you might resolve some Windows-related installation issues.

Resolving Windows-related instabilities

Many large sites run Perforce servers on Windows without incident. There are also sites in which a Perforce service or server installation appears to be unstable; the server dies mysteriously, the service can't be started, and in extreme cases, the system crashes. In most of these cases, this is an indication of recent changes to the machine or a corrupted operating system.

Though not all Perforce failures are caused by OS-level problems, a number of symptoms can indicate the OS is at fault. Examples include: the system crashing, the Perforce server exiting without any error in its log and without Windows indicating that the server crashed, or the Perforce service not starting properly.

In some cases, installing third-party software *after* installing a service pack can overwrite critical files installed by the service pack; reinstalling your most-recently installed service pack can often correct these problems. If you've installed another application after your last service pack, and server stability appears affected since the installation, consider reinstalling the service pack.

Resolving issues with P4EDITOR or P4DIFF

Your Windows users might experience difficulties using the Perforce Command-Line Client (**`p4.exe`**) if they use the `P4EDITOR` or `P4DIFF` environment variables.

The reason for this is that Perforce applications sometimes use the DOS shell (**`cmd.exe`**) to start programs such as user-specified editors or diff utilities. Unfortunately, when a Windows command is run (such as a GUI-based editor like **`notepad.exe`**) from the shell, the shell doesn't always wait for the command to complete before terminating. When this happens, the Perforce client then mistakenly behaves as if the command has finished and attempts to continue processing, often deleting the temporary files that the editor or diff utility had been using, leading to error messages about temporary files not being found, or other strange behavior.

You can get around this problem in two ways:

- Unset the environment variable `SHELL`. Perforce applications under Windows use **`cmd.exe`** only when `SHELL` is set; otherwise they call `spawn()` and wait for the Windows programs to complete.
- Set the `P4EDITOR` or `P4DIFF` variable to the name of a batch file whose contents are the command:

```
start /wait program %1 %2
```

where ***program*** is the name of the editor or diff utility you want to invoke. The **/wait** flag instructs the system to wait for the editor or diff utility to terminate, enabling the Perforce application to behave properly.

Some Windows editors (most notably, Wordpad) do not exhibit proper behavior, even when instructed to wait. There is presently no workaround for such programs.

Chapter 9

Tuning Perforce for Performance

Your Perforce server should normally be a light consumer of system resources. As your installation grows, however, you might want to revisit your system configuration to ensure that it is configured for optimal performance.

This chapter briefly outlines some of the factors that can affect the performance of a Perforce server, provides a few tips on diagnosing network-related difficulties, and offers some suggestions on decreasing server load for larger installations:

- It describes the variables that affect performance: operating system, disk subsystem, file system, CPU, memory, network connectivity settings, journal and archive location, use patterns, the use of read-only clients, and parallel processing for submits and syncs.
- It explains how you can improve performance with lockless reads.
- It explains how you can diagnose slow response times.
- It describes the factors that create server swamp.
- It explains how you can improve performance by rebalancing B-trees.

Tuning for performance

In general, Perforce performs well on any server-class hardware platform. The following variables can affect the performance of your Perforce server.

Operating systems

32-bit operating systems might not be able to address large amounts of physical memory, which can restrict the effective size of the filesystem cache. The various 64-bit operating systems each have their own performance characteristics that can favor a particular Perforce workload. In general, Linux distributions using later Linux 2.6 64-bit kernels have good performance characteristics for most Perforce workloads.

Disk subsystem

For I/O requests that must be satisfied from beyond the filesystem cache, there might be several improvements possible for the I/O subsystem. The storage subsystem containing the `db.*` files should have a memory cache; maximizing the storage subsystem's memory cache is also a good recommendation. For best performance, write-back caching should be enabled, which of course requires that the storage subsystem's memory have battery backup power. I/O latency to the logical drive where the `db.\*` files are located should be minimized, including the rotational latency of the physical drives themselves. Minimizing I/O latency might require direct connections between the host and the storage subsystem, and usually requires physical drives with the fastest rotational speed (such as 15K RPM).

RAID 1+0 (or RAID 10) is usually the better performing RAID configuration, and is recommended for the logical drive where the `db.*` files are located. The number of physical drives in the logical drive can also have an affect on *`p4d`* performance. Generally, performance improves as the number of physical drives in the logical drive increases. For a given amount of disk space required, better performance

might result from using more smaller-capacity physical drives. The stripe size for the logical drive can also affect performance; the optimal stripe size might be dependent upon the number of physical drives in the logical drive.

Hardware-based RAID implementations (that is, RAID logic that is not implemented as software running on the host) usually have good performance characteristics. Software-based RAID implementations can require CPU cycles that might otherwise be needed for **p4d** processes. Therefore, software-based RAID implementations should be avoided.

File systems

Filesystem performance is an important component of operating system performance. The various operating systems usually offer several filesystems, each with their own performance characteristics that can favor a particular Perforce workload. For best **p4d** performance, the **db.*** files should be located on a high-performance filesystem. In general, the XFS filesystem has good performance characteristics for most Perforce workloads. The XFS filesystem is available on several operating systems, including Linux distributions using later Linux 2.6 64-bit kernels.

Reading pages into a cache in anticipation of being requested is an optimization that is often implemented within various I/O subsystem components. This optimization is commonly known as "read-ahead". In some implementations, read-ahead can be tuned, which might result in better performance. But tuning read-ahead can be a bit of an art. For example, increasing the read-ahead size might result in better performance for operations requiring mostly sequential reads. But the same increased read-ahead size applied consistently during random reads might unnecessarily discard previously-cached data that might have satisfied subsequent requests.

CPU

CPU resource consumption can be adversely affected by compression, lockless reads, or a badly designed protections table. In general, there is a trade-off between speed and the number of cores. A minimum of 2.4 GHZ and 8 cores is recommended. With greater speed, fewer cores will do: for example, a 3.2 GHZ and 4-core processor will also work.

Faster processors and memory in the machine where **p4d** executes might result in faster execution of **p4d** commands. Since portions of some commands acquire and hold resources that might block other commands, it is important that these portions of the commands execute as fast as possible. For example, most **p4d** commands have a compute phase, during which shared locks are acquired and held on some of the **db.*** files. A shared lock on a **db.*** file blocks an operation that writes to the same **db.*** file. If the data needed for a command's compute phase is cached within the operating system's filesystem cache, only the processor and memory speed constrains the compute phase.

If you are using lockless reads, CPU speed is not as critical, but can still be helpful for good performance. Since some readers will no longer block a writer (and a writer will no longer block some readers), speeding commands through the server might not be as critical from a concurrency point of view. And since more commands might now run concurrently through the Perforce Server, more CPU cores might be better utilized.

The complexity of the site's protections table and of client views can affect CPU requirements. You can monitor CPU utilization using OS utilities such as **top** (on Linux and Unix) and **perfmon** (on Windows). Installations with high CPU utilization on the machine where **p4d** executes that are already using faster

processors might need more processors and/or processors with more cores while maintaining the speed of the processors.

Note	If you are using SSL to secure client-server connections, choose a CPU that supports the AES instruction set. Perforce normally uses AES-256 to encrypt its SSL connections, so using a CPU that supports AES will minimize the encryption overhead: in most CPUs, it will eliminate the performance penalty.

Some processors and operating systems support dynamic frequency scaling, which allows the processor to vary power consumption by dynamically adjusting the processor voltage and core frequency. As more demand is placed on the processor, the voltage and core frequency increase. Until the processor is ramped up to full speed, **p4d** performance might be impacted. Although the power-saving capability of the dynamic frequency scaling feature is useful for mobile computers, it is not recommended for the machine where **p4d** executes.

Examples of dynamic frequency scaling include the following:

- Intel SpeedStep - available on some Xeon processors and generally available on mobile computers
- AMD PowerNow! - available on an array of AMD processors, including server-level processors

Both features are supported on Linux (and enabled by default in some SuSE distributions), Windows, and Mac OS X platforms. If this feature is enabled on the machine where **p4d** executes, we recommend disabling it. In some Linux distributions, such as SuSE, this feature can be disabled by setting the "powersaved" service to "off".

You might be able to determine the current speed of the processors on your computer. On Linux, the current speed of each core is reported on the "cpu MHz" line in the output from the **cat /proc/cpuinfo** OS command.

Memory

Server performance is highly dependent upon having sufficient memory. Two bottlenecks are relevant. The first bottleneck can be avoided by ensuring that the server doesn't page when it runs large queries, and the second by ensuring that the **db.rev** table (or at least as much of it as practical) can be cached in main memory:

- Determining memory requirements for large queries is fairly straightforward: the server requires about 1 kilobyte of RAM per file to avoid paging; 10,000 files will require 10 MB of RAM.
- To cache **db.rev**, the size of the **db.rev** file in an existing installation can be observed and used as an estimate. New installations of Perforce can expect **db.rev** to require about 150-200 bytes per revision, and roughly three revisions per file, or about 0.5 kilobytes of RAM per file.
- I/O requests that can be satisfied from a larger filesystem cache complete faster than requests that must be satisfied from beyond the filesystem cache.

Thus, if there is 1.5 kilobytes of RAM available per file, or 150 MB for 100,000 files, the server does not page, even when performing operations involving all files. It is still possible that multiple large operations can be performed simultaneously and thus require more memory to avoid paging. On the other hand, the vast majority of operations involve only a small subset of files.

One way to determine if you have allocated sufficient memory is to look at the physical read rate on the device that contains *only* the database files. This read rate should be trivial.

Network

Perforce can run over any TCP/IP network. For remote users or distributed configurations, Perforce offers options like proxies and the commit/edge architecture that can enhance performance over a WAN. Compression in the network layer can also help.

Perforce uses a TCP/IP connection for each client interaction with the server. The server's port address is defined by `P4PORT`, but the TCP/IP implementation picks a client port number. After the command completes and the connection is closed, the port is left in `TIME_WAIT` state for two minutes. Although the port number ranges from `1025` to `32767`, generally only a few hundred or thousand can be in use simultaneously. It is therefore possible to occupy all available ports by invoking a Perforce command many times in rapid succession, such as with a script.

By default, idle connections are not kept alive. If your network silently drops idle connections, this behavior may cause unexpected connectivity issues. (Consider a **`p4 pull`** thread that transfers data between a master server and a replica at a remote site; depending on each site's respective business hours and user workloads, such a connection could be idle for several hours per day.) Four configurables are available to manage the state of idle connections.

Configurable	Default Value	Meaning
`net.keepalive.disable`	`0`	If non-zero, disable the sending of TCP keepalive packets.
`net.keepalive.idle`	`0`	Idle time (in seconds) before starting to send keepalives.
`net.keepalive.interval`	`0`	Interval (in seconds) between sending keepalive packets.
`net.keepalive.count`	`0`	Number of unacknowledged keepalives before failure.

If your network configuration requires keepalive packets, consider setting `net.keepalive.idle` to a suitably long value, for example 3,600 seconds (1 hour), and an interval measured in tens of minutes.

Journal and archive location

For recoverability, the live journal should not be on the same physical device that contains the `db.*` files. Separating the live journal and the `db.\*` files also improves performance. During operations that write to the `db.*` files, entries are written to the live journal as records are written to the `db.\*` files. If the live journal and the `db.*` files are on the same physical device, the I/O throughput to the `db.\*` files is degraded. For best performance, the live journal should be on a separate storage subsystem connected to a separate host adapter. The live journal should be on a logical drive and filesystem that is optimized for sequential writes.

The versioned files should be located on a separate logical drive than the logical drives where the `db.*` files and the live journal are located. For best performance, the logical drive where the versioned files

are located should be on a separate storage subsystem connected to a separate host adapter. Since the versioned files typically require significantly more disk space and the I/O throughput is not as critical as for the `db.\*` files, a more economical RAID configuration, such as RAID 5, can be used for the logical drive where the versioned files are located.

Use patterns

Perforce usage can affect performance. There are several usage patterns that can have a direct effect on performance. Since the depot filenames are the leading portion of the key in several important `db.\*` files (`db.rev`, `db.revhx`, and `db.integed` are among the more notable), the length of paths in the depot filenames have a direct effect on performance. As the length of paths increase, performance decreases. It is therefore prudent to discourage the use of overly-descriptive paths in the depot filenames.

The development methodology can also have a direct effect on performance. If the development methodology calls for frequent creation of full branches (perhaps branching for each bug fix), then the amount of metadata rapidly increases, resulting in more levels within the `db.*` file B-trees. As the number of levels increase, more key comparisons and I/O requests are required to traverse to the leaf pages, which will impact performance. Creating full branches also requires more metadata read and written; the additional metadata read and written might affect the filesystem cache to the detriment of other Perforce tasks. Rather than frequent creation of full branches, it might be prudent to branch only those files needed for each bug fix, or consider a development methodology in which multiple bug fixes can occur on the same branch.

Using read-only clients in automated builds

Build automation scripts, which routinely create, sync, and tear down clients, may fragment the `db.have` table over time. To avoid this, you can specify the type `readonly` for these clients. Such clients cannot add, delete, edit, integrate, or submit files, but this should not be an issue in build scripts.

A readonly client is assigned its own personal `db.have` database table, and the location of this table is specified using the `client.readonly.dir` configurable.

To set up a read-only client:

1. Set the `client.readonly.dir` configurable to the directory where the `db.*` tables for the client should be stored.

   For example, if you create a read-only client whose name is `myroc` and you set `client.readonly.dir` to `/perforce/1`, then syncing files using this client will write to the following database

   ```
   /perforce/1/server.dbs/client/hashdir/db.myroc
   ```

2. Set the `Type` field of the client spec to `readonly`.

Using parallel processing for submits and syncs

You can configure the server to transfer files in parallel for submit and sync processing. Parallel processing is most effective with long-haul, high latency networks or with other network configuration that prevents the use of available bandwidth with a single TCP flow. Parallel processing might also

be appropriate when working with large compressed binary files, where the client must perform substantial work to decompress the file.

- Use the `net.parallel.max` configurable to transfer files in parallel during the submit process. For this feature to work, you must have both server and client upgraded to version 2015.1.
- Use the `net.parallel.max` configurable to speed up sync processing by having the **`p4 sync`** command transfer files using multiple threads. You do this by setting the `net.parallel.max` configuration variable to a value greater than one and by using the `--parallel` option to the **`p4 sync`** command.

For more information see the **`p4 submit`** command and the **`p4 sync`** command in *P4 Command Reference*.

Improving concurrency with lockless reads

Prior to Release 2013.3, commands that only read data from the database take a read-lock on one (or more) database tables. Although other commands can read from the tables at the same time, any commands attempting to write to the read-locked tables are forced to wait for the read-lock to complete before writing could begin. Currently, the default behavior is to allow some commands to perform lock-free reads (or "peeks") on these tables, without sacrificing consistency or isolation. This provides significant performance improvement by ensuring that write operations on these tables can run immediately, rather than being held until the read-lock is released.

Note

Lockless reads require that server locks be enabled. Because this can cause issues for long duration syncs, the default value for controlling the 'sync' server lock (`server.locks.sync`) is currently disabled by default.

`maxlocktime` has been changed when peeking is enabled. To revert to the old behavior, set the `dbpeeking.usemaxlock` configurable to `1`.

To change the setting of lockless reads on your Perforce Server, use the **`p4 configure set db.peeking=N`** command.

Any change to `db.peeking` requires a server restart to take effect.

Possible values for `db.peeking` are as follows:

`db.peeking`	Meaning
`0`	If `db.peeking` is unset or `0`, the old database locking order is used and lockless reads ("peeking") are disabled. This corresponds to the behavior of Perforce at release 2013.2 and below.
`1`	If `db.peeking` is set to `1`, the new database locking order is used, but peeking remains disabled. This configuration is intended primarily for diagnostic purposes.
`2` (default)	If `db.peeking` is set to `2`, the new database locking order is used and lockless reads ("peeking") are enabled.

db.peeking	Meaning
	This configuration is expected to provide the best performance results for most sites. It is the default value.
3	If db.peeking is set to 3, the new database locking order is used and lockless reads ("peeking") are enabled, but optimizations for the db.revhx and db.revdx tables are bypassed. This configuration involves a trade-off between concurrency and command completion speed; in general, if a repository has many revisions per file, then some commands will complete more slowly with db.peeking=3, but will no longer require read locks on the db.revhx and db.revdx tables. If read locks on these tables are in fact the bottleneck, overall performance may still be better with db.peeking=3. One guideline: if you have lots of history, use the default; if you have lots of single revision branch data, try db.peeking=3; if you max out cpu, go back to the default (2).

Commands implementing lockless reads

When peeking is enabled, the following commands run lockless:

Command	Notes
annotate	
branches	
changes	
clients	
counters	
depots	
describe	
diff	
diff2	
dir2	
filelog	
files	Applies to **files -a**
fixes	
fstat	when db.peeking=3

Command	Notes
`have`	
`interchanges`	
`integ`	
`integed`	
`istat`	
`jobs`	
`keys`	
`labels`	
`merge`	
`streams`	
`sizes`	Applies to **`sizes -a`**
`sync`	when `db.peeking=3`
`print`	Applies to **`print -a`**
`resolved`	
`users`	
`verify`	

The following commands run partially lockless; in most cases these commands will operate lock-free, but lockless operation is not guaranteed:

Command	Notes
`copy`	
`cstat`	
`fstat`	when `db.peeking=2`
`interchanges`	in the context of `copy` operations
`istat`	in the context of `copy` operations
`opened`	
`sync`	when `db.peeking=2`

Overriding the default locking behavior

You can override the `db.peeking` setting on a per-command basis by using the `-Zpeeking=` flag followed by your preferred value. For example, to disable peeking for one command, run the following command:

```
$ p4 -Zpeeking=1 fstat
```

and compare the results with:

```
$ p4 -Zpeeking=2 fstat
```

Observing the effect of lockless reads

To determine whether read locks are impacting performance (and the extent to which enabling lockless reads has improved performance), you can examine the server logs, or you can use the `-Ztrack` flag to output, for any given command, the lines that would be written to the `P4LOG`. For example:

```
$ p4 -Zpeeking=1 -Ztrack sync
```

produces output for 11 database tables. The relevant lines here are those that refer to "`locks read/write`".

```
...
--- db.counters
---   pages in+out+cached 3+0+2
---   locks read/write 1/0 rows get+pos+scan put+del 1+0+0 0+0
--- db.user
---   pages in+out+cached 3+0+2
---   locks read/write 1/0 rows get+pos+scan put+del 1+0+0 0+0
...
```

The `1` appearing in ("`locks read/write 1/0`") every table's locking results shows one read lock taken per table. By contrast, the diagnostic output from:

```
$ p4 -Zpeeking=2 -Ztrack sync
```

```
...
--- db.counters
---   pages in+out+cached 3+0+2
---   locks read/write 0/0 rows get+pos+scan put+del 1+0+0 0+0
...
```

shows that the sync operation completed without any read or write locks required on `db.counters` (if you try it yourself, on many other tables); when peeking is enabled, many commands will show `read/write 0/0` locks (or at least, fewer locks) taken.

Side-track servers must have the same db.peeking level

A single Perforce instance can detect and ignore inadvertent attempts to override `db.peeking` that would change table locking order and risk deadlock. (For example, if you attempt to use `db.peeking=3` on a server for which peeking is disabled by having `db.peeking` set to `0` (or unset), the service ignores the attempt altogether and the command proceeds with the old behavior.

In the case of "side-track servers" described in the following Knowledge Base article:

http://answers.perforce.com/articles/KB_Article/Setting-Up-a-Side-track-Server

this protection is not available.

Warning	All side-track servers must have the same `db.peeking` setting as the main server. Server deadlock may result.

Diagnosing slow response times

Perforce is normally a light user of network resources. Although it is possible that an extremely large user operation could cause the Perforce server to respond slowly, consistently slow responses to **`p4`** commands are usually caused by network problems. Any of the following can cause slow response times:

1. Misconfigured domain name system (DNS)
2. Misconfigured Windows networking
3. Difficulty accessing the **`p4`** executable on a networked file system

A good initial test is to run **`p4 info`**. If this does not respond immediately, then there is a network problem. Although solving network problems is beyond the scope of this manual, here are some suggestions for troubleshooting them.

Hostname vs. IP address

Try setting `P4PORT` to the service's IP address instead of its hostname. For example, instead of using:

```
P4PORT=host.domain:1666
```

try using:

```
P4PORT=1.2.3.4:1666
```

with your site-specific IP address and port number.

On most systems, you can determine the IP address of a host by invoking:

```
$ ping hostname
```

If **p4 info** responds immediately when you use the IP address, but not when you use the hostname, the problem is likely related to DNS.

Windows wildcards

In some cases, **p4** commands on Windows can result in a delayed response if they use unquoted file patterns with a combination of depot syntax and wildcards, such as:

```
$ p4 files //depot/*
```

You can prevent the delay by putting double quotes around the file pattern, like this:

```
$ p4 files "//depot/*"
```

The cause of the problem is the **p4** command's use of a Windows function to expand wildcards. When quotes are not used, the function interprets `//depot` as a networked computer path and spends time in a futile search for a machine named `depot`.

DNS lookups and the hosts file

On Windows, the `%SystemRoot%\system32\drivers\etc\hosts` file can be used to hardcode IP address-hostname pairs. You might be able to work around DNS problems by adding entries to this file. The corresponding UNIX file is `/etc/hosts`.

Location of the p4 executable

If none of the above diagnostic steps explains the sluggish response time, it's possible that the **p4** executable itself is on a networked file system that is performing very poorly. To check this, try running:

```
$ p4 -V
```

This merely prints out the version information, without attempting any network access. If you get a slow response, network access to the **p4** executable itself might be the problem. Copying or downloading a copy of **p4** onto a local filesystem should improve response times.

Working over unreliable networks

To set a hard upper bound on how long a connection is willing to wait on any single network read or write, set the `net.maxwait` configurable to the number of seconds to wait before disconnecting with a network error. Users working over unreliable connections can set `net.maxwait` value either in their

`P4CONFIG` files, or use `-vnet.maxwait=t` on a per-command basis, where *`t`* is the number of seconds to wait before timing out.

Note

Although `net.maxwait` can be set on the Perforce server, it is generally inadvisable to do so. For example, if `net.maxwait` is set to `60` on the server, users of the Command-Line Client must complete every interactive form within one minute before the command times out. If, however, individual users set `net.maxwait` in their own `P4CONFIG` files (which reside on their own workstations) their connections are not subject to this limitation; commands only fail if the versioning service takes more than 60 seconds to respond to their requests.

It is useful to combine `net.maxwait` with the `-rN` global option, where *`N`* is the number of times to attempt reconnection in the event that the network times out. For example:

```
$ p4 -r3 -vnet.maxwait=60 sync
```

attempts to sync the user's workspace, making up to three attempts to resume the sync if interrupted. The command fails after the third 60-second timeout.

Because the format of the output of a command that times out and is restarted cannot be guaranteed (for example, if network connectivity is broken in the middle of a line of output), avoid the use of `-r` on any command that reads from standard input. For example, the behavior of the following command, which reads a list of files from stdin and passes it to **`p4 add`**, can result in the attempted addition of "half a filename" to the depot.

```
$ find . -print | p4 -x - -r3 add
```

To prevent this from happening (for example, if adding a large number of files over a very unreliable connection), consider an approach like the following:

```
$ find directoryname -type f -exec p4 -r5 -vmax.netwait=60 add {} \;
```

All files (`-type f`) in *`directoryname`* are found, and added one at a time, by invoking the command "**`p4 -r5 -vmax.netwait=60 add`**" for each file individually.

After all files have been added, assign the changelist a changelist number with **`p4 change`**, and submit the numbered atomically with:

```
$ p4 -r5 -vmax.netwait=60 submit -c changenum
```

If connectivity is interrupted, the numbered changelist submission is resumed.

Preventing server swamp

Generally, Perforce's performance depends on the number of files a user tries to manipulate in a single command invocation, not on the size of the depot. That is, syncing a client view of 30 files from a

3,000,000-file depot should not be much slower than syncing a client view of 30 files from a 30-file depot.

The number of files affected by a single command is largely determined by the following factors:

- **`p4`** command-line arguments (or selected folders in the case of GUI operations)

  Without arguments, most commands operate on, or at least refer to, all files in the client workspace view.

- Client views, branch views, label views, and protections

  Because commands without arguments operate on all files in the workspace view, it follows that the use of unrestricted views and unlimited protections can result in commands operating on all files in the depot.

When the server answers a request, it locks down the database for the duration of the computation phase. For normal operations, this is a successful strategy, because the server can "get in and out" quickly enough to avoid a backlog of requests. Abnormally large requests, however, can take seconds, sometimes even minutes. If frustrated users press **CTRL**+**C** and retry, the problem gets even worse; the server consumes more memory and responds even more slowly.

Warning

The **p4 obliterate** command scans the entire database once per file argument and locks the entire database while scanning. It is best to do this during off hours for large sites.

At sites with very large depots, unrestricted views and unqualified commands make a Perforce server work much harder than it needs to. Users and administrators can ease load on their servers by the following:

- Using "tight" views
- Assigning protections
- Limiting `maxresults`
- Limiting simultaneous connections with `server.maxcommands`
- Unloading infrequently-used metadata
- Writing efficient scripts
- Using compression efficiently
- Other server configurables

The following sections examine each of these solutions:

Using tight views

The following "loose" view is trivial to set up but could invite trouble on a very large depot:

```
//depot/...       //workspace/...
```

In the loose view, the entire depot was mapped into the client workspace; for most users, this can be "tightened" considerably. The following view, for example, is restricted to specific areas of the depot:

```
//depot/main/srv/devA/...          //workspace/main/srv/devA/...
//depot/main/drv/lport/...         //workspace/main/dvr/lport/...
//depot/rel2.0/srv/devA/bin/...    //workspace/rel2.0/srv/devA/bin/...
//depot/qa/s6test/dvr/...          //workspace/qa/s6test/dvr/...
```

Client views, in particular, but also branch views and label views, should also be set up to give users just enough scope to do the work they need to do.

Client, branch, and label views are set by a Perforce administrator or by individual users with the **`p4 client`**, **`p4 branch`**, and **`p4 label`** commands, respectively.

Two of the techniques for script optimization (described in "Using branch views" on page 165 and "Using a temporary client workspace" on page 166) rely on similar techniques. By limiting the size of the view available to a command, fewer commands need to be run, and when run, the commands require fewer resources.

Assigning protections

Protections (see "Authorizing access" on page 83) are actually another type of Perforce view. Protections are set with the **`p4 protect`** command and control which depot files can be affected by commands run by users.

Unlike client, branch, and label views, however, the views used by protections can be set only by Perforce superusers. (Protections also control read and write permission to depot files, but the permission levels themselves have no impact on server performance.) By assigning protections in Perforce, a Perforce superuser can effectively limit the size of a user's view, even if the user is using "loose" client specifications.

Protections can be assigned to either users or groups. For example:

```
write    user     sam          *    //depot/admin/...
write    group    rocketdev    *    //depot/rocket/main/...
write    group    rocketrel2   *    //depot/rocket/rel2.0/...
```

Perforce groups are created by superusers with the **`p4 group`** command. Not only do they make it easier to assign protections, they also provide useful fail-safe mechanisms in the form of `maxresults` and `maxscanrows`, described in the next section.

Limiting database queries

Each Perforce group has an associated *maxresults*, *maxscanrows*, and *maxlocktime* value. The default for each is `unset`, but a superuser can use **`p4 group`** to limit it for any given group.

`MaxResults` prevents the server from using excessive memory by limiting the amount of data buffered during command execution. Users in limited groups are unable to run any commands that buffer more database rows than the group's `MaxResults` limit. (For most sites, `MaxResults` should be larger than the largest number of files anticipated in any one user's individual client workspace.)

Like `MaxResults`, `MaxScanRows` prevents certain user commands from placing excessive demands on the server. (Typically, the number of rows scanned in a single operation is roughly equal to `MaxResults` multiplied by the average number of revisions per file in the depot.)

Finally, `MaxLockTime` is used to prevent certain commands from locking the database for prolonged periods of time. Set `MaxLockTime` to the number of milliseconds for the longest permissible database lock.

To set these limits, fill in the appropriate fields in the **`p4 group`** form. If a user is listed in multiple groups, the *highest* of the `MaxResults` (or `MaxScanRows`, or `MaxLockTime`) limits (including `unlimited`, but *not* including the default `unset` setting) for those groups is taken as the user's `MaxResults` (or `MaxScanRows`, or `MaxLockTime`) value.

Example 9.1. Effect of setting maxresults, maxscanrows, and maxlocktime.

As an administrator, you want members of the group `rocketdev` to be limited to operations of 20,000 files or less, that scan no more than 100,000 revisions, and lock database tables for no more than 30 seconds:

```
Group:        rocketdev
MaxResults:   20000
MaxScanRows:  100000
MaxLockTime:  30000
Timeout:      43200
Subgroups:
Owners:
Users:
        bill
        ruth
        sandy
```

Suppose that Ruth has an unrestricted (*loose*) client view. She types:

```
$ p4 sync
```

Her **`sync`** command is rejected if the depot contains more than 20,000 files. She can work around this limitation either by restricting her client view, or, if she needs all of the files in the view, by syncing smaller sets of files at a time, as follows:

```
$ p4 sync //depot/projA/...
$ p4 sync //depot/projB/...
```

Either method enables her to sync her files to her workspace, but without tying up the server to process a single extremely large command.

Ruth tries a command that scans every revision of every file, such as:

```
$ p4 filelog //depot/projA/...
```

If there are fewer than 20,000 revisions, but more than 100,000 integrations (perhaps the `projA` directory contains 1,000 files, each of which has fewer than 20 revisions and has been branched more than 50 times), the `MaxResults` limit does not apply, but the `MaxScanRows` limit does.

Regardless of which limits are in effect, no command she runs will be permitted to lock the database for more than the `MaxLockTime` of 30,000 milliseconds.

To remove any limits on the number of result lines processed (or database rows scanned, or milliseconds of database locking time) for a particular group, set the `MaxResults` or `MaxScanRows`, or `MaxLockTime` value for that group to `unlimited`.

Because these limitations can make life difficult for your users, do not use them unless you find that certain operations are slowing down your server. Because some Perforce applications can perform large operations, you should typically set `MaxResults` no smaller than 10,000, set `MaxScanRows` no smaller than 50,000, and `MaxLockTime` to somewhere within the 1,000-30,000 (1-30 second) range.

For more information, including a comparison of Perforce commands and the number of files they affect, type:

```
$ p4 help maxresults
$ p4 help maxscanrows
$ p4 help maxlocktime
```

from the command line.

MaxResults, MaxScanRows and MaxLockTime for users in multiple groups

As mentioned earlier, if a user is listed in multiple groups, the highest numeric `MaxResults` limit of all the groups a user belongs to is the limit that affects the user.

The default value of `unset` is *not* a numeric limit; if a user is in a group where `MaxResults` is set to `unset`, he or she is still limited by the highest numeric `MaxResults` (or `MaxScanRows` or `MaxLockTime`) setting of the other groups of which he or she is a member.

A user's commands are truly unlimited only when the user belongs to no groups, or when any of the groups of which the user is a member have their `MaxResults` set to `unlimited`.

Limiting simultaneous connections

If monitoring is enabled (`p4 configure set monitor=1` or higher), you can set the `server.maxcommands` configurable to limit the number of simultaneous command requests that the service will attempt to handle.

Ideally, this value should be set low enough to detect a runaway script or denial of service attack before the underlying hardware resources are exhausted, yet high enough to maintain a substantial margin of safety between the typical average number of connections and your site's peak activity.

If `P4LOG` is set, the server log will contain lines of the form:

```
Server is now using nnn active threads.
```

You can use the server log to determine what levels of activity are typical for your site. As a general guideline, set `server.maxcommands` to *at least* 200-500% of your anticipated peak activity.

Unloading infrequently-used metadata

Over time, a Perforce server accumulates metadata associated with old projects that are no longer in active development. On large sites, reducing the working set of data, (particularly that stored in the `db.have` and `db.labels` tables) can significantly improve performance.

Create the unload depot

To create an unload depot named `//unload`, enter **`p4 depot unload`**, and fill in the resulting form as follows:

```
Depot:      unload
Type:       unload
Map:        unloaded/...
```

In this example, unloaded metadata is stored in flat files in the `/unloaded` directory beneath your server root (that is, as specified by the `Map:` field).

After you have created the unload depot, you can use **`p4 unload`** and **`p4 reload`** to manage your installation's handling of workspace and label-related metadata.

Unload old client workspaces, labels, and task streams

The **`p4 unload`** command transfers infrequently-used metadata from the versioning engine's `db.*` files to a set of flat files in the unload depot.

Individual users can use the `-c`, `-l`, and `-s` flags to unload client workspaces, static labels, or task streams that they own. For example, maintainers of build scripts that create one workspace and/or label per build, particularly in continuous build environments, should be encouraged to unload the labels after each build:

```
$ p4 unload -c oldworkspace
$ p4 unload -l oldlabel
```

Similarly, developers should be encouraged to unload (**`p4 unload -s oldtaskstream`**) or delete (**`p4 stream -d oldtaskstream`**) task streams after use.

To manage old or obsolete metadata in bulk, administrators can use the `-a`, `-al`, or `-ac` flags in conjunction with the `-d` *`date`* and/or `-u` *`user`* flags to unload all static labels and workspaces older than a specific *`date`*, owned by a specific *`user`*, or both.

By default, only unlocked labels or workspaces are unloaded; use the `-L` flag to unload locked labels or workspaces.

To unload or reload a workspace or label, a user must be able to scan *all* the files in the workspace's have list and/or files tagged by the label. Set `MaxScanrows` and `MaxResults` high enough (see

"MaxResults, MaxScanRows and MaxLockTime for users in multiple groups" on page 162) that users do not need to ask for assistance with **p4 unload** or **p4 reload** operations.

Accessing unloaded data

By default, Perforce commands such as **p4 clients**, **p4 labels**, **p4 files**, **p4 sizes**, and **p4 fstat** ignore unloaded metadata. Users who need to examine unloaded workspaces and labels (or other unloaded metadata) can use the **-U** flag when using these commands. For more information, see the *P4 Command Reference*.

Reloading workspaces and labels

If it becomes necessary to restore unloaded metadata back into the **db.have** or **db.labels** table, use the **p4 reload** command.

Scripting efficiently

The Perforce Command-Line Client, **p4**, supports the scripting of any command that can be run interactively. The Perforce server can process commands far faster than users can issue them, so in an all-interactive environment, response time is excellent. However, **p4** commands issued by scripts — triggers, or command wrappers, for example — can cause performance problems if you haven't paid attention to their efficiency. This is not because **p4** commands are inherently inefficient, but because the way one invokes **p4** as an interactive user isn't necessarily suitable for repeated iterations.

This section points out some common efficiency problems and solutions.

Iterating through files

Each Perforce command issued causes a connection thread to be created and a **p4d** subprocess to be started. Reducing the number of Perforce commands your script runs might make it more efficient if the command is lockless. Depending on the use of shared locks however, it might be more efficient to have several commands operate on smaller sets of files than having one command operate on a large set of files.

To minimize the number of commands, try this approach:

```
for i in p4 diff2 path1/... path2/...
do
    [process diff output]
done
```

Instead of an inefficient approach like:

```
for i in p4 files path1/...
do
    p4 diff2 path1/$i path2/$i
    [process diff output]
done
```

Using list input files

Any Perforce command that accepts a list of files as a command-line argument can also read the same argument list from a file. Scripts can make use of the list input file feature by building up a list of files first, and then passing the list file to **`p4 -x`**.

For example, if your script might look something like this:

```
for components in header1 header2 header3
do
    p4 edit ${component}.h
done
```

A more efficient alternative would be:

```
for components in header1 header2 header3
do
    echo ${component}.h >> LISTFILE
done
p4 -x LISTFILE edit
```

The `-x` *`file`* flag instructs **`p4`** to read arguments, one per line, from the named file. If the file is specified as `-` (a dash), the standard input is read.

By default, the server processes arguments from `-x` *`file`* in batches of 128 arguments at a time; you can change the number of arguments processed by the server by using the `-b` *`batchsize`* flag to pass arguments in different batch sizes.

Using branch views

Branch views can be used with **`p4 integrate`** or **`p4 diff2`** to reduce the number of Perforce command invocations. For example, you might have a script that runs:

```
$ p4 diff2 pathA/src/...   pathB/src/...
$ p4 diff2 pathA/tests/... pathB/tests/...
$ p4 diff2 pathA/doc/...   pathB/doc/...
```

You can make it more efficient by creating a branch view that looks like this:

```
Branch:        pathA-pathB
View:
        pathA/src/...      pathB/src/...
        pathA/tests/...    pathB/tests/...
        pathA/doc/...      pathB/doc/...
```

…and replacing the three commands with one:

```
$ p4 diff2 -b pathA-pathB
```

Limiting label references

Repeated references to large labels can be particularly costly. Commands that refer to files using labels as revisions will scan the whole label once for each file argument. To keep from hogging the Perforce server, your script should get the labeled files from the server, and then scan the output for the files it needs.

For example, this:

```
$ p4 files path/...@label | egrep "path/f1.h|path/f2.h|path/f3.h"
```

imposes a lighter load on the Perforce server than either this:

```
$ p4 files path/f1.h@label path/f1.h@label path/f3.h@label
```

or this:

```
$ p4 files path/f1.h@label
$ p4 files path/f2.h@label
$ p4 files path/f3.h@label
```

The "temporary client workspace" trick described below can also reduce the number of times you have to refer to files by label.

On large sites, consider unloading infrequently-referenced or obsolete labels from the database. See "Unloading infrequently-used metadata" on page 163.

Using a temporary client workspace

Most Perforce commands can process all the files in the current workspace view with a single command-line argument. By making use of a temporary client workspace with a view that contains only the files on which you want to work, you might be able to reduce the number of commands you have to run, or to reduce the number of file arguments you need to give each command.

For instance, suppose your script runs these commands:

```
$ p4 sync pathA/src/...@label
$ p4 sync pathB/tests/...@label
$ p4 sync pathC/doc/...@label
```

You can combine the command invocations and reduce the three label scans to one by using a client workspace specification that looks like this:

```
Client:        XY-temp
View:
        pathA/src/...      //XY-temp/pathA/src/...
        pathB/tests/...    //XY-temp/pathB/tests/...
        pathC/doc/...      //XY-temp/pathC/doc/...
```

Using this workspace specification, you can then run:

```
$ p4 -c XY-temp sync @label
```

Using compression efficiently

There are cases where compression is automatically handled:

- By default, revisions of files of type **binary** are compressed when stored on the Perforce server. Some file formats (for example, .GIF and .JPG images, .MPG and .AVI media content, files compressed with **.gz** compression) include compression as part of the file format.

  Attempting to compress such files on the Perforce server results in the consumption of server CPU resources with little or no savings in disk space. To disable server storage compression for these file types, specify such files as type **binary+F** (binary, stored on the server in full, without compression) either from the command line or from the **p4 typemap** table.

  For more about **p4 typemap**, including a sample typemap table, see "Defining filetypes with p4 typemap" on page 33.

- By default compression is enabled between the Perforce server and the proxy; if this connection is going across a VPN that is already doing compression at a lower layer, you might want to disable the compression for the proxy (**-c** flag).

Other server configurables

The Perforce server has many configurables that may be changed for performance purposes.

A complete list of configurables may be found by running **p4 help configurables**.

Checkpoints for database tree rebalancing

Perforce's internal database stores its data in structures called Bayer trees, more commonly referred to as B-trees. While B-trees are a very common way to structure data for rapid access, over time, the process of adding and deleting elements to and from the trees can eventually lead to imbalances in the data structure.

Eventually, the tree can become sufficiently unbalanced that performance is degraded. The Perforce checkpoint and restore processes (see "Backup and recovery concepts" on page 105) re-create the trees in a balanced manner, and consequently, you might see some improvement in server performance following a backup, a removal of the **db.*** files, and the re-creation of the **db.*** files from a checkpoint.

Given the length of time required for the trees to become unbalanced during normal Perforce use, we expect that the majority of sites will never need to restore the database from a checkpoint (that is, rebalance the trees) to improve performance.

(The changes to the B-trees between Perforce 2013.2 and 2013.3 require that any upgrade that crosses this release boundary must be performed by taking a checkpoint with the older release and restoring that checkpoint with the newer release. See "Upgrading p4d - between 2013.2 and 2013.3" on page 30 for details.)

Chapter 10

Customizing Perforce: Job Specifications

Perforce's jobs feature enables users to link changelists to enhancement requests, problem reports, and other user-defined tasks. Perforce also offers P4DTG (Perforce Defect Tracking Gateway) as a means to integrate third-party defect tracking tools with Perforce. See "Working with third-party defect tracking systems" on page 176 for details.

The Perforce user's use of **p4 job** is discussed in the *Helix Versioning Engine User Guide*. This chapter covers administrator modification of the jobs system.

Perforce's default jobs template has five fields for tracking jobs. These fields are sufficient for small-scale operations, but as projects managed by Perforce grow, the information stored in these fields might be insufficient. To modify the job template, use the **p4 jobspec** command. You must be a Perforce administrator to use **p4 jobspec**.

This chapter discusses the mechanics of altering the Perforce job template.

Warning

Improper modifications to the Perforce job template can lead to corruption of your server's database. Recommendations, caveats, and warnings about changes to job templates are summarized at the end of this chapter.

The default Perforce job template

To understand how Perforce jobs are specified, consider the default Perforce job template. The examples that follow in this chapter are based on modifications to the this template.

A job created with the default Perforce job template has this format:

```
# A Perforce Job Specification.
#
#  Job:         The job name.  'new' generates a sequenced job number.
#  Status:      Either 'open', 'closed', or 'suspended'. Can be changed.
#  User:        The user who created the job. Can be changed.
#  Date:        The date this specification was last modified.
#  Description: Comments about the job.  Required.
Job:    new
Status: open
User:   edk
Date:   2011/06/03 23:16:43
Description:
        <enter description here>
```

The template from which this job was created can be viewed and edited with **p4 jobspec**. The default job specification template looks like this:

```
# A Perforce Job Specification.
#
#  Updating this form can be dangerous!
#  See 'p4 help jobspec' for proper directions.
Fields:
      101 Job word 32 required
      102 Status select 10 required
      103 User word 32 required
      104 Date date 20 always
      105 Description text 0 required
Values:
      Status open/suspended/closed
Presets:
      Status open
      User $user
      Date $now
      Description $blank
Comments:
      # A Perforce Job Specification.
      #
      # Job: The job name. 'new' generates a sequenced job number.
      # Status: Either 'open', 'closed', or 'suspended'. Can be changed.
      # User: The user who created the job. Can be changed.
      # Date: The date this specification was last modified.
      # Description: Comments about the job. Required.
```

The job template's fields

There are four fields in the **`p4 jobspec`** form. These fields define the template for all Perforce jobs stored on your server. The following table shows the fields and field types.

Field / Field Type	Meaning
`Fields:`	A list of fields to be included in each job. Each field consists of an ID#, a name, a datatype, a length, and a setting. Field names must not contain spaces.
`Values:`	A list of fields whose datatype is `select`. For each `select` field, you must add a line containing the field's name, a space, and its list of acceptable values, separated by slashes.
`Presets:`	A list of fields and their default values. Values can be either literal strings or variables supported by Perforce.
`Comments:`	The comments that appear at the top of the **`p4 job`** form. They are also used by P4V, the Perforce Visual Client, to display tooltips.

The Fields: field

The **`p4 jobspec`** field `Fields:` lists the fields to be tracked by your jobs and specifies the order in which they appear on the **`p4 job`** form.

The default `Fields:` field includes these fields:

```
Fields:
      101 Job word 32 required
      102 Status select 10 required
      103 User word 32 required
      104 Date date 20 always
      105 Description text 0 required
```

Warning

Do not attempt to change, rename, or redefine fields 101 through 105. Fields 101 through 105 are used by Perforce and should not be deleted or changed. Use **`p4 jobspec`** only to add new fields (106 and above) to your jobs.

Each field must be listed on a separate line. A field is defined by a line containing each of the following five field descriptors.

Field descriptor	Meaning
ID#	A unique integer identifier by which this field is indexed. After a field has been created and jobs entered into the system, the name of this field can change, but the data becomes inaccessible if the ID number changes. ID numbers must be between `106` and `199`.
Name	The name of the field as it should appear on the **`p4 job`** form. No spaces are permitted.
Data type	One of six datatypes (`word`, `text`, `line`, `select`, `date` or `bulk`), as described in the next table.
Length	The recommended size of the field's text box as displayed in P4V, the Perforce Visual Client. To display a text box with room for multiple lines of input, use a length of `0`; to display a single line, enter the `Length` as the maximum number of characters in the line. The value of this field has no effect on jobs edited from the Perforce command line, and it is not related to the actual length of the values stored by the server.
Field type	Determines whether a field is read-only, contains default values, is required, and so on. The valid values for this field are: • `optional:` the field can take any value or can be deleted. • `default:` a default value is provided, but it can be changed or erased.

Field descriptor	Meaning
	• `required:` a default is given; it can be changed but the field can't be left empty. • `once:` read-only; the field is set once to a default value and is never changed. • `always:` read-only; the field value is reset to the default value when the job is saved. Useful only with the `$now` variable to change job modification dates, and with the `$user` variable to change the name of the user who last modified the job.

Fields have the following six datatypes.

Field Type	Explanation	Example
`word`	A single word (a string without spaces).	A userid: `edk`
`text`	A block of text that can span multiple lines.	A job's description.
`line`	One line of text.	A user's real name: `Ed K.`
`select`	One of a set of user-defined values. Each field with datatype `select` must have a corresponding line in the `Values:` field entered into the job specification.	A job's status. One of: `open/suspended/closed`
`date`	A date value: *year*/*month*/*day*:*hours*:*minutes*:*seconds*	The date and time of job creation: `1998/07/15:13:21:46`
`bulk`	A block of text that can span multiple lines, but which is not indexed for searching with **`p4 jobs -e`**.	Alphanumeric data for which text searches are not expected.

The Values: fields

You specify the set of possible values for any field of datatype `select` by entering lines in the `Values:` field. Each line should contain the name of the field, a space, and the list of possible values, separated by slashes.

In the default Perforce job specification, the `Status:` field is the only `select` field, and its possible values are defined as follows:

```
Values:
      Status open/suspended/closed
```

The Presets: field

All fields with a field type of anything other than `optional` require default values. To assign a default value to a field, create a line in the jobspec form under `Presets`, consisting of the field name to which you're assigning the default value. Any single-line string can be used as a default value.

The following variables are available for use as default values.

Variable	Value
`$user`	The Perforce user creating the job, as specified by the `P4USER` environment variable, or as overridden with **`p4 -u username job`**
`$now`	The date and time at the moment the job is saved.
`$blank`	The text `<enter description here>` When users enter jobs, any fields in your jobspec with a preset of `$blank` must be filled in by the user before the job is added to the system.

The lines in the `Presets`: field for the standard jobs template are:

```
Presets:
      Status open
      User $user
      Date $now
      Description $blank
```

Using Presets: to change default fix status

The `Presets:` entry for the job status field (field 102) has a special syntax for providing a default fix status for **`p4 fix`**, **`p4 change -s`**, and **`p4 submit -s`**.

To change the default fix status from `closed` to some other ***`fixStatus`*** (assuming that your preferred ***`fixStatus`*** is already defined as a valid `select` setting in the `Values:` field), use the following syntax:

```
Presets:
      Status openStatus,fix/fixStatus
```

In order to change the default behavior of **`p4 fix`**, **`p4 change`**, and **`p4 submit`** to leave job status unchanged after fixing a job or submitting a changelist, use the special ***`fixStatus`*** of `same`. For example:

```
Presets:
      Status open,fix/same
```

The Comments: field

The `Comments:` field supplies the comments that appear at the top of the **`p4 job`** form. Because **`p4 job`** does not automatically tell your users the valid values of `select` fields, which fields are required, and so on, your comments must tell your users everything they need to know about each field.

Each line of the `Comments:` field must be indented by at least one tab stop from the left margin, and must begin with the comment character `#`.

The comments for the default **`p4 job`** template appear as:

```
Comments:
      # A Perforce Job Specification.
      # Job: The job name. 'new' generates a sequenced job number.
      # Status: Either 'open', 'closed', or 'suspended'. Can be changed
      # User: The user who created the job. Can be changed.
      # Date: The date this specification was last modified.
      # Description: Comments about the job. Required.
```

These fields are also used by P4V, the Perforce Visual Client, to display tooltips.

Caveats, warnings, and recommendations

Although the material in this section has already been presented elsewhere in this chapter, it is important enough to bear repeating. Please follow the guidelines presented here when editing job specifications with **`p4 jobspec`**.

Warning	Please read and understand the material in this section before you attempt to edit a job specification.

- Do not attempt to change, rename, or redefine fields 101 through 105. These fields are used by Perforce and should not be deleted or changed. Use **`p4 jobspec`** only to add new fields (106 and above) to your jobs.

  Field 101 is required by Perforce and cannot be renamed nor deleted.

  Fields 102 through 105 are reserved for use by Perforce applications. Although it is possible to rename or delete these fields, it is highly undesirable to do so. Perforce applications may continue to set the value of field 102 (the `Status:` field) to `closed` (or some other value defined in the `Presets:` for field 102) upon changelist submission, even if the administrator has redefined field 102 for use as a field that does not contain `closed` as a permissible value, leading to unpredictable and confusing results.

- After a field has been created and jobs have been entered, do not change the field's ID number. Any data entered in that field through **`p4 job`** will be inaccessible.

- Field names can be changed at any time. When changing a field's name, be sure to also change the field name in other **`p4 jobspec`** fields that reference this field name. For example, if you create a new

field **106** named **severity** and subsequently rename it to **bug-severity**, then the corresponding line in the jobspec's **Presets:** field must be changed to **bug-severity** to reflect the change.

- The comments that you write in the **Comments:** field are the only way to let your users know the requirements for each field. Make these comments understandable and complete. These comments are also used to display tooltips in P4V, the Perforce Visual Client.

Example: a custom template

The following example shows a more complicated jobspec and the resulting job form:

```
# A Custom Job Specification.
#
#  Updating this form can be dangerous!
#  See 'p4 help jobspec' for proper directions.
Fields:
        101 Job word 32 required
        102 Status select 10 required
        103 User word 32 required
        104 Date date 20 always
        111 Type select 10 required
        112 Priority select 10 required
        113 Subsystem select 10 required
        114 Owned_by word 32 required
        105 Description text 0 required
Values:
        Status open/closed/suspended
        Type bug/sir/problem/unknown
        Priority A/B/C/unknown
        Subsystem server/gui/doc/mac/misc/unknown
Presets:
        Status open
        User setme
        Date $now
        Type setme
        Priority unknown
        Subsystem setme
        Owned_by $user
        Description $blank
Comments:
        # Custom Job fields:
        # Job:    The job name. 'new' generates a sequenced job number.
        # Status: Either 'open', 'closed', or 'suspended'. Can be changed
        # User:   The user who created the job. Can be changed.
        # Date:   The date this specification was last modified.
        # Type:          The type of the job. Acceptable values are
        #                'bug', 'sir', 'problem' or 'unknown'
        # Priority:      How soon should this job be fixed?
        #                Values are 'a', 'b', 'c', or 'unknown'
        # Subsystem:     One of server/gui/doc/mac/misc/unknown
        # Owned_by:      Who's fixing the bug
        # Description: Comments about the job. Required.
```

The order of the listing under `Fields:` in the **`p4 jobspec`** form determines the order in which the fields appear to users in job forms; fields need not be ordered by numeric identifier.

Running **`p4 job`** against the example custom jobspec displays the following job form:

```
# Custom Job fields:
# Job:    The job name. 'new' generates a sequenced job number.
# Status: Either 'open', 'closed', or 'suspended'. Can be changed
# User:   The user who created the job. Can be changed.
# Date:   The date this specification was last modified.
# Type:          The type of the job. Acceptable values are
#                'bug', 'sir', 'problem' or 'unknown'
# Priority:      How soon should this job be fixed?
#                Values are 'a', 'b', 'c', or 'unknown'
# Subsystem:     One of server/gui/doc/mac/misc/unknown
# Owned_by:      Who's fixing the bug
# Description: Comments about the job. Required.
Job:    new
Status: open
User:   setme
Type:   setme
Priority:       unknown
Subsystem:      setme
Owned_by:       edk
Description:
        <enter description here>
```

Working with third-party defect tracking systems

Perforce currently offers two independent platforms to integrate Perforce with third-party defect tracking systems. Both platforms allow information to be shared between Perforce's job system and external defect tracking systems.

P4DTG, The Perforce Defect Tracking Gateway

P4DTG, the Perforce Defect Tracking Gateway, is an integrated platform that includes both a graphical configuration editor and a replication engine.

The P4DTG includes a graphical configuration editor that you can use to control the relationship between Perforce jobs and the external system. Propagation of the data between the two systems is coordinated by a replication engine. P4DTG comes with plug-ins for HP Quality Center, JIRA, Redmine, and Bugzilla.

For more information, see the product page at:

http://www.perforce.com/product/components/defect_tracking_gateway

Available from this page are an overview of P4DTG's capabilities, the download for P4DTG itself, and a link to the *Defect Tracking Gateway Guide*, which describes how to install and configure the gateway to replicate data between a Perforce server and a defect tracker.

Building your own integration

Even if you don't use Perforce's integrations as your starting point, you can still use Perforce's job system as the interface between Perforce and your defect tracker. Depending on the application, the interface you set up will consist of one or more of the following:

- A trigger or script on the defect tracking system side that adds, updates, or deletes a job in Perforce every time a bug is added, updated, or deleted in the defect tracking system.

  The third-party system should generate the data and pass it to a script that reformats the data to resemble the form used by a manual (interactive) invocation of **`p4 job`**. The script can then pipe the generated form to the standard input of a **`p4 job -i`** command.

  The **`-i`** flag to **`p4 job`** is used when you want **`p4 job`** to read a job form directly from the standard input, rather than using the interactive "form-and-editor" approach typical of user operations. Further information on automating Perforce with the **`-i`** option is available in the *P4 Command Reference*.

- A trigger on the Perforce side that checks changelists being submitted for any necessary bug fix information.

For more about triggers, including examples, see Chapter 11, "Using triggers to customize behavior" on page 179.

Chapter 11

Using triggers to customize behavior

Perforce *triggers* are user-written programs or scripts that are called by a Perforce server whenever certain operations (such as changelist submits, changes to forms, attempts by users to log in or change passwords) are performed. If the script returns a value of `0`, the operation continues; if the script returns any other value, the operation fails.

Triggers allow you to extend or customize Perforce functionality. Consider the following common uses:

- To validate changelist contents beyond the mechanisms afforded by the Perforce protections table. For example, you can use a pre-submit trigger to ensure that whenever `file1` is submitted in a changelist, `file2` is also submitted.
- To perform some action before or after the execution of a particular Perforce command.
- To validate forms, or to provide customized versions of Perforce forms. For example, you can use form triggers to generate a customized default workspace view when users run the **`p4 client`** command, or to ensure that users always enter a meaningful workspace description.
- To configure Perforce to work with external authentication mechanisms such as LDAP or Active Directory.

  You might prefer to enable LDAP authentication by using an LDAP specification. For more information, see section "Authentication options" on page 69.
- To retrieve content from data sources archived outside of the Perforce repository.

For simplicity's sake, this guide refers to trigger scripts and programs as *triggers*.

Note	If the API level is 79 or greater, canonical filetypes are now displayed by default for all commands that display filetypes. If the API level is 78 or lower, filetype aliases are displayed instead. If your script depends on the display of filetype aliases, you will need either to change the API level or to change your script.

Creating triggers

This section explains the basic workflow used to create a trigger, describes a sample trigger, discusses the trigger definition, and examines a trigger's execution environment.

To create a trigger and have Perforce execute it, you must do the following:

1. Write the program or script. Triggers can be written in a shell script such as Perl, Python, or Ruby; or they can be written in any programming language that can interface with Perforce, including UNIX shell and compiled languages like C/C+.

   Triggers have access to *trigger variables* that can be used to get server state information, execution context, client information, information about the parameters passed to the trigger, and so on. For information about trigger variables, see "Trigger script variables" on page 223.

   Triggers communicate with the server using trigger variables or by using a dictionary of key/value pairs accessed via STDIN and STDOUT. For more information on these methods, see "Communication between a trigger and the server" on page 185.

Triggers can also use the command-line client (**p4.exe**) or the Perforce scripting API's (P4-Ruby, P4-Python, P4-PHP) when data is needed that cannot be accessed by trigger variables. For more information, see *APIs for Scripting*.

Triggers can be located on the server's file system or in the depot itself, for information on using a trigger that is located in the depot, see "Storing triggers in the depot" on page 188.

Triggers can be written for portability across servers. For more information, see "Writing triggers to support multiple Perforce servers" on page 190.

2. Use the **p4 triggers** command to create a trigger definition that determines when the trigger will fire. Trigger definitions are composed of four fields: these specify the trigger name, the event type that must occur, the location of the trigger and, in some cases, some file pattern that must be matched in order to fire the trigger.

   For more information, see "Trigger definition" on page 181.

Warning

When you use trigger scripts, remember that Perforce commands that write data to the depot are dangerous and should be avoided. In particular, do not run the **p4 submit** command from within a trigger script.

It's also important to avoid recursion and to watch out for client workspace locks. A trigger running commands as the requesting user could accidentally stall if it hits a lock.

Sample trigger

The following code sample is a bash **auth-check** type trigger that tries to authenticate a user (passed to the script using the **%user%** variable) using the Active Directory. If that fails, all users have the same "secret" password, and special user bruno is able to authenticate without a password.

```
USERNAME=$1
echo "USERNAME is $USERNAME"

# read user-supplied password from stdin
read USERPASS
echo Trying AD authentication for $USERNAME
echo $USERPASS | /home/perforce/p4auth_ad 192.168.100.80 389 DC=ad,DC=foo,DC=com $USERNAME
if [ $? == 0 ]
then
     # Successful AD
     echo Active Directory login successful
     exit 0
fi
# Compare user-supplied password with correct password, "secret"
PASSWORD=secret
if [ "$USERPASS" = $PASSWORD ]
then
     # Success
    exit 0
fi
if [ "$USERNAME" = "bruno" ]
then
    # Always let user bruno in
    exit 0
fi
# Failure
# password $USERPASS for $USERNAME is incorrect;
exit 1
```

To define this trigger, use the **`p4 triggers`** command, and add a line like the following to the triggers form:

```
bypassad auth-check auth "/home/perforce/bypassad.sh %user%"
```

The auth-check trigger is fired, if it exists, after a user executes the **`p4 login`** command. For authentication triggers, the password is sent on STDIN.

Note Use an `auth-check` trigger rather than the `service-check` trigger for operator users.

Trigger definition

After you have written a trigger, you create the trigger definition by issuing the **`p4 triggers`** command and providing trigger information in the triggers form. You must be a Perforce superuser to run this command. The **`p4 triggers`** form looks like this:

```
Triggers:
  relnotecheck change-submit //depot/bld/...  "/usr/bin/rcheck.pl %user%"
  verify_jobs  change-submit //depot/...      "/usr/bin/job.py %change%"
```

As with all Perforce commands that use forms, field names (such as `Triggers:`) must be flush left (not indented) and must end with a colon, and field values (that is, the set of lines you add, one for each trigger) must be indented with spaces or tabs on the lines beneath the field name.

Each line in the trigger form you fill out when you use the **`p4 triggers`** command has four fields. These are briefly described in the following table. Values for three of these fields vary with the trigger type; these values are described in additional detail in the sections describing each type of trigger. The *`name`* field uses the same format for all trigger types.

Field	Meaning
`name`	The user-defined name of the trigger. To use the same trigger script with multiple file patterns, list the same trigger multiple times on contiguous lines in the trigger table. Use exclusionary mappings to prevent files from activating the trigger script; the order of the trigger entries matters, just as it does when exclusionary mappings are used in views. In this case, only the *`command`* of the first such trigger line that matches a path is used.
`type`	Triggers are divided into ten categories: submit triggers, push triggers, command triggers, journal-rotate triggers, shelve triggers, edge-server triggers, fix triggers, form triggers, authentication triggers, and archive triggers. One or more types is defined for each of these categories. For example, submit triggers include the `change-submit`, `change-content`, `change-commit`, and `change-failed` types. Please consult the section describing the category of interest to determine which types relate to that trigger.
`path`	The use of this field varies with the trigger type. For example, for submit, edge server, and shelve triggers, this field is a file pattern in depot syntax. When a user submits a changelist that contains files that match this pattern, the trigger script executes. Please consult the section describing the trigger of interest to determine which path is appropriate for that trigger.
`command`	The trigger for the Perforce server to run when the conditions implied by the trigger definition is satisfied. You must specify the name of the trigger script or executable in ASCII, even when the server is running in Unicode mode and passes arguments to the trigger script in UTF8. Specify the trigger in a way that allows the Perforce server to locate and run the command. The *`command`* (typically a call to a script) must be quoted, and can take as arguments any argument that your *`command`* is capable of parsing, including any applicable Perforce trigger variables. On those platforms where the operating system does not know how to run the trigger, you will need to specify an interpreter in the command field. For example, Windows does not know how to run `.pl` files. `lo form-out label  "perl //myscripts/validate.pl"`

Field	Meaning
	When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as **`//depot/scripts/myScript.pl`**, the corresponding value for the command field might be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information.

Triggers are run in the order listed in the trigger table; if a trigger script fails for a specified type, subsequent trigger scripts also associated with that type are not run.

The **`p4 triggers`** command has a very simple syntax:

`p4 triggers [ -i | -o ]`

- With no flags, the user's editor is invoked to specify the trigger definitions.
- The `-i` flag reads the trigger table from standard input.
- The `-o` flag displays all the trigger definitions stored in the trigger table.

Execution environment

When testing and debugging triggers, remember that any **`p4`** commands invoked from within the script will run within a different environment (`P4USER`, `P4CLIENT`, and so on) than that of the calling user. You must therefore take care to initialize the environment you need from within the trigger script and not inherit these values from the current environment. For example:

```
export P4USER=george
export P4PASSWD=abracadabra
cd /home/pforce/database

p4 admin checkpoint
ls -l checkpoint.* journal*
```

In general, it is good practice to observe the following guidelines:

- Wherever possible, use the full path to executables.
- For path names that contain spaces, use the short path name.

  For example, `C:\Program Files\Perforce\p4.exe` is most likely located in `C:\PROGRA~1\Perforce\p4.exe`.

- Unicode settings affect trigger scripts that communicate with the server. You should check your trigger's use of file names, directory names, Perforce identifiers, and files that contain Unicode characters, and make sure that these are consistent with the character set used by the server.
- Login tickets may not be located in the same place as they were during testing; for testing, you can pass in data with **`p4 login < input.txt`**.

- If you are using LDAP authentication, or authentication triggers, you must authenticate using tickets (as with security level 3). It then follows that you cannot store a plaintext password value in `P4PASSWD`: you should set `P4PASSWD` to a ticket value obtained from **`p4 login -p`** instead.

- For troubleshooting, log output to a file. For example:

  ```
  date /t >> trigger.log
  p4 info >> trigger.log
  C:\PROGRA~1\Perforce\p4.exe -p myServer:1666 info
  ```

  If a trigger fails to execute, the event is now logged in the Server log and an error is sent to the user.

- Perforce commands in trigger scripts are always run by a specific Perforce user. If no user is specified, an extra Perforce license for a user named `SYSTEM` (or on UNIX, the user that owns the **`p4d`** process) is assumed. To prevent this from happening:

  - Pass a `%user%` argument to the trigger that calls each Perforce command to ensure that each command is called by that user. For example, if Joe submits a changelist that activates trigger script **`trigger.pl`**, and **`trigger.pl`** calls the **`p4 changes`** command, the script can run the command as **`p4 -u %user% changes`**.

  - Set `P4USER` for the account that runs the trigger to the name of an existing user. (If your Perforce server is installed as a service under Windows, note that Windows services cannot have a `P4USER` value; on Windows, you must therefore pass a user value to each command as described above.)

- You can access the following environment variables from a trigger: `P4USER`, `P4CLIENT`, `P4HOST`, `P4LANGUAGE`, `CWD`, `OS`.

- Timeouts associated with the trigger user might affect trigger execution. To prevent an unwanted timeout, place the user running the trigger in a group that will not time out.

  Timeout is the login ticket duration as defined by the group spec of the user the trigger is using to run commands; the ticket is the one created for use with the trigger. For example, the default login ticket duration is 8 hours, so if that is left unchanged for the trigger user, the trigger will have stopped working by the next day. Consider disabling the timeout so the trigger is not concerned about logins while it has access to the ticket file.

- By default, the Perforce service runs under the Windows local `System` account. The `System` account may have different environmental configurations (including not just Perforce-related variables, but `PATH` settings and file permissions) than the one in which you are using to test or write your trigger.

- Because Windows requires a real account name and password to access files on a network drive, if the trigger script resides on a network drive, you must configure the service to use a real userid and password to access the script.

- On Windows, standard input does not default to binary mode. In text mode, line ending translations are performed on standard input, which is inappropriate for binary files.

  If you are using archive triggers against binary files on a Windows machine, you *must* prevent unwanted line-ending translations by ensuring that standard input is changed to binary mode (`O_BINARY`).

- When using triggers on Windows, `%formfile%` and other variables that use a temp directory should use the `TMP` and `TEMP` system variables in Windows, *not* the user's `TEMP` variables.

Trigger basics

This section contains information for working with triggers. Detailed information about implementing each type of trigger is found in the sections that follow. The information in this section applies to all types of triggers.

- "Communication between a trigger and the server" on page 185 describes how to select the method used for communication and how to parse dictionary input.
- "Storing triggers in the depot" on page 188 describes how to format depot paths if you want to run a trigger from the depot.
- "Using multiple triggers" on page 189 explains how Perforce interprets and processes the trigger table when it includes multiple trigger definitions.
- "Writing triggers to support multiple Perforce servers" on page 190 describes how you can write a trigger so that it is portable across Perforce servers.
- "Triggers and distributed architecture" on page 190 explains the issues you must address when locating triggers on replicas.

For information about debugging triggers, see http://answers.perforce.com/articles/KB/1249

Communication between a trigger and the server

Triggers can communicate with the server in one of two ways: by using the variables described in "Trigger script variables" on page 223 or by using a dictionary of key/value pairs accessed via `STDIN` and `STDOUT`. The setting of the `triggers.io` configuration variable determines which method is used. The method chosen determines the content of `STDIN` and `STDOUT` and also affects how trigger failure is handled. The following table summarizes the effect of these settings. *Client* refers to the client application (Swarm, P4V, P4, etc) that is connected to the server where the trigger executes.

	triggers.io = 0	**triggers.io = 1**
Trigger succeeds	The trigger communicates with the server using trigger variables. STDIN is only used by archive or authentication triggers. It is the file content for an archive trigger, and it is the password for an authentication trigger. The trigger's STDOUT is sent as an unadorned message to the client for all triggers except archive triggers; for archive triggers, the command's standard output is the file content.	The trigger communicates with the server using STDIN and STDOUT. STDIN is a textual dictionary of name-value pairs of all the trigger variables except for `%clienthost%` and `%peerhost%`. This setting does not affect STDIN values for archive and authentication triggers. The trigger should exit with a zero value.

	triggers.io = 0	triggers.io = 1
	The trigger should exit with a zero value.	
Trigger fails	The trigger's STDOUT and STDERR are sent to the client as the text of a trigger failure error message. The trigger should exit with a non-zero value.	STDOUT is a textual dictionary that contains error information. STDERR is merged with STDOUT. Failure indicates that the trigger script can't be run, that the output dictionary includes a failure message, or that the output is mis-formatted. The execution error is logged by the server, and the server sends the client the information specified by STDOUT. If no dictionary is provided, the server sends the client a generic message that something has gone wrong.

The dictionary format is a sequence of lines containing key:value pairs. Any non-printable characters must be percent-encoded. Data is expected to be UTF8-encoded on unicode-enabled servers. Here are some examples of how the %client%, %clientprog%, %command%, and %user% variables would be represented in the %dictionary:

```
client:jgibson-aaaatchoooo
clientprog:P4/LINUX45X86_128/2017.9.MAIN/1773263782 (2017/OCT/09).
command:user-dwim
user:jgibson
```

The example above shows only a part of the dictionary. When variables are passed in this way, all the variables described in "Trigger script variables" on page 223 are passed in STDIN, and the trigger script must read all of STDIN even if the script only references some of these variables. If the script does not read all of STDIN, the script will fail and the server will see errors like this:

```
write: yourTriggerScript: Broken pipe
```

The trigger must send back a dictionary to the server via STDOUT. The dictionary must at a minimum contain an action with an optional message. The action is either `pass` or `fail`. Non-printable characters must be percent encoded. For example:

```
action:fail
message:too bad!
```

Malformed trigger response dictionaries and execution problems are reported to the client with a generic error. A detailed message is recorded in the server log.

The introduction to this section suggested that the two ways of communicating with the server were mutually exclusive. In general, they are. There is one case, however, in which you must specify

variables on the command line even if you set `triggers.io` to 1. This is when you want to reference the `%peerhost%` or `%clienthost%` variables. These variables are very expensive to pass. For their values to be included in the dictionary, you must specify one or both on the command line.

The following is a sample Perl program that echoes its input dictionary to the user:

```
use strict;
use warnings FATAL=>"all";
use open qw/ :std :utf8 /;
use Data::Dumper;
use URI::Escape;

$Data::Dumper::Quotekeys = 0;
$Data::Dumper::Sortkeys  = 1;

my %keys = map { /(.*):(.*)/ } <STDIN>;

print "action:pass\nmessage:" . uri_escape Dumper \ %keys;
```

The listing begins with some code that sets Perl up for basic Unicode support and adds some error handling. The gist of the program is in line 8. `<STDIN>` is a file handle that is applied to the `map{}`, where the map takes one line of input at a time and runs the function between the map's {}. The expression `(.*):(.*)` is a regular expression with a pair of capture groups that are split by the colon. No key the server sends has a colon in it, so the first `.*` will not match. Since most non-printable characters (like newline) are percent-encoded in the dictionary, a trigger can expect every key/value pair to be a single line; hence the single regular expression can extract both the key and the value. The return values of the regular expression are treated as the return values for the map's function, which is a list of strings. When a list is assigned to a hash, Perl tries to make it into a list of key/value pairs. Because we know it's an even list, this works and we've gotten our data. The `print` command makes the result dictionary and sends it to the server. Calling it a pass action tells the server to let the command continue and that the message to send the user is the formated hash of the trigger's input dictionary.

Exceptions

Setting `triggers.io` to 1 does not affect authentication and archive triggers; these behave as if `triggers.io` were set to 0 no matter what the actual setting is.

Compatibility with old triggers

When you set the `triggers.io` variable to 1, it affects how the server runs all scripts, both old and new. If you don't want to rewrite your old trigger scripts, you can insert a shim between the trigger table and the old trigger script, which collects trigger output and formats it as the server now expects it. That is, the shim runs the old trigger, captures its output and return code, and then emits the appropriate dictionary back to the server. The following Perl script illustrates such a shim:

```
t form-out label unset "perl shim.pl original_trigger.exe orig_args..."
```

The **`shim.pl`** program might look like this:

```
use strict;
use warnings FATAL => "all";
use open qw/ :std :utf8 /;
use URI::Escape;
use IPC::Run3;

@_=<STDIN>;
run3 \@ARGV, undef, \$_, \$_;
print 'action:' . (? ? 'fail' : 'pass' ) . "\nmessage:" . uri_escape $_;
```

Storing triggers in the depot

You can store a trigger in the depot. This has two advantages:

- It allows you to version the trigger and be able to access prior versions if needed.
- In a distributed architecture, it enables Perforce to propagate the latest trigger script to every replica without your having to manually update the file in the filesystem of each server.

When you store a trigger in the depot, you must specify the trigger name in a special way in the `command` field of the trigger definition by enclosing the file path of the file containing the trigger in % signs. If you need to pass additional variables to the trigger, add them in the command field as you usually do. The server will create a temporary file that holds the contents of the file path name you have specified for the command field. (Working with a temporary file is preferable for security reasons and because depot files cannot generally be executed without some further processing.)

Multiple files can be loaded from the depot. In the next trigger definition, two depot paths are provided. Multiple depot paths may be used to load multiple files out of the depot when the trigger executes. For example, the triggers script might require a configuration file that is stored next to the script in the depot:

```
lo form-out label  "perl %//admin/validate.pl% %//admin/validate.conf%"
```

The depot file must already exist to be used as a trigger. All file types are acceptable so long as the content is available. For text types on unicode-enabled servers, the temporary file will be in UTF8. Protections on the depot script file must be such that only trusted users can see or write the content.

If the file path name contains spaces or if you need to pass additional parameters, you must enclose the `command` field in quotes.

In the next trigger definition, note that an interpreter is specified for the trigger. Specifying the interpreter is needed for those platforms where the operating system does not know how to run the trigger. For example, Windows does not know how to run .pl files.

```
lo form-out label  "perl %//admin/validate.pl%"
```

In the next trigger definition, the depot path is quoted because of the revision number. The absence of an interpreter value implies that the operating system knows how to run the script directly.

```
lo form-out branch "%//depot/scripts/validate.exe#123%"
```

Warning

A depot file path name may not contain reserved characters. This is because the hex replacement contains a percent sign, which is the terminator for a `%var%`. For example, no file named `@myScript` can be used because it would be processed as `%40myScript` inside a var `%%40myScript%`.

Using multiple triggers

Submit and form triggers are run in the order in which they appear in the triggers table. If you have multiple triggers of the same type that fire on the same path, each is run in the order in which it appears in the triggers table. If one of these triggers fails, no further triggers are executed.

Example 11.1. Multiple triggers on the same file

All `*.c` files must pass through the scripts `check1.sh`, `check2.sh`, and `check3.sh`:

```
Triggers:
  check1 change-submit //depot/src/*.c "/usr/bin/check1.sh %change%"
  check2 change-submit //depot/src/*.c "/usr/bin/check2.sh %change%"
  check3 change-submit //depot/src/*.c "/usr/bin/check3.sh %change%"
```

If any trigger fails (for instance, `check1.sh`), the submit fails immediately, and none of the subsequent triggers (that is, `check2.sh` and `check3.sh`) are called. Each time a trigger succeeds, the next matching trigger is run.

To link multiple file specifications to the same trigger (and trigger type), list the trigger multiple times in the trigger table.

Example 11.2. Activating the same trigger for multiple filespecs

```
Triggers:
  bugcheck change-submit //depot/*.c   "/usr/bin/check4.sh %change%"
  bugcheck change-submit //depot/*.h   "/usr/bin/check4.sh %change%"
  bugcheck change-submit //depot/*.cpp "/usr/bin/check4.sh %change%"
```

In this case, the `bugcheck` trigger runs on the `*.c` files, the `*.h` files, and the `*.cpp` files.

Multiple submit triggers of different types that fire on the same path fire in the following order:

1. `change-submit` (fired on changelist submission, before file transmission)
2. `change-content` triggers (after changelist submission and file transmission)
3. `change-commit` triggers (on any automatic changelist renumbering by the server)

Similarly, form triggers of different types are fired in the following order:

1. `form-out` (form generation)

2. `form-in` (changed form is transmitted to the server)
3. `form-save` (validated form is ready for storage in the Perforce database)
4. `form-delete` (validated form is already stored in the Perforce database)

Writing triggers to support multiple Perforce servers

To call the same trigger script from more than one Perforce server, use the `%serverhost%`, `%serverip%`, and `%serverport%` variables to make your trigger script more portable.

For instance, if you have a script that uses hardcoded port numbers and addresses…

```
#!/bin/sh
# Usage: jobcheck.sh changelist
CHANGE=$1
P4CMD="/usr/local/bin/p4 -p 192.168.0.12:1666"
$P4CMD describe -s $1 | grep "Jobs fixed...\n\n\t" > /dev/null
```

…and you call it with the following line in the trigger table…

```
jc1 change-submit //depot/qa/... "jobcheck.sh %change%"
```

…you can improve portability by changing the script as follows…

```
#!/bin/sh
# Usage: jobcheck.sh changelist server:port
CHANGE=$1
P4PORT=$2
P4CMD="/usr/local/bin/p4 -p $P4PORT"
$P4CMD describe -s $1 | grep "Jobs fixed...\n\n\t" > /dev/null
```

…and passing the server-specific data as an argument to the trigger script:

```
jc2 change-submit //depot/qa/... "jobcheck.sh %change% %serverport%"
```

Note that the `%serverport%` variable can contain a transport prefix: `ssl`, `tcp6`, or `ssl6`.

For a complete list of variables that apply for each trigger type, see "Trigger script variables" on page 223.

Triggers and distributed architecture

Triggers installed on the master server must also exist on any of its replicas.

- The trigger definition is automatically propagated to all replicas.
- It is your responsibility to make sure that the program file that implements the trigger exists on every replica where the trigger might be activated. Its location on every replica must correspond to the location provided in the `command` field of the trigger definition.

You can do this either by placing the trigger script in the same location in the file system on every server, or you can do it by storing it in the depot on the master or commit server and using depot syntax to specify the file name. In this case, the file is automatically propagated to all the replicas. For more information, see "Storing triggers in the depot" on page 188.

Triggers installed on the replicas must have the same execution environment for the triggers and the trigger bodies. This might typically include trigger login tickets or trigger script runtimes like Perl or Python.

Note

Edge servers have triggers that fire between client and edge server communication, and between edge server and commit server communication. For more information, see *Helix Versioning Engine Administrator Guide: Multi-site Deployment*.

Triggering on submits

To configure Perforce to run trigger scripts when users submit changelists, use *submit triggers*: these are triggers of type `change-submit`, `change-content`, and `change-commit`. You can also use `change-failed` triggers for the **`p4 submit`** or the **`p4 populate`** command.

You might want to take into consideration file locking behavior associated with submits: Before committing a changelist, **`p4 submit`** briefly locks all files being submitted. If any file cannot be locked or submitted, the files are left open in a numbered pending changelist. By default, the files in a failed submit operation are left locked unless the `submit.unlocklocked` configurable is set. Files are unlocked even if they were manually locked prior to submit if submit fails when `submit.unlocklocked` is set.

The following table describes the fields of a submit trigger. For sample definitions, see the subsequent sections, describing each trigger subtype.

Field	Meaning
type	• `change-submit`: Execute a submit trigger after changelist creation, but before file transfer. Trigger may not access file contents. • `change-content`: Execute a submit trigger after changelist creation and file transfer, but before file commit. To obtain file contents, use the revision specifier `@=`***change*** (where ***change*** is the changelist number of the pending changelist as passed to the script in the `%changelist%` variable) with commands such as **`p4 diff2`**, **`p4 files`**, **`p4 fstat`**, and **`p4 print`**. • `change-commit`: Execute a submit trigger after changelist creation, file transfer, and changelist commit. • `change-failed`: Execute a submit trigger if the **`p4 submit`** or the **`p4 populate`** command fails. This trigger only fires on errors that occur after a commit process has started. It does not fire for early usage errors, or due to errors from the submit form. That is, if an edge or change trigger could have run, then the `change-failed` trigger will fire if that commit fails. When using **`p4 diff2`** in a change-content trigger:

Field	Meaning

	• The first file argument can be either ***file@change*** or ***file#headrev***, but NOT ***file@=change***.
	• The second file argument (typically the change being submitted) must use the ***file@=change*** syntax to report differences successfully. (Using ***file@change*** without the equals sign reports the file revisions as identical, which is wrong.)
	For example, to submit a file **//depot/foo** as change 1001, and the previously submitted change was 1000, with a head revision of 25, both these revision specifier formats should work correctly if generated and called in the trigger script:
	`p4 diff2 //depot/foo@1000 file@=1001`
	`p4 diff2 //depot/foo#25 file@=1001`

path	A file pattern in depot syntax. When a user submits a changelist that contains any files that match this file pattern, the trigger specified in the ***command*** field is run. Use exclusionary mappings to prevent triggers from running on specified files.
command	The trigger for the Perforce server to run when a user submits a changelist that contains any file patterns specified by ***path***. Specify the command in a way that allows the Perforce server account to locate and run the command. The ***command*** (typically a call to a script) must be quoted, and can take as arguments anything that your ***command*** is capable of parsing, including any applicable Perforce trigger variables. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as **//depot/scripts/myScript.pl**, the corresponding value for the command field might be **"/usr/bin/perl %//depot/scripts/myScript.pl%"**. See "Storing triggers in the depot" on page 188 for more information. For **change-submit** and **change-content** triggers (and their corresponding edge server triggers), changelist submission does not continue if the trigger fails. For **change-commit** triggers, changelist submission succeeds regardless of trigger success or failure, but subsequent **change-commit** triggers do not fire if the script fails.

Even when a **change-submit** or **change-content** trigger script succeeds, the submit can fail because of subsequent trigger failures, or for other reasons. Use **change-submit** and **change-content** triggers only for validation, and use **change-commit** triggers for operations that are contingent on the successful completion of the submit.

Be aware of edge cases: for example, if a client workspace has the **revertunchanged** option set, and a user runs **p4 submit** on a changelist with no changed files, a changelist has been submitted with files contents, but no changes are actually committed. (That is, a **change-submit** trigger fires, a **change-content** trigger fires, but a **change-commit** trigger does not.)

Change-submit triggers

Use the `change-submit` trigger type to create triggers that fire after changelist creation, but before files are transferred to the server. Because change-submit triggers fire before files are transferred to the server, these triggers cannot access file contents. Change-submit triggers are useful for integration with reporting tools or systems that do not require access to file contents.

In addition to the **`p4 submit`** command, the **`p4 populate`** command, which does an implicit submit as part of its branching action, fires a change-submit trigger to allow for validation before submission.

Example 11.3. The following change-submit trigger is an MS-DOS batch file that rejects a changelist if the submitter has not assigned a job to the changelist. This trigger fires only on changelist submission attempts that affect at least one file in the //depot/qa branch.

```
@echo off

rem REMINDERS
rem - If necessary, set Perforce environment vars or use config file
rem - Set PATH or use full paths (C:\PROGRA~1\Perforce\p4.exe)
rem - Use short pathnames for paths with spaces, or quotes
rem - For troubleshooting, log output to file, for instance:
rem - C:\PROGRA~1\Perforce\p4 info >> trigger.log

if not x%1==x goto doit
echo Usage is %0[change#]

:doit
p4 describe -s %1|findstr "Jobs fixed..." > nul
if errorlevel 1 echo No jobs found for changelist %1
p4 describe -s %1|findstr "Jobs fixed..." > nul
```

To use the trigger, add the following line to your triggers table:

```
sample1   change-submit //depot/qa/...   "jobcheck.bat %changelist%"
```

Every time a changelist is submitted that affects any files under `//depot/qa`, the `jobcheck.bat` file is called. If the string "Jobs fixed…" (followed by two newlines and a tab character) is detected, the script assumes that a job has been attached to the changelist and permits changelist submission to continue. Otherwise, the submit is rejected.

The second `findstr` command ensures that the final error level of the trigger script is the same as the error level that determines whether to output the error message.

Change-content triggers

Use the `change-content` trigger type to create triggers that fire after changelist creation and file transfer, but prior to committing the submit to the database. Change-content triggers can access file contents by using the **`p4 diff2`**, **`p4 files`**, **`p4 fstat`**, and **`p4 print`** commands with the **`@=`*`change`***

revision specifier, where ***change*** is the number of the pending changelist as passed to the trigger script in the `%changelist%` variable.

Use change-content triggers to validate file contents as part of changelist submission and to abort changelist submission if the validation fails.

Even when a `change-submit` or `change-content` trigger script succeeds, the submit can fail because of subsequent trigger failures, or for other reasons. Use `change-submit` and `change-content` triggers only for validation, and use `change-commit` triggers for operations that are contingent on the successful completion of the submit.

Example 11.4. The following change-content trigger is a Bourne shell script that ensures that every file in every changelist contains a copyright notice for the current year.

The script assumes the existence of a client workspace called `copychecker` that includes all of `//depot/src`. This workspace does not have to be synced.

```
#!/bin/sh
# Set target string, files to search, location of p4 executable...
TARGET="Copyright 'date +%Y' Example Company"
DEPOT_PATH="//depot/src/..."
CHANGE=$1
P4CMD="/usr/local/bin/p4 -p 1666 -c copychecker"
XIT=0
echo ""
# For each file, strip off #version and other non-filename info
# Use sed to swap spaces w/"%" to obtain single arguments for "for"
for FILE in '$P4CMD files $DEPOT_PATH@=$CHANGE | \
  sed -e 's/\(.*\)\#[0-9]* - .*$/\1/' -e 's/ /%/g''
do
  # Undo the replacement to obtain filename...
  FILE="'echo $FILE | sed -e 's/%/ /g''"
# ...and use @= specifier to access file contents:
  # p4 print -q //depot/src/file.c@=12345
  if $P4CMD print -q "$FILE@=$CHANGE" | grep "$TARGET" > /dev/null
  then echo ""
  else
      echo "Submit fails: '$TARGET' not found in $FILE"
      XIT=1
  fi
done
exit $XIT
```

To use the trigger, add the following line to your triggers table:

```
sample2  change-content //depot/src/... "copydate.sh %change%"
```

The trigger fires when any changelist with at least one file in `//depot/src` is submitted. The corresponding `DEPOT_PATH` defined in the script ensures that of all the files in the triggering changelist, only those files actually under `//depot/src` are checked.

Change-commit triggers

Use the `change-commit` trigger type to create triggers that fire after changelist creation, file transfer, and changelist commission to the database. Use change-commit triggers for processes that assume (or require) the successful submission of a changelist.

Warning

When a `change-commit` trigger fires, any file in the committed changelist has already been submitted and could be changed by a user while the `change-commit` trigger executes.

Example 11.5. A change-commit trigger that sends emails to other users who have files open in the submitted changelist.

```
#!/bin/sh
# mailopens.sh - Notify users when open files are updated
changelist="$1
workspace="$2"
user="$3"
p4 fstat -e "$changelist" //... | while read -r line
do
   # Parse out the name/value pair.
   name=$(echo "$line" | sed 's/[\. ]\+\([^ ]\+\) .\+/\1/')
   value=$(echo "$line" | sed 's/[\. ]\+[^ ]\+ \(.\+\)/\1/')
   if [ "$name" = "depotFile" ]
   then
     # Line is "... depotFile <depotFile>". Parse to get depotFile.
     depotfile="$value"
   elif [ "$(echo "$name" | cut -b-9)" = "otherOpen" ] && \
        [ "$name" != "otherOpen" ]
   then
     # Line is "... ... otherOpen[0-9]+ <otherUser@otherWorkspace>".
     # Parse to get otherUser and otherWorkspace.
     otheruser=$(echo "$value" | sed 's/\(.\+\)@.\+/\1/')
     otherworkspace=$(echo "$value" | sed 's/.\+@\(.\+\)/\1/')
     # Get email address of the other user from p4 user -o.
     othermail=$(p4 user -o "$otheruser" | grep "Email:" | \
         grep -v \# | cut -b8-)

     # Mail other user that a file they have open has been updated
     mail -s "$depotfile was just submitted" "$othermail" <<EOM
The Perforce file: $depotfile
was just submitted in changelist $changelist by Perforce user $user
from the $workspace workspace.  You have been sent this message
because you have this file open in the $otherworkspace workspace.
EOM
   fi
done
exit 0
```

To use the trigger, add the following line to your triggers table:

```
sample3  change-commit //... "mailopens.sh %change% %client% %user%"
```

Whenever a user submits a changelist, any users with open files affected by that changelist receive an email notification.

Triggering on pushes and fetches

To configure Perforce to run trigger scripts when the **`p4 push`**, **`p4 unzip`**, or **`p4 fetch`** commands are invoked, use *push triggers*: these include triggers of type **`push-submit`**, **`push-content`**, and **`push-commit`**.

This section describes the triggers that can be used when initiating a push or fetch. See "Additional triggers for push and fetch commands" on page 204 for a description of the triggers that can be used by the server receiving the pushed items or responding to the fetch request.

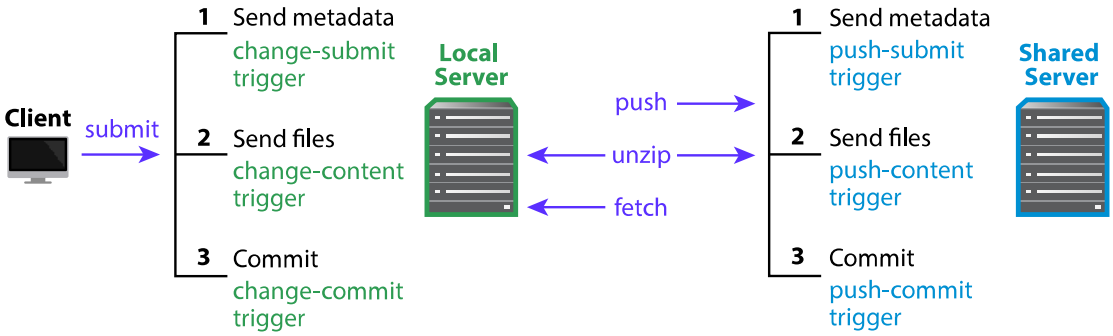
Because during a push, the local server acts as the client of the shared server, there are many similarities between the processing of submits and that of pushes:

- Push actions are atomic: either everything is pushed or nothing is pushed.
- Pushes occur in three distinct phases and different types of push triggers are appropriate for each phase.

(It is also the case that push triggers differ from change triggers; these differences affect the possible content of push triggers and influence the kind of trigger you want to use to customize the processing of changes. We will describe these differences shortly.)

The following figure illustrates the path of submitted files, via a changelist, from the client, to the local server, and finally, to the shared server. It also shows the types of triggers that may be run during each phase of these processes. There is no requirement that any triggers be run at any point in the submission or push process: the figure includes all possible types of triggers to illustrate the similarities between submits and pushes.

Figure 11.1. Change and push triggers



The three phases of submits and pushes include the following:

1. Metadata is sent.

   Following this phase, a change-submit or push-submit trigger may test to see whether the user is allowed to perform the action, whether the file type is acceptable, and so on. Anything one can query about the metadata is actionable.

2. Files are sent but changes are not yet committed.

   Following this phase, a `content-submit` or `push-submit` trigger may parse the contents of the files and take appropriate action depending on what it discovers. During this phase, one might look to see whether the submitted files adhere to coding conventions or other policies.

3. The changes are committed.

   Following this phase, the commit is irrevocable, but the trigger may take some action: send a notification, do some clean up, and so on.

Turning to look at the differences between submits and pushes, we discover the following:

- While both submits and pushes are atomic, a submit encompasses a single changelist; a push may contain multiple changelists. Thus the failure of a push is more costly.
- Submits are unidirectional; pushes (which might happen as the result of a **`p4 push`**, **`p4 fetch`**, or **`p4 unzip`**) are bidirectional; depending on the command that causes the trigger to execute, either the local server or the shared server might play the role of client.
- During the first phase of a push, metadata is read into memory, which limits the data that the `push-commit` trigger (which is a separate process with its own per-instance memory) can access. See "Push-submit triggers" on page 198 for more information.
- If a submit fails, you're left with work in progress that you can change and retry. Having a push operation fail requires that you retrace your steps prior to the submit to the local server. For this reason, you might want to run triggers during the submit operation rather than the push operation if possible.
- Change triggers are involved in the processing of **`p4 submit`** commands only. Push triggers are invoked in the context of three somewhat different scenarios: the execution of **`p4 push`**, **`p4 fetch`**, or **`p4 unzip`** commands.

You should keep these differences in mind when you decide how to define your triggers and at what stage to run them.

The following table describes the fields of a push trigger. For sample definitions, see the subsequent sections, describing each push trigger type.

Field	Meaning
`type`	• `push-submit`: Execute this trigger after changelist creation, but before file transfer. Trigger may not access file contents. • `push-content`: Execute this trigger after changelist creation and file transfer, but before file commit. To obtain file contents, use the revision specifier `@=`***`change`*** (where ***`change`*** is the changelist number of the pending changelist as passed to the script in the `%changelist%` variable) with commands such as **`p4 diff2`**, **`p4 files`**, **`p4 fstat`**, and **`p4 print`**. • `push-commit`: Execute this trigger after changelist creation, file transfer, and changelist commit.

Field	Meaning
path	A file pattern in depot syntax. When a user uses the **p4 push**, **p4 unzip**, or **p4 fetch** commands to submit a changelist that contains any files that match this file pattern, the trigger specified in the ***command*** field is run. Use exclusionary mappings to prevent triggers from running on specified files.
command	The trigger for the Perforce server to run when a user invokes the **p4 push**, **p4 unzip**, or **p4 fetch** commands to submit a changelist that contains any file patterns specified by *path*. Specify the command in a way that allows the Perforce server account to locate and run the command. The ***command*** (typically a call to a script) must be quoted, and can take as arguments anything that your ***command*** is capable of parsing, including any applicable Perforce trigger variables. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as `//depot/scripts/myScript.pl`, the corresponding value for the command field might be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information. For `push-submit` and `push-content` triggers, changelist submission does not continue if the trigger fails. For `push-commit` triggers, changelist submission succeeds regardless of trigger success or failure, but subsequent `push-commit` triggers do not fire if the script fails.

Even when a `push-submit` or `push-content` trigger script succeeds, the submission that caused the trigger to run can fail because of subsequent trigger failures, or for other reasons. Use `push-submit` and `push-content` triggers only for validation, and use `push-commit` triggers or for operations that are contingent on the successful completion of the push or fetch.

Push-submit triggers

Use the `push-submit` trigger type to create triggers that fire after changelist creation, but before files are transferred to the shared server. Because push-submit triggers fire before files are transferred to the server, these triggers cannot access file contents. Push-submit triggers are useful for integration with reporting tools or systems that do not require access to file contents.

As mentioned in the previous section where submit and push processing was described, push processing limits the commands you can run in a push-submit trigger. Please use the following commands only:

```
p4 change -o %changelist%

p4 describe -s %changelist%

p4 files //path/...@=%changelist%

p4 fstat //path/...@=%changelist%
```

Example 11.6. The following push-submit trigger is an MS-DOS batch file that rejects a changelist being pushed if the changelist description does not contain a line of the form Reviewed and signed off by: XXXXXXXX .

```
@echo off

if not x%1==x goto doit
echo Usage is %0[change#]
exit 1
:doit
p4 describe -s %1 | findstr "Reviewed and signed off" > nul
if errorlevel 1 echo "Changelist %1 missing review information."
```

To use the trigger, add the following line to your triggers table:

```
sample1   push-submit //depot/qa/...   "reviewcheck.bat %changelist%"
```

Every time a changelist is pushed that affects any files under `//depot/qa`, the `reviewcheck.bat` file is called. If the string "`Reviewed and signed off`" is detected, the script assumes that the required review has happened and permits the changelist push to continue. Otherwise the push is rejected.

Note

The **`p4 change`** and **`p4 describe`** commands do not display associated fixes when run from the push-submit or push-content triggers, even if the changes being pushed have associated fixes that are added as part of the push.

Push-content triggers

Use the `push-content` trigger type to create triggers that fire after changelist creation and file transfer, but prior to committing the push to the database. Push-content triggers can access file contents by using the **`p4 diff2`**, **`p4 files`**, **`p4 fstat`**, and **`p4 print`** commands with the **`@=`*`change`*** revision specifier, where ***`change`*** is the number of the pending changelist as passed to the trigger script in the `%changelist%` variable.

Use push-content triggers to validate file contents as part of changelist submission and to abort changelist submission if the validation fails.

Even when a `push-submit` or `push-content` trigger script succeeds, the push can fail because of subsequent trigger failures, or for other reasons. Use `push-submit` and `push-content` triggers only for validation, and use `push-commit` triggers for operations that are contingent on the successful completion of the push.

Example 11.7. The following push-content trigger is a Bourne shell script that ensures that every file in every changelist contains a copyright notice for the current year.

The script assumes the existence of a client workspace called `copychecker` that includes all of `//depot/src`. This workspace does not have to be synced.

```
#!/bin/sh
# Set target string, files to search, location of p4 executable...
TARGET="Copyright 'date +%Y' Example Company"
DEPOT_PATH="//depot/src/..."
CHANGE=$1
P4CMD="/usr/local/bin/p4 -p 1666 -c copychecker"
XIT=0
echo ""
# For each file, strip off #version and other non-filename info
# Use sed to swap spaces w/"%" to obtain single arguments for "for"
for FILE in '$P4CMD files $DEPOT_PATH@=$CHANGE | \
  sed -e 's/\(.*\)\#[0-9]* - .*$/\1/' -e 's/ /%/g''
do
  # Undo the replacement to obtain filename...
  FILE="'echo $FILE | sed -e 's/%/ /g''"
# ...and use @= specifier to access file contents:
  # p4 print -q //depot/src/file.c@=12345
  if $P4CMD print -q "$FILE@=$CHANGE" | grep "$TARGET" > /dev/null
  then echo ""
  else
      echo "Submit fails: '$TARGET' not found in $FILE"
      XIT=1
  fi
done
exit $XIT
```

To use the trigger, add the following line to your triggers table:

```
sample2  push-content //depot/src/... "copydate.sh %change%"
```

The trigger fires when any changelist with at least one file in `//depot/src` is pushed. The corresponding **`DEPOT_PATH`** defined in the script ensures that of all the files in the triggering changelist, only those files actually under `//depot/src` are checked.

Note

The **`p4 change`** and **`p4 describe`** commands do not display associated fixes when run from the push-submit or push-content triggers, even if the changes being pushed have associated fixes that are added as part of the push.

Push-commit triggers

Use the `push-commit` trigger type to create triggers that fire after changelist creation, file transfer, and changelist commission to the database. Use push-commit triggers for processes that assume (or require) the successful push of a changelist.

Example 11.8. A push-commit trigger that sends emails to other users who have files open in the pushed changelist.

```
#!/bin/sh
# mailopens.sh - Notify users when open files are updated
changelist=$1
workspace=$2
user=$3
p4 fstat @$changelist,@$changelist | while read line
do
  # Parse out the name/value pair.
  name='echo $line | sed 's/[\. ]\+\([^ ]\+\) .\+/\1/''
  value='echo $line | sed 's/[\. ]\+[^ ]\+ \(.\+\)/\1/''
  if [ "$name" = "depotFile" ]
  then
    # Line is "... depotFile <depotFile>". Parse to get depotFile.
    depotfile=$value
  elif [ "'echo $name | cut -b-9'" = "otherOpen" -a \
    "$name" != "otherOpen" ]
  then
    # Line is "... ... otherOpen[0-9]+ <otherUser@otherWorkspace>".
    # Parse to get otherUser and otherWorkspace.
    otheruser='echo $value | sed 's/\(.\+\)@.\+/\1/''
    otherworkspace='echo $value | sed 's/.\+@\(.\+\)/\1/''
    # Get email address of the other user from p4 user -o.
    othermail='p4 user -o $otheruser | grep Email: \
      | grep -v \# | cut -b8-'

    # Mail other user that a file they have open has been updated
    mail -s "$depotfile was just submitted" $othermail <<EOM
The Perforce file: $depotfile
was just pushed in changelist $changelist by Perforce user $user
from the $workspace workspace.  You have been sent this message
because you have this file open in the $otherworkspace workspace.
EOM
  fi
done
exit 0
```

To use the trigger, add the following line to your triggers table:

```
sample3  push-commit //... "mailopens.sh %change% %client% %user%"
```

Whenever a user pushes a changelist, any users with open files affected by that changelist receive an email notification.

The section "Triggering before or after commands" on page 202 describes some additional options you have for triggers with push and fetch actions.

Triggering before or after commands

Triggers of type `command` allow you to configure Perforce to run a trigger before or after a given command executes. Generally, you might want to execute a script before a command runs to prevent that command from running; you might want to run a script after a command if you want to connect its action with that of another tool or process.

Note	You may use command type triggers with **`p4 push`** and **`p4 fetch`** commands.

The following table describes the fields of the `command` trigger.

Field	Meaning
`type`	`command` The command to execute is specified in the ***`path`*** field.

`path`	The`pre-user-*command*` value specifies the command before which the trigger should execute. The **`post-user-`*`command`*** value specifies the command after which the trigger should execute. ***`command`*** can be a regular expression. For additional information about the grammar of regular expressions, see **`p4 help grep`**.
	
	Here are examples of possible path values:
	
	`pre-user-login \\ before the login command`
	`post-user-(add|edit) \\ after the add or edit command`
	`pre-user-obliterate \\ before the obliterate command`
	`(pre|post)-user-sync \\ before or after the sync command`
	
	If you want to match a command name that's a substring of another valid command, you should use the end-of-line meta-character to terminate matching. For example, use **`change$`** so you don't also match **`changes`**.
	
	For additional information about path values with **`p4 push`** and **`p4 change`** commands, see "Additional triggers for push and fetch commands" on page 204.
	
	You cannot create a **`pre-user-info`** trigger.`

`command`	The trigger for the Perforce server to run when the condition implied by ***`path`*** is satisfied. Specify the command in a way that allows the Perforce server to locate and run the command. The ***`command`*** (typically a call to a script) must be quoted, and can take as arguments anything that your ***`command`*** is capable of parsing, including any applicable Perforce trigger variable. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as **`//depot/scripts/myScript.pl`**, the corresponding value for the command field might

Field	Meaning
	be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information.

Parsing the input dictionary

One thing you might need to do in a command trigger is to parse the input dictionary. The following code sample does just that, putting the key/value store in a Perl data structure ready for access, and it shows how to send data back to the server.

```
use strict
use warnings FATAL => "all";
use open qw / :std :utf8 /;
use Data::Dumper;
use URI::Escape;

$Data::Dumper::Quotekeys = 0;
$Data::Dumper::Sortkeys = 1;

my %keys = map
{  /([^:]*):(.*)/  }
<STDIN>;

print "action:pass\nmessage:"  . uri_escape Dumper \ %keys;
```

The listing is a bit bigger than it needs to be in order to illustrate good trigger coding practice: it begins with some code that sets Perl up for basic Unicode support and adds some error handling. The gist of the program is in line 8. `<STDIN>` is a file handle that is applied to the `map{}`, where the map takes one line of input at a time and runs the function between the map's `{}`. The expression `(.*):(.*)` is a regular expression with a pair of capture groups that are split by the colon. No key the server sends has a colon in it, so the first `.*` will not match. Since most non-printable characters (like newline) are percent-encoded in the dictionary, a trigger can expect every key/value pair to be a single line; hence the single regular expression can extract both the key and the value. The return values of the regular expression are treated as the return values for the map's function, which is a list of strings. When a list is assigned to a hash, Perl tries to make it into a list of key/value pairs. Because we know it's an even list, this works and we've gotten our data.

The `print` command makes the result dictionary and sends it to the server. Calling it a pass action tells the server to let the command continue and that the message to send the user is the formated hash of the trigger's input dictionary.

After you write the script, you can add it to the trigger table by editing the **`p4 triggers`** form.

```
Triggers:
  myTrig command  post-user-move "perl /usr/bin/test.pl "
```

After the **`p4 move`** command executes, this trigger fires.

Additional triggers for push and fetch commands

The section "Triggering on pushes and fetches" on page 196 describes the triggers that you can run during the various phases of the **p4 push** and **p4 fetch** commands. These are triggers that are run by the server initiating the push or the fetch. However, for every initiator, there is a responder:

- For every push by server A to server B, there is a server B receiving the items pushed by A.
- For every fetch by server A from server B, there is a sever B that is being fetched from.

This creates additional trigger opportunities for the server receiving the push and the server responding to the fetch request. You can use `command` type triggers to take advantage of these opportunities. Within this context, `pre-user` and `post-user` actions refer to the server initiating the push or fetch; `pre-rmt` and `post-rmt` actions refer to the responding server. The following table lists the triggers that can be used by the responding, or remote, server.

Trigger	Meaning
`pre-rmt-Push`	Run this trigger on the remote server before it receives pushed content.
`post-rmt-Push`	Run this trigger on the remote server after it receives pushed content. Two special variables are available for use with post remote push triggers: • `%%firstPushedChange%%` specifies the first new changelist number • `%%lastPushedChange%%` specifies the last new changelist number
`pre-rmt-Fetch`	Run this trigger on the remote server before it responds to a fetch request.
`post-rmt-Fetch`	Run this trigger on the remote server after it responds to a fetch request.

Triggering on journal rotation

To configure Perforce to run trigger scripts when journals are rotated, use the `journal-rotate` and `journal-rotate-lock` type triggers. Journal-rotate triggers are executed after the journal is rotated on a running server, but only if journals are rotated with the **p4 admin journal** or **p4 admin checkpoint** commands. Journal rotate triggers will not execute when journals are rotated with the **p4d -jc** or **p4d --jj** commands.

Journal-rotate triggers allow you to run maintenance routines on servers after the journal has been rotated, either while the database tables are still locked or after the locks have been released. These triggers are intended to be used on replicas or edge servers where journal rotation is triggered by journal records. The server must be running for these triggers to be invoked.

The following table describes the fields of a journal-rotate trigger:

Field	Meaning
type	• `journal-rotate-lock`: Execute the trigger after the journal is rotated but while the database files are still locked. • `journal-rotate`: Execute the trigger after the journal is rotated and data base file locks are released.
path	The server on which the triggers should be run. One of the following: • `any` • *`serverid`*- run on the specified server
command	The trigger for the Perforce server to run when the server matching *`path`* is found for the trigger type. Specify the command in a way that allows the Perforce server account to locate and run the command. The *`command`* (typically a call to a script) must be quoted, and can take as arguments anything that your *`command`* is capable of parsing, including any applicable Perforce trigger variables. Journal-rotate triggers can process two variables: `%journal%` and `%checkpoint%`. These specify the names of the rotated journal and the new checkpoint if a checkpoint was taken. If no checkpoint was taken, `%checkpoint%` is an empty string. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as `//depot/scripts/myScript.pl`, the corresponding value for the command field might be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information.

Triggering on shelving events

To configure Perforce to run trigger scripts when users work with shelved files, use *shelve triggers*: these are triggers of type `shelve-submit`, `shelve-commit`, and `shelve-delete`.

The following table describes the fields of a shelving type trigger:

Field	Meaning
type	• `shelve-submit`: Execute a pre-shelve trigger after changelist has been created and files locked, but prior to file transfer. • `shelve-commit`: Execute a post-shelve trigger after files are shelved. • `shelve-delete`: Execute a shelve trigger prior to discarding shelved files.
path	A file pattern in depot syntax. If a shelve contains any files in the specified path, the trigger fires. To prevent some shelving operations from firing these triggers, use an exclusionary mapping in the path.

Field	Meaning
command	The trigger for the Perforce server to run when a matching ***path*** applies for the trigger type. Specify the command in a way that allows the Perforce server account to locate and run the command. The ***command*** (typically a call to a script) must be quoted, and can take as arguments anything that your ***command*** is capable of parsing, including any applicable Perforce trigger variables. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as `//depot/scripts/myScript.pl`, the corresponding value for the command field might be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information.

Shelve-submit triggers

The shelve-submit trigger works like the `change-submit` trigger; it fires after the shelved changelist is created, but before before files are transferred to the server. Shelve-submit triggers are useful for integration with reporting tools or systems that do not require access to file contents.

Example 11.9. A site administrator wants to prohibit the shelving of large disk images; the following shelve-submit trigger rejects a shelving operation if the changelist contains .iso files.

```
#!/bin/sh

# shelve1.sh - Disallow shelving of certain file types

# This trigger always fails: when used as a shelve-submit trigger
# with a specified path field, guarantees that files matching that
# path are not shelved

echo "shelve1.sh: Shelving operation disabled by trigger script."

exit 1
```

To use the trigger, add the following line to your triggers table, specifying the path for which shelving is to be prohibited in the appropriate field, for example:

```
shelving1   shelve-submit   //....iso   shelve1.sh
```

Every time a changelist is submitted that affects any `.iso` files in the depot, the `shelve1.sh` script runs, and rejects the request to shelve the disk image files.

Shelve-commit triggers

Use the `shelve-commit` trigger to create triggers that fire after shelving and file transfer. Use shelve-commit triggers for processes that assume (or require) the successful submission of a shelving operation.

Example 11.10. A shelve-commit trigger that notifies a user (in this case, reviewers) about a shelved changelist.

```
#!/bin/sh
# shelve2.sh - Send email to reviewers when open files are shelved
changelist=$1
workspace=$2
user=$3

mail -s "shelve2.sh: Files available for review" reviewers << EOM
   $user has created shelf from $workspace in $changelist"
EOM

exit 0
```

To use the trigger, add the following line to your triggers table:

```
shelving2  shelve-commit //... "shelve2.sh %change% %client% %user%"
```

Whenever a user shelves a changelist, reviewers receive an email notification.

Shelve-delete triggers

Use the `shelve-delete` trigger to create triggers that fire after users discard shelved files.

Example 11.11. A shelve-delete trigger that notifies reviewers that shelved files have been abandoned.

```
#!/bin/sh
# shelve3.sh - Send email to reviewers when files deleted from shelf
changelist=$1
workspace=$2
user=$3

mail -s "shelve3.sh: Shelf $changelist deleted" reviewers << EOM
   $user has deleted shelved changelist $changelist"
EOM

exit 0
```

To use the trigger, add the following line to your triggers table:

```
shelving3  shelve-delete //... "shelve3.sh %change% %client% %user%"
```

Whenever a user deletes files from the shelf, reviewers receive an email notification. A more realistic example might check an external (or internal) data source to verify that code review was complete complete before permitting the user to delete the shelved files.

Triggering on fixes

To configure Perforce to run trigger scripts when users add or delete fixes from changelists, use *fix triggers*: these are triggers of type `fix-add` and `fix-delete`.

The special variable `%jobs%` is available for expansion with fix triggers; it expands to one argument for every job listed on the **`p4 fix`** command line (or in the `Jobs:` field of a **`p4 change`** or **`p4 submit`** form), and must therefore be the last argument supplied to the trigger script.

Note

Fix-add triggers might be also be run following the submission of a changelist if the job associated with the changelist exists both on the personal and the shared servers. For more information on push triggers, see "Triggering on pushes and fetches" on page 196.

The following table describes the fields used for a fix trigger definition.

Field	Meaning
type	• `fix-add`: Execute fix trigger prior to adding a fix. • `fix-delete`: Execute fix trigger prior to deleting a fix.
path	Use `fix` as the path value.
command	The trigger for the Perforce server to run when a user adds or deletes a fix. Specify the command in a way that allows the Perforce server account to locate and run the command. The ***`command`*** (typically a call to a script) must be quoted, and can take as arguments any argument that your ***`command`*** is capable of parsing, including any applicable Perforce trigger variables. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as `//depot/scripts/myScript.pl`, the corresponding value for the command field might be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information. For `fix-add` and `fix-delete` triggers, fix addition or deletion continues whether the script succeeds or fails.

Fix-add and fix-delete triggers

Example 11.12. The following script, when copied to fixadd.sh and fixdel.sh, fires when users attempt to add or remove fix records, whether by using the p4 fix command, or

by modifying the Jobs: field of the forms presented by the p4 change and p4 submit commands.

```
#!/bin/bash
# fixadd.sh, fixdel.sh - illustrate fix-add and fix-delete triggers

COMMAND=$0
CHANGE=$1
NUMJOBS=$(($# - 1 ))

echo $COMMAND: fired against $CHANGE with $NUMJOBS job arguments.
echo $COMMAND: Arguments were: $*
```

These `fix-add` and `fix-delete` triggers fire whenever users attempt to add (or delete) fix records from changelists. To use the trigger, add the following lines to the trigger table:

```
sample4   fix-add    fix "fixadd.sh %change% %jobs%"
sample5   fix-delete fix "fixdel.sh %change% %jobs%"
```

Using both copies of the script, observe that **`fixadd.sh`** is triggered by **`p4 fix`**, the **`fixdel.sh`** script is triggered by **`p4 fix -d`**, and either script may be triggered by manually adding (or deleting) job numbers from within the `Jobs:` field in a changelist form - either by means of **`p4 change`** or as part of the **`p4 submit`** process.

Because the `%jobs%` variable is expanded to one argument for every job listed on the **`p4 fix`** command line (or in the `Jobs:` field of a **`p4 change`** or **`p4 submit`** form), it must be the last argument supplied to any `fix-add` or `fix-delete` trigger script.

Triggering on forms

To configure Perforce to run trigger scripts when users edit forms, use *form triggers*: these are triggers of type `form-save`, `form-in`, `form-out`, `form-delete`, and `form-commit`.

Use form triggers to generate customized field values for users, to validate data provided on forms, to notify other users of attempted changes to form data, and to otherwise interact with process control and management tools.

The `%specdef%` variable is defined for form triggers: it is expanded to the spec string of the form in question. This allows derived APIs to parse forms as part of triggers by loading the spec string as an argument.

If you write a trigger that fires on trigger forms, and the trigger fails in such a way that the **`p4 triggers`** command no longer works, the only recourse is to remove the `db.triggers` file in the server root directory.

The following table describes the fields of a form trigger definition:

Field	Meaning
type	• `form-save`: Execute a form trigger after the form contents are parsed, but before the contents are stored in the Perforce database. The trigger cannot modify the form specified in `%formfile%` variable. • `form-out`: Execute form trigger upon generation of form to end user. The trigger can modify the form. • `form-in`: Execute form trigger on edited form before contents are parsed and validated by the Perforce server. The trigger can modify the form. • `form-delete`: Execute form trigger after the form contents are parsed, but before the form is deleted from the Perforce database. The trigger cannot modify the form. • `form-commit`: Execute form trigger after the form has been committed for access to automatically-generated fields such as jobname, dates, etc.
path	The name of the type of form, (`branch`, `change`, `client`, `depot`, `group`, `job`, `label`, `protect`, `server`, `spec`, `stream`, `triggers`, `typemap`, or `user`).
command	The trigger for the Perforce server to run when the type of form specified in the *path* field is processed. Specify the command in a way that allows the Perforce server account to locate and run the command. The *command* (typically a call to a script) must be quoted, and can take as arguments any argument that your *command* is capable of parsing, including any applicable Perforce trigger variables. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as `//depot/scripts/myScript.pl`, the corresponding value for the command field might be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information. For `form-in`, `form-out`, `form-save`, and `form-delete` triggers, the data in the specification becomes part of the Perforce database if the script succeeds. Otherwise, the database is not updated.

Form-save triggers

Use the `form-save` trigger type to create triggers that fire when users send changed forms to the server. Form-save triggers are called after the form has been parsed by the server but before the changed form is stored in the Perforce metadata.

Example 11.13. To prohibit certain users from modifying their client workspaces, add the users to a group called lockedws and use the following form-save trigger.

This trigger denies attempts to change client workspace specifications for users in the **lockedws** group, outputs an error message containing the user name, IP address of the user's workstation, and the name of the workspace on which a modification was attempted, and notifies an administrator.

```
#!/bin/bash
NOAUTH=lockedws
USERNAME=$1
WSNAME=$2
IPADDR=$3

GROUPS='p4 groups "$1"'

if echo "$GROUPS" | grep -qs $NOAUTH
then
   echo "$USERNAME ($IPADDR) in $NOAUTH may not change $WSNAME"
   mail -s "User $1 workspace mod denial" admin@127.0.0.1
   exit 1
else
   exit 0
fi
```

This **form-save** trigger fires on **client** forms only. To use the trigger, add the following line to the trigger table:

```
sample6   form-save  client  "ws_lock.sh %user% %client% %clientip%"
```

Users whose names appear in the output of **p4 groups lockedws** have changes to their client workspaces parsed by the server, and even if those changes are syntactically correct, the attempted change to the workspace is denied, and an administrator is notified of the attempt.

Form-out triggers

Use the **form-out** trigger type to create triggers that fire whenever the Perforce server generates a form for display to the user.

Warning	Never use a Perforce command in a **form-out** trigger that fires the same **form-out** trigger, or infinite recursion will result. For example, never run **p4 job -o** from within a **form-out** trigger script that fires on **job** forms.

Example 11.14. The default Perforce client workspace view maps the entire depot // depot/... to the user's client workspace. To prevent novice users from attempting to sync

the entire depot, this Perl script changes a default workspace view of //depot/... in the p4 client form to map only the current release codeline of //depot/releases/main/...

```
#!/usr/bin/perl
# default_ws.pl - Customize the default client workspace view.
$p4 = "p4 -p localhost:1666";
$formname = $ARGV[0];  # from %formname% in trigger table
$formfile = $ARGV[1];  # from %formfile% in trigger table
# Default server-generated workspace view and modified view
# (Note: this script assumes that //depot is the only depot defined)
$defaultin = "\t//depot/... //$formname/...\n";
$defaultout = "\t//depot/releases/main/... //$formname/...\n";
# Check "p4 clients": if workspace exists, exit w/o changing view.
# (This example is inefficient if there are millions of workspaces)
open CLIENTS, "$p4 clients |" or die "Couldn't get workspace list";
while ( <CLIENTS> )
{
        if ( /^Client $formname .*/ ) { exit 0; }
}
# Build a modified workspace spec based on contents of %formfile%
$modifiedform = "";
open FORM, $formfile or die "Trigger couldn't read form tempfile";
while ( <FORM> )
{       ## Do the substitution as appropriate.
        if ( m:$defaultin: ) { $_ = "$defaultout"; }
        $modifiedform .= $_;
}
# Write the modified spec back to the %formfile%,
open MODFORM, ">$formfile" or die "Couldn't write form tempfile";
print MODFORM $modifiedform;
exit 0;
```

This `form-out` trigger fires on `client` workspace forms only. To use the trigger, add the following line to the trigger table:

```
sample7   form-out  client  "default_ws.pl %formname% %formfile%"
```

New users creating client workspaces are presented with your customized default view.

Form-in triggers

Use the `form-in` trigger type to create triggers that fire when a user attempts to send a form to the server, but before the form is parsed by the Perforce server.

Example 11.15. All users permitted to edit jobs have been placed in a designated group called jobbers. The following Python script runs p4 group -o jobbers with the -G (Python

marshaled objects) flag to determine if the user who triggered the script is in the jobbers group.

```
import sys, os, marshal

# Configure for your environment
tuser = "triggerman"   # trigger username
job_group = "jobbers"  # Perforce group of users who may edit jobs

# Get trigger input args
user = sys.argv[1]

# Get user list
# Use global -G flag to get output as marshaled Python dictionary
CMD = "p4 -G -u %s -p 1666 group -o %s" % \
        (tuser, job_group)
result = {}
result = marshal.load(os.popen(CMD, 'r'))

job_users = []
for k in result.keys():
        if k[:4] == 'User': # user key format: User0, User1, ...
                u = result[k]
                job_users.append(u)

# Compare current user to job-editing users.
if not user in job_users:
        print "\n\t>>> You don't have permission to edit jobs."
        print "\n\t>>> You must be a member of '%s'.\n" % job_group
        sys.exit(1)
else: # user is in job_group -- OK to create/edit jobs
        sys.exit(0)
```

This `form-in` trigger fires on `job` forms only. To use the trigger, add the following line to the trigger table:

```
sample8   form-in  job  "python jobgroup.py %user%"
```

If the user is in the `jobbers` group, the `form-in` trigger succeeds, and the changed job is passed to the Perforce server for parsing. Otherwise, an error message is displayed, and changes to the job are rejected.

Form-delete triggers

Use the `form-delete` trigger type to create triggers that fire when users attempt to delete a form, after the form is parsed by the Perforce server, but before the form is deleted from the Perforce database.

Example 11.16. An administrator wants to enforce a policy that users are not to delete jobs from the system, but must instead mark such jobs as closed.

```
#!/bin/sh

echo "Jobs may not be deleted. Please mark jobs as closed instead."
exit 1
```

This `form-delete` trigger fires on `job` forms only. To use the trigger, add the following line to the trigger table:

```
sample9   form-delete  job  "nodeljob.sh"
```

Whenever a user attempts to delete a job, the request to delete the job is rejected, and the user is shown an error message.

Form-commit triggers

Unlike the other form triggers, the `form-commit` trigger fires after a form is committed to the database. Use these triggers for processes that assume (or require) the successful submission of a form. In the case of job forms, the job's name is not set until after the job has been committed to the database; the `form-commit` trigger is the only way to obtain the name of a new job as part of the process of job creation.

Example 11.17. The following script, when copied to newjob.sh, shows how to get a job name during the process of job creation, and also reports the status of changelists associated with job fixes.

```
#!/bin/sh
# newjob.sh - illustrate form-commit trigger

COMMAND=$0
USER=$1
FORM=$2
ACTION=$3

echo $COMMAND: User $USER, formname $FORM, action $ACTION >> log.txt
```

To use the trigger, add the following line to the trigger table:

```
sample10  form-commit  job   "newjob.sh %user% %formname% %action%"
```

Use the `%action%` variable to distinguish whether or not a change to a job was prompted by a user directly working with a job by means of **`p4 job`**, or indirectly by means of fixing the job within the context of **`p4 fix`** or the `Jobs:` field of a changelist.

The simplest case is the creation of a new job (or a change to an existing job) with the **p4 job** command; the trigger fires, and the script reports the user, the name of the newly-created (or edited) job. In these cases, the **%action%** variable is null.

The trigger also fires when users add or delete jobs to changelists, and it does so regardless of whether the changed jobs are being manipulated by means of **p4 fix**, **p4 fix -d**, or by editing the **Jobs:** field of the changelist form provided by **p4 change** or **p4 submit** form). In these cases, the **%action%** variable holds the status of the changelist (**pending** or **submitted**) to which the jobs are being added or deleted. The **form-commit** trigger does not run if zero jobs are attached to the changelist.

Because the **%action%** variable is not always set, it must be the last argument supplied to any **form-commit** trigger script.

Triggering to use external authentication

To configure Perforce to work with an external authentication manager (such as LDAP or Active Directory), use *authentication triggers* (**auth-check**, **auth-check-sso**, **service-check**, and **auth-set**). These triggers fire on the **p4 login** and **p4 passwd** commands, respectively.

Note

You might prefer to enable LDAP authentication by using an LDAP specification. This option is recommended: it is easier to use, no external scripts are required, it provides greater flexibility in defining bind methods, it allows users who are not in the LDAP directory to be authenticated against Perforce's internal user database, and it is more secure. For more information, see "Authentication options" on page 69.

That being said, you also have the option of using **auth-check-sso** triggers when LDAP authentication is enabled. In this case, users authenticated by LDAP can define a client-side SSO script instead of being prompted for a password. If the trigger succeeds, the active LDAP configurations are used to confirm that the user exists in at least one LDAP server. The user must also pass the group authorization check if it is configured. Triggers of type **auth-check-sso** will not be called for users who do not authenticate against LDAP.

Authentication triggers differ from changelist and form triggers in that passwords typed by the user as part of the authentication process are supplied to authentication scripts as standard input; never on the command line. (The only arguments passed on the command line are those common to all trigger types, such as **%user%**, **%clientip%**, and so on.)

Warning

Be sure to spell the trigger name correctly when you add the trigger to the trigger table because a misspelling can result in all users being locked out of Perforce.

Be sure to fully test your trigger and trigger table invocation prior to deployment in a production environment.

Contact Perforce Technical Support if you need assistance with restoring access to your server.

The examples in this book are for illustrative purposes only. For a more detailed discussion, including links to sample code for an LDAP environment, see "Setting Up External Authentication Triggers" in the Perforce knowledge base:

http://answers.perforce.com/articles/KB_Article/Setting-Up-External-Authentication-Triggers

You must restart the Perforce server after adding an `auth-check` (or `service-check`) trigger in order for it to take effect. You can, however, change an existing `auth-check` trigger table entry (or trigger script) without restarting the server.

After an `auth-check` trigger is in place and the server restarted, the Perforce `security` configurable is ignored; because authentication is now under the control of the trigger script, the server's default mechanism for password strength requirements is redundant.

The following table describes the fields of an authentication trigger definition.

Field	Meaning
type	• `auth-check`: Execute an authentication check trigger to verify a user's password against an external password manager during login, or when setting a new password. If an `auth-check` trigger is present, the Perforce `security` configurable (and any associated password strength requirement) is ignored, as authentication is now controlled by the trigger script. You must restart the Perforce server after adding an `auth-check` trigger. • `auth-check-sso`: Facilitate a single sign-on user authentication. • `auth-set`: Execute an authentication set trigger to send a new password to an external password manager. • `service-check`: Execute a trigger to verify the password of a service user, rather than a standard user. Service check triggers work in the same way that `auth-check` triggers do. Do not use this type of trigger for an operator user; use the `auth-check` type trigger instead. You must restart the Perforce server after adding a `service-check` trigger.
path	Use `auth` as the path value.
command	The trigger for the Perforce server to run. See the following sections about specific authentication trigger types for more information on when the trigger is fired. In most cases, it is when the **`p4 login`** command executes. Specify the command in a way that allows the Perforce server account to locate and run the command. The ***`command`*** (typically a call to a script) must be quoted, and can take as arguments any argument that your ***`command`*** is capable of parsing, including any applicable Perforce trigger variables. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as `//depot/scripts/myScript.pl`, the corresponding value for the command field might

Field	Meaning
	be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information. For `auth-check` and `service-check` triggers (fired by **`p4 login`** from standard/operator users and service users respectively), the user's typed password is supplied to the trigger command as standard input. If the trigger executes successfully, the Perforce ticket is issued. The user name is available as `%user%` to be passed on the command line. For `auth-check-sso` triggers, (fired by **`p4 login`** for all users) the output of the client-side script (specified by `P4LOGINSSO`) is sent to the server-side script in cleartext. For `auth-set` triggers, (fired by **`p4 passwd`**, but only after also passing an `auth-check` trigger check) the user's old password and new password are passed to the trigger as standard input. The user name is available as `%user%` to be passed on the command line.

Auth-check and service-check triggers

Triggers of type `auth-check` fire when standard or operator users run the **`p4 login`** command. Similarly, `service-check` triggers fire when service users users run the **`p4 login`** command. If the script returns `0`, login is successful, and a ticket file is created for the user.

The `service-check` trigger works exactly like an `auth-check` trigger, but applies only to users whose `Type:` has been set to `service`. The `service-check` trigger type is used by Perforce administrators who want to use LDAP to authenticate other Perforce servers in replicated and other multi-server environments.

Warning	If you are using `auth-check` triggers, the Perforce superuser must also be able to authenticate against the remote authentication database. (If you, as the Perforce superuser, cannot use the trigger, you may find yourself locked out of your own server, and will have to (temporarily) overwrite your auth-check trigger with a script that always passes in order to resolve the situation.)

Example 11.18. A trivial authentication-checking script.

All users must enter the password "secret" before being granted login tickets. Passwords supplied by the user are sent to the script on STDIN.

```
#!/bin/bash
# checkpass.sh - a trivial authentication-checking script

# in this trivial example, all users have the same "secret" password
USERNAME=$1
PASSWORD=secret

# read user-supplied password from stdin
read USERPASS

# compare user-supplied password with correct password
if [ "$USERPASS" = $PASSWORD ]
then
    # Success
    exit 0
fi

# Failure
echo checkpass.sh: password $USERPASS for $USERNAME is incorrect
exit 1
```

This **auth-check** trigger fires whenever users run **p4 login**. To use the trigger, add the following line to the trigger table:

```
sample11  auth-check  auth  "checkpass.sh %user%"
```

Users who enter the "secret" password are granted login tickets.

Single signon and auth-check-sso triggers

Triggers of type **auth-check-sso** fire when standard users run the **p4 login** command. Two scripts are run: a client-side script is run on the user's workstation, and its output is passed (in plaintext) to the Perforce Server, where the server-side script runs.

- On the user's client workstation, a script (whose location is specified by the **P4LOGINSSO** environment variable) is run to obtain the user's credentials or other information verifiable by the Perforce Server. The **P4LOGINSSO** contains the name of the client-side script and zero or more of the following trigger variables, passed as parameters to the script: **%user%**, **%serverAddress%**, and **%P4PORT%**. For example:

  ```
  export P4LOGINSSO="/path/to/sso-client.sh %user% %serverAddress% %P4PORT%"
  ```

  Where **%user%** is the Perforce client user, **%serverAddress%** is the address of the target Perforce server, and **%P4PORT%** is an intermediary between the client and the server.

- On the server, the output of the client-side script is passed to the server-side script as standard input. The server-side script specified in the trigger table runs, and the server returns an exit status of 0 if successful.

  With a distributed configuration in which a proxy or broker acts as an intermediary between the client and the server, the **%serverAddress%** variable will hold the address/port of the server and the

`%P4PORT%` variable will hold the port of the intermediary. It is up to the script to decide what to do with this information.

Example 11.19. Interaction between client-side and server-side scripts.

An `auth-check-sso` trigger fires whenever users run **`p4 login`**. The system administrator might add the following line to the trigger table to specify the script that should run on the server side:

```
sample13  auth-check-sso  auth  "serverside.sh %user%"
```

and each end user sets the following environment variable on the client side:

```
export P4LOGINSSO=/usr/local/bin/clientside.sh %serverAddress%
```

When the user attempts to log on, the `P4LOGINSSO` script runs on the user's workstation:

```
##!/bin/bash
# clientside.sh - a client-side authentication script
#
# if we use %serverAddress% in the command-line like this:
#    p4 -E P4LOGINSSO=clientside.sh %serverAddress%
# then this script receives the serverAddress as $1, and the user
# can use it for multiple connections to different Perforce servers.
#
# In this example, we simulate a client-side authentication process
# based on whether the user is connecting to the same Perforce Server
# as is already configured in his or her environment.
# (We also output debugging information to a local file.)

input_saddr=$1

env_saddr=`p4 info | grep "Server address" | awk '{printf "%s", $3}'`

if test "$input_saddr" == "$env_saddr"
  then
    # User is connected to the server specified by P4PORT - pass
    echo "sso pass"; echo pass "$input_saddr" >> debug.txt; exit 0
  else
    # User is attempting to connect to another server - fail
    echo "no pass"; echo fail "$input_saddr" >> debug.txt; exit 1
fi
```

If the user is connected to the same Perforce Server as specified by `P4PORT` (that is, if the server address passed from the Server to this script matches the server that appears in the output of a plain **`p4 info`** command), client-side authentication succeeds. If the user is connected to another Perforce Server (for example, by running **`p4 -p host:port login`** against a different Perforce Server), client-side authentication fails.

The server-side script is as follows:

```
#!/bin/bash
#
# serverside.sh - a server-side authentication script
#

if test $# -eq 0
  then
    echo "No user name passed in.";
    exit 1;
fi

read msg </dev/stdin

if test "$msg" == ""
  then
    echo "1, no stdin"
    exit 1
fi

if test "$msg" == "sso pass"
  then
    exit 0
  else
    exit 1
fi
```

In a more realistic example, the end user's `P4LOGINSSO` script points to a **`clientside.sh`** script that contacts an authentication service to obtain a token of some sort. The client-side script then passes this token to Perforce Server's trigger script, and **`serverside.sh`** uses the single-signon service to validate the token.

In this example, **`clientside.sh`** merely checks whether the user is using the same connection as specified by `P4PORT`, and the output of **`clientside.sh`** is trivially checked for the string "`sso pass`"; if the string is present, the user is permitted to log on.

Triggering for external authentication

Triggers of type `auth-set` fire when users (standard users or service users) run the **`p4 passwd`** command and successfully validate their old password with an `auth-check` (or `service-check`) trigger. The process is as follows:

1. A user invokes **`p4 passwd`**.

2. The Perforce server prompts the user to enter his or her old password.

3. The Perforce server fires an `auth-check` trigger to validate the old password against the external authentication service.

4. The script associated with the `auth-check` trigger runs. If the `auth-check` trigger fails, the process ends immediately: the user is not prompted for a new password, and the `auth-set` trigger never fires.

5. If the `auth-check` trigger succeeds, the server prompts the user for a new password.

6. The Perforce server fires an `auth-set` trigger and supplies the trigger script with both the old password and the new password on the standard input, separated by a newline.

> **Note**
>
> In most cases, users in an external authentication environment will continue to set their passwords without use of Perforce. The `auth-set` trigger type is included mainly for completeness.

Because the Perforce server must validate the user's current password, you must have a properly functioning `auth-check` trigger before attempting to write an `auth-set` trigger.

Example 11.20. A trivial authentication-setting script

```
#!/bin/bash
# setpass.sh - a trivial authentication-setting script

USERNAME=$1

read OLDPASS
read NEWPASS

echo setpass.sh: $USERNAME attempted to change $OLDPASS to $NEWPASS
```

This `auth-set` trigger fires after users run **`p4 passwd`** and successfully pass the external authentication required by the `auth-check` trigger. To use the trigger, add the following two lines to the trigger table:

```
sample11  auth-check  auth  "checkpass.sh %user%"
sample12  auth-set    auth  "setpass.sh %user%"
```

This trivial example doesn't actually change any passwords; it merely reports back what the user attempted to do.

Triggering to affect archiving

The `archive` trigger type is used in conjunction with the `+X` filetype modifier in order to replace the mechanism by which the Perforce Server archives files within the repository. They are used for storing, managing, or generating content archived outside of the Perforce repository. See "Execution environment" on page 183 for platform-specific considerations.

The following table describes the fields of an archive trigger definition:

Field	Meaning
type	`archive`: Execute the script when a user accesses any file with a filetype containing the `+X` filetype modifier. The script can read, write, or delete files in the archive.

Field	Meaning
	The script is run once per file requested. For `read` operations, scripts should deliver the file to the user on standard output. For `write` operations, scripts receive the file on standard input.
`path`	A file pattern to match the name of the file being accessed in the archive.
`command`	The trigger for the Perforce server to run when a file matching *`path`* is found in the archive. Specify the command in a way that allows the Perforce server account to locate and run the command. The *`command`* (typically a call to a script) must be quoted, and can take as arguments any argument that your *`command`* is capable of parsing, including any applicable Perforce trigger variables. When your trigger script is stored in the depot, its path must be specified in depot syntax, delimited by percent characters. For example, if your script is stored in the depot as `//depot/scripts/myScript.pl`, the corresponding value for the command field might be `"/usr/bin/perl %//depot/scripts/myScript.pl%"` . See "Storing triggers in the depot" on page 188 for more information. If the command succeeds, the command's standard output is the file content. If the command fails, the command standard output is sent to the client as the text of a trigger failure error message.

Example 11.21. An archive trigger

This `archive` trigger fires when users access files that have the `+X` (archive) modifier set.

```
#!/bin/sh
# archive.sh - illustrate archive trigger

OP=$1
FILE=$2
REV=$3

if [ "$OP" = read ]
then
    cat ${FILE}${REV}
fi

if [ "$OP" = delete ]
then
    rm ${FILE}${REV}
fi

if [ "$OP" = write ]
then
    # Create new file from user's submission via stdin
    while read LINE; do
        echo ${LINE} >> ${FILE}${REV}
    done
    ls -t ${FILE}* |
    {
        read first; read second;
        cmp -s $first $second
        if [ $? -eq 0 ]
        then
            # Files identical, remove file, replace with symlink.
            rm ${FILE}${REV}
            ln -s $second $first
        fi
    }
fi
```

To use the trigger, add the following line to the trigger table:

```
arch  archive  path  "archive.sh %op% %file% %rev%"
```

When the user attempts to submit (write) a file of type `+X` in the specified ***`path`***, if there are no changes between the current revision and the previous revision, the current revision is replaced with a symlink pointing to the previous revision.

Trigger script variables

You can use trigger script variables to pass data to a trigger script. All data is passed as a string; it is up to the trigger to interpret and use these appropriately.

It is also possible to have the server and trigger communicate using STDIN and STDOUT. For more information, see "Communication between a trigger and the server" on page 185.

The *maxError…* variables refer to circumstances that prevented the server from completing a command; for example, an operating system resource issue. Note also that client-side errors are not always visible to the server and might not be included in the `maxError` count.

The `terminated` and `termType` variables indicate whether the command exited early and why.

Note

The processing of unknown variables has changed. Previously, unknown variables were removed from the trigger invocation. Currently they are left as is. This preserves the trigger argument ordering, and might be a clue to authors that data they assumed to be available is not.

Argument	Description	Available for type
`%action%`	Either null or a string reflecting an action taken to a changelist or job. For example,"`pending change 123 added`" or "`submitted change 124 deleted`" are possible `%action%` values on `change` forms, and "`job000123 created`" or "`job000123 edited`" are possible `%action%` values for `job` forms.	`form-commit`
`%argc%`	Command argument count.	all except archive
`%args%`	Command argument string.	all except archive
`%argsQuoted%`	Command argument string that contains the command arguments as a percent-encoded comma-separated list.	all except archive

`%changelist%`, `%change%`	The number of the changelist being submitted. The abbreviated form `%change%` is equivalent to `%changelist%`.
	
	A `change-submit` trigger is passed the pending changelist number; a `change-commit` trigger receives the committed changelist number.
	
	A `shelve-commit` or `shelve-delete` trigger receives the changelist number of the shelf.
		`change-submit`
		`push-submit`
		`change-content`
		`push-content`
		`change-commit`
		`push-commit`
		`fix-add`,
		`fix-delete`,
		`form-commit`,
		`shelve-commit`,
		`shelve-delete`

`%changeroot%`	The root path of files submitted.	`change-commit`
		
		`push-commit`

`%client%`	Triggering user's client workspace name.	all
`%clientcwd%`	Client's current working directory.	all except archive

Argument	Description	Available for type
`%clienthost%`	Hostname of the user's workstation (even if connected through a proxy, broker, replica, or an edge server.)	all
`%clientip%`	The IP address of the user's workstation (even if connected through a proxy, broker, replica, or an edge server.)	all
`%clientprog%`	The name of the user's client application. For example, P4V, P4Win, etc.	all
`%clientversion%`	The version of the user's client application.	all
`%command%`	Command name.	all except archive
`%file%`	Path of archive file based on depot's `Map:` field. If the `Map:` field is relative to `P4ROOT`, the `%file%` is a server-side path relative to `P4ROOT`. If the `Map:` field is an absolute path, the `%file%` is an absolute server-side path.	`archive`
`%firstPushedChange%`	First new changelist number. See "Additional triggers for push and fetch commands" on page 204 for more information.	`command`
`%formfile%`	Path to temporary form specification file. To modify the form from an `in` or `out` trigger, overwrite this file. The file is read-only for triggers of type `save` and `delete`.	`form-commit`, `form-save`, `form-in`, `form-out`, `form-delete`
`%formname%`	Name of form (for instance, a branch name or a changelist number).	`form-commit`, `form-save`, `form-in`, `form-out`, `form-delete`
`%formtype%`	Type of form (for instance, `branch`, `change`, and so on).	`form-commit`, `form-save`, `form-in`, `form-out`, `form-delete`
`%groups%`	List of groups to which the user belongs, space-separated.	all except archive
`%intermediateService%`	A broker or proxy is present.	all except archive

Argument	Description	Available for type
`%jobs%`	A string of job numbers, expanded to one argument for each job number specified on a **`p4 fix`** command or for each job number added to (or removed from) the **`Jobs:`** field in a **`p4 submit`**, or **`p4 change`** form.	`fix-add`, `fix-delete`
`%lastPushedChange%`	Last new changelist number. See "Additional triggers for push and fetch commands" on page 204 for more information.	`command`
`%maxErrorSeverity%`	One of `empty`, `error`, or `warning`.	all except archive
`%maxErrorText%`	Error number and text.	all except archive
`%maxLockTime%`	A user-specified value that specifies the number of milliseconds for the longest permissible database lock. If this variable is set, it means the user has overridden the group setting for this value.	all except archive
`%maxResults%`	A user-specified value that specifies the amount of data buffered during command execution. If this variable is set, it means the user has overridden the group setting for this value.	all except archive
`%maxScanRows%`	A user-specified value that specifies the maximum number of rows scanned in a single operation. If this variable is set, it means the user has overridden the group setting for this value.	all except archive

`%oldchangelist%`	If a changelist is renumbered on submit, this variable contains the old changelist number.	`change-commit`
		
		`push-commit`

`%op%`	Operation: `read`, `write`, or `delete`.	`archive`
`%peerhost%`	If the command was sent through a proxy, broker, replica, or edge server, the hostname of the proxy, broker, replica, or edge server. (If the command was sent directly, `%peerhost%` matches `%clienthost%`)	all
`%peerip%`	If the command was sent through a proxy, broker, replica, or edge server, the IP address of the proxy, broker, replica, or edge	all

Argument	Description	Available for type
	server. (If the command was sent directly, `%peerip%` matches `%clientip%`)	
`%P4PORT%`	The host port to which the client connects. If the client connects to the server through an intermediary, this will hold the port number of the intermediary. If there's no intermediary, this will hold the same value as the `%serverAddress%` variable.	`auth-check-sso` (client-side script only)
`%quote%`	A double quote character.	all
`%rev%`	Revision of archive file	`archive`
`%serverAddress%`	The IP address and port of the Perforce server, passable only in the context of a client-side script specified by `P4LOGINSSO`.	`auth-check-sso` (client-side script only)
`%serverhost%`	Hostname of the Perforce server.	all
`%serverid%`	The value of the Perforce server's `server.id`. See **`p4 serverid`** in the *P4 Command Reference* for details.	all
`%serverip%`	The IP address of the server.	all
`%servername%`	The value of the Perforce server's `P4NAME`.	all
`%serverport%`	The transport, IP address and port of the Perforce server, in the format *`prefix:ip_address:port`*. *`prefix`* can be one of `ssl`, `tcp6`, or `ssl6`. This means that the command **`p4 -p %serverport%`** can be used to connect to the server no matter which type of connection the server uses.	all
`%serverroot%`	The `P4ROOT` directory of the Perforce server.	all
`%serverservices%`	A string specifying the role of the server. One of the following: • `standard` • `replica` • `broker` • `proxy`	all except archive

Argument	Description	Available for type
	• `commit-server` • `edge-server` • `forwarding-replica` • `build-server` • `P4AUTH` • `P4CHANGE`	
`%serverVersion%`	Version string for the server that terminated if the command exited early. Reason for termination is given in `%termType%`.	all except archive
`%specdef%`	Expanded to the spec string of the form in question.	form
`%submitserverid%`	If this is not a distributed installation, `%submitserverid%` is always empty. In a distributed installation, for any change trigger: • if the submit was run on the commit server, `%submitserverid%` equals `%serverid%`. • if the submit was run on the edge server, `%submitserverid%` does not equal `%serverid%`. In this case, `%submitserverid%` holds the edge server's server id. If there is a forwarding replica between the commit server and the edge server, then `%submitserverid%` actually holds the forwarding replica's server id. See **`p4 serverid`** in the *P4 Command Reference* for details.	`change-submit`, `change-content`, `change-commit`, Not available for push-* triggers.
`%terminated%`	The value of 0 indicates that the command completed. A value of 1 indicates that the command did not complete.	
`%termType%`	The reason for early termination. This might be one of the following: • `'p4 monitor terminate'`	all except archive

Argument	Description	Available for type
	• `client disconnect` • `maxScanRows` • `maxLockTime` • `maxResults` See also `%serverVersion%`.	
`%triggerMeta_action%`	Command to execute when trigger is fired. Last field of trigger definition. Set only when you run a script from the depot.	all except archive
`%triggerMeta_depotFile%`	Third field in trigger definition. Its meaning varies with the trigger type. For a change-submit trigger, it is the path for which the trigger is expected to match. For a form-out trigger, it might be the form type to which the trigger is expected to apply. See the description of the trigger types for more information on the meaning of this field.	all except archive
`%triggerMeta_name%`	Trigger name: first field from trigger definition. Set only when you run a script from the depot.	all except archive
`%triggerMeta_trigger%`	Trigger type: second field in trigger definition. Set only when you run a script from the depot.	all except archive
`%user%`	Perforce username of the triggering user.	all

Appendix

Perforce Server (p4d) Reference

Synopsis

Start the Perforce service or perform checkpoint/journaling (system administration) tasks.

Syntax

```
p4d [ options ]
p4d.exe [ options ]
p4s.exe [ options ]
p4d -j? [ -z | -Z ] [ args ... ]
```

Description

The first three forms of the command invoke the background process that manages the Perforce versioning service. The fourth form of the command is used for system administration tasks involving checkpointing and journaling.

On UNIX and Mac OS X, the executable is **p4d**.

On Windows, the executable is **p4d.exe** (running as a server) or **p4s.exe** (running as a service).

Exit Status

After successful startup, **p4d** does not normally exit. It merely outputs the following startup message:

```
Perforce server starting...
```

and runs in the background.

On failed startup, **p4d** returns a nonzero error code.

Also, if invoked with any of the **-j** checkpointing or journaling options, **p4d** exits with a nonzero error code if any error occurs.

Options

Server options	Meaning
-d	Run as a daemon (in the background)
-f	Run as a single-threaded (non-forking) process
-i	Run from **inetd** on UNIX
-q	Run quietly (no startup messages)

Server options	Meaning
`--pid-file[=file]`	Write the PID of the server to a file named `server.pid` in the directory specified by `P4ROOT`, or write the PID to the file specified by ***`file`***. This makes it easier to identify a server instance among many. The ***`file`*** parameter can be a complete path specification. The file does not have to reside in `P4ROOT`.
`-xi`	Irreversibly reconfigure the Perforce server (and its metadata) to operate in Unicode mode. Do not use this option unless you know you require Unicode mode. See the *Release Notes* and *Internationalization Notes* for details.
`-xu`	Run database upgrades and exit. This will no longer run automatically if there are fewer than 1000 changelists. Upgrades must be run manually unless the server is a DVCS personal server; in this case, any upgrade steps are run automatically.
`-xv`	Run low-level database validation and quit.
`-xvU`	Run fast verification; do not lock database tables, and verify only that the unlock count for each table is zero.
`-xD [ serverID ]`	Display (or set) the server's ***`serverID`*** (stored in the `server.id` file) and exit.

General options	Meaning
`-h`, `-?`	Print help message.
`-V`	Print version number.
`-A auditlog`	Specify an audit log file. Overrides `P4AUDIT` setting. Default is null.
`-Id description`	A server description for use with **`p4 server`**. Overrides `P4DESCRIPTION` setting.
`-In name`	A server name for use with **`p4 configure`**. Overrides `P4NAME` setting.
`-J journal`	Specify a journal file. Overrides `P4JOURNAL` setting. Default is `journal`. (Use `-J off` to disable journaling.)
`-L log`	Specify a log file. Overrides `P4LOG` setting. Default is `STDERR`.
`-p port`	Specify a port to listen to. Overrides `P4PORT`. Default `1666`.
`-r root`	Specify the server root directory. Overrides `P4ROOT`. Default is current working directory.

General options	Meaning
`-v` *`subsystem=level`*	Set trace options. Overrides value `P4DEBUG` setting. Default is null.
`-C1`	Force the service to operate in case-insensitive mode on a normally case-sensitive platform.
`--pid-file[=`*`name`*`]`	Write the server's PID to the specified file. Default name for the file is `server.pid`

Checkpointing options	Meaning
`-c` *`command`*	Lock database tables, run *`command`*, unlock the tables, and exit.
`-jc [` *`prefix`* `]`	Journal-create; checkpoint and `.md5` file, and save/truncate journal. In this case, your checkpoint and journal files are named *`prefix`*`.ckp.`*`n`* and *`prefix`*`.jnl.`*`n`* respectively, where *`prefix`* is as specified on the command line and *`n`* is a sequence number. If no *`prefix`* is specified, the default filenames `checkpoint.`*`n`* and `journal.`*`n`* are used. You can store checkpoints and journals in the directory of your choice by specifying the directory as part of the prefix. **Warning** If you use this option, it must be the last option on the command line.
`-jd` *`file`*	Journal-checkpoint; create checkpoint and `.md5` file without saving/truncating journal.
`-jj [` *`prefix`* `]`	Journal-only; save and truncate journal without checkpointing.
`-jr` *`file`*	Journal-restore; restore metadata from a checkpoint and/or journal file. If you specify the `-r $P4ROOT` option on the command line, the `-r` option must precede the `-jr` option.
`-jv` *`file`*	Verify the integrity of the checkpoint or journal specified by *`file`* as follows: • Can the checkpoint or journal be read from start to finish? • If it's zipped can it be successfully unzipped? • If it has an MD5 file with its MD5, does it match? • Does it have the expected header and trailer? This command does not replay the journal.

Checkpointing options	Meaning
	Use the `-z` option with the `-jv` option to verify the integrity of compressed journals or compressed checkpoints.
`-z`	Compress (in `gzip` format) checkpoints and journals. When you use this option with the `-jd` option, Perforce automatically adds the `.gz` extension to the checkpoint file. So, the command: `p4d -jd -z myCheckpoint` creates two files: `myCheckpoint.gz` and `myCheckpoint.md5`.
`-Z`	Compress (in `gzip` format) checkpoint, but leave journal uncompressed for use by replica servers. That is, it applies to `-jc`, not `-jd`.

Journal restore options	Meaning
`-jrc` *file*	Journal-restore with integrity-checking. Because this option locks the database, this option is intended only for use by replica servers started with the **`p4 replicate`** command.
`-jrF` *file*	Allow replaying a checkpoint over an existing database. (Bypass the check done by the `-jr` option to see if a checkpoint is being replayed into an existing database directory by mistake.)
`-b` *bunch* `-jr` *file*	Read *bunch* lines of journal records, sorting and removing duplicates before updating the database. The default is `5000`, but can be set to `1` to force serial processing. This combination of options is intended for use with by replica servers started with the **`p4 replicate`** command.
`-f -jr` *file*	Ignore failures to delete records; this meaning of `-f` applies only when `-jr` is present. This combination of options is intended for use with by replica servers started with the **`p4 replicate`** command. By default, journal restoration halts if record deletion fails. As with all journal-restore commands, if you specify the `-r $P4ROOT` option on the command line, the `-r` option must precede the `-jr` option.
`-m -jr` *file*	Schedule new revisions for replica network transfer. Required only in environments that use **`p4 pull -u`** for archived files, but **`p4 replicate`** for metadata. Not required in replicated environments based solely on **`p4 pull`**.
`-s` *-jr file*	Record restored journal records into regular journal, so that the records can be propagated from the server's journal to any replicas downstream of the server. This combination of options is intended for use in conjunction with Perforce technical support.

Replication and multi-server options	Meaning
`-a` *`host:port`*	In multi-server environments, specify an authentication server for licensing and protections data. Overrides `P4AUTH` setting. Default is null.
`-g` *`host:port`*	In multi-server environments, specify a changelist server from which to obtain changelist numbers. Overrides `P4CHANGE` setting. Default is null.
`-t` *`host:port`*	For replicas, specify the target (master) server from which to pull data. Overrides `P4TARGET` setting. Default is null.
`-u` *`serviceuser`*	For replicas, authenticate as the specified ***`serviceuser`*** when communicating with the master. The service user must have a valid ticket before replica operations will succeed.

Journal dump/restore filtering	Meaning
`-jd` *`file`* `db.`*`table`*	Dump **`db.`*`table`*** by creating a checkpoint ***`file`*** that contains only the data stored in **`db.`*`table`*** This command can also be used with non-journaled tables.
`-k db.`*`table1`*`,db.`*`table2`*`,... -jd` *`file`*	Dump a set of named tables to a single dump ***`file`***.
`-K db.`*`table1`*`,db.`*`table2`*`,... -jd` *`file`*	Dump all tables except the named tables to the dump ***`file`***.
`-P` *`serverid`* `-jd` *`file`*	Specify filter patterns for **`p4d -jd`** by specifying a ***`serverid`*** from which to read filters (see **`p4 help server`**, or use the older syntax described in **`p4 help export`**.) This option is useful for seeding a filtered replica.
`-k db.`*`table1`*`,db.`*`table2`*`,... -jr` *`file`*	Restore from ***`file`***, including only journal records for the tables named in the list specified by the `-k` option.
`-K db.`*`table1`*`,db.`*`table2`*`,... -jr` *`file`*	Restore from ***`file`***, excluding all journal records for the tables named in the list specified by the `-K` option.

Certificate Handling	Meaning
`-Gc`	Generate SSL credentials files for the server: create a private key and certificate file in `P4SSLDIR`, and then exit. Requires that `P4SSLDIR` be set to a directory that is owned by the user invoking the command, and that is readable only by that user. If

Certificate Handling	Meaning
	`config.txt` is present in `P4SSLDIR`, generate a self-signed certificate with specified characteristics.
`-Gf`	Display the fingerprint of the server's public key, and exit. Administrators can communicate this fingerprint to end users, who can then use the **`p4 trust`** command to determine whether or not the fingerprint (of the server to which they happen to be connecting) is accurate.

Configuration options	Meaning
`-cshow`	Display the contents of `db.config` without starting the service. (That is, run **`p4 configure show allservers`**, but without a running service.)

`-cset` *`server`*`#var=val`	Set a Perforce configurable without starting the service, optionally specifying the server for which the configurable is to apply. For example,
	
	`p4d -r . "-cset replica#P4JOURNAL=off"`
	
	`p4d -r .  "-cset replica#P4JOURNAL=off replica#server=3"`
	
	It is best to include the entire ***`variable=value`*** expression in quotation marks.

`-cunset` *`server`*`#var`	Unset the specified configurable.

Usage Notes

- On all systems, journaling is enabled by default. If `P4JOURNAL` is unset when **`p4d`** starts, the default location for the journal is `$P4ROOT`. If you want to manually disable journaling, you must explicitly set `P4JOURNAL` to `off`.
- Take checkpoints and truncate the journal often, preferably as part of your nightly backup process.
- Checkpointing and journaling preserve only your Perforce metadata (data *about* your stored files). The stored files themselves (the files containing your source code) reside under `P4ROOT` and must be also be backed up as part of your regular backup procedure.
- It is best to keep journal files and checkpoints on a different hard drive or network location than the Perforce database.
- If your users use triggers, don't use the `-f` (non-forking mode) option; the Perforce service needs to be able to spawn copies of itself ("fork") in order to run trigger scripts.

- After a hardware failure, the options required for restoring your metadata from your checkpoint and journal files can vary, depending on whether data was corrupted.
- Because restorations from backups involving loss of files under `P4ROOT` often require the journal file, we strongly recommend that the journal file reside on a separate filesystem from `P4ROOT`. This way, in the event of corruption of the filesystem containing `P4ROOT`, the journal is likely to remain accessible.
- The database upgrade option (`-xu`) can require considerable disk space. See the *Release Notes* for details when upgrading.

Related Commands

To start the service, listening to port `1999`, with journaling enabled and written to `journalfile`.	`p4d -d -p 1999 -J /opt/p4d/journalfile`

To checkpoint a server with a non-default journal file, the `-J` option (or the environment variable `P4JOURNAL`) must match the journal file specified when the server was started.	Checkpoint with:
	`p4d -J /p4d/jfile -jc`
	or
	`P4JOURNAL=/p4d/jfile ; export P4JOURNAL; p4d -jc`

To create a compressed checkpoint from a server with files in directory `P4ROOT`	`p4d -r $P4ROOT -z -jc`
To create a compressed checkpoint with a user-specified prefix of "ckp" from a server with files in directory `P4ROOT`	`p4d -r $P4ROOT -z -jc ckp`
To restore metadata from a checkpoint named `checkpoint.3` for a server with root directory `P4ROOT`	`p4d -r $P4ROOT -jr checkpoint.3` (The `-r` option must precede the `-jr` option.)
To restore metadata from a compressed checkpoint named `checkpoint.3.gz` for a server with root directory `P4ROOT`	`p4d -r $P4ROOT -z -jr checkpoint.3.gz` (The `-r` option must precede the `-jr` option.)

Appendix

Moving a Perforce server to a new machine

How you move an existing Perforce server from one machine to another depends on the following factors:

- whether the machines use the same byte order
- whether the machines use different byte ordering, but the same text file (CR/LF) format
- whether the machines use different byte order *and* a different text file format.

Additional considerations apply if the new machine has a different IP address/hostname.

The Perforce server stores two types of data under the Perforce root directory: *versioned files* and a *database* containing *metadata* describing those files. Your versioned files are the ones created and maintained by your users, and your database is a set of Perforce-maintained binary files holding the history and present state of the versioned files. In order to move a Perforce server to a new machine, both the versioned files and the database must be successfully migrated from the old machine to the new machine.

For more about the distinction between versioned files and database, as well as for an overview of backup and restore procedures in general, see Chapter 6, "Backup and Recovery" on page 105.

For more information, see "Moving a Perforce Server" in the Perforce knowledge base:

http://answers.perforce.com/articles/KB_Article/Moving-a-Perforce-Server

Moving between machines of the same byte order

If the architecture of the two machines uses the same byte order (for example, SPARC/SPARC, x86/x86, or even 32-bit Windows to 64-bit Windows), the versioned files and database can be copied directly between the machines, and you only need to move the server root directory tree to the new machine. You can use **`tar`**, **`cp`**, **`xcopy.exe`**, or any other method. Copy everything in and under the `P4ROOT` directory - the `db.*` files (your database) as well as the depot subdirectories (your versioned files).

1. Back up your server (including a **`p4 verify`** before the backup) and take a checkpoint.
2. On the old machine, stop **`p4d`**.
3. Copy the contents of your old server root (`P4ROOT`) and all its subdirectories on the old machine into the new server root directory on the new machine.
4. Start **`p4d`** on the new machine with the desired flags.
5. Run **`p4 verify`** on the new machine to ensure that the database and your versioned files were transferred correctly to the new machine.

(Although the backup, checkpoint, and subsequent **`p4 verify`** are not strictly necessary, it's always good practice to verify, checkpoint, and back up your system before any migration and to perform a subsequent verification after the migration.)

Moving between different byte orders that use the same text format

If the internal data representation (big-endian vs. little-endian) convention differs between the two machines (for example, Linux-on-x86/SPARC), but their operating systems use the same CR/LF text file conventions, you can still simply move the server root directory tree to the new machine.

Although the versioned files are portable across architectures, the database, as stored in the **db.*** files, is not. To transfer the database, you will need to create a checkpoint of your Perforce server on the old machine and use that checkpoint to re-create the database on the new machine. The checkpoint is a text file that can be read by a Perforce server on any architecture. For more details, see "Creating a checkpoint" on page 106.

After you create the checkpoint, you can use **tar**, **cp**, **xcopy.exe**, or any other method to copy the checkpoint file and the depot directories to the new machine. (You don't need to copy the **db.*** files, because they will be re-created from the checkpoint you took.)

1. On the old machine, use **p4 verify** to ensure that the database is in a consistent state.
2. On the old machine, stop **p4d**.
3. On the old machine, create a checkpoint:

   ```
   p4d -jc checkpointfile
   ```

4. Copy the contents of your old server root (P4ROOT) and all its subdirectories on the old machine into the new server root directory on the new machine.

   (To be precise, you don't need to copy the **db.*** files, just the checkpoint and the depot subdirectories. The **db.*** files will be re-created from the checkpoint. If it's more convenient to copy everything, then copy everything.)

5. On the new machine, if you copied the **db.*** files, be sure to remove them from the new P4ROOT before continuing.
6. Re-create a new set of **db.*** files suitable for your new machine's architecture from the checkpoint you created:

   ```
   p4d -jr checkpointfile
   ```

7. Start **p4d** on the new machine with the desired flags.
8. Run **p4 verify** on the new machine to ensure that the database and your versioned files were transferred correctly to the new machine.

Moving between Windows and UNIX

In this case, both the architecture of the system *and* the CR/LF text file convention may be different. You still have to create a checkpoint, copy it, and re-create the database on the new platform, but when you move the depot subdirectories containing your versioned files, you also have to address the issue of the differing linefeed convention between the two platforms.

Depot subdirectories can contain both text and binary files. The text files (in RCS format, ending with "`,v`") and binary files (directories of individual binary files, each directory ending with "`,d`") need to be transferred in different ways in order to translate the line endings on the text files while leaving the binary files unchanged.

As with all other migrations, be sure to run **`p4 verify`** after your migration.

Warning	Windows is a case-insensitive operating system. Files that differ by case only on a UNIX server will occupy the same namespace when transferred to a Windows machine. For instance, files `Makefile` and file `makefile` on a UNIX server will appear to be the same file on a Windows machine. Due to the risk of data loss due to case collision, migrations from UNIX servers to Windows are not supported.

Contact Perforce Technical Support for assistance when migrating a Perforce server from Windows to UNIX.

Changing the IP address of your server

If the IP address of the new machine is not the same as that of the old machine, you will need to update any IP-address-based protections in your protections table. See "Authorizing access" on page 83 for information on setting protections for your Perforce server.

If you are a licensed Perforce customer, you will also need a new license file to reflect the server's new IP address. Contact Perforce Technical Support to obtain an updated license.

Changing the hostname of your server

If the hostname of the new machine serving Perforce is different from that of its predecessor, your users must change their `P4PORT` settings. If the old machine is being retired or renamed, consider setting an alias for the new machine to match that of the old machine, so that your users won't have to change their `P4PORT` settings.

Appendix

Perforce Server Control (p4dctl)

The Perforce Service Control (**p4dctl**) utility allows you to manage Perforce services running on the local host. Non-root users can administer the services they own, while **root** may administer all services, but may not own any.

Note	**p4dctl** can only be obtained as part of the UNIX package installation. It is not supported on Windows.

You use the **p4dctl** utility to configure the environment in which services run and to manage the services themselves. The basic workflow for an administrator using the **p4dctl** utility is as follows:

1. Edit a configuration file that defines the environment for the services you want to control.
2. Execute **p4dctl** commands to start and stop services, to get information about services, and to checkpoint services.

You can use a single **p4dctl** command to manage all services or an arbitrary group of services by assigning them a common *name* in the **p4dctl** configuration file.

p4dctl introduces no new environment variables; it enforces strict control of the environment of any service it starts according to the directives in the **p4dctl** configuration file. This prevents failures that stem from the differences between the user's environment and that of **root**.

Installation

p4dctl is installed as part of the UNIX package installation. The installation process automatically creates a master configuration file located at `/etc/perforce/p4dctl.conf`.

As part of the package install, **p4dctl** is installed as a **setuid** root executable because it uses root privileges to maintain pid files for compatibility with systems that use them. For all other operations, **p4dctl** runs with the privileges of the executing user. This allows non-root users to start and stop the services they own while having the pid file remain up to date.

Configuration file format

p4dctl uses a configuration file, `p4dctl.conf`, to control the following:

- service settings for the services started with the **p4dctl** command.
- settings for the **p4dctl** utility itself
- service processes managed by **p4dctl**, for example checkpointing and journal rotation
- the environment in which managed services are running

  The environment is configured using environment variables that may be defined globally or for a specific service. The service type determines which variables must be defined. See "Service types and required settings" on page 246 for information on the requirements for each type.

A **p4dctl** configuration file is made up of an **environment** block and one or more **server** type blocks. The following sections describe each type in detail.

The configuration file may also contain comments; these are designated by starting the comment line with the **#** sign.

Settings specified outside of a server block are global and are merged into the settings of all services. They take the following form:

setting_name = value

For example:

```
PATH = /bin:/user/bin
```

Environment block

An environment block defines environment variables that are applied to one or more services. You can have more than one environment block. Server-specific environment blocks settings override corresponding settings in global environment blocks.

An environment block is defined using the following syntax:

```
Environment
{
   variable = value
}
```

An environment block may be used outside of a server block or inside of it.

- If the block is outside a server block, the variables it contains are applied to the environment of all processes created by **p4dctl**.
- If the block is inside a server block, the variables it defines are set only in the environment of that server's processes, but they do override corresponding settings at the environment level.

For example, the following settings outside a server block ensure that the owner is set to `perforce`, logging is enabled, and the correct `P4CONFIG` files are used.

```
Environment
{
     P4DEBUG     =  "server=1" # Embedded = requires quotes
     P4LOG       =  log
     P4CONFIG    =  .p4config
}
```

Server block

A server block defines settings and variables that apply only to the specified type of service. Type may be one of the following:

Type	Meaning
`p4d`	Perforce server
`p4p`	Perforce proxy server
`p4broker`	Perforce broker
`p4ftp`	Perforce FTP plugin
`p4web`	Perforce web client
`other`	Any other service

A server block is defined using the following syntax:

```
server_type name
{
   setting = value
   Environment
   {
       variable = value
   }
}
```

The specified `name` name must refer to services of a given type, but the name can include different types of servers. This allows you to control or query groups of heterogeneous servers that share the same name.

For example, a configuration that defines p4d, proxy, and p4ftp services all using the name `main` can use a command like the following to stop all these services without affecting any other services.

```
$ p4dctl stop main
```

You can define the following variables within server blocks. `Owner` and `Execute` are required for all server types.

Setting	Meaning
`Owner`	The owner of the service. The service is started under the owner's account and with their privileges. The user can also use **`p4dctl`** to manage the server they own. Required.
`Execute`	The path to the binary to execute when starting this server. Required.
`Args`	A string containing the arguments to be passed to the binary specified with `Execute`.

Setting	Meaning
	The string *must* be quoted if it contains a space.
`Enabled`	Set to `FALSE` to disable the service and not start it with the **`p4dctl start`** command. Default: `TRUE`
`Umask`	An octal value specifying the `umask` to be applied to the child processes for this service. The default `umask` on most Linux/Unix systems is 022, which means all new files are readable by all users. Setting this variable to 077 ensures that the files created by this service are only accessible to the owner of the service.
`Prefix`	A string containing a prefix to apply when checkpointing the server or rotating the journal. This prefix is passed down to the relevant **`p4d`** command if needed. Default: none

`PrettyNames`	Set to `true` to have **`p4dctl`** format the names of the server processes it starts, in an informative way.
	
	In the following example, the **`p4d`** process is qualified with its host and port name when `PrettyNames` is set to true.
	
	`PrettyNames=true`
	`  perforce callto:21397%201%200%2010[21397 1  0 10]:48 ?  00:00:00 p4d`
	`                                                [blacksphere/1666]`
	`PrettyNames=false`
	`  perforce callto:21725%201%200%2010[21725 1  0 10]:50 ?  00:00:00`
	`                                                 /usr/sbin/p4d`
	
	Default: true

Service types and required settings

Each service type requires that you define the `owner` of the server (which cannot be `root`) and the `execute` path where its binary can be found. For example, for the `p4d` type, you specify the path to the **`p4d`** binary, for the broker, you must provide the path to the **`p4broker`** binary, and so on.

For each service type, you must also define certain environment variables; these are listed in the following subsections.

Type	Variable	Setting
`p4d`	`P4PORT`	Port to use for this service
	`P4ROOT`	Path to the server's root directory

Type	Variable	Setting
	PATH	Search path to be used for this service
p4p	PORT	Port to use for this service
	P4TARGET	Address of the target Perforce service
	P4ROOT	Path to the server's root directory
	PATH	Search path to be used for this service
p4broker	P4BROKEROPTIONS	Command line options to pass to this broker
p4ftp	PORT	Address of the target Perforce service
	P4FTPPORT	Port to use for serving FTP requests
p4web	PORT	Address of the target Perforce server
	P4WEBPORT	Port to use for serving HTTP requests
	P4ROOT	Path to the server's root directory
	PATH	Search path to be used for this service

Configuration file examples

The following example shows a basic Perforce server (**p4d**) configuration file.

```
p4d minimum
{
  Owner   = perforce
  Execute = /usr/bin/p4d
  Environment

  {
   P4ROOT     = /home/perforce/p4-main
   P4PORT     = 1666
   PATH       = /bin:/usr/bin:/usr/local/bin
  }
}
```

In the following example, the PATH environment variable is defined once, globally for both the service and its proxy. Note how the name 'test' is used to refer to both.

```
Environment
{
  PATH       = /bin:/usr/bin:/usr/local/bin
}

p4d test
{
  Owner   = perforce
  Execute = /usr/bin/p4d

Environment
  {
   P4ROOT     = /home/perforce/p4-main
   P4PORT     = "localhost:1667"
  }
}

p4p test
{
  Owner   = perforce
  Execute = /usr/bin/p4p

  Environment
  {
   P4ROOT     = /home/perforce/proxy-main
   P4PORT     = 1666
   P4TARGET   = "localhost:1667"
  }
}
```

Using multiple configuration files

You can modularize your configuration by creating multiple configuration files and directories and including these in your configuration.

- To include a specific file, use the following syntax:

```
include pathToFile
```

- To include directories, use the following syntax:

```
include directoryPath
```

  When including directories, **p4dctl** requires that names for files included end in `.conf`.

The following example shows a multiple file configuration.

```
Environment
{
  PATH       = /bin:/usr/bin:/usr/local/bin
}

  include /etc/perforce/p4dctl.conf.d
```

p4dctl commands

p4dctl commands can be divided into three categories: commands that stop and start services, commands that checkpoint services, and commands that return information about services.

The **p4dctl checkpoint** command is similar to the **p4d -jc** command.

The following table presents a summary of command syntax for each category. The parameter **-a** specifies all servers.

Category	Syntax
Control services	**p4dctl [*options*] start [-t *type*] -a** **p4dctl [*options*] start [-t *type*] *name*** **p4dctl [*options*] stop [-t *type*] -a** **p4dctl [*options*] stop [-t *type*] *name*** **p4dctl [*options*] restart [-t *type*] -a** **p4dctl [*options*] restart [-t *type*] *name***
Checkpoints and journals	**p4dctl [*options*] checkpoint -a** **p4dctl [*options*] checkpoint *name***
Query services	**p4dctl [*options*] status [-t *type*] -a** **p4dctl [*options*] status [-t *type*] *name*** **p4dctl [*options*] list [-t *type*]** **p4dctl [*options*] list [-t *type*] *name*** **p4dctl [*options*] env [-t *type*] -a *var* [*var*…]** **p4dctl [*options*] status [-t *type*] *name var* [*var*…]**

Options to **p4dctl** commands are described in the following table. The meaning of variable names other than option names is explained in "Configuration file format" on page 243.

Options	Meaning
`-c` *configFile*	Path to the configuration file Default: `/etc/perforce/p4dctl.conf`
`-p` *pidDir*	Path to the pid file directory. Default: `/var/run`

Options	Meaning
`-q`	Send output to syslog instead of `STDOUT` or `STDERR`
`-v` *`level`*	Set debug level (0-9) For more information, see the description of the `P4DEBUG` environment variable in *P4 Command Reference*.
`-V`	Display version and exit.

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/).

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/).